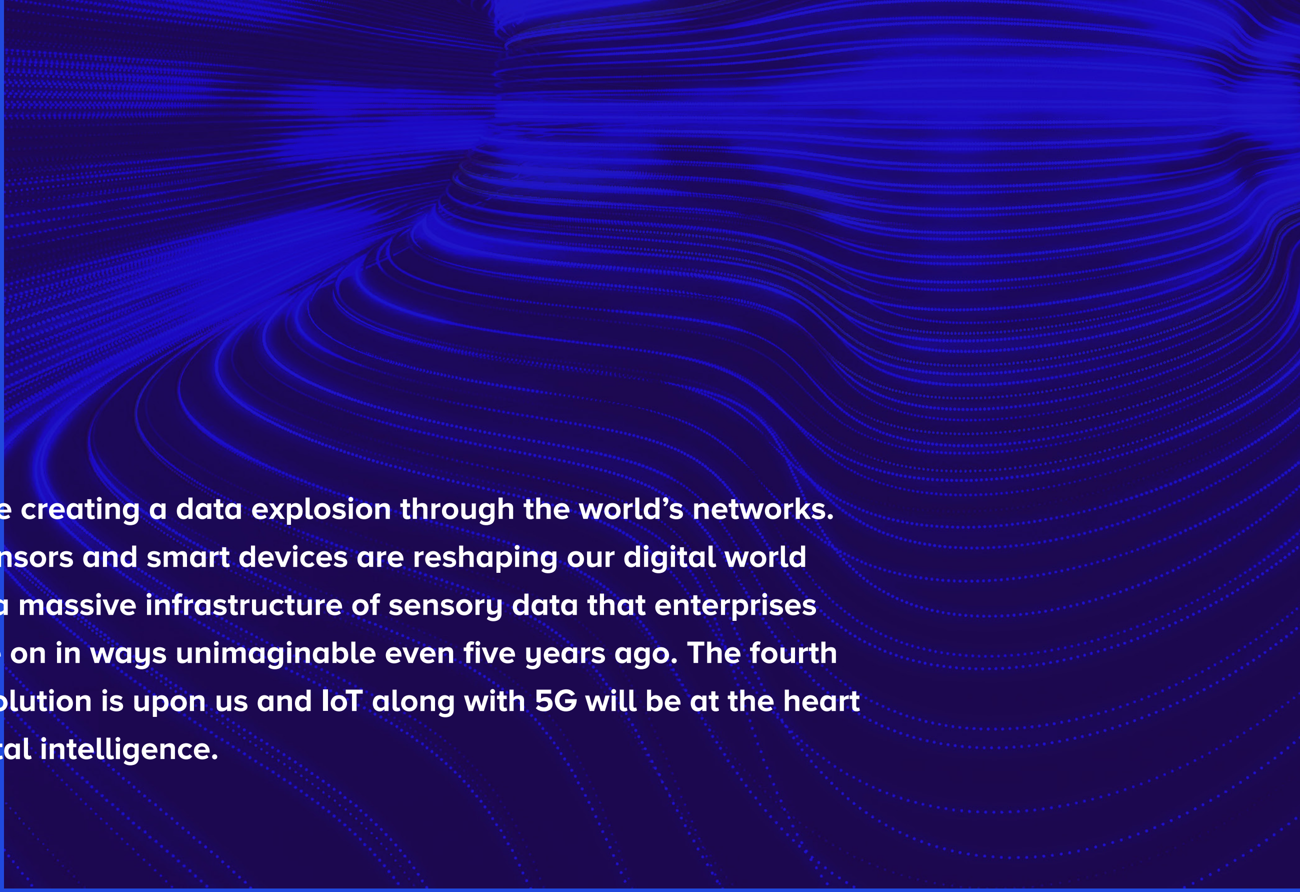
WHITEPAPER

Solving IoT Data Management Challenges

DATASTAX



IoT and 5G are creating a data explosion through the world's networks. Myriads of sensors and smart devices are reshaping our digital world and creating a massive infrastructure of sensory data that enterprises can capitalize on in ways unimaginable even five years ago. The fourth industrial revolution is upon us and IoT along with 5G will be at the heart of AI and digital intelligence.

