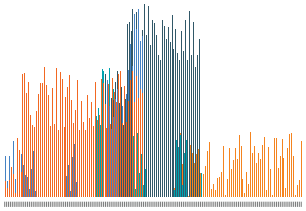
CLOUDERA

WHITE PAPER

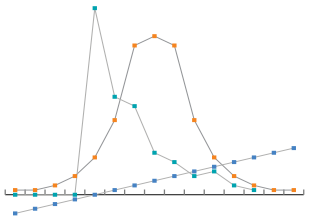
TIME SERIES DATA WAREHOUSE

A reference architecture utilizing a modern data warehouse, based on Cloudera HDP 3

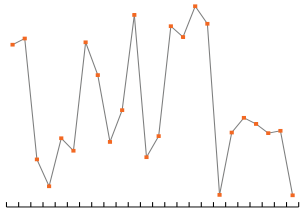
Time series is different from traditional statistical analysis



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

Large pharmaceutical and chemical manufacturers optimize yield by using time series data analysis at large scale.

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

Table of Contents

Requirements

The architecture outlined in this document describes a reference solution for time series use cases on the Cloudera HDP 3 platform. 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

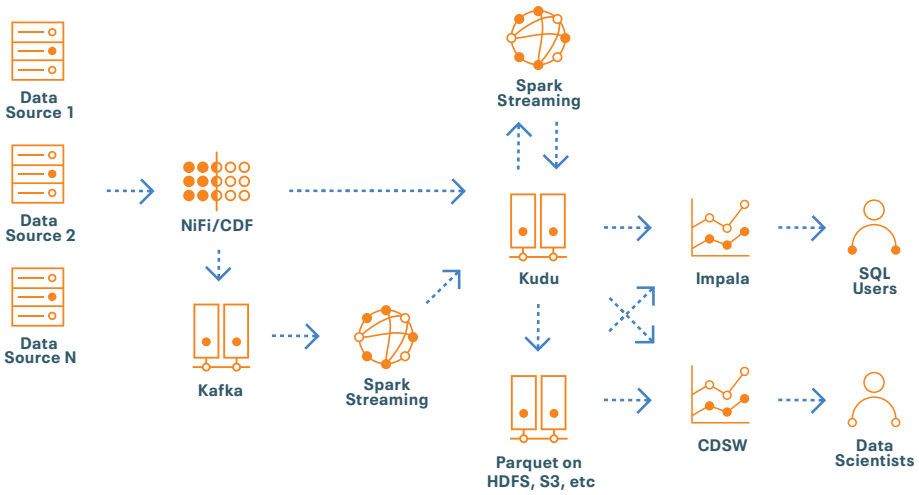
High-level architecture

The HDP 3 platform provides a variety of open source components that can be configured together to build applications. This reference architecture uses multiple HDP 3 components to solve the time series requirements that were described in the previous section:

- Apache NiFi, for ingestion of measurements
- Apache Kafka, for buffered storage of ingested measurements
- Apache Spark, for measurement pre-processing, if required
- Apache Druid, for permanent storage of measurements
- Apache Hive, for user query of measurements

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. Cloudera advises to start a HDP 3 time series proof of concept with this reference application and make required modifications from that point, instead of starting a new application from scratch.
The default AWS cluster configuration of the reference application is capable of ingesting 1.5 million measurements per second, with spare compute capacity for tens of queries per second. Note that query throughput is highly dependent on the specific queries that are submitted.

Data model

The data model is the central construct of a data warehouse architecture, and time series applications are no exception. The design of the time series data model will have a major impact on the performance, simplicity, and cost-effectiveness of the overall solution.

This section outlines a reference data model for a time series data warehouse. This data model will need to be customized for each implementation based on application requirements.

In this architecture the data is stored in two storage layers:

Druid, which stores new measurements

- Druid, which stores measurements to serve user queries
- Kafka, which buffers measurements that have been ingested but yet to be processed

Druid

Druid is a database in the HDP 3 platform that is designed from the ground up for high velocity ingestion, aggregation, and querying of time series events. Druid can be configured to ingest events directly from Kafka. Druid can also be configured to automatically roll up events by a specific time granularity so that many queries that span long periods of time can be answered very quickly.

The equivalent of a table in Druid is a "data source", which contains a single timestamp field (`__time`) along with fields for dimensions and measures. Druid data sources are optimized for aggregation queries that return a small number of records, and so are best modeled as highly denormalized to remove the need for external joins at query time.

