# Streaming Data Pipeline to Transform, Store and Explore Healthcare Dataset With Apache Kafka API, Apache Spark, Apache Drill, JSON and MapR-DB

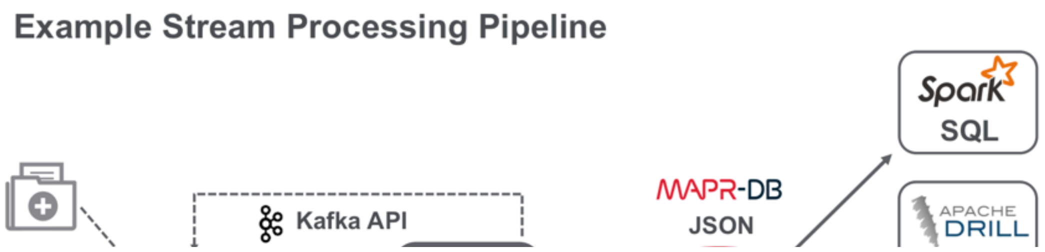
Contributed by Carol McDonald

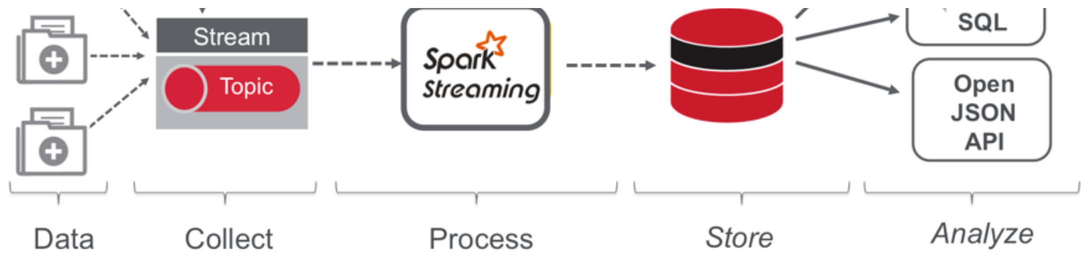
In the past, big data was interacted with in batch on a once-a-day basis. Now data is dynamic and data driven businesses need instant results from continuously changing data. Data Pipelines, which combine real-time Stream processing with the collection, analysis and storage of large amounts of data, enable modern, real-time applications, analytics and reporting.

This post is based on a recent workshop I helped develop and deliver at a large health services and innovation company's analytics conference. This company is combining streaming data pipelines with data science on top of the MapR Converged Data Platform to improve healthcare outcomes, improve access to appropriate care, better manage cost, and reduce fraud, waste and abuse.

In this post we will:

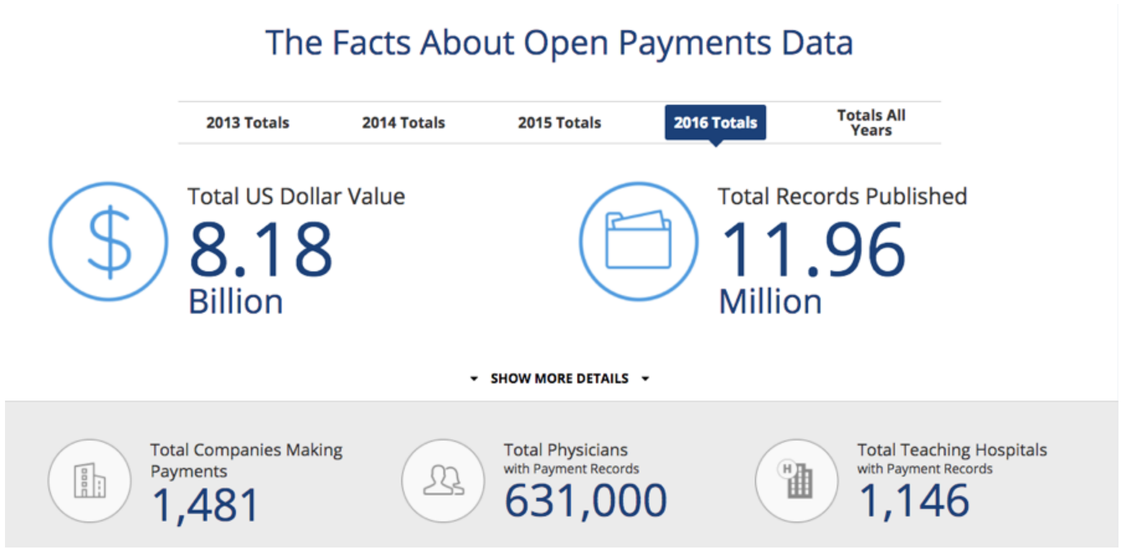
- Use Apache Spark streaming to consume Medicare Open payments data using the Apache Kafka API
- Transform the streaming data into JSON format and save to the MapR-DB document database.
- Query the MapR-DB JSON table with Apache Spark SQL, Apache Drill, and the Open JSON API (OJAI) and Java.