Scope

In this white paper, we will examine the different components of an IoT system that need a data management layer to be able to ingest large volumes of high-velocity data. An IoT system can be broken down into three main areas:

1. Device connectivity

2. IoT hub

3. Business layer

Each of these areas provides different functionality and outputs. The devices are the things that need to be managed. The gateways can provide a single layer to allow immediate insights into those devices. The central hub serves as the administration and operations area for the gateways and devices and also provides reporting and insights that allow for predictive maintenance and remote actions. The BUSIVNESS layer connects actionable data to backend systems and provides 360 dashboards with KPIs for business users.

01

Device Management

A device can be any object that has telemetry data. Each device may also have multiple sensors producing different types of data for different time intervals.

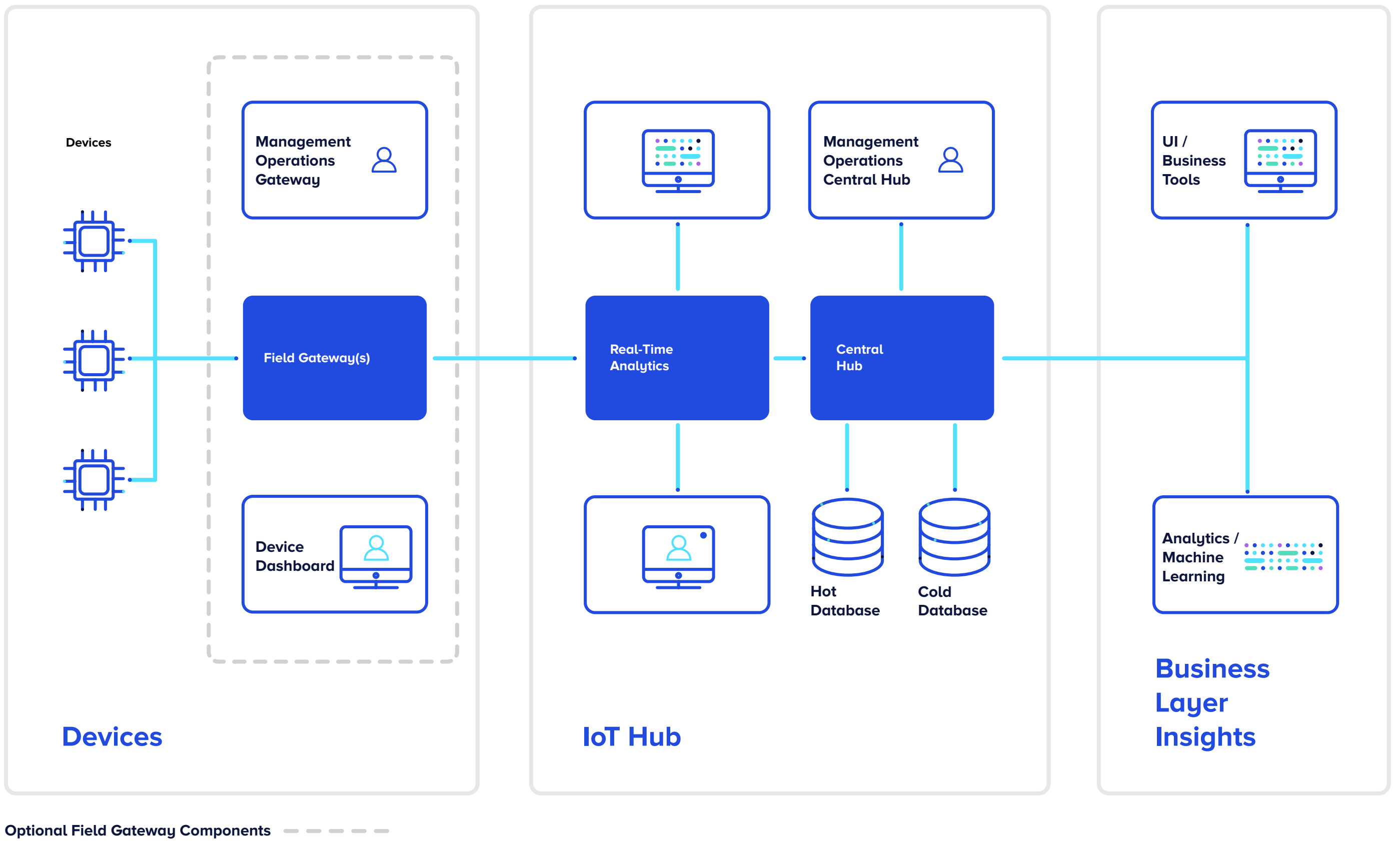


Figure 1. IoT Reference Architecture

To understand the devices and their sensors better, metadata and information for each device deployed needs to be managed, including:

Device Registry

This registry contains device-related metadata attributes and reference data for provisioned devices. The device registry serves as an index for device discoverability and is used by the solution backend components and UI. Typically, the device registry contains only slowly changing data. To implement this, we use a simple entity table holding the values associated with the device's protocols and credentials.

Device Configuration

A device may go through a number of different configurations in its lifetime. In most cases, it's important to understand the difference between each configuration and when they were successfully applied to the device in question.

Device Commands and Interactions

The central hub will often need to interact with the devices, sometimes directly and sometimes through a field gateway acting as a proxy. As such, it will be necessary to have a table hold this information, which works as an audit log for all the commands, whether successful or unsuccessful. An example of a simple command would be telling the device to perform a reset or look for setting updates from the hub.

Device Metadata

Metadata contains information about a device. This may include the device ID, model, manufacturer, and more. This information helps the system determine which devices are similar to others. For example. if we know a particular device created in a particular batch may have defects at a certain point in time, it is important to be able to find other devices from the same batch that could be similarly affected. For this reason, this table will be an entity search table.

Device State

The device state store contains the current state of the device. This is different from the latest configuration as at any time the desired configuration and current status can be different. This uses a simple entity table to hold the current state configuration of the device.

Device Shadows/Digital Twins

A digital twin or device shadow is a digital representation of a physical object or system. These can be useful to run simulations for testing state and usability changes remotely before changing the device software itself. Digital twins require all the above information as well as processes to help mirror what the physical device would do.

This registry contains device-related metadata attributes and reference data for provisioned devices.

02

Field Gateway

A field gateway is a specialized device or software that acts as a communication enabler and as a local device control system and device data processing hub. A field gateway can perform local processing and control functions for the devices. It can filter, aggregate, and transform the device telemetry it collects from its local devices making it easier to create a consistent layer for the central hub to collect.

For example, a local device can collect data from different protocols such as MQTT, CoAP, and AMQP. The field gateway can then transform this data into a common protocol like HTTP for data transfer to the central hub. The processing of the data at the edge on the gateway device reduces the amount of overall data transferred to the cloud backend.

Gateways may assist in device provisioning, data filtering, batching and aggregation, buffering of data, protocol translation, and event processing. If we think of a company with hundreds of remote oil rigs, we would use a field gateway on each of the oil rigs to help operate all the devices local to it. A central hub would then communicate using the field gateways as a proxy to the devices themselves. Field gateways are not always needed but can be useful when there is limited connectivity and bandwidth. Holding all this data for the central hub requires the field gateway to be stateful—hence using DataStax as its backend.

Gateways may assist in device provisioning, data filtering, batching and aggregation, buffering of data, protocol translation, and event processing.

Alerts, Filtering, and Aggregation

If we take the example of oil rigs a little further, each oil rig may keep the raw data set for all the devices on each oil rig, saving communication bandwidth by not sending all the data to a central hub. The field gateway can filter, aggregate, and summarize the data so that the central hub can see a high-level view of all the oil rigs' devices at any time. If there is a need to drill down on any of the components, the central hub can view or ask for the raw data to see more information. In comparison, any alerts or notifications from the gateway or devices need to be sent to the central hub in real time to allow actions to be taken.

Hybrid Cloud Environments

In most cases, field gateways are geographically distributed around the globe, sometimes on the other side of the world from the central hub. With cloud providers now providing services in all regions of the world, they can be used to collect, store, and process all of the data in a cost-effective and bandwidth-efficient manner with a low latency.

03

Central Hub

The central hub is where all the data comes together to allow operations, administration, and insights to take place. It contains two or more types of storage. One hot layer for ingesting and holding recent data and a cold layer for older data. The central hub can be on premises, close to the edge, or in the cloud, depending on the scale of the data or latency requirements.

