CLOUDERA

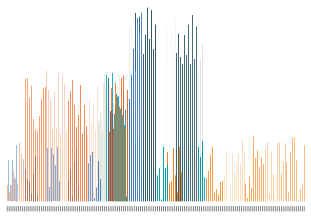
WHITE PAPER

TIME SERIES DATA WAREHOUSE

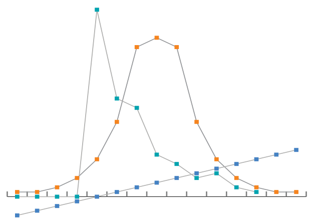
A reference architecture utilizing a modern data warehouse, based on Cloudera 6

TIME SERIES DATA ANALYSIS

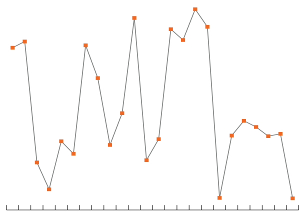
Time series is different from traditional statistical analysis



HIGH-DIMENSIONAL
100'S OF PARAMETERS
Oil yield with temperature, pressure. Medical, with heart rate, blood pressure. Network quality with weather, traffic, events.



NOT A NORMAL DISTRIBUTION
COMPLICATED. MULTIPLE PROCESSORS
Smart meters weekday vs. weekend. Complex manufacturing processes.



NON-STATIONARY
SLIDING WINDOWS OF DATA
Continuously shifting

DESIRE TO DETECT PATTERN
Real time data vs. historical data. Input keeps changing at rapid pace.

Introduction

Cloudera's data platform is commonly used by Fortune 2000 companies all over the world, for a variety of large scale ETL/data curation, ad-hoc reporting, ML model training, and other data-driven, mission critical applications. In this document, we will share how these enterprises utilize the same platform and data warehousing technology for high scale, time series applications.

A time series data warehouse provides the ability to report and analyze across data generated by a large number of data sources that generate data at regular intervals, such as sensors, devices, IoT entities, and financial markets. This data can also be queried, in real time, on the Cloudera platform in conjunction with other data sources from the organization and historical data, or to perform advanced analytics workloads such as statistical modeling and machine learning.

Predictive maintenance

Large manufacturers collect sensor data from each manufacturing robot at each factory floor, to correlate patterns and understand what leads up to specific events, that later causes downtime due to need of maintenance. These organizations are looking to optimize processes so that spare parts can be available in a timely manner, to avoid downtime.

Capacity planning and optimization

Large utilities and telecom organizations, as well as broadcasting and supply-chain dependent organizations, use time series data warehousing to better plan / replan manufacturing pipelines and supply chains, to optimize / plan for peak hours (to prevent downtimes), or to do better pricing.

Quality optimization

A frequent add-on use case to the ones above is optimization of quality and quality processes, by collecting samples from tests and other sensors. The aim is to prevent the significant costs or fines related to having to pull products later in the production pipeline (or, in the worst-case scenario, off the market), and to avoid liability and penalty costs.

Yield optimization

In large pharmaceuticals and chemical manufacturing, time series data analysis at large scale is done to optimize yield.

Modern Data Warehousing

REQUIRES ANALYSIS OF TIME SERIES DATA

- Automated maintenance and continuous plant uptime
- Targeted and personalized customer service and promotion
- Automated network utility and cost optimization
- Real-time fraud prevention and threat detection
- Quality and yield optimization
- Continuous operations dashboards

MEDICAL DATA

OIL YIELDS

SMART METERS

Table of Contents

Requirements

The architecture outlined in this document describes a reference solution for time series use cases on the Cloudera 6 platform (CDH 6). The solution addresses these high- level time series requirements:

- Extraction of time series measurements from data sources. A measurement is generally a single value from a single source (e.g., a sensor) at a single timestamp
- Data sources span a wide variety of interfaces, e.g., files, message queues, IoT hubs, REST
- Data sources can span a wide variety of data serializations, e.g. CSV, JSON, XML
- The total of all data sources of an application can be a very high rate of measurement, e.g., millions of measurements per second
- Enrichment of measurements with additional information, e.g., the device that a sensor is attached to. This allows users to query the measurements in more meaningful ways than only what was provided by the data source
- Querying of the measurements by application users
- Measurements should be available across a deep history, from the very latest to those far in the past
- Queries over measurements should execute quickly, enabling an interactive and exploratory user mindset that encourages driving the most value out of the data
- Queries over measurements should take advantage of existing tooling, such as standard business intelligence software
- Measurements should quickly become visible for user queries, e.g. within 30 seconds since the measurement took place
- Advanced querying of the measurements with statistics and machine learning libraries, and widely used languages, such as Python and R
- Automatic handling of updates. These could be intentional, such as data corrections, or unintentional, such as duplicates
- Ability to deploy the application either on premises or on a public cloud
- Data secured and governed, including authorization, authentication, auditing, and encryption