Druid provides a comprehensive REST interface for administrators and end users. Druid does provide a native SQL interface for end user queries, however this is limited to simple SQL queries over a single Druid data source. Instead, for end user queries, the HDP 3 platform provides integration between Hive and Druid so that SQL queries can be sent to Druid from the same Hive interface as queries to other storage layers, such as HDFS. Cloudera recommends that end users query Druid via the Hive integration.

Modeling

Data source creation by Hive

The Hive-Druid integration provides the functionality to create a table in Hive that in turn creates a data source in Druid. This includes the option to configure data ingestion from Kafka.

For example,

```
CREATE EXTERNAL TABLE telemetry (
    `__time` TIMESTAMP
  , sensor_id STRING
  , value DOUBLE)
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
TBLPROPERTIES (
  "kafka.bootstrap.servers" = "localhost:9092",
  "kafka.topic" = "telemetry",
  "druid.query.granularity" = "MINUTE",
  "druid.segment.granularity" = "DAY"
);
```

Using the Hive-Druid integration to create the Druid data source has some important limitations:

- Numeric fields are assumed to be measures
- Non-numeric fields are assumed to be dimensions
- Measures can only be rolled up on the sum aggregation

If a numeric dimension field is required, then it should be converted to a string before loading into Druid. The field can be converted to a string within the NiFi flow that ingests the events.

Measures are generally numeric, so if ingested events contain numeric measures as strings, they should be converted to a numeric data type before loading into Druid.

The limitation of only rolling up measures on the sum aggregation will prevent users from querying the count of events and any other aggregations that depend on a c ount, such as averages. This impact on user queries is discussed further in the Queries section. To include the count aggregation the data source must be created first in Druid, as described in the next section.

Data source creation by Druid

An alternative approach to creating Hive-Druid tables/data sources is to first use the Druid REST interface to directly create the Druid data source, and to then create a Hive table that points to that data source. This provides full flexibility over the data source specification, at the expense of some complexity for the developer to need to use the REST interface.

Cloudera recommends to create Druid data sources through Hive if the above stated limitations are acceptable, or to otherwise create them directly through Druid.

For example,

```
DRUID_OVERLORD_HOSTNAME=...
HIVESERVER2_HOSTNAME=...

curl -X POST -H 'Content-Type: application/json' -d @supervisor.
json http://${DRUID_OVERLORD_HOSTNAME}:8090/druid/indexer/v1/
supervisor

beeline -u "jdbc:hive2://${HIVESERVER2_HOSTNAME}:10501/
default;transportMode=http;httpPath=cliservice" -e "CREATE
EXTERNAL TABLE telemetry STORED BY 'org.apache.hadoop.hive.
druid.DruidStorageHandler' TBLPROPERTIES (\"druid.datasource\" =
\"default.telemetry\")"
```

Cloudera recommends creating Druid data sources through Hive if the above stated limitations are acceptable, or to otherwise create them directly through Druid.

For example,

```
DRUID_OVERLORD_HOSTNAME=...
HIVESERVER2_HOSTNAME=...

curl -X POST -H 'Content-Type: application/json' -d @supervisor.
json http://${DRUID_OVERLORD_HOSTNAME}:8090/druid/indexer/v1/
supervisor
```

```
beeline -u "jdbc:hive2://${HIVESERVER2_HOSTNAME}:10501/
default;transportMode=http;httpPath=cliservice" -e "CREATE
EXTERNAL TABLE telemetry STORED BY 'org.apache.hadoop.hive.
druid.DruidStorageHandler' TBLPROPERTIES (\"druid.datasource\" =
\"default.telemetry\")"
```

A base supervisor JSON file can be generated for your schema by first temporarily creating the data source via Hive, then navigating to the Druid Overlord UI and clicking the "payload" link for the data source. When the base supervisor JSON file has been copied the temporary Hive table can then be dropped. Refer to the Druid documentation for changes that can be made to the base supervisor JSON file.

A common configuration to add to a supervisor JSON file is the count, min, and max aggregations, so that end users can query for the count of events and for the average, minimum, and maximum of event measures. When these aggregations are included in the data source the corresponding Hive table will include the corresponding columns as named by the `name` property of each aggregation. These aggregations can be added with JSON similar to:

```
...
    "metricsSpec": [
      {
        "type": "doubleSum",
        "name": "value",
        "fieldName": "value",
        "expression": null
      },
      {
        "type": "count",
        "name": "count"
      },
      {
        "type": "doubleMin",
        "name": "value_min",
        "fieldName": "value",
      },
      {
        "type": "doubleMax",
        "name": "value_max",
        "fieldName": "value",
      }
    ],
...
```

Kafka ingestion

The Druid data source can be configured to automatically load events from Kafka. Cloudera recommends this whenever possible so that external Druid ingestion jobs do not need to be developed and maintained.