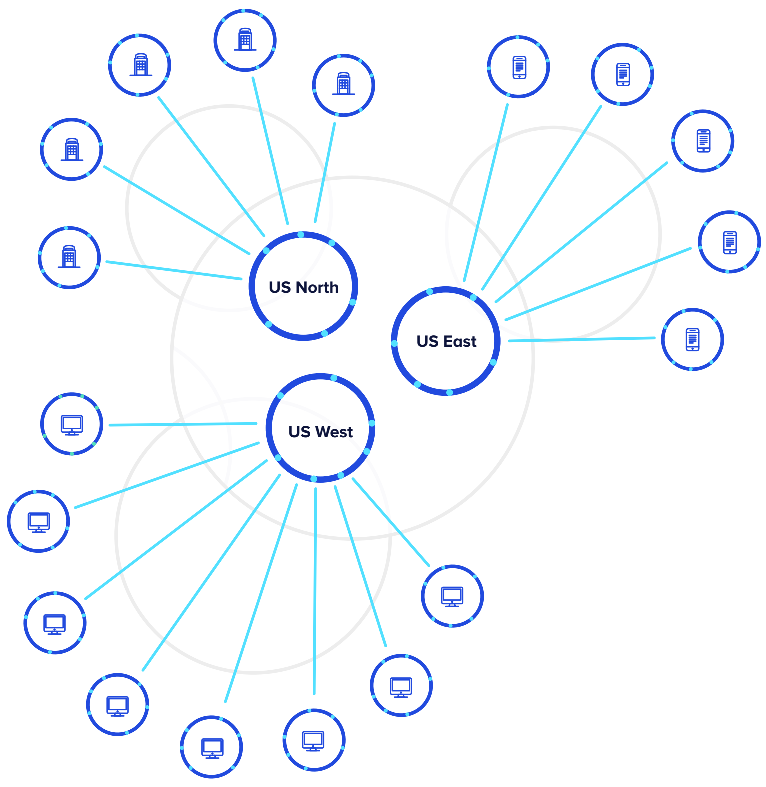


Figure 2. **Advanced Data Replication in DataStax.**

The diagram to the left shows an architecture where a central hub is created using data centers in three separate regions for business continuity and latency requirements. Each of the field gateways have two-way communication with the central hub, allowing for real-time alerting and statistics to be passed to the central hub, where device management can also be orchestrated.

Real-Time Analytics

When data arrives into the central hub it is usually streamed using one of the many event streaming technologies. This allows for complex event processing tasks such as data aggregation, data enrichment through correlation with reference data, as well as analytics tasks such as detection of threshold limits or anomalies and generation of alerts.

Batch Analytics

Slower batch-style analytics is ideal for cases where large amounts of data must be analyzed with batch queries or even ad hoc queries. Training machine learning models in batch (as opposed to incremental real-time training) or building monthly roll up reports are both use cases batch analytics is well suited for. The resulting model and data structures often feed in the real-time analytics reference data allowing for a fully automated feedback loop.

Each of the field gateways have two-way communication with the central hub, allowing for real-time alerting and statistics to be passed to the central hub, where device management can also be orchestrated.

04

Business Layer

Actionable Insights

The business component layer brings together data from the devices, gateways, and other sources to provide analysis for actionable insights. This analysis provides the ability to spot anomalies, explore trends, and measure operational efficiency. Other tools like machine learning and AI form the basis for predictive maintenance and automation, and help build the next generation of applications and services.

The business component layer brings together data from the devices, gateways, and other sources to provide analysis for actionable insights.

05

Why Apache Cassandra™ / DataStax

Apache Cassandra is a distributed database that delivers the high availability, high performance, and linear scalability today's most demanding applications require. It offers operational simplicity and effortless replication across cloud service providers, data centers, and geographies, and it can handle petabytes of information and thousands of concurrent operations per second across hybrid cloud environments.

