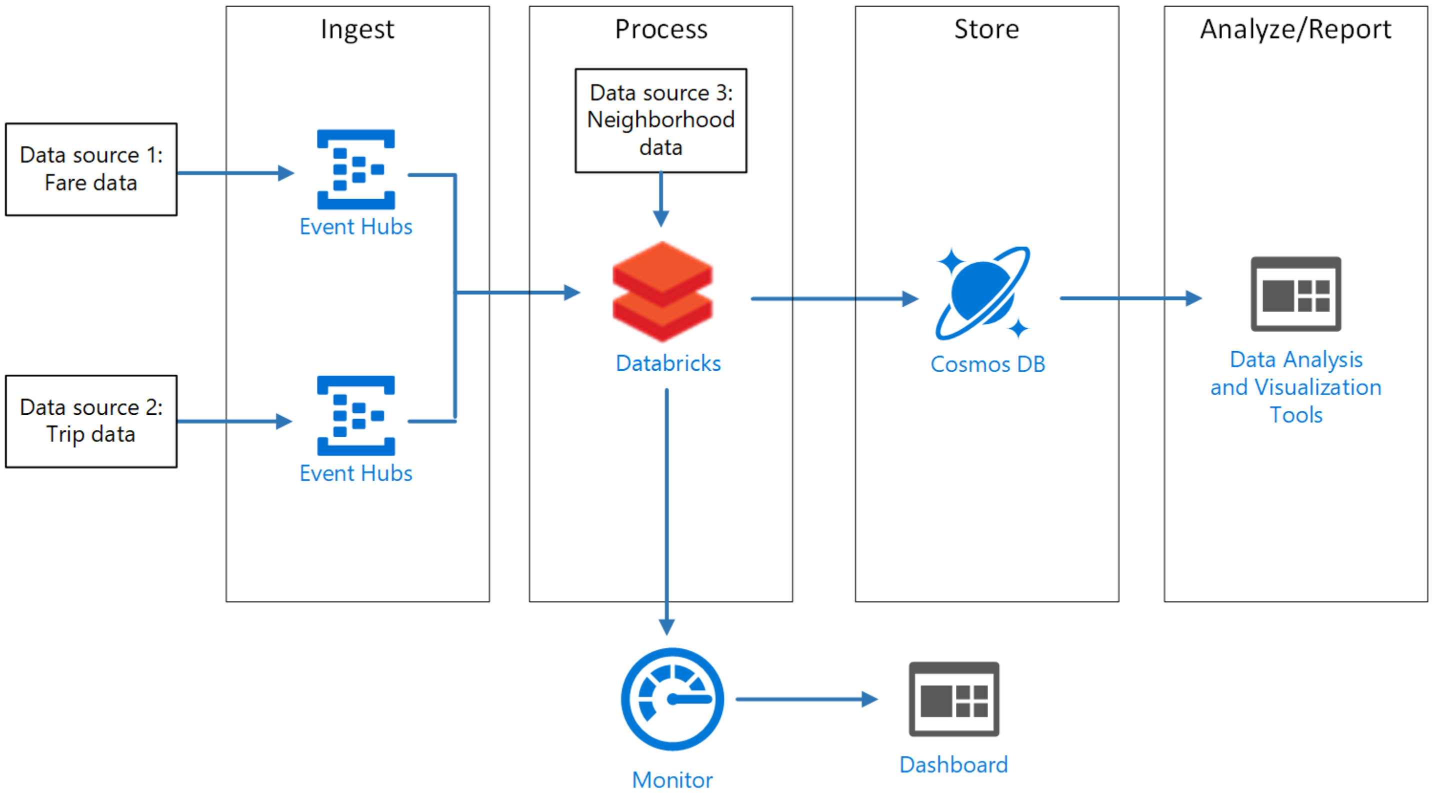
Create a stream processing pipeline with Azure Databricks

11/30/2018 • 9 minutes to read • Contributors

In this article

This reference architecture shows an end-to-end stream processing pipeline. This type of pipeline has four stages: ingest, process, store, and analysis and reporting. For this reference architecture, the pipeline ingests data from two sources, performs a join on related records from each stream, enriches the result, and calculates an average in real time. The results are stored for further analysis.