The Druid-Kafka ingestion can be configured as part of the data source creation, and then enabled with a subsequent Hive alter table statement.

For example,

```
CREATE EXTERNAL TABLE telemetry (`__time` timestamp, sensor_id
string, value double)
```

```
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
TBLPROPERTIES (
  "kafka.bootstrap.servers" = "localhost:9092",
  "kafka.topic" = "telemetry",
  "druid.kafka.ingestion.taskCount" = "10",
  ...
);

ALTER TABLE telemetry SET TBLPROPERTIES("druid.kafka.ingestion"
= 'START');
```

A Druid-Kafka ingestion job is run as one-to-many tasks on the Druid middle manager processes, and is managed by the Druid overlord process.

By default ingestion will only use one task on one middle manager, which can be a considerable bottleneck for high velocity ingestion. The Hive-Druid configuration druid.kafka.ingestion.taskCount (as in the above example) can be used to increase the number of ingestion tasks for the Druid data source. Note that running more tasks than there are Kafka topic partitions will not improve ingestion performance, because a Kafka topic partition can only be consumed by one task at once.

The capacity of tasks that a Druid middle manager can run is set by the middle manager configuration druid.worker.capacity, which by default is set to 1. Generally, if the middle manager is on a dedicated host, set the worker capacity closer to the number of cores on the host. For example, to enable capacity for 60 tasks over 5 middle managers, set druid.worker.capacity to 12.

Rollups

One of the main benefits of using Druid is that it can automatically roll up ingested events to a larger time granularity than the raw events, so that aggregations in user queries that span over long time ranges can be served with greatly improved performance relative to scanning the raw ingested events. For example, if events are ingested at the rate of one per second, rolling up events to the per-minute granularity can reduce the size of the data to be aggregated in a user query by as much as 60x.

The primary tradeoff to querying a rolled up data source is that the raw events are not retained and so questions that require the original time granularity cannot be answered. For many time series applications that tradeoff is acceptable and so Druid provides these applications with very high performance without the application developers requiring to manually implement rollup logic.

In cases where rolled up events are required for the acceptable performance of some queries, but raw events are also required for other queries, Cloudera recommends a dual ingestion approach where events are ingested into Druid for rollups and into another storage layer for raw events, such as HDFS or an object store such as S3 or ADLS. While it is technically possible to store raw events in Druid, it is generally not recommended to store high scales of raw events in Druid as it is not optimized for returning very large result sets from user queries, which would be necessary for many queries that require raw events (e.g. machine learning algorithms). Dual ingestion can be enabled through an additional sink in the NiFi flow.

Automated rollups can be configured in the Druid data source definition. The granularity at which Druid segments the data source can also be set in a similar way.

For example,

```
CREATE EXTERNAL TABLE telemetry (`__time` timestamp, sensor_id
string, value double)
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
TBLPROPERTIES (
  ...
  "druid.query.granularity" = "MINUTE",
  "druid.segment.granularity" = "DAY"
);
```

In this example the measures of the events of each minute of the `__time` field will be automatically aggregated for each combination of dimension values.

Ingestion serialization format

By default, Hive-Druid assumes that events being ingested are serialized as JSON objects. If the events in Kafka are serialized with delimited text (e.g. CSV) or Avro then this can be set with the Hive-Druid `druid.parseSpec.format` configuration. The serialization configurations are indirectly documented in the Hive-Druid code.

If ingested events are serialized in another format then they should be converted by NiFi to a supported serialization, ideally Avro for best performance.

For example, for Avro

```
CREATE EXTERNAL TABLE telemetry (
  `__time` TIMESTAMP,
  sensor_id STRING,
  value DOUBLE,
  plant STRING,
  machine STRING,
  sensor_type STRING
)
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
TBLPROPERTIES (
  ...
  "druid.parseSpec.format" = "avro",
  "avro.schema.literal" =
"{\"type\":\"record\",\"name\":\"
telemetry\",\"fields\":[{\"name\":\"sensor_id\",\"type\":\"string\"
},{\"name\":\"__time\",\"type\":\"long\"},{\"name\":\"value\",\"-
type\":\"double\"},{\"name\":\"sensor_type\",\"type\":[\"null\",\"stri
ng\"]},{\"name\":\"plant\",\"type\":[\"null\",\"string\"]},{\"name\":\
"machine\",\"type\":[\"null\",\"string\"]}]}"
);
```

Kafka

Kafka should be used as the buffer between data ingestion by NiFi and data loading by Druid. The time series measurements that are ingested by NiFi should be serialized in a common format and written to a Kafka measurements topic.

