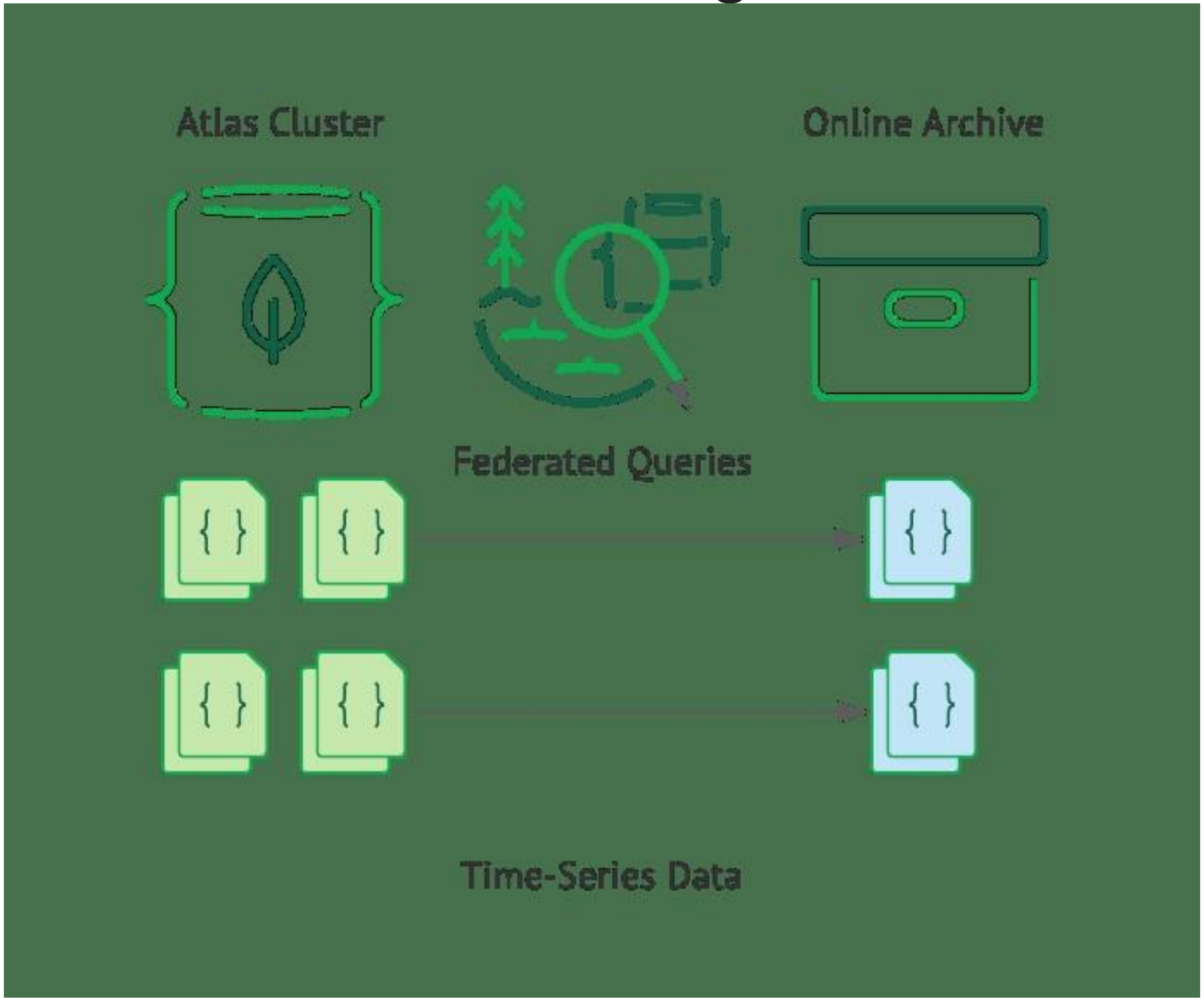
Time Series Data in MongoDB



Introduction to Time-Series Data: Time-series data in MongoDB refers to datasets where each data point is associated with a timestamp. This type of data is prevalent in applications that record events over time, such as IoT sensor readings, financial transactions, or system logs. Efficiently managing and analyzing time-series data is crucial for extracting valuable insights from temporal patterns.