A reference implementation for this architecture is available on GitHub.



Scenario: A taxi company collects data about each taxi trip. For this scenario, we assume there are two separate devices sending data. The taxi has a meter that sends information about each ride — the duration, distance, and pickup and dropoff locations. A separate device accepts payments from customers and sends data about fares. To spot ridership trends, the taxi company wants to calculate the average tip per mile driven, in real time, for each neighborhood.

Architecture

The architecture consists of the following components.

Data sources. In this architecture, there are two data sources that generate data streams in real time. The first stream contains ride information, and the second contains fare information. The reference architecture includes a simulated data generator that reads from a set of static files and pushes the data to Event Hubs. The data sources in a real application would be devices installed in the taxi cabs.

Azure Event Hubs. Event Hubs is an event ingestion service. This architecture uses two event hub instances, one for each data source. Each data source sends a stream of data to the associated event hub.

Azure Databricks. Databricks is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Databricks is used to correlate of the taxi ride and fare data, and also to enrich the correlated data with neighborhood data stored in the Databricks file system.

Cosmos DB. The output from Azure Databricks job is a series of records, which are written to Cosmos DB using the Cassandra API. The Cassandra API is used because it supports time series data modeling.

Azure Log Analytics. Application log data collected by Azure Monitor is stored in a Log Analytics workspace. Log Analytics queries can be used to analyze and visualize metrics and inspect log messages to identify issues within the application.

Data ingestion

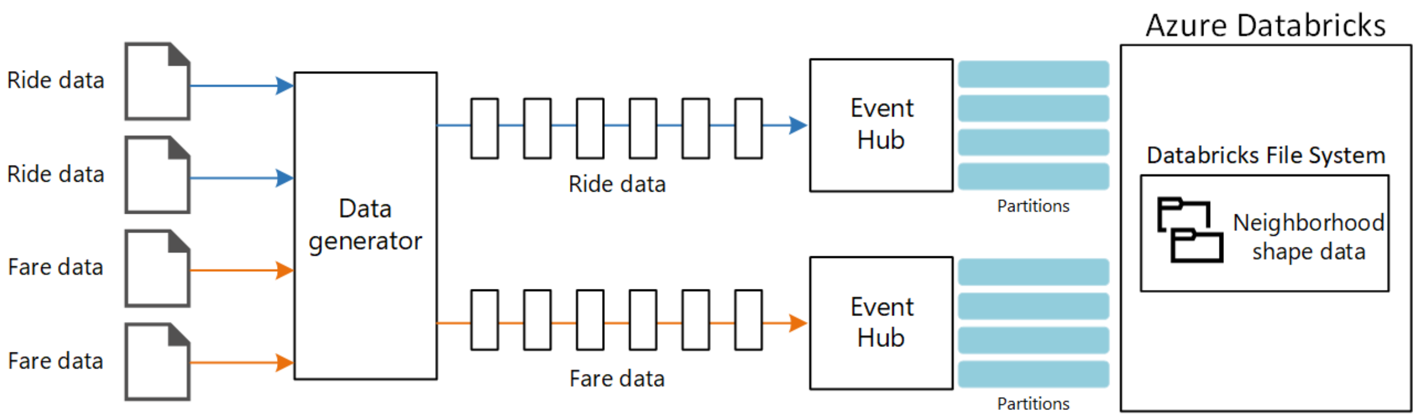
To simulate a data source, this reference architecture uses the New York City Taxi Data dataset[1]. This dataset contains data about taxi trips in New York City over a four-year period (2010 – 2013). It contains two types of record: Ride data and fare data. Ride data includes trip duration, trip distance, and pickup and dropoff location. Fare data includes fare, tax, and tip amounts. Common fields in both record types include medallion number, hack license, and vendor ID. Together these three fields uniquely identify a taxi plus a driver. The data is stored in CSV format.