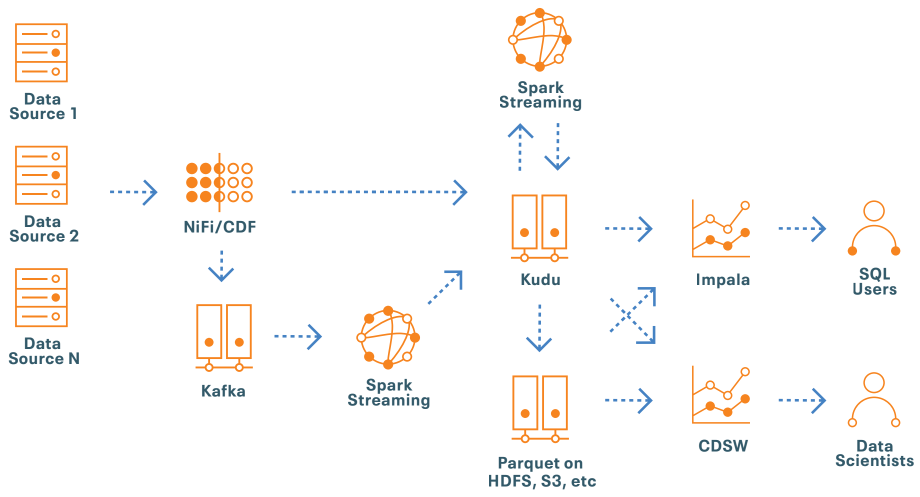
High-level architecture

The CDH 6 platform provides a variety of open source components that can be configured together to build applications. This reference architecture uses multiple CDH 6 components to solve the time series requirements that were described in the previous section:

- Apache NiFi, for data ingestion
- Apache Kafka, for storage of pre-processed measurements
- Apache Spark, for data pre-processing, if required
- Apache Kudu, for storage of recent measurements
- Apache Parquet, for storage of historical measurements
- Apache Impala, for user query of all measurement

This architecture is capable of scaling to:

- Ingestion and processing of millions of measurements per second
- Latency in the seconds from real world measurement to user query
- Dozens of concurrent user queries
- Petabytes of available measurements history



The following sections detail four major concerns of the architecture:

- The data model, which describes how the data is stored at rest
- The ingestion flow, which describes how the data is extracted from source systems and loaded into the cluster
- The processing job, which describes how the data is transformed prior to user query
- The user queries, which describes how users can ask questions of the data

Note that the intention of this document is to describe application architecture decisions that are specific to time series use cases and does not intend to cover the full breadth of CDH application architectures. For deeper consultation on CDH application architectures please contact your Cloudera account team or send an email to sales@cloudera.com.

Reference application

This reference architecture is accompanied by a reference application, ***available from Cloudera,*** that implements and demonstrates the functionality described by this architecture. This reference application can be considered an out-of-the-box proof of concept of many of the topics described in this document. It is advisable to start a CDH 6 time series proof of concept with this reference application and make required modifications from that point, instead of starting a new application from scratch.

Kudu

Kudu is a database in the CDH 6 platform that is suitable for high velocity ingest, high volume scans, and fast key lookups. These characteristics make it ideal for landing and serving new measurements to user queries. If required, as the data in Kudu ages it can be rolled off to Parquet files on HDFS or object stores to take advantage of cheaper storage and larger capacity scalability.

This section describes some Kudu tables that would typically be found in a time series data warehouse. Not all of these may be required for your application, and likewise some may be required but with variations to meet specific requirements not captured by this document.