Time-Series Collections and Indexing:

1. **Time-Series Collections:**
   - In MongoDB, dedicated collections are often used to organize time-series data.
   - Each document within the collection represents a specific data point, and the timestamp associated with it plays a central role in temporal ordering.
2. **Timestamp as Primary Key:**
   - Designing time-series collections with the timestamp as the primary key ensures natural chronological ordering of data.
   - This design choice simplifies and accelerates queries that involve time-based filtering and analysis.
3. **Indexing for Performance:**

- To optimize queries involving time-based criteria, it's essential to create indexes on the timestamp field.
- MongoDB's compound indexes can be leveraged for scenarios where queries involve additional fields along with time-based conditions.

4. **TTL Index for Data Expiry:**
    - Time-To-Live (TTL) indexes provide a convenient mechanism for automatically removing documents older than a specified duration.
    - This feature is beneficial for managing storage and maintaining the relevance of time-series data over time.

Perform Time-Based Analytics:

1. **Aggregation for Time Intervals:**
    - MongoDB's aggregation framework is a powerful tool for performing analytics on time-series data.
    - Aggregation pipelines can be constructed to summarize data over specific time intervals, such as hourly, daily, or monthly aggregations.
2. **Temporal Queries:**
    - Temporal queries involve filtering and analyzing data within defined time ranges.
    - Operators like **$gte** (greater than or equal) and **$lte** (less than or equal) facilitate precise time-based filtering, allowing users to extract data for specific periods.
3. **Window Functions:**
    - Window functions enable calculations over a moving window of time, facilitating computations like moving averages or rolling sums.
    - This capability is valuable for understanding trends and patterns within dynamic datasets.
4. **Bucketing for Granularity:**
    - Bucketing involves grouping data into intervals or bins based on time, allowing for detailed analysis at different levels of granularity.
    - Aggregating data in buckets provides insights into temporal patterns and trends.
5. **Time-Series Indexes:**
    - MongoDB introduces specialized time-series indexes, such as the **Time Series Collections** feature in MongoDB 5.0.
    - These indexes are tailored to enhance performance and storage efficiency for time-series data, further optimizing query execution.

Introduction to MongoDB and QuestDB

Founded in 2009, MongoDB is an open-source, NoSQL, document-oriented database, subject to Server Side Public License (commercial license also available). It leads the ranking for document stores (Feb 2023) on DB-Engines. MongoDB is implemented in C++, with time-series data support added in version 5.0.

QuestDB is an open-source time-series database licensed under Apache License 2.0. It is designed for high throughput ingestion and fast SQL queries. It supports schema-agnostic ingestion using the InfluxDB line protocol, PostgreSQL wire protocol, and a REST API for bulk imports and exports.

A brief comparison based on DB-Engines:

Name	**MongoDB**	**QuestDB**
Primary database model	Document store	Time Series DBMS
Overall rank	#5	#146
Other rank	#1 Document stores	#11 Time Series DBMS
Implementation language	C++	Java (low latency, zero-GC), C++
SQL #	Read-only SQL queries via the MongoDB Connector for BI	SQL-like query language
APIs and other access methods	proprietary protocol using JSON	InfluxDB Line Protocol, Postgres Wire Protocol, HTTP, JDBC

MongoDB's default storage engine is WiredTiger, which integrates concepts from both the B-tree and the LSM-tree storage models.

Unlike a traditional RDBMS, data in MongoDB is not organized in tables, rows, and columns, but in collections of documents that contain fields. MongoDB supports Time-Series Collections, which are behind-the-scenes materialized views backed by an internal collection.

With its column-based storage engine, QuestDB stores data densely in arrays. It packs SIMD instructions that speed up queries by leveraging modern CPUs. The data is sorted by time natively and stored in time partitions. Those partitions are append-only and versioned. QuestDB only lifts the latest version of the partition in memory for any given time-based query, and the query performance does not slow down as more data is inserted.

Benchmark

As usual, we use the industry standard Time Series Benchmark Suite (TSBS) as the benchmark tool. Unfortunately, TSBS upstream does not support MongoDB time series collections. Hence, we use the patch from this pull request.