[1] Donovan, Brian; Work, Dan (2016): New York City Taxi Trip Data (2010-2013). University of Illinois at Urbana-Champaign. https://doi.org/10.13012/J8PN93H8

The data generator is a .NET Core application that reads the records and sends them to Azure Event Hubs. The generator sends ride data in JSON format and fare data in CSV format.

Event Hubs uses partitions to segment the data. Partitions allow a consumer to read each partition in parallel. When you send data to Event Hubs, you can specify the partition key explicitly. Otherwise, records are assigned to partitions in round-robin fashion.

In this scenario, ride data and fare data should end up with the same partition ID for a given taxi cab. This enables Databricks to apply a degree of parallelism when it correlates the two streams. A record in partition *n* of the ride data will match a record in partition *n* of the fare data.



In the data generator, the common data model for both record types has a `PartitionKey` property that is the concatenation of `Medallion`, `HackLicense`, and `VendorId`.

C#

```csharp
public abstract class TaxiData
{
    public TaxiData()
    {
    }

    [JsonProperty]
    public long Medallion { get; set; }

    [JsonProperty]
    public long HackLicense { get; set; }

    [JsonProperty]
    public string VendorId { get; set; }

    [JsonProperty]
    public DateTimeOffset PickupTime { get; set; }

    [JsonIgnore]
    public string PartitionKey
    {
        get => $"{Medallion}_{HackLicense}_{VendorId}";
    }
```

This property is used to provide an explicit partition key when sending to Event Hubs:

C#

```csharp
using (var client = pool.GetObject())
{
    return client.Value.SendAsync(new EventData(Encoding.UTF8.GetBytes(
        t.GetData(dataFormat))), t.PartitionKey);
}
```

Event Hubs

The throughput capacity of Event Hubs is measured in throughput units. You can autoscale an event hub by enabling auto-inflate, which automatically scales the throughput units based on traffic, up to a configured maximum.

Stream processing

In Azure Databricks, data processing is performed by a job. The job is assigned to and runs on a cluster. The job can either be custom code written in Java, or a Spark notebook.

In this reference architecture, the job is a Java archive with classes written in both Java and Scala. When specifying the Java archive for a Databricks job, the class is specified for execution by the Databricks cluster. Here, the **main** method of the **com.microsoft.pnp.TaxiCabReader** class contains the data processing logic.

Reading the stream from the two event hub instances

The data processing logic uses Spark structured streaming to read from the two Azure event hub instances:

Scala

```scala
val rideEventHubOptions = EventHubsConf(rideEventHubConnectionString)
      .setConsumerGroup(conf.taxiRideConsumerGroup())
      .setStartingPosition(EventPosition.fromStartOfStream)
    val rideEvents = spark.readStream
      .format("eventhubs")
```

```
        .options(rideEventHubOptions.toMap)
        .load

      val fareEventHubOptions = EventHubsConf(fareEventHubConnectionString)
        .setConsumerGroup(conf.taxiFareConsumerGroup())
        .setStartingPosition(EventPosition.fromStartOfStream)
      val fareEvents = spark.readStream
        .format("eventhubs")
        .options(fareEventHubOptions.toMap)
        .load
```

Enriching the data with the neighborhood information

The ride data includes the latitude and longitude coordinates of the pick up and drop off locations. While these coordinates are useful, they are not easily consumed for analysis. Therefore, this data is enriched with neighborhood data that is read from a shapefile.

The shapefile format is binary and not easily parsed, but the GeoTools library provides tools for geospatial data that use the shapefile format. This library is used in the **com.microsoft.pnp.GeoFinder** class to determine the neighborhood name based on the pick up and drop off coordinates.