Measurement table

This table stores the processed measurements as one record per measurement.
The columns of a Kudu measurement record generally consist of:

- The identifier of the metric that is being measured over time, such as the identifier of an individual sensor. For this reference architecture we will name this `metric_id`
- The timestamp of the measurement. For this reference architecture we will name this `time`
- The value of the metric at the timestamp. For this reference architecture we will name this `value`
- Optionally, one or more additional metadata fields about the measurement or about the metric

The names of the columns can be customized for each individual application. The length of the names will not impact storage costs or query performance. Where multiple metrics are measured together at the same timestamp, and provided on the same measurement message to the application, then the multiple values could be modeled as individual value columns on the same Kudu record. If the multiple metrics are at different timestamps, or provided on separate measurement messages, then it is typically more efficient to store them as separate records in the Kudu table.

Data types

- The metric ID column should use the `INT` data type. This will allow up to approximately four billion unique metrics.
- The timestamp column should use the `TIMESTAMP` data type. This will allow the user queries to make best use of the available time-related functions and syntax.
- The value column should generally use the `FLOAT` or `DOUBLE` data type, depending on the range and precision required. `FLOAT` will use at most 4 bytes, and `DOUBLE` will use at most eight bytes, however Kudu column encoding may considerably reduce the average size of the value.
- Note that a floating point data type such as `FLOAT` and `DOUBLE` cannot store all numbers in its range at full precision. This is generally acceptable for physical sensor readings. For time series applications that require exact values, such as financial market prices, use the `DECIMAL` data type.
- If there are multiple possible data types for a measurement’s value, then there should be separate value fields for each possible data type. Only one of these value fields should be non-null for each measurement record. Optionally, a column can be added that indicates which value column to use for that measurement.

- Metadata fields should use numeric or boolean types where possible. Long strings that are mostly unique should generally be avoided as they will greatly increase the size of the table on disk.

Primary key

The primary key of the Kudu measurement table should generally be two fields: `time` and `metric_id`.

It is important that the `time` field is the first column of the primary key to ensure scalability of ingestion as the application accumulates a large history of measurements. Kudu stores records on disk in primary key order, and because new measurements tend to be higher in timestamp value than almost all existing measurements in the table, new measurements tend to be written to the very end of the table's partitions, which minimizes the need for Kudu to run expensive compactions for new data. This also has the benefit that user queries that select short time slices will only scan table records within that time slice.

Partitioning

The Kudu measurement table should be partitioned with a combination of hash and range partitioning. This helps ensure that the scale of the cluster is utilized for large workloads, and without over-allocating that scale for small workloads. For more information on Kudu partitioning design, see the Kudu documentation.
The two values that need to be specified for hash-range partitioning is the number of hash buckets and the width of a range partition.

The number of hash buckets will determine the maximum level of parallelism for scans and ingest within a single range. It is important to model this parallelism for a single range because generally the majority of queries will only scan the most recent range.

A good measure for the number of hash buckets for very large tables, such as time series measurements, is to set the number of hash buckets as a small multiple of the number of worker nodes in the Kudu cluster. This will provide parallelism across all worker nodes, and a small level of parallelism within each worker node to make use of the multiple cores in each node.

For example, in a 20-worker-node cluster, where a measurement table will occasionally be used, having 40 hash buckets would be sensible. In another example, in a 50-worker-node cluster, where a measurement table is used very often, having 250 buckets would be sensible.

If the cluster is planned to grow over time, it is a good idea to consider the number of worker nodes for this calculation as the number at some point in the future, such as a year or two ahead. This is important because the number of hash partitions cannot be changed after the table has been created.

The second value that then needs to be determined is the width of each range partition. The width is not strictly required to be a round value such as one day or one month, but doing this will make it simpler to offload data to Parquet. The width of each range partition can also be different, but again it will be simpler to keep them the same unless there is a significant change in the volume of data arriving into the table.

A good measure for the width of a range partition for very large tables, such as time series measurements, is to set the width of the range partition such that the size of the range divided by the number of hash buckets is roughly 4GB. This will create tablets on disk of that size, and in Cloudera's experience that rough size is a good balance between the overhead of having many tablets and the performance of scanning larger tablets.

For example, where the hash bucket calculation above yielded 120 buckets, and where one day of measurements in the Kudu table consumes about 500GB on disk, then the width of the range partition should be:

```
range-width = #buckets / size-per-day / 4GB days

            = 120 / 500GB / 4GB days
            = 120 / 125 days
            = approx. 1 day
```

The above calculation indicates that the range width should be one day.

