Continental Tire

Project Central

Contents

Telemetry Backbone (TBB) 2

Introduction 2

Architecture 4

DataSource 5

DataSource Extractor 5

Ingestion Zone 6

TBB Processor 6

TBB Python Bindings 6

Cassandra and Spark collocated clusters 7

Telemetry Backbone (TBB)

Introduction

The Telemetry Backbone's main purpose is to support Data Scientists with a solid and easily accessible layer of all available and enriched telemetry data.

Nowadays the connections between users and network connected devices are generating large quantities of data. The data usually is created by the Internet of Things (IoT): "*a system of interrelated computing devices, mechanical and digital machines provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction (Wikipedia)*".

With the emergence of the Internet of Things, large volumes of data are generated today. All types of hardware can deliver information about their state and how they are being used. This can allow for completely new products and services such as "Predictive Maintenance". However, the amount and structure of data have changed, as well as the data delivery mechanisms (from vast amounts provided by batch interfaces to highly frequent small data sets being constantly streamed).

These new types of data can be used for all kinds of insights, for example:

- Performance of newly created products
- Usage of new products
- Performance of infrastructure to create new products

To facilitate these use cases, data must be readily available to analysts trying to get specific information. "Availability" here means:

- data should be in one single location
- data can be easily understood
- fetching different datasets is technically non challenging
- known data flows have been removed and data has been aligned automatically

This is what the Telemetry Backbone is trying to provide:

- a single location allowing data scientists to find the data they are seeking without the need to focus on different delivery strategies

- a platform to easily grasp the structural information of all datasets without the need to request that information from stakeholders
- a tool to easily consume the data in a technical way without the need to learn new APIs and interfaces for each individual dataset

The Telemetry Backbone with its underlying architecture is meant to provide Continental telemetry data of any kind. The Telemetry Backbone holds data from different loggers (e.G. Yard Reader, FlexBox, Avisaro, etc.) and aims at providing data enrichments such as CRM Master data or data from other providers such as altitude or weather conditions for GPS data. Data is usually provided in its initial aggregation (usually transactional level) to allow for better analysis.

If necessary, data can also be processed (e.g. aligned, cleaned, etc.) to allow for better quality.

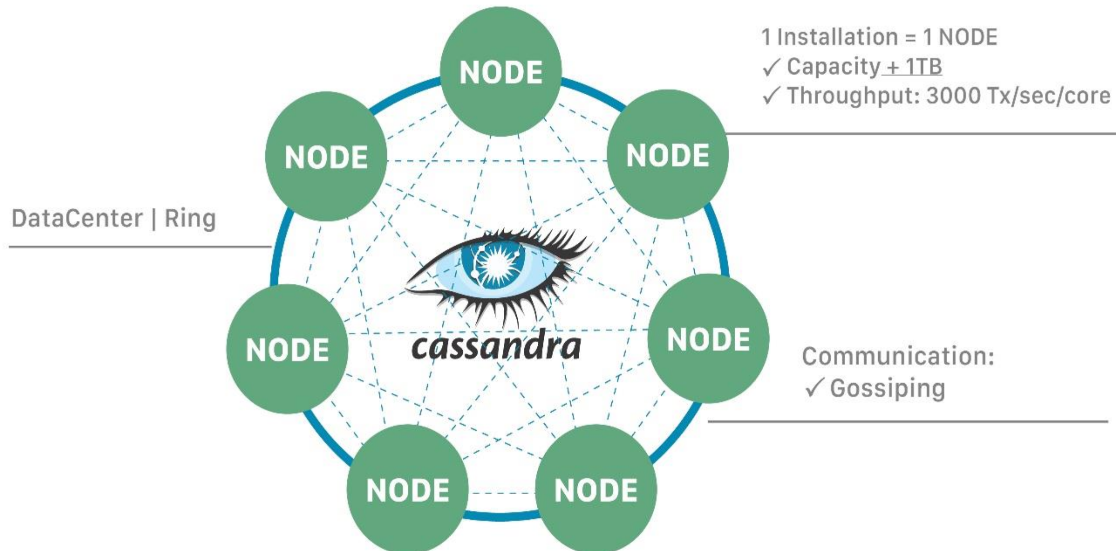
Data Scientists are the main consumers of the Telemetry Backbone. They are the main target group for the provided data. To comply with their specific needs to work on non-aggregated and non-selective data, the Telemetry Backbone focuses on delivering raw data, at most enriched with other datasets that can help the Data Scientist to detect relevant business information.