Scala

```scala
val neighborhoodFinder = (lon: Double, lat: Double) => {
      NeighborhoodFinder.getNeighborhood(lon, lat).get()
    }
```

Joining the ride and fare data

First the ride and fare data is transformed:

Scala

```scala
      val rides = transformedRides
        .filter(r => {
          if (r.isNullAt(r.fieldIndex("errorMessage"))) {
            true
          }
          else {
            malformedRides.add(1)
            false
          }
        })
        .select(
          $"ride.*",
          to_neighborhood($"ride.pickupLon", $"ride.pickupLat")
            .as("pickupNeighborhood"),
          to_neighborhood($"ride.dropoffLon", $"ride.dropoffLat")
            .as("dropoffNeighborhood")
        )
        .withWatermark("pickupTime", conf.taxiRideWatermarkInterval())

      val fares = transformedFares
        .filter(r => {
          if (r.isNullAt(r.fieldIndex("errorMessage"))) {
            true
          }
          else {
            malformedFares.add(1)
            false
          }
        })
        .select(
          $"fare.*",
          $"pickupTime"
        )
        .withWatermark("pickupTime", conf.taxiFareWatermarkInterval())
```

And then the ride data is joined with the fare data:

```scala
val mergedTaxiTrip = rides.join(fares, Seq("medallion", "hackLicense", "vendorId",
"pickupTime"))
```

Processing the data and inserting into Cosmos DB

The average fare amount for each neighborhood is calculated for a given time interval:

```scala
val maxAvgFarePerNeighborhood = mergedTaxiTrip.selectExpr("medallion", "hackLicense", "ven-
dorId", "pickupTime", "rateCode", "storeAndForwardFlag", "dropoffTime", "passengerCount",
"tripTimeInSeconds", "tripDistanceInMiles", "pickupLon", "pickupLat", "dropoffLon", "dropoff-
Lat", "paymentType", "fareAmount", "surcharge", "mtaTax", "tipAmount", "tollsAmount", "totalAm-
ount", "pickupNeighborhood", "dropoffNeighborhood")
      .groupBy(window($"pickupTime", conf.windowInterval()), $"pickupNeighborhood")
      .agg(
        count("*").as("rideCount"),
        sum($"fareAmount").as("totalFareAmount"),
        sum($"tipAmount").as("totalTipAmount")
      )
      .select($"window.start", $"window.end", $"pickupNeighborhood", $"rideCount", $"totalFare-
Amount", $"totalTipAmount")
```

Which is then inserted into Cosmos DB:

```scala
maxAvgFarePerNeighborhood
      .writeStream
      .queryName("maxAvgFarePerNeighborhood_cassandra_insert")
      .outputMode(OutputMode.Append())
      .foreach(new CassandraSinkForeach(connector))
      .start()
      .awaitTermination()
```

Security considerations

Access to the Azure Database workspace is controlled using the administrator console. The administrator console includes functionality to add users, manage user permissions, and set up single sign-on. Access control for workspaces, clusters, jobs, and tables can also be set through the administrator console.

Managing secrets

Azure Databricks includes a secret store that is used to store secrets, including connection strings, access keys, user names, and passwords. Secrets within the Azure Databricks secret store are partitioned by **scopes**:

```bash
databricks secrets create-scope --scope "azure-databricks-job"
```

Secrets are added at the scope level:

```bash
databricks secrets put --scope "azure-databricks-job" --key "taxi-ride"
```

> **Note**
>
> An Azure Key Vault-backed scope can be used instead of the native Azure Databricks scope. To learn more, see Azure Key Vault-backed scopes.

In code, secrets are accessed via the Azure Databricks secrets utilities.

Monitoring considerations

Azure Databricks is based on Apache Spark, and both use log4j as the standard library for logging. In addition to the default logging provided by Apache Spark, this reference architecture sends logs and metrics to Azure Log Analytics.