Note that if the cluster is small and the velocity of arriving data is high, the calculated size of a range partition may be very small. This indicates that the scaling limits of the Kudu cluster will be quickly reached, and so arriving data will need to be rapidly aged off to Parquet files. It also likely indicates that the query performance will not be suitable and that a larger cluster should be provisioned instead. For example, where the hash bucket calculation above yielded 10 buckets, and where one day of measurements in the Kudu table consumes about 2TB on disk, then the width of the range partition would be:

```
range-width = #buckets / size-per-day / 4GB days
            = 10 / 2TB / 4GB days
            = 10 / 500 days
            = 0.02 days
            = approx. 0.5 hours
```

Example

As an example, this Impala statement will create a Kudu measurements table:

```
CREATE TABLE measurements (
  time TIMESTAMP,
  metric_id INT,
  value DOUBLE,
 plant INT,
  machine INT,
  sensor_type STRING,
  PRIMARY KEY (time, metric_id)
)
PARTITION BY
HASH(metric_id) PARTITIONS 60
RANGE(time)
(
  PARTITION '2019-06-20' <= VALUES < '2019-06-21',
  PARTITION '2019-06-21' <= VALUES < '2019-06-22',
  PARTITION '2019-06-22' <= VALUES < '2019-06-23'
)
STORED AS KUDU;
```

Metrics table

This table stores information about the metrics as one record per metric. This table can be accessed by the Spark processing job to enrich the ingested measurements with additional fields.

The primary key should generally be the `metric_id` column.

The remaining columns of the metrics table should then contain information about the metric. Commonly these would form one or more hierarchies of fields about the metric. For example, the machine that a sensor is on, and the plant that the machine is in. These hierarchy levels provide opportunities for application users to query measurements at varying levels of aggregation.

If the metrics table will never require more than 4GB on disk the table should not need to be partitioned. If the metrics table will grow to more than 4GB on disk the table should be hash partitioned with the number of hash partitions set to the target size divided by 4GB. For example, if it is expected that the metrics table will use up to 20GB on disk, it should be hash partitioned with five hash partitions. The size of a Kudu table on disk can be found in Cloudera Manager under the Charts Library of the Kudu service.

For example,

```
CREATE TABLE metrics (
  metric_id INT PRIMARY KEY,
  plant INT,
  machine INT,
  sensor_type STRING
)
PARTITION BY HASH PARTITIONS 10
STORED AS KUDU;
```

Hardware considerations

As general Kudu best practice the write-ahead-log (WAL) directory of a Kudu tablet server should be on a dedicated disk.

For bare metal clusters it is recommended to use SSD disks for the tablet server WAL and spinning hard disks for the master WAL and the master and tablet server data directories. Tablet server data directories can be colocated with HDFS data directories.

For AWS EBS it is recommended to use the "io1" volume type for the tablet server WAL, and the "st1" volume type for the master WAL and the master and tablet server data directories.

The Kudu scaling guide should be used to allocate memory, threads, and other configurations.

Scalability limits

While Kudu supports high velocity ingest, high volume scans, and fast key lookups, it does have limits in storage capacity. As these limits are approached older data should be aged off to Parquet files on HDFS or object stores so that the limits are not breached. This is described further in the next section.

Review the CDH Kudu documentation to find the limits for your CDH version. As of CDH 6.2 the primary scalability limits per cluster are:

- Maximum 100 tablet servers
- Maximum 2000 tablets per tablet server
- Maximum 8TB per tablet server

If the design or volumes of your time series application would require going beyond the documented limits above, please contact your Cloudera account team, or the Kudu mailing lists, for advice on how to safely expand beyond these limits.

Kafka

If processing of measurements is required before data is visible to users, Kafka should be used as the buffer between data ingestion by NiFi and data processing by Spark. The time series measurements that are ingested by NiFi should be serialized in a common format and written to the Kafka measurements topic. The Spark data processing job then reads the standardized measurements from the topic. In this way, Kafka acts as an abstraction of the upstream data sources and serializations so that the business logic (e.g. enrichment) can be implemented independently of those concerns.

Kafka can also be used for providing messages downstream to other systems, if required. These requirements are specific to each application. For example, a system external to CDH may require notification when a sensor has been above a certain value for more than an hour. This notification could be a message on a Kafka topic that that external system subscribes to.

Measurement topic

The measurements topic holds all time series measurements ingested by NiFi, but not yet processed by Spark. One message in the topic should represent one measurement.

Topic design

The number of topic partitions is an important factor in message throughput. A reasonable initial value is one partition per 10,000 measurements per second. For example, for an application that ingests 500,000 measurements per second, start with a topic of 50 partitions. Further tuning of the partition count may be required after initial performance testing of the developed application, but ideally before entering production.