## Example Use Case Data Set

Since 2013, Open Payments is a federal program that collects information about the payments drug and device companies make to physicians and teaching hospitals for things like travel, research, gifts, speaking fees, and meals.


The Facts About Open Payments Data

2013 Totals | 2014 Totals | 2015 Totals | 2016 Totals | Totals All Years

Total US Dollar Value
8.18
Billion

Total Records Published
11.96
Million

SHOW MORE DETAILS

Total Companies Making Payments
1,481

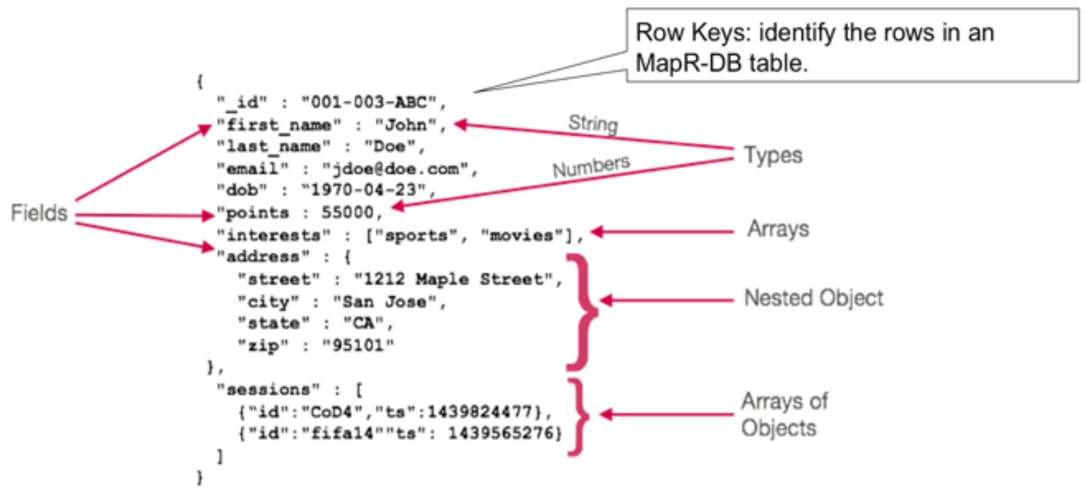
Total Physicians
with Payment Records
631,000

Total Teaching Hospitals
with Payment Records
1,146

A large Health payment dataset, JSON, Apache Spark, MapR-ES, and MapR-DB are an interesting combination for a health analytics workshop because:

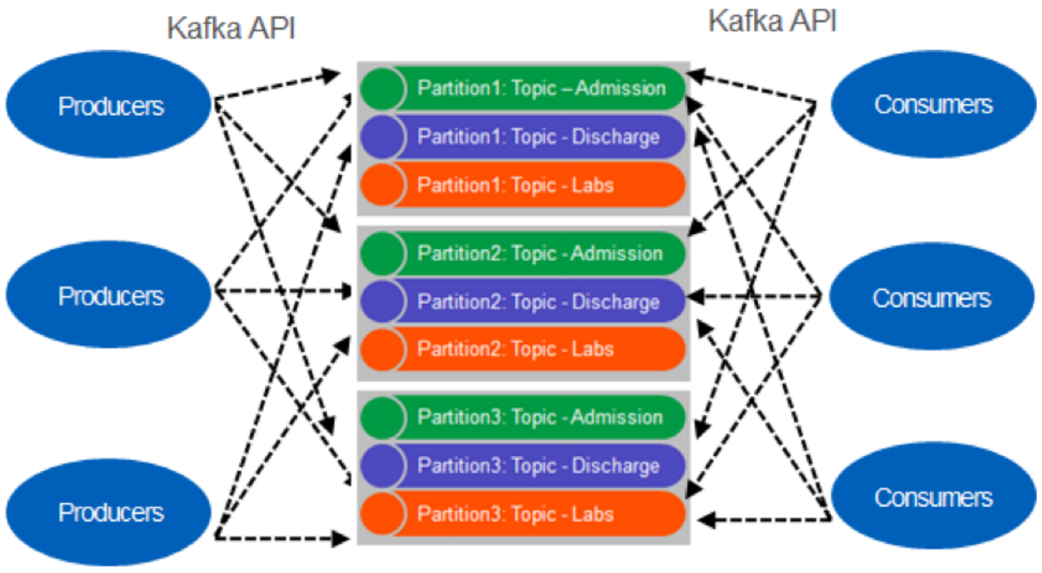
- JSON is an open-standard and efficient format that is easy for computer