In addition to data science users, the Telemetry Backbone can also work as a consolidated access layer to telemetry data for analytical applications. Examples are: "Vehicle Location Tools" that require information about selected vehicles or "Customer Care Tools" that provide predictive maintenance services and need information about different loggers and their measurements.

The Telemetry Backbone is intended for Data Scientists and analytical applications. It is not intended for classical online transactional applications.

The architectural concept of the Telemetry Backbone supports analytical rather than transactional uses. Data is being stored and provided for analytical processes. Instead of working with small chunks of data (normalized and structured) in a read-write manner, the users work on large datasets in a read-process. Additionally, the Telemetry Backbone does not perceive itself as the main analytical layer. It does not allow complex queries of data. This is due to its persistence infrastructure where the very fast and scalable storage capacities come with the price of less querying functionalities. Instead, the Telemetry Backbone works as an integrated and large data storage from which Data Scientists can easily select large logical chunks of data (e.g. all logger data from one vehicle or all data from a particular time) and promote that data into their respective analytical tool set. That is where extensive analysis can then take place.

Architecture



The TBB uses Apache **C**assandra as persistence layer without single point of failure serves as storage for the data provided. Cassandra scales linearly.

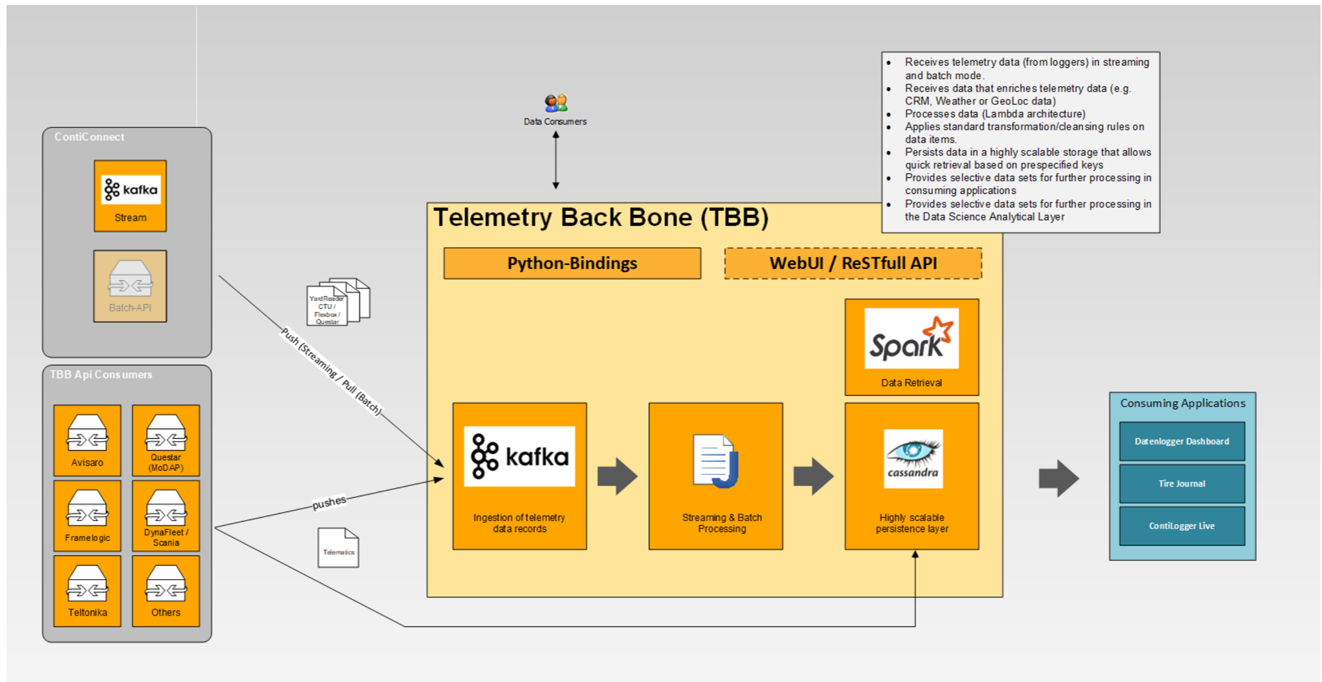
Data is currently being ingested via stream (mainly from the ContiConnect Kafka Broker) or via Batch process from different batch engines (e.g. Synetrics Leno).