The topic should have a replication factor of 3 for data durability and availability in failure scenarios.

Message serialization

The measurement messages in the topic should be serialized with Apache Avro, which is a compact binary serialization that is well supported in the Cloudera platform. Avro allows the schema of the messages to evolve over time without necessarily breaking compatibility with earlier messages. The use of the Schema Registry component in Cloudera Stream Processing can assist with managing this schema evolution, including supporting the messages to be written without each embedding the full schema. The NiFi ingestion flow can be configured to write messages to Kafka in Avro format, including integration with the schema registry.

It is possible for the measurements messages to be serialized with other formats, such as JSON; however these may take up more disk space, may be slower for downstream processing to parse, and may not support schema evolution.

Parquet

As data ages in the Kudu measurement table it can be rolled off to Parquet files so that Kudu's advantages are dedicated to storing the more recent data. The offloaded Parquet files can be stored on HDFS, or object storage such as S3 or ADLS.

Note that this offload is not always mandatory for a successful time series application. If the measurement data will remain comfortably under the scalability limits of the Kudu cluster then there should typically be less of a hard requirement to introduce the complexity of rolling off data to Parquet files. However, the cost differences of object storage for Parquet (e.g. S3) vs file system storage for Kudu (e.g. ext4 on EBS) may be a factor even when the scalability limits are not a concern.

The two locations for the overall data set can be made transparent to users by reading from both tables in an Impala view. Where a user query spans a range of time that exists in only one storage layer then only that storage layer will be scanned.

For more information on rolling data off to Parquet files, and to maintain the table partitions and view, see the Cloudera blog article on the topic.

Data ingestion

Ingestion is the extract of data from source systems and the load of that data into the CDH 6 platform. From data warehousing terminology this is the "E" and the "L" of ETL.

NiFi is the recommended data integration component, which is provided by Cloudera Data Flow (CDF) for CDH 6. NiFi is very well suited to high velocity ingest of time series data.

The source systems that time series data is ingested from can take many different forms and interfaces. Some of these include:

- Raw TCP
- HTTP, e.g. REST
- Historians
- Upstream collectors, e.g. Azure IoT Hub
- Relational databases
- Flat files
- Custom format files

NiFi has the ability to read all of these source systems and interfaces, including extensibility for data formats that NiFi does not currently support. Consult the NiFi documentation to find the processors that match your source system interfaces.

If processing of the measurements is required before data is visible to users, NiFi should load the extracted data into the measurement topic in Kafka. Each message in the measurement topic should represent one time series measurement. All ingested measurements should be serialized as Avro records for the Kafka messages.

If this pre-processing is not required, NiFi should write the measurements directly into the Kudu measurements table.

Data processing

In this architecture, the primary processing job reads the ingested measurements from Kafka, transforms the measurements according to the required business rules of the application, and writes the processed results to Kudu.

If no transformations are required prior to user queries the NiFi flow should write the measurements directly to the Kudu measurements table.

The transformations required for data processing will vary for each application. A decision needs to be made for each proposed transformation as to whether to calculate it once up front in data processing jobs, or to have it recalculated in user queries each time their results are requested. There are trade-offs to either approach, but generally transformations should be calculated up front in data processing jobs if they are commonly requested in user queries, and they are slow to execute, and their results do not take up a large amount of additional disk space.

Spark is the recommended component for time series data processing on the CDH 6 platform. Spark can run as a continuous loop of "micro-batches" that are typically a few seconds to one minute in duration. In this architecture each micro-batch will retrieve the latest data from Kafka, transform that data as required, and then write the results to Kudu tables. The data that is written to Kudu is then immediately visible to user queries.

For Spark Streaming time series measurement pipelines it is recommended to set these Spark configurations:

CONFIGURATION	VALUE	NOTES
`spark.dynamicAllocation.enabled`	`false`	Dynamic allocation of executors is not supported in Spark Streaming mode
`spark.streaming.backpressure.enabled`	`true`	Backpressure sizes micro-batches to attempt to finish within the defined duration. Without this the micro-batches may exceed their duration when there are volume spikes
`spark.streaming.kafka.maxRatePerPartition`	max # records per Kafka partition per second	This avoids initial micro-batches becoming too large after returning from downtime
`spark.locality.wait`	`1ms`	Improves streaming task scheduling so that timing issues do not cause all tasks of a micro-batch to be run on a single node
`spark.speculation`	`true`	Improves resiliency when a node is slow or failing. Is suitable for this application because the writes are idempotent, and will not fail on double writes of the same data

To create a processing pipeline directly against Spark requires developing an application that calls the various Spark APIs. For stream processing it is recommended to use either the Scala or Java Spark APIs. The approach of developing against the Spark API provides the most flexibility for application design but requires a solid familiarity with the Spark internals to quickly develop features.