Measurements topic

The measurements topic holds time series measurements ingested by NiFi, but not yet loaded by Druid. 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 generally take up more disk space, are slower for Druid to parse, and may not support schema evolution.

Data ingestion

Ingestion is the extract of data from source systems and the load of that data into the HDP 3 platform. From data warehousing terminology this is the "E" and the "L" of ETL, or ELT. NiFi is the recommended data integration component, which is provided by Cloudera Data Flow (CDF) for HDP 3. 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

- 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.

NiFi should load the extracted data into the measurements 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 required, NiFi can also be used to pre-process measurements before loading them into Kafka. This can include simple transformations such as data type conversion and field renames, or more complex transformations such as enrichment via lookups.

In some cases, the complexity of a required transformation is beyond what is expressable in NiFi, such as a join or window function. In these cases, Cloudera recommends to use the stream processing engine Apache Spark to run these transformations between the data being ingested by NiFi and the data being loaded into Druid. Kafka can be used as a buffer between these components, just as it is used in this architecture document as a buffer between NiFi and Druid.

User queries

With the measurements loaded into Druid the users can immediately query them with Hive. This can be done using hand-written SQL in the Data Analytics Studio (DAS) user interface for HDP 3, or through third-party business intelligence tools that support 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 same time series data using Spark.
This section provides some example SQL queries that users might run to ask questions of the time series application.

Raw

In this context "raw" data refers to the individual records stored within Druid. If the measurements have not been rolled up on ingestion, then these will be the individual measurements.
If the measurements have been rolled up on ingestion, for example from per-second events to the one-minute granularity, then these will be the partial aggregation results (i.e. rollups) stored in Druid. These records are a slice of the time and dimensions fields of the measurements. It is not guaranteed that Druid will always store one rollup record per slice of the time and the dimensions, nor that Druid will always maintain the same number of rollup records per slice of time and dimensions. In general, this means that is it not particularly useful to select raw records from rollup tables, and instead queries that aggregate by a slice of time and/or dimensions should be submitted instead.
For example, within a relative window of time, note that sometimes there are multiple records per timestamp:

```
SELECT `__time`, count, value
FROM telemetry
WHERE sensor_id = 100
AND `__time` >= CURRENT_TIMESTAMP() - INTERVAL 10 MINUTES;
```

```
+---------------------------+--------+----------------------+
|           __time          | count  |        value         |
+---------------------------+--------+----------------------+
| 2019-10-22 14:46:00.0 UTC | 60     | -16219.843261338903  |
| 2019-10-22 14:47:00.0 UTC | 1      | -6.846895856402227   |
| 2019-10-22 14:47:00.0 UTC | 59     | -14912.613028755055  |
| 2019-10-22 14:48:00.0 UTC | 60     | -17470.1292797956    |
| 2019-10-22 14:49:00.0 UTC | 1      | -470.02321050059106  |
| 2019-10-22 14:49:00.0 UTC | 59     | -19866.89021500123   |
| 2019-10-22 14:50:00.0 UTC | 60     | -19556.180216432906  |
| 2019-10-22 14:51:00.0 UTC | 1      | -299.80085607997455  |
| 2019-10-22 14:51:00.0 UTC | 59     | -15906.440247964943  |
| 2019-10-22 14:52:00.0 UTC | 60     | -14923.916427317667  |
| 2019-10-22 14:53:00.0 UTC | 1      | -112.41589126252131  |
| 2019-10-22 14:53:00.0 UTC | 59     | -17371.277289287562  |
| 2019-10-22 14:54:00.0 UTC | 60     | -20344.369541141623  |
| 2019-10-22 14:55:00.0 UTC | 1      | -586.1111970069903   |
| 2019-10-22 14:55:00.0 UTC | 32     | -6269.855211856371   |
+---------------------------+--------+----------------------+
```

Or within a specific range of time:

```
SELECT `__time`, count, value
FROM telemetry
WHERE sensor_id = 100
AND `__time` BETWEEN '2019-10-22 14:50:00' AND '2019-10-22
14:52:00';

+----------------------------+--------+----------------------+
|           __time           | count  |        value         |
+----------------------------+--------+----------------------+
| 2019-10-22 14:50:00.0 UTC  | 60     | -19556.180216432906  |
| 2019-10-22 14:51:00.0 UTC  | 1      | -299.80085607997455  |
| 2019-10-22 14:51:00.0 UTC  | 59     | -15906.440247964943  |
| 2019-10-22 14:52:00.0 UTC  | 60     | -14923.916427317667  |
+----------------------------+--------+----------------------+
```

Downsamples

Downsampling involves aggregating measurements to a larger time granularity so that values can be determined for wider timespans than what is already rolled up in the table. Within Druid these are known as 'timeseries' queries.