The Telemetry Backbone utilizes a message broker (Kafka) and provides dedicated topics per data source as ingestion zone for incoming telemetry data.

Usually, when new loggers are added, they can be easily made available in the TBB, especially when they use one of the existing Kafka topics.

Some data are being delivered in a batch mode, e.g., Questar data. When they are being provided by a non-streaming API, they are usually fetched by Synetrics Leno and directly forwarded to the corresponding Kafka topic within the Telemetry Backbone.

Data is directly being stored in the Cassandra storage. There are always untouched copies of received data payloads (raw data). Data is stored in structured models in Cassandra. This process is preceded by the design of the desired structure and access paths, and by deciding how this data will be fetched and transported into an analytical layer from where it can be extensively queried.



For each data source, there is an extractor process responsible for fetching the correct data from its source, and for forwarding it (raw) to its dedicated Kafka topic (ingestion zone) in the TBB.

The logical unit of TBB Processor(s) handles all incoming sources, which might consist of several micro processes with intermediate topics and Spark jobs. The TBB Processor then stores raw and processed data into Cassandra.

DataSource

A DataSource is a (remote) service or storage which serves telemetry data. The data can be provided via streaming (e.g., via a message broker) or on a request basis (batch processing). Each DataSource provides an API which depends on the technology and access rules implemented by the Data Source owner (e.g., REST, FTP, Fileshare, Kafka, etc.).

The structure and format of the served data, the API specifications, sample data and additional details of a DataSource are documented in the data dictionary.

DataSource Extractor

The DataSource Extractor (aka API consumer) is responsible for fetching / retrieving data from a DataSource or multiple DataSources on a regular basis, either as an always online streaming application (e.g. when the DataSource is a message broker) or as a scheduled batch process (e.g. DB, File or REST). Each message received is considered as a raw payload. The DataSource

Extractor can transform/filter these messages as necessary and eventually forward them to the TBB's ingestion zone.

Once these raw payloads arrive in TBB, the TBB Processor will process and persist these messages.

Ingestion Zone

The Ingestion Zone is a set of Kafka topics. The DataSource Extractors produce messages and send them to the Kafka topics. The TBB Processor consumes messages from the KAfka topics. For each payload type there is a dedicated Kafka topic, depending on the DataSource and the implemented DataSource Extractor.

In most cases, there is one Kafka topic per DataSource Extractor. In some exceptional cases there might be multiple topics per DataSource Extractor. This is required when different payload types cannot be distinguished easily, e.g. the format and structure for GPS data is the same as for vehicle data and there is no clear discriminator in the payload. Or, if payloads are delivered in different file formats, e.g. csv vs. binary.

TBB Processor

The TBB Processor receives all telemetry data (produced by DataSource Extractors) from multiple Kafka topics in the Ingestion Zone. Each message is persisted in raw format (as received) so they can be re-processed any time when needed.

Additionally, if implemented, each message is further processed:

- persist each message in a structured format (Cassandra tables) for optimized query access
- homogenize data
- enrich data

The granularity of data remains untouched (there is no aggregation).

TBB Python Bindings

The TBB python bindings are an implementation of the DataStax Python Driver for Apache Cassandra. This driver works exclusively with the Cassandra Query Language v3 (CQL3) and Cassandra’s native protocol.

The TBB python bindings provide:

- Abstraction of storage technology, with the focus on data

- Easy access to raw sensor readings
- Predefined functions for standard use cases
- Interface to allow searches for telemetry data
- Fallback option for direct queries (data model and technology skills a must)

The TBB python bindings use the cassandra.cluster library from the DataStax Python Driver API to create a connection to the TBB Cassandra cluster, and the cassandra.auth (Cassandra authentication) library to create an Authenticator class instance with the necessary credentials. A Cluster instance is the main entry point to the TBB Cassandra cluster. The Cluster class constructor takes the cluster information (list of IP addresses) and the authentication information as parameters.