An alternative option is to use the Envelope framework to create the processing pipeline mostly or entirely using configuration. Envelope is an open source data processing abstraction of Spark. Envelope is provided by Cloudera but is not currently included in the supported CDH 6 platform. The reference application that accompanies this reference architecture has its processing pipeline built on Envelope.

The following subsections describe some of the transformations that may be required from the processing pipeline. The transformations that each time series application requires will vary considerably. For examples of how this business logic may be implemented, see the reference application.

Enrichment

A good example of a transformation that often makes sense to calculate up front in data processing jobs is enrichment of the measurements with commonly required metadata fields. This will avoid adding a join to the user query to access those fields and allows more efficient filtering by pushing filters down into Kudu.

For example, in a financial markets time series use case the pricing measurements may contain the stock symbol. Enrichment could then use that symbol to look up and add related fields such as the company name or the exchange that the equity trades on.

The Spark processing job should use the Spark-Kafka integration to read the measurements from Kafka, join them to the Kudu metrics table to append the additional fields for enrichment, and write the enriched measurements to the Kudu measurements table.

Metric registration

It may be required for the Spark processing job to detect when measurements contain metric identifiers that are not yet present in the metrics table, and add new placeholder entries to the metrics table until they can be later fully populated.

Aggregation

It may be required to pre-aggregate results for improved performance of user queries. This is most likely to be required if user queries tend to span a long range of time, which would require a large amount of input data to calculate results that may be too slow to recalculate with each user query.
These aggregation results should be written to a separate Kudu table. The same primary key and partitioning logic applies to an aggregation table as the original measurements table. While this aggregation can in theory be built into the Spark processing job that reads from Kafka and writes to Kudu, recalculating aggregations every few seconds can either heavily reduce processing performance or require significantly more cluster resources. Instead, it is typically more efficient to develop a separate batch Spark job that recalculates aggregates for the period of time since the previous batch job. This batch job would be scheduled to run periodically, for example every 10 minutes.

User queries

With the processed measurements loaded into Kudu the users can immediately query them with Impala. This can be done using hand-written SQL in the Hue user interface for CDH 6, or through almost any third-party business intelligence tool that supports JDBC or ODBC data sources. Custom SQL can be useful for exploratory analytics, and BI tools can be useful for live monitoring and dashboarding.

In addition to time series data warehousing, data scientists can access the same data through the Cloudera Data Science Workbench tool. This enables distributed machine learning algorithms to be executed over the time series data using Spark.

This section provides some example SQL queries that users might run to ask questions of the time series application. These could be submitted manually through Hue, or an equivalent submitted by a BI tool.

Raw

A typical query in a time series data warehouse would be to retrieve all values for a single metric within a relative window of time:

```
SELECT time, value
FROM measurements
   WHERE metric_id = 1000 AND time >= NOW() - INTERVAL 1 HOUR
   ORDER BY time;
   +-------------------------------+--------------------+
   | time                          | value              |
   +-------------------------------+--------------------+
   | 2019-06-03 00:48:24.964000000 | -665.2692997894557 |
   | 2019-06-03 00:48:26.987000000 | -687.818101900486  |
   | 2019-06-03 00:48:30.020000000 | -698.1934639523939 |
   | 2019-06-03 00:48:32.043000000 | -685.8743759190486 |
   | 2019-06-03 00:48:34.067000000 | -662.0600881819529 |
```

Or within a specific range of time:

```
SELECT time, value
FROM measurements
WHERE metric_id = 1000 AND time BETWEEN '2019-06-03 01:00:00'
AND '2019-06-03 01:01:00'
ORDER BY time;
+-------------------------------+--------------------+
| time                          | value              |
+-------------------------------+--------------------+
| 2019-06-03 01:00:00.887000000 | 959.4854494673297  |
| 2019-06-03 01:00:02.913000000 | 941.3869754231239  |
| 2019-06-03 01:00:04.938000000 | 911.9405173709874  |
...
| 2019-06-03 01:00:54.502000000 | -685.8743756099201 |
| 2019-06-03 01:00:56.526000000 | -662.0600839755724 |
| 2019-06-03 01:00:58.550000000 | -627.084895326232  |
+-------------------------------+--------------------+
```

Downsamples

A common challenge when working with raw measurements is that timestamps are not exactly aligned to time boundaries, even when they are being generated at a regular interval. The previous example shows this – they are all generated roughly every two seconds, but the millisecond component of the timestamps is more or less uncorrelated from one measurement to the next.