Hive provides "floor" functions to round down timestamps to the nearest granularity timestamp, e.g. for the floor minute function, from '2019-01-01 12:34:56' to '2019-01-01 12:34:00'.

For example, to downsample to a per hour granularity:

```
    SELECT
    FLOOR_HOUR(`__time`) `__time`
  , SUM(count) count
  , SUM(value) value
FROM telemetry
WHERE sensor_id = 100
GROUP BY FLOOR_HOUR(`__time`);
+----------------------------+--------+----------------------+
|           __time           | count  |        value         |
+----------------------------+--------+----------------------+
| 2019-10-22 14:00:00.0 UTC  | 894    | -262296.3874278831   |
| 2019-10-22 15:00:00.0 UTC  | 3600   | -1062112.2920872918  |
| 2019-10-22 16:00:00.0 UTC  | 3600   | -1062137.711899869   |
| 2019-10-22 17:00:00.0 UTC  | 3599   | -1061534.043815015   |
| 2019-10-22 18:00:00.0 UTC  | 1954   | -577525.0118555583   |
+----------------------------+--------+----------------------+
```

Or to downsample to a per 15-minute granularity:

```
SELECT
    CAST(FLOOR_MINUTE(`__time`) AS TIMESTAMP) -
    INTERVAL (EXTRACT(minute FROM `__time`) % 15) MINUTES
    AS `__time`
  , SUM(count) count
  , SUM(value) value
FROM telemetry
WHERE sensor_id = 100
GROUP BY CAST(FLOOR_MINUTE(`__time`) AS TIMESTAMP) -
  INTERVAL (EXTRACT(minute FROM `__time`) % 15) MINUTES
ORDER BY `__time`;
```

```
+-------------------------+--------+----------------------+
|          __time         | count  |        value         |
+-------------------------+--------+----------------------+
| 2019-10-22 14:45:00.0   | 894    | -262296.3874278831   |
| 2019-10-22 15:00:00.0   | 900    | -265527.93449713575  |
| 2019-10-22 15:15:00.0   | 900    | -265535.0565255714   |
| 2019-10-22 15:30:00.0   | 900    | -265525.1983720285   |
| 2019-10-22 15:45:00.0   | 900    | -265524.1026925563   |
| 2019-10-22 16:00:00.0   | 900    | -265531.57272412465  |
| 2019-10-22 16:15:00.0   | 900    | -265531.25440185587  |
| 2019-10-22 16:30:00.0   | 900    | -265538.47855463775  |
| 2019-10-22 16:45:00.0   | 900    | -265536.4062192506   |
| 2019-10-22 17:00:00.0   | 899    | -264938.3456768413   |
| 2019-10-22 17:15:00.0   | 900    | -265532.40692064306  |
| 2019-10-22 17:30:00.0   | 900    | -265534.8937905999   |
| 2019-10-22 17:45:00.0   | 900    | -265528.3974269307   |
| 2019-10-22 18:00:00.0   | 900    | -265534.2945895997   |
| 2019-10-22 18:15:00.0   | 783    | -227960.73054807374  |
+-------------------------+--------+----------------------+
```

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.

```
CREATE VIEW telemetry_hour AS
SELECT
    FLOOR_HOUR(`__time`) `__time`
  , sensor_id
  , SUM(count) count
  , SUM(value) value
FROM telemetry
GROUP BY FLOOR_HOUR(`__time`), sensor_id;
```

The view can then be queried as:

```
SELECT `__time`, count, value
FROM telemetry_hour
WHERE sensor_id = 100
ORDER BY `__time`;
+---------------------------+--------+----------------------+
|           __time          | count  |        value         |
+---------------------------+--------+----------------------+
| 2019-10-22 14:00:00.0 UTC | 894    | -262296.3874278831   |
| 2019-10-22 15:00:00.0 UTC | 3600   | -1062112.2920872918  |
| 2019-10-22 16:00:00.0 UTC | 3600   | -1062137.711899869   |
| 2019-10-22 17:00:00.0 UTC | 3599   | -1061534.043815015   |
| 2019-10-22 18:00:00.0 UTC | 2118   | -625490.7404606077   |
+---------------------------+--------+----------------------+
```

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.