Cassandra and Spark collocated clusters

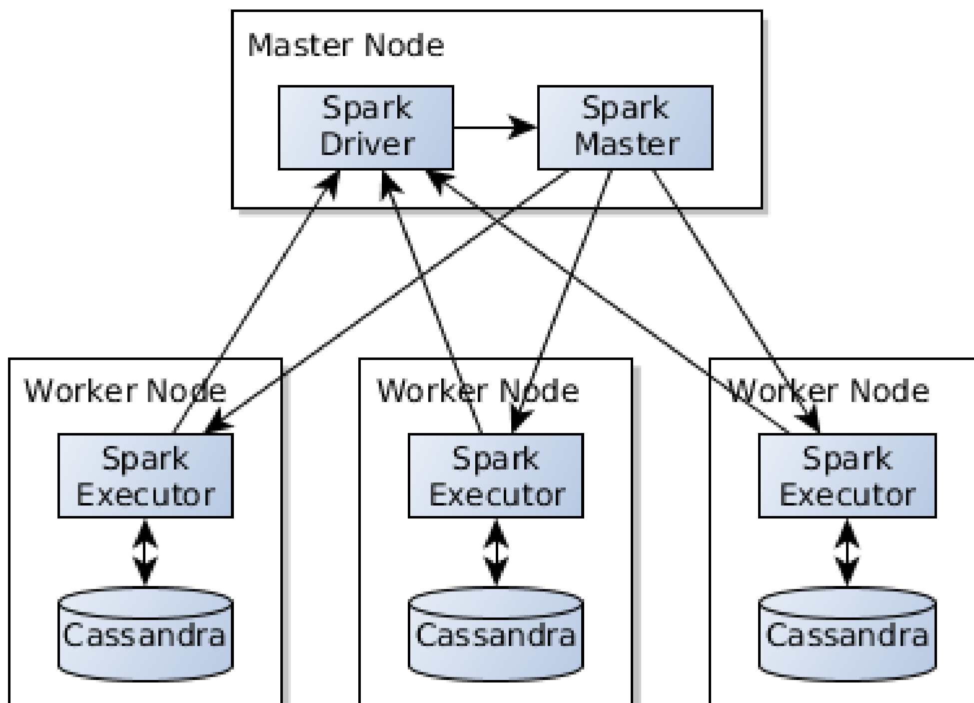
Apache Spark is a big data processing and analysis platform such as Hadoop MapReduce. Compared to Hadoop MapReduce, Spark offers several advantages, including:

Speed: Spark is designed to be much faster than Hadoop MapReduce, especially when it comes to iterative algorithms and interactive data analysis. Spark's in-memory computing capabilities allow for faster processing and analysis of data.

Ease of use: Spark's API is designed to be more user-friendly than Hadoop MapReduce, which can be complex and difficult to work with. Spark's API is built on top of the popular programming languages like Scala, Java, Python and R making it more accessible to developers.

Flexibility: Spark is a more versatile platform than Hadoop MapReduce, it can handle a wide range of workloads including batch processing, real-time streaming, interactive queries, and machine learning.

Apache Spark was installed in the same cluster as the TBB Cassandra database.



There are several advantages to using Apache Spark and Cassandra together on the same cluster:

Data locality: When Spark is installed on the same cluster as Cassandra, it can take advantage of data locality, which means that data can be read and processed locally rather than being transferred over the network. This can lead to significant performance improvements, particularly for large data sets.

Scalability: Both Cassandra and Spark are designed to scale horizontally, which means that they can handle increasing amounts of data and processing power by adding more nodes to the cluster. This allows for linear scalability and high availability, which is crucial for big data applications.

High availability: Cassandra is a highly available NoSQL database, meaning that it can handle node failures without any interruption. Combining with Spark allows us to handle the same level of high availability for the processing power which is crucial for big data applications.

Data modeling: Cassandra's data model is based on a key-value store, which makes it well suited for storing and processing large amounts of data. Spark can work with Cassandra's data model, allowing it to process and analyze data in an efficient and flexible way.