This makes it difficult to retrieve values for a particular point in time:

```
SELECT time, sensor_type, value
FROM measurements
WHERE metric_id IN (1002, 1005) AND time = '2019-06-03
01:00:56';
[no results]
```

While it may not be possible to deduce the exact value of a metric at a timestamp that has not reported a measurement, an estimated value for the timestamp can be calculated by downsampling the raw metrics so that the metrics are aggregated per interval.

For example, to downsample to the next second down, using an average of the values within a second interval:

```
SELECT
    DATE_TRUNC('second', time) AS time
  , sensor_type
  , AVG(value) AS value
FROM measurements
WHERE metric_id IN (1002, 1005)
AND time >= '2019-06-03 01:00:56' AND time < '2019-06-03
01:00:57'
GROUP BY 1, 2 ORDER BY 1, 2;
+---------------------+-------------+--------------------+
| time                | sensor_type | value              |
+---------------------+-------------+--------------------+
| 2019-06-03 01:00:56 | depth       | -185.2137726433043 |
| 2019-06-03 01:00:56 | speed       | 466.9444886563384  |
+---------------------+-------------+--------------------+
```

Similarly, to downsample to the next five-second interval down:

```
SELECT
    DATE_TRUNC('second', time) - INTERVAL (EXTRACT(time,
'second') % 5) SECONDS AS time
  , sensor_type
  , AVG(value) AS value
FROM measurements
WHERE metric_id IN (1002, 1005)
AND time >= '2019-06-03 01:00:50' AND time < '2019-06-03
01:01:10'
GROUP BY 1, 2 ORDER BY 1, 2;
+---------------------+-------------+--------------------+
| time                | sensor_type | value              |
+---------------------+-------------+--------------------+
| 2019-06-03 01:00:50 | depth       | -212.2521216402649 |
| 2019-06-03 01:00:50 | speed       | 461.1362196282832  |
| 2019-06-03 01:00:55 | depth       | -181.3314399444706 |
| 2019-06-03 01:00:55 | speed       | 468.0732078473125  |
| 2019-06-03 01:01:00 | depth       | -170.9435884955582 |
| 2019-06-03 01:01:00 | speed       | 472.8436200197602  |
| 2019-06-03 01:01:05 | depth       | -183.1460075965676 |
| 2019-06-03 01:01:05 | speed       | 475.4202629257637  |
+---------------------+-------------+--------------------+
```

These queries can be wrapped in views to simplify user queries. Multiple views can be created for different granularities, such as one per second, one per minute, and one per hour.

For example, with a view created for downsampling as an average per second:

```
CREATE VIEW downsampled_1s AS
SELECT
    DATE_TRUNC('second', time) AS time
  , metric_id
  , plant
  , machine
  , sensor_type
  , AVG(value) AS value
FROM measurements
GROUP BY 1, 2, 3, 4, 5;
```

The view can then be queried as:

```
SELECT time, sensor_type, value
FROM downsampled_1s
WHERE metric_id IN (1002, 1005)
AND time = '2019-06-03 01:00:56'
ORDER BY time, sensor_type;
+---------------------+-------------+--------------------+
| time                | sensor_type | value              |
+---------------------+-------------+--------------------+
| 2019-06-03 01:00:56 | depth       | -185.2137726433043 |
| 2019-06-03 01:00:56 | speed        | 466.9444886563384  |
+---------------------+--------------+--------------------+
```

Pivot/transpose

The above examples provide the multiple metrics per timestamp on separate records. A pivot, otherwise known as a transpose, can be used where multiple metrics per timestamp are required on the same record.