The hardware we use for the benchmark is the following:

- c6a.12xlarge EC2 instance with 48 vCPU and 96 GB RAM
- 500GB gp3 volume configured for the maximum settings (16,000 IOPS and 1,000 MB/s throughput)

The software side of the benchmark is the following:

- Ubuntu 22.04
- MongoDB Community Edition 6.0.4 with the default configuration
- QuestDB 7.0 with the default configuration

Ingestion

To benchmark ingestion, we used the following commands:

```
$ ./tsbs_generate_data --use-case="cpu-only" --seed=123 --scale=4000 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-01-03T00:00:00Z" --log-interval="10s" --format="mongo" > /tmp/mongo_data
```

```
$ ./tsbs_load_mongo --file=/tmp/mongo_data --document-per-event=true --timeseries-collection=true --workers=10
```

Here, we use a cpu-only scenario with two days of CPU data from 4,000 simulated hosts. We use 10 client connections for the benchmark in both cases. In MongoDB, we also use time series collections, as recommended by MongoDB for time-series data.

The diagram below summarizes the results for ingestion performance:

To load just over 691 million metrics, MongoDB requires more than 2,000 seconds, while it takes less than 90 seconds for QuestDB; QuestDB is 24 times faster than MongoDB.

It is also worthwhile noting that MongoDB recommends ingesting measures in batches. For better performance, the measures are grouped by metadata rather than time. Otherwise, ingestion would be less efficient. Therefore, ingestion performance is dependent on metadata and sensitive to the order of the fields in the document. Insertion is completed in BSON format, a serialization format used to store documents and make remote procedure calls in MongoDB.

By contrast, QuestDB's ingestion is not affected by the order of metadata in a batch. This is due to QuestDB's column-based storage model. Ingesting data is less error-prone for developers.

Query performance

QuestDB is designed for time-series data; as such, it is not surprising that QuestDB outperforms MongoDB for ingestion. However, query performance is equally important for time-series data analysis.

As part of the standard TSBS benchmark, we compare both databases for a few types of queries, which are popular for time-series data:

- lastpoint: The last reading for each host
- groupby-orderby-limit: The last 5 aggregate readings (across time) before a randomly chosen endpoint
- high-cpu-all: All the readings where one metric is above a threshold across all hosts
- double-groupby-1: Aggregate across both time and host, giving the average of 1 CPU metric per host per hour for 24 hours
- single-groupby-1-8-1: Simple aggregate (MAX) on one metric for 8 hosts, every 5 mins for 1 hour
- cpu-max-all-1: Aggregate across all CPU metrics per hour over 1 hour for a single host

Here are some sample commands to generate and run the queries:

```
$ ./tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=4000 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-01-03T00:00:00Z" --queries=1000 --query-type="lastpoint" --format="mongo" > /tmp/mongo_query_lastpoint
```

```
$ ./tsbs_run_queries_mongo --file=/tmp/mongo_query_lastpoint --workers=10
```

Again, we use 10 concurrent client connections for the comparison. We measure how much time each query takes, in milliseconds per query:

Two conclusions from the benchmark:

- MongoDB performs slightly better for simple aggregations (single-groupby-1-8-1 and cpu-max-all-1).
- QuestDB outperforms MongoDB significantly for more complicated queries.

QuestDB's underperformance for simple aggregations may be partly explained by the fact that TSBS is not designed to use bind variables. Instead, it sends lots of unique queries making QuestDB's query cache redundant. Moreover, the query engine uses bytecode generation in these particular queries, so lots of one-off queries put some pressure on JVM metaspace.

Ease of use

Time-series data can provide great insight if you know what to look for and how to query the data to reveal otherwise hidden trends. Ideally, the database should encourage interactive exploration so developers can experiment with different query iterations effortlessly. Therefore, we will now compare both databases for their query language.

The query language for data aggregation

MongoDB is NoSQL and supports the MongoDB query language (MQL), which is JavaScript-based and unique to MongoDB. The learning curve can be steep. MongoDB recently started to support SQL, but this is not the case for time-series queries.