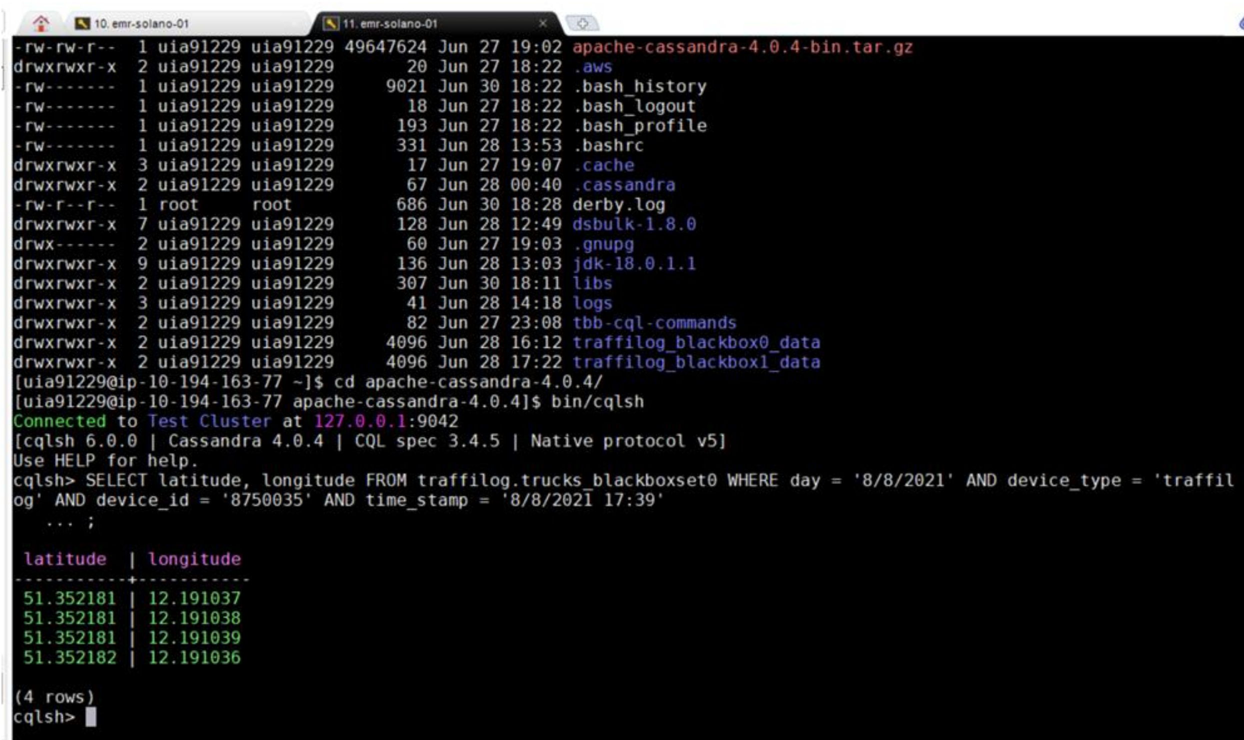
Integrated analytics: Spark can be used to perform advanced analytics and machine learning on data stored in Cassandra, providing a powerful integrated analytics solution.

High performance: Cassandra's write-heavy workloads and Spark's read-heavy workloads complement each other, they both can handle large amounts of data and high-throughput operations, so combining them together can provide better performance than using them separately.

In summary, using Apache Spark and Cassandra together on the same cluster allows for efficient data processing and analysis, high scalability, high-availability and provides an integrated analytics solution for big data applications.