Impala does not currently provide a pivot function, but it can be implemented with this syntax (assuming we have a view `downsampled_1s` that downsamples measurements per second, as described in the previous section):

```
SELECT
    time
  , MAX(CASE WHEN sensor_type = 'pressure' THEN value END)
pressure
  , MAX(CASE WHEN sensor_type = 'brightness' THEN value END)
brightness
FROM downsampled_1s
WHERE machine = 705 AND time >= NOW() - INTERVAL 1 HOUR
GROUP BY time
ORDER BY time;
+---------------------+-------------------+--------------------+
| time                | pressure          | brightness         |
+---------------------+-------------------+--------------------+
| 2019-06-03 00:48:24 | 46.31959579101573 | -399.1159619067209 |
| 2019-06-03 00:48:25 | 47.17996759211917 | -399.9728049675076 |
| 2019-06-03 00:48:26 | 47.73523223287668 | -399.6272832668245 |
| 2019-06-03 00:48:28 | 47.92830080609275 | -395.3348250960566 |
| 2019-06-03 00:48:29 | 47.56588908407019 | -391.3973229393663 |
...
```

Where a set of metrics is commonly pivoted together then a view could be created for that pivot query:

```
CREATE VIEW probe_metrics AS
SELECT
    time
  , plant
  , machine
  , MAX(CASE WHEN sensor_type = 'pressure' THEN value END)
pressure
  , MAX(CASE WHEN sensor_type = 'brightness' THEN value END)
brightness
FROM downsampled_1s
GROUP BY time, plant, machine;
```

Users could then query for probe metrics per timestamp with:

```
SELECT time, pressure, brightness
FROM probe_metrics
WHERE machine = 705
AND time >= NOW() - INTERVAL 1 HOUR
ORDER BY time;
+---------------------+-------------------+--------------------+
| time                | pressure          | brightness         |
+---------------------+-------------------+--------------------+
| 2019-06-03 00:48:24 | 46.31959579101573 | -399.1159619067209 |
| 2019-06-03 00:48:25 | 47.17996759211917 | -399.9728049675076 |
| 2019-06-03 00:48:26 | 47.73523223287668 | -399.6272832668245 |
| 2019-06-03 00:48:28 | 47.92830080609275 | -395.3348250960566 |
| 2019-06-03 00:48:29 | 47.56588908407019 | -391.3973229393663 |
| 2019-06-03 00:48:30 | 46.89793899373325 | -386.2763078513929 |
| 2019-06-03 00:48:31 | 45.92519684787266 | -379.9830358708207 |
| 2019-06-03 00:48:32 | 44.64874896045262 | -372.5313384227952 |
| 2019-06-03 00:48:33 | 43.07002057534766 | -363.9376006885211 |
| 2019-06-03 00:48:34 | 41.19077563451083 | -354.2207113810819 |
```

Latest values

To retrieve the latest value for each metric it is important to restrict how far back to look for each metric, otherwise the query may be slow as it looks for metrics that have not reported measurements in a long time.

For example, to create a view that provides the latest values for metrics that have provided measurements for the previous five minutes:

```
CREATE VIEW latest AS
SELECT metric_id, machine, plant, time, value
FROM (
  SELECT metric_id, machine, plant, time, value,
    ROW_NUMBER() OVER (PARTITION BY metric_id ORDER BY time
DESC) = 1 AS is_latest
  FROM measurements
  WHERE time >= NOW() - INTERVAL 5 MINUTES
) x
WHERE is_latest = true;
```

Users can then select the latest values for specific tags like:

```
SELECT metric_id, time, value
FROM latest
WHERE machine = 11340;
```

Aggregations

Impala supports many common aggregate functions. These can be used on the raw measurements table, or an aggregated measurements table if that is being maintained by the time series application. For example, to retrieve the average, minimum, maximum, and approximate median values of a metric over the previous day:

```
SELECT
    metric_id
  , AVG(value) avg_value
  , MIN(value) min_value
  , MAX(value) max_value
  , APPX_MEDIAN(value) appx_median_value
FROM measurements
WHERE time >= NOW() - INTERVAL 1 DAY
GROUP BY metric_id;
```

Interpolation

Interpolation is a transformation that provides values for timestamps that were not directly measured. For example, if two consecutive measurements were spaced five seconds apart, interpolation provides values for the four individual seconds between them. In some cases, these values are exact, such as carrying forward financial market prices, and in other cases these values are estimated, such as what would have been the readings from a physical sensor.

Impala does not currently provide interpolation functionality, however Cloudera Data Science Workbench (CDSW) for the CDH 6 platform can be used as an interactive Spark interface. Spark provides a flexible API for scalable data transformations. Interpolation is included in various open source time series libraries for Spark, such as Flint.