DataStax Enterprise (DSE) is a platform with Cassandra at its core. DSE also integrates complementary technologies like search, analytics, and graph that could be used to simplify data when accessing and processing in comparison with open source Cassandra. Additionally, DSE provides at no additional cost a Kafka connector which allows data to seamlessly move from Apache Kafka to DSE in event-driven architectures. When exploring the various processes in an IoT system, we will discuss how they map to the feature set of Cassandra and DSE. Before we do this, though, we need to clarify some of the terms we will talk about later in this paper. It's helpful for the reader to have a fundamental understanding of Cassandra before reading any further.

Cassandra uses a columnar format to store data, which is ideal for collecting time series data. This model allows for data to be grouped together in a relationship table with a particular key having one or many rows associated with it. This key is called the partition key and each partition has one or many rows. This design allows data to be stored per source in an ascending or descending order, depending on the queries to be performed. The main thing to determine is the most common queries that will be used to retrieve the data. If it's a consumer application, then usually the device ID will make up the majority of the primary key which will always be provided when querying the data.

Cassandra uses a columnar format to store data, which is ideal for collecting time series data.

Some concepts that we need to define before describing table definitions are the following:

- **Table Types** – There are usually two types of tables that are created: entity tables and relationship tables. Entity tables are usually objects that have a number of values associated with them. An entity may also be searchable so that we can query any of the items in the table rather than just the primary key. Typical examples of this are a user or a device. A relationship table holds information regarding a relationship between an entity and objects that are in some way time related. Examples of this include a user's transactions or sensor values for a device.
- **Compactions Strategies** – Each data table has a feature called a compaction strategy. Compaction is the process of combining data files to ensure an optimized read path. This is very important in time series type models when compaction strategies change using entity and relationship tables. In Cassandra, there are three types of compaction strategies: one for time series, one for read-heavy data, and one for slowly growing data, which is the default.
- **Data Tiering** – Most big data databases move data between data tiers at some point to provide a better return on investment. Data tiers are usually based on speed and cost, so the hot tier will be fast but expensive, the warm will be cheaper and slower, and the cold will be the cheapest and slowest. So it makes sense to have a process that moves data through the tiers as the data becomes less critical.
- **Long-term Storage** – There are many long-term storage options available both in the cloud and on premises. A lot of these are designed as data lakes or warehouses. Technically, they can be distributed file systems, blob storage, or objects stores, or a combination of some or all of these.

Apache Cassandra is a distributed database that delivers the high availability, high performance, and linear scalability today's most demanding applications require. It offers operational simplicity and effortless replication across cloud service providers, data centers, and geographies, and it can handle petabytes of information and thousands of concurrent operations per second across hybrid cloud environments.

06

Data Management Best Practices

Telemetry Data

The telemetry data has the largest data footprint in the IoT system by far and, as such, requires specific data strategies to accommodate its size, speed, and retention period. For example, to create a simple table to collect many rows of data from a device, we can use the following table definition:

```
Create table datastax.device_data (
     id text,
     time timestamp,
     value int,
     PRIMARY KEY ((id), time)
) with clustering order by (time desc);

Insert into datastax.device_data (id, time, value) values (‘12312312’,
‘2019-08-10 00:00:00Z’, 13);
Insert into datastax.device_data (id, time, value) values (‘12312312’,
‘2017-08-10 00:01:00Z’, 11);
```

Partition Size

In Cassandra, we try to keep the partition size down to a max of 100MB of data or 100,000 rows. In this case, the partition is the device itself, so we will keep all the values associated with this device on the same partition. This design is obviously dependant on the data in the rows and is a quick guideline. To achieve this, if we suspect the data will get larger than these limits, we change the data model to incorporate a bucket into the partition key. This bucket can be something simple like a day or a month.

For example, if we were getting one value for each sensor every minute, we would expect every sensor to have 1,440 values each day. After 70 days, this volume will exceed the guideline for the partition size. If we want to keep less than 70 days’ worth of data, we can simply delete it when it’s no longer needed. If we want to keep this data for longer (e.g., five years), we can introduce a bucket of “year-month” to the partition key to ensure a maximum of 31 days belongs to one partition.

In Cassandra, we try to keep the partition size down to a max of 100MB of data or 100,000 rows.