With this collocation of Cassandra and Spark, it is possible to:

- query Cassandra from Spark using the CQL CassandraConnector API from DataStax to create a session and execute a query. Screenshots show the same query from Cassandra CQL shell and from the Spark Scala Shell):



```
10. emr-solano-01
scala> import com.datastax.spark.connector.cql.CassandraConnector
import com.datastax.spark.connector.cql.CassandraConnector

scala> val c = CassandraConnector(sc.getConf)
c: com.datastax.spark.connector.cql.CassandraConnector = com.datastax.spark.connector.cql.CassandraConnector@342daa77

scala> val rs = c.withSessionDo ( session => session.execute(" SELECT latitude, longitude FROM traffilog.trucks_blackboxset0
 WHERE day = '8/8/2021' AND device_type = 'traffilog' AND device_id = '8750035' AND time_stamp = '8/8/2021 17:39' "))
rs: com.datastax.oss.driver.api.core.cql.ResultSet = com.datastax.oss.driver.internal.core.cql.SinglePageResultSet@29dbf03f

scala> import java.util.function.Consumer;
import java.util.function.Consumer

scala> :paste
// Entering paste mode (ctrl-D to finish)

class RowConsumer extends java.util.function.Consumer[com.datastax.oss.driver.api.core.cql.Row] {
    def accept(row: com.datastax.oss.driver.api.core.cql.Row)
    {
        System.out.println(row.getFormattedContents);
    }
}
val rowConsumer = new RowConsumer
rs.forEach(rowConsumer)

// Exiting paste mode, now interpreting.

[latitude:51.352181, longitude:12.191037]
[latitude:51.352181, longitude:12.191038]
[latitude:51.352181, longitude:12.191039]
[latitude:51.352182, longitude:12.191036]
defined class RowConsumer
rowConsumer: RowConsumer = RowConsumer@7c59fe87

scala>
```

- use the CassandraTable and the CassandraJoinRDD APIs from DataStax to read data sources from Cassandra and manipulate the data: e.g., select data, join two tables (note: JOINS are NOT supported by Cassandra):

```
10. emr-solano-01
scala>

scala> import com.datastax.spark.connector._
import com.datastax.spark.connector._

scala> val rdd = sc.cassandraTable("traffilog", "trucks_blackboxset0")
rdd: com.datastax.spark.connector.rdd.CassandraTableScanRDD[com.datastax.spark.connector.CassandraRow] = CassandraTableScan
RDD[8] at RDD at CassandraRDD.scala:18

scala>

scala>

scala>

scala>

scala>

scala>

scala>

scala>

scala> val bbJoin = sc.cassandraTable("traffilog","trucks_blackboxset0").joinWithCassandraTable("traffilog","trucks_blackbo
xset1")
bbJoin: com.datastax.spark.connector.rdd.CassandraJoinRDD[com.datastax.spark.connector.CassandraRow,com.datastax.spark.conn
ector.CassandraRow] = CassandraJoinRDD[10] at RDD at CassandraRDD.scala:18

scala> bbJoin.toDebugString
res9: String = (6) CassandraJoinRDD[10] at RDD at CassandraRDD.scala:18 []

scala>
```

- use the Spark SQL API to run queries NOT allowed in Cassandra (e.g., using columns in the WHERE clause that are NOT part of the partition key)

10. emr-solano-01

```
scala>

scala>

scala>

scala>

scala>

scala>

scala> val a_sql = spark.sql(" SELECT latitude, longitude FROM bb0 WHERE day = '8/8/2021' AND device_type = 'traffilog' AND
 device_id = '8750035' AND time_stamp = '8/8/2021 17:39' ")
a_sql: org.apache.spark.sql.DataFrame = [latitude: decimal(38,18), longitude: decimal(38,18)]

scala> val a_sql = spark.sql(" SELECT day, latitude, longitude FROM bb0 WHERE day = '8/8/2021' AND device_id = '8750035' AN
D time_stamp = '8/8/2021 17:39' ")
a_sql: org.apache.spark.sql.DataFrame = [day: string, latitude: decimal(38,18) ... 1 more field]

scala> val a_sql = spark.sql(" SELECT day, latitude, longitude FROM bb0 WHERE day = '8/8/2021' AND device_id = '8750035' ")

a_sql: org.apache.spark.sql.DataFrame = [day: string, latitude: decimal(38,18) ... 1 more field]

scala> val a_sql = spark.sql(" SELECT altitude, latitude, longitude FROM bb0 WHERE day = '8/8/2021' AND device_id = '875003
5' and altitude > 100")
a_sql: org.apache.spark.sql.DataFrame = [altitude: decimal(38,18), latitude: decimal(38,18) ... 1 more field]

scala>

scala>

scala>

scala>
```

m by subscribing to the professional edition here: https://mobaxterm.mobatek.net