Querying in QuestDB is done via standard SQL with time-series extensions.

Looking at MongoDB's documentation, we found a sample to calculate the total consumption of AC units per building per hour on a sample dataset:

```
var pipelineBuildingsSummary = [

// Calculate each unit's energy consumed over the last hour for each reading

{"$setWindowFields": {

  "partitionBy": "$deviceID",

  "sortBy": {"timestamp": 1

  },

"output": {

  "consumedKilowattHours": {

    "$integral": { "input": "$powerKilowatts", "unit": "hour", },
      "window": { "range": [-1, "current"], "unit": "hour", },
      },
   },
}},
// Sort each reading by unit/device and then by timestamp
{"$sort": { "deviceID": 1, "timestamp": 1, }},
// Group readings together for each hour for each device using
// the last calculated energy consumption field for each hour
{"$group": {
   "_id": {
      "deviceID": "$deviceID",
      "date": { "$dateTrunc": { "date": "$timestamp", "unit": "hour", } }, },
      "buildingID": {"$last": "$buildingID"},
      "consumedKilowattHours": {"$last": "$consumedKilowattHours"},
 }},
// Sum together the energy consumption for the whole building
// for each hour across all the units in the building
{"$group": {
   "_id": {
      "buildingID": "$buildingID",
      "dayHour": {"$dateToString": {"format": "%Y-%m-%d %H", "date": "$_id.date"}}, },
      "consumedKilowattHours": {"$sum": "$consumedKilowattHours"},
 }},
// Sort the results by each building and then by each hourly summary
{"$sort": { "_id.buildingID": 1, "_id.dayHour": 1, }},
// Make the results more presentable with meaningful field names

{"$set": {

   "buildingID": "$_id.buildingID",

   "dayHour": "$_id.dayHour", "_id": "$$REMOVE",

 }},

];
```

With QuestDB, the query syntax is shorter and simpler:

```
SELECT timestamp, buildingID, SUM(powerKilowats)

FROM device_readings

SAMPLE BY 1h ALIGN TO CALENDAR;
```

Any developer familiar with SQL should understand this query. The only part that needs an explanation is SAMPLE BY, which is specifically designed to aggregate large datasets based on a time unit.

In addition, QuestDB provides the FILL keyword, one of the many ways to identify and fill missing data, which is a common first step in data analysis:

```
SELECT timestamp, buildingID, SUM(powerKilowats)

FROM device_readings

SAMPLE BY 1h FILL(LINEAR) ALIGN TO CALENDAR;
```

MongoDB provides two aggregation operators, $densify and $fill, to complete similar tasks. However, they are not as straightforward as using SAMPLE BY. The operator $densify identifies missing data and fills it in, but it does not resample the dataset.

If you want to fill gaps and values, you would need to use a densify and then a fill aggregation, which would add about 20 extra lines of code to the pipeline.

Other considerations

We have covered the two main factors for choosing the right time-series database: performance and usability. However, there are other aspects to consider when building a system with a Time-Series Database (TSDB).

Lifecycle policies

For most time-series data applications, the value of each data point diminishes over time. Therefore, we should also look at data lifecycle management as part of the database evaluation.

QuestDB offers the ability to detach partitions for cheaper storage and to reattach them later if required. In addition, the QuestDB engineering team is working on automating a feature for automating this process.

MongoDB allows auto-deletion of data. Once removed, data can only be recovered from a backup. In QuestDB, if you want to delete rather than keep older data for future use, you can invoke the DROP PARTITION statement periodically.

Ecosystem and support

QuestDB is a younger database. However, QuestDB builds on the immensely popular SQL language and is compatible with the Postgres wire protocol. In addition, QuestDB has a vibrant and supportive community: users get to interact with the engineering team building QuestDB.

MongoDB has an impressive ecosystem and has been around for a long time. As a more mature product, MongoDB is the most popular Document store database.

Conclusions:

By achieving these learning objectives, participants will gain the skills necessary to work with time-series data in MongoDB. They will understand how to model, ingest, and query time-series collections, implement indexing strategies, and perform advanced analytics on temporal data. This knowledge is essential for applications that involve the analysis and visualization of time-dependent trends and patterns.