The changed data model would look like this:

```
Create table datastax.device_data_bucket (
     id text,
     year_month text,
     time timestamp,
     value int,
     PRIMARY KEY ((id, year_month), time)
) with clustering order by (time desc);

Insert into datastax.device_data_bucket (id, year_month, time, value) values
('12312312', '201710',  '2017-10-10 00:00:00Z', 13);
Insert into datastax.device_data_bucket (id, year_month, time, value) values
('12312312', '201710',  '2017-10-11 00:00:00Z', 11);

select * from datastax.device_data_bucket where id ='12312312' and year_month='201710';
```

Collecting data can be very different than retrieving it. When collecting data, we are usually collecting millions of points from thousands or millions of devices. When querying, we are usually requesting the data for one device over a certain period of time. This allows us to perform some optimizations on long-term storage if needed. So, for example, older data can be stored in a warm compressed state with extremely fast retrieval times while very recent data can be held the way it's collected in the hot store.

So, for example, older data can be stored in a warm compressed state with extremely fast retrieval times while very recent data can be held the way it's collected in the hot store.

For example, we collect one value per minute for our sensors. When analyzing these values, the minimum granularity we look at is one day. In most cases, then, it would be advisable to move all the data points for a day into one single compressed field (i.e., a single column value or cell) after the data is collected in any format (e.g., bytes, JSON, and Java serializable). This allows data from a specific day to be retrieved by reading just one field, which can be extremely performant in the query while also reducing the historical data footprint. On top of this, it can benefit the underlying database processes that manage the data, such as repair, compaction, and streaming. Making these operations more lightweight frees up node resources and computational power to better serve user requests. (Note: There are also more complex time series compression algorithms like Gorilla that can be used but are outside the scope of this white paper.)

Retrieving the daily data to store it in a compressed fashion also gives us a chance to sample values, compute statistics, or aggregate data in a way that may be useful at a higher level. These calculated aggregates may include the number of values per day, the sum of the values, and maybe the average per day, per week, or per month, for example.

Keep in mind that the extent of the period over which the data is compressed doesn't have to be a day. It depends on the desired granularity when retrieving it, as well as on the expected size of the data over that period. The lower the granularity, the more concurrent reads will be needed for longer periods. The higher the granularity, the bigger the reads will need to be. Depending on the use case, the choice should be simple.

As an example of this approach, we could potentially have an additional table that contains data older than a month. Here's an example that shows only one measure field for the ID and time period:

```
Create table datastax.device_data_compressed (

id text,
     year_month text,
     measurements text,
     PRIMARY KEY ((id, year_month))
);

Insert into datastax.device_data_compressed (id, year_month, measurements) values
('12312312', '201710', "{'2017-10-10 00:00:00Z'-13, '2017-10-10 00:01:00Z'-15, …. }" );
```

Device Summary and Statistics Table

```
Create table datastax.device_stats (

     id text,
     year_month int,
     stat_name text,
     stat_value int,
     PRIMARY KEY (id, year_month, stat_name)
) with clustering order by (year_month desc, stat_name asc);

Insert into datastax.device_stats (id, year_month, stat_name, stat_value) values
('12312312', 201810, 'Avg', 13);
Insert into datastax.device_stats (id, year_month, stat_name, stat_value) values
('12312312', 201810, 'Sum', 1383);
```

In the above example, we need a process that collects all the data for a device over the previous month and compresses it, storing into the "measurements" column in the new compressed table. This would happen on or after the first of every month after all the data of the previous month has been collected. At this point, data could be sampled out and statistics or aggregates could be computed and stored for future use.

What if we receive data late? If data for a particular device arrives out of order for any reason, the process to analyze and compress the data can be run for individual devices and on the specific historical days. This could also happen if there is a need to reload data from backups.

The telemetry data has the largest data footprint in the IoT system by far and, as such, requires specific data strategies to accommodate its size, speed, and retention period.

Purging Data

In all database systems, there comes a time when some data may need to be deleted or purged, especially if we are moving data to a different tiering layer. DSE has a unique way of purging data with each column having a time to live (TTL) associated with it. This can be set to be any length of time, e.g., one day, seven days, or three months. When the TTL is over, the data will automatically be excluded from queries and removed from the database. The time window compaction strategy, created specifically for purging large volumes of cold data, also helps to remove data effectively.