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

```
    SELECT
    FLOOR_HOUR(`__time`) `__time`
  , MAX(CASE WHEN sensor_type = 'temperature' THEN value END)
temperature
  , MAX(CASE WHEN sensor_type = 'pressure' THEN value END)
pressure
FROM telemetry
WHERE sensor_id IN (100, 101)
AND `__time` >= CURRENT_TIMESTAMP() - INTERVAL 4 HOURS
GROUP BY FLOOR_HOUR(`__time`)
ORDER BY FLOOR_HOUR(`__time`);
+----------------------+----------------------+--------------------+
|        __time        |      temperature     |      pressure      |
+----------------------+----------------------+--------------------+
| 2019-10-22 14:00:00.0 | -14919.459924611458  | 1901.2905602971227 |
| 2019-10-22 15:00:00.0 | -6035.789051408976   | 3362.989233184454  |
| 2019-10-22 16:00:00.0 | -5961.867724867079   | 3358.183901721163  |
| 2019-10-22 17:00:00.0 | -3.7921024546802755  | 3353.33377443314   |
| 2019-10-22 18:00:00.0 | -0.22397884804610158 | 3322.994063408674  |
+----------------------+----------------------+--------------------+
```

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 |
...
```

Aggregations

Hive supports many common aggregate functions, however it does not always translate these into queries on Druid in a way that reflects the intention of the user. The following sub-sections describe how to derive the intended aggregation results.

Sums

The `SUM` function is understood correctly by Hive to be pushed down to Druid, and so it will automatically generate the correct results.

For example,

```
SELECT SUM(value) sum
FROM telemetry
WHERE `__time` >= CURRENT_TIMESTAMP() - INTERVAL 1 DAY
AND sensor_id = 100;
+----------------------+
|         sum          |
+----------------------+
| -157826.08752537577  |
+----------------------+
```

Counts and averages

For Hive-Druid tables the Hive `COUNT` and `AVG` functions will use the count of the multiple partial aggregation records of a single time/dimension slice, instead of using the count column created by the count aggregation in the Druid data source. Because there are an arbitrary number of partial aggregation records for a single time/dimension slice, it is not meaningful to use the count of those records for query results.
Instead, to correctly derive the count and average aggregations for a Druid-backed table the `SUM` function must be used on the `count` column.

For example, to retrieve the average value and the count of measurements of a metric over the previous day, and to also demonstrate the unintended way to aggregate values on Druid tables:

```
SELECT
    SUM(value) / SUM(count) avg
  , SUM(count) count
  , AVG(value) wrong_avg
  , COUNT(*) wrong_count
FROM telemetry
WHERE `__time` >= CURRENT_TIMESTAMP() - INTERVAL 1 DAY
AND sensor_id = 100;
+--------------------+--------+---------------------+-------------+
|        avg         | count  |      wrong_avg      | wrong_count |
+--------------------+--------+---------------------+-------------+
| -294.97244278095854 | 15846  | -16753.166051279815  | 279         |
+--------------------+--------+---------------------+-------------+
```

Minimums and maximums

By default, a Hive-Druid table will not maintain the minimum or maximum values within each rollup. This means that when using the Hive MIN or MAX function on a measure the result will be the minimum and maximum of the summed measures for each slice of time/dimensions, which generally is not what is intended when selecting a minimum or maximum.

Instead, to correct derive the minimum and maximum of a measure for a Druid-backed table the Druid data source must be created with minimum and maximum aggregators. These are included in the same way as the count aggregator. See the Druid documentation for more information on these aggregators.

With the minimum and maximum aggregators included in the Hive-Druid table the minimums and maximums can be derived by using the Hive MIN and MAX function over the min and max columns created by the aggregators.

For example, where the minimum and maximum aggregators are named `value_min` and `value_max`:

```
SELECT
    MIN(value_min) min
  , MAX(value_max) max
FROM telemetry
WHERE `__time` >= CURRENT_TIMESTAMP() - INTERVAL 1 DAY
AND sensor_id = 100;
+---------------------+---------------------+
|         min         |         max         |
+---------------------+---------------------+
| -699.9823943886912  | 1023.9863627726648  |
+---------------------+---------------------+
```