The **com.microsoft.pnp.TaxiCabReader** class configures the Apache Spark logging system to send its logs to Azure Log Analytics using the values in the **log4j.properties** file. While the Apache Spark logger messages are strings, Azure Log Analytics requires log messages to be formatted as JSON. The **com.microsoft.pnp.log4j.LogAnalyticsAppender** class transforms these messages to JSON:

```Scala

    @Override
    protected void append(LoggingEvent loggingEvent) {
        if (this.layout == null) {
            this.setLayout(new JSONLayout());
        }

        String json = this.getLayout().format(loggingEvent);
        try {
            this.client.send(json, this.logType);
        } catch(IOException ioe) {
            LogLog.warn("Error sending LoggingEvent to Log Analytics", ioe);
        }
    }

```

As the **com.microsoft.pnp.TaxiCabReader** class processes ride and fare messages, it's possible that either one may be malformed and therefore not valid. In a production environment, it's important to analyze these malformed messages to identify a problem with the data sources so it can be fixed quickly to prevent data loss. The **com.microsoft.pnp.TaxiCabReader** class registers an Apache Spark Accumulator that keeps track of the number of malformed fare and ride records:

```Scala
    @transient val appMetrics = new AppMetrics(spark.sparkContext)
    appMetrics.registerGauge("metrics.malformedrides", AppAccumulators.getRideInstance(spark.s-
parkContext))
    appMetrics.registerGauge("metrics.malformedfares", AppAccumulators.getFareInstance(spark.s-
parkContext))
    SparkEnv.get.metricsSystem.registerSource(appMetrics)
```

Apache Spark uses the Dropwizard library to send metrics, and some of the native Dropwizard metrics fields are incompatible with Azure Log Analytics. Therefore, this reference architecture includes a custom Dropwizard sink and reporter. It formats the metrics in the format expected by Azure Log Analytics. When Apache Spark reports metrics, the custom metrics for the malformed ride and fare data are also sent.

The last metric to be logged to the Azure Log Analytics workspace is the cumulative progress of the Spark Structured Streaming job progress. This is done using a custom StreamingQuery listener implemented in the **com.microsoft.pnp.StreamingMetricsListener** class. This class is registered to the Apache Spark Session when the job runs:

Scala Copy

```scala
spark.streams.addListener(new StreamingMetricsListener())
```

The methods in the StreamingMetricsListener are called by the Apache Spark runtime whenever a structured steaming event occurs, sending log messages and metrics to the Azure Log Analytics workspace. You can use the following queries in your workspace to monitor the application:

Latency and throughput for streaming queries

shell Copy

```shell
taxijob_CL
| where TimeGenerated > startofday(datetime(<date>)) and TimeGenerated <
endofday(datetime(<date>))
| project  mdc_inputRowsPerSecond_d, mdc_durationms_triggerExecution_d
| render timechart
```

Exceptions logged during stream query execution

shell Copy

```shell
taxijob_CL
| where TimeGenerated > startofday(datetime(<date>)) and TimeGenerated <
endofday(datetime(<date>))
| where Level contains "Error"
```

Accumulation of malformed fare and ride data

shell Copy

```shell
SparkMetric_CL
| where TimeGenerated > startofday(datetime(<date>)) and TimeGenerated <
endofday(datetime(<date>))
| render timechart
| where name_s contains "metrics.malformedrides"

SparkMetric_CL
| where TimeGenerated > startofday(datetime(<date>)) and TimeGenerated <
endofday(datetime(<date>))
| render timechart
| where name_s contains "metrics.malformedfares"
```

Job execution to trace resiliency

shell Copy

```shell
SparkMetric_CL
| where TimeGenerated > startofday(datetime(<date>)) and TimeGenerated <
endofday(datetime(<date>))
```

```
| render timechart
| where name_s contains "driver.DAGScheduler.job.allJobs"
```

For more information, see Monitoring Azure Databricks.

Deploy the solution

To the deploy and run the reference implementation, follow the steps in the GitHub readme.