Here's an example of inserting a one-day TTL, which is specified in seconds:

```
Insert into datastax.device_data (id, time, value) values ('12312311', '2017-10-10
00:00:00Z', 13) USING TTL 86400;
```

Another simple way to purge data is by using DSE Analytics or DSE Search to remove data after a certain time range. DSE Analytics will use Apache Spark to distribute the delete over many nodes at the same time.

An example of using Spark SQL to remove data in a large batch process is as follows:

```
spark.sql(s"select id, year_month, time from datastax.device_data where time <
cast('2017-09-05 00:00:00' as timestamp") .rdd.deleteFromCassandra("datastax",
"device_data", keyColumns = SomeColumns("id", "year_month", "time"))

sc.cassandraTable("test", "test").select("id", "time").where("time < '2017-09-
05 00:00:00'") .rdd.deleteFromCassandra("datastax", "device_data", keyColumns =
SomeColumns("id", "time"))
```

This provides an easy and effective way to delete lots of data for a specific time frame from the database in a distributed manner.

Data tiering is an important part of any data management system that has time-based data.

Data Tiering

Data tiering is an important part of any data management system that has time-based data. The data tiers are usually based on speed and cost, so the hot tier will be fast but expensive, the warm will be cheaper and slower, and the cold will be the cheapest and slowest.

When collecting the data, we'll use the hot tier for our fastest data, which is usually data that is currently being ingested and data that is being read for alerting and monitoring purposes. This will include aggregated and summary data from the central hub and field gateways if these are being used. The warm tier will be the first compressed layer of telemetry data, which will be generally immutable and represent a device's data for a certain time period. Finally, the cold tier will be a blob or file storage system that holds the data in long-term storage for mostly analytical and reporting purposes.

An example of moving the data from the hot tier to the warm tier was shown above using the compressed data table. This can be done periodically at a set time. Similarly, we can move data from the hot/warm tier to the cold tier when we want to archive it to a long retention store. The most efficient way to do this would be to use DSE Analytics. For example, if we decide we want to archive the data to a HDFS-compatible store using the Parquet format, we could do something like the following:

```
//Collect all sensor data for a month
val data = sqlContext.sql("Select year(time) as year_date, month(time) as month_date,
* from datastax.device_data_bucket where year_month='201908'")

//write data partitioned by year and month for time related queries
data.write.mode(SaveMode.Append)
.partitionBy("year_date","month_date").parquet("partitionedbyyearmonth/sensor_values.pqt")
```

Similarly, we could partition the data on a substring of the device ID to allow for better lookups of individual devices over longer periods of time.

Automation

Depending on the desired setup of the tiering layers of the storage system, each device's metadata can contain details around how often data is moved through various layers of storage tiers to allow flexibility for high- and low-velocity devices.

Real-Time Analytics

DSE Analytics uses Apache Spark to process events in real time. This integration enables real-time insights while processing the sensor data. Using data locality and data streaming, data analytics can process gigabytes of data quickly to run reports, process machine learning algorithms, and provide real-time alerting. In a typical deployment, an event processing system like Apache Kafka or Microsoft Azure Event Hub would be used.

Batch Analytics

DSE Analytics also handles more traditional large batch processing jobs, which can be used for predictive maintenance or reporting purposes. These jobs usually take a long time and require significant processing resources, and as they are not critical processes, they can usually be achieved using cold data storage and low-priority machines.

DSE Analytics uses Apache Spark to process events in real time. This integration enables real-time insights while processing the sensor data.

Conclusion

Data management already plays a key role in the success of the modern organization. That role will only take on more prominence as we move further into the future. Data management has never been easy. But in the coming years, it will get even more complicated, with sensors and devices around the world creating trillions of data points daily—each of which needs to be stored in a database. To succeed, organizations will need to manage three issues:

01 **How to create a data architecture that can manage the speed at which data will arrive from IoT devices**

02 **How to integrate that data with other internal and external data sources**

03 **How to handle the volume and speed of the data that is being generated**

In both industrial and consumer settings, organizations will need to prepare their data architecture to cope with the scale of data from IoT devices created and distributed across 5G networks in this new industrial digital era. Enterprises need to create strategies for consolidating and analyzing sensor data so that they can unlock the higher value of IoT data and provide actionable insights.

DataStax has already helped some of the world's largest companies with their IoT and device-based platforms, paving the way for the data management of the future.