languages to manipulate. Newer standards for exchanging healthcare information such as FHIR are easier to implement because they use a modern suite of API technology, including JSON.
- Apache Spark SQL, Dataframes, and Datasets make it easy to load, process, transform, and analyze JSON data. MapR-ES is a distributed messaging system for streaming event data at scale. MapR-ES integrates with Spark Streaming via the Kafka API.
- MapR-DB, a high performance NoSQL database, supports JSON documents as a native data store. MapR-DB makes it easy to store, query and build applications with JSON documents. The Spark connector makes it easy to build real-time or batch pipelines between your JSON data and MapR-DB and leverage Spark within the pipeline.


```
{
  "_id" : "001-003-ABC",
  "first_name" : "John",
  "last_name" : "Doe",
  "email" : "jdoe@doe.com",
  "dob" : "1970-04-23",
  "points : 55000,
  "interests" : ["sports", "movies"],
  "address" : {
    "street" : "1212 Maple Street",
    "city" : "San Jose",
    "state" : "CA",
    "zip" : "95101"
 },
  "sessions" : [
    {"id":"CoD4","ts":1439824477},
    {"id":"fifa14""ts": 1439565276}
  ]
}
```

## How Do You Build a Data Pipeline that handles millions of events in Real Time at Scale?

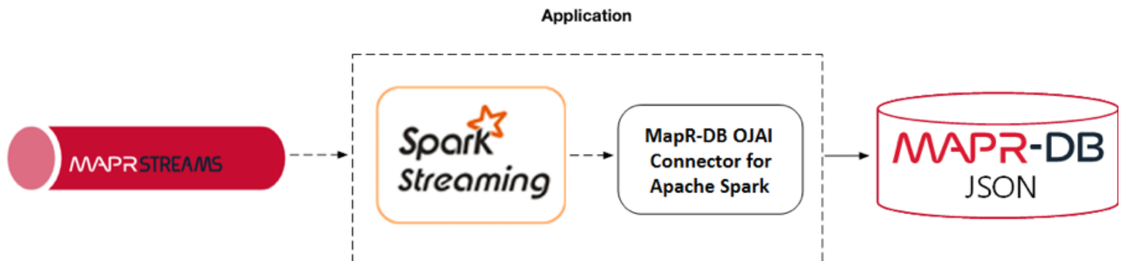
A common data pipeline architecture pattern is event sourcing using an append only publish subscribe event stream such as MapR Event Streams (which provides a Kafka API). MapR-ES Topics are logical collections of events, which organize events into categories and decouple producers from consumers, making it easy to add new producers and consumers. Topics are partitioned for throughput and scalability, producers are load balanced and consumer can be grouped to read in parallel. MapR-ES can scale to very high throughput levels, easily delivering millions of messages per second using very modest hardware.


Kafka API
Kafka API
Producers
Producers
Producers
Consumers
Consumers
Consumers
Partition1: Topic – Admission
Partition1: Topic - Discharge
Partition1: Topic - Labs
Partition2: Topic - Admission
Partition2: Topic - Discharge
Partition2: Topic - Labs
Partition3: Topic - Admission
Partition3: Topic - Discharge
Partition3: Topic - Labs

## Processing Streaming Data with Spark

Apache Spark Streaming is an extension of the core Spark API that enables continuous data stream processing. Data streams can be processed with Spark's core, SQL, GraphX, or machine learning APIs, and can be persisted to a file system, HDFS, MapR-XD, MapR-DB, HBase, or any data source offering a Hadoop OutputFormat or Spark connector. Stream processing of events is useful for filtering, transforming, creating counters and aggregations, correlating values, joining streams together, machine learning, and publishing to a different topic for pipelines.

MapR Event Streams integrates with Spark Streaming via the Kafka direct approach. The MapR-DB OJAI Connector for Apache Spark enables you to use MapR-DB as a sink for Apache Spark Data Streams.



The incoming data is in CSV format, an example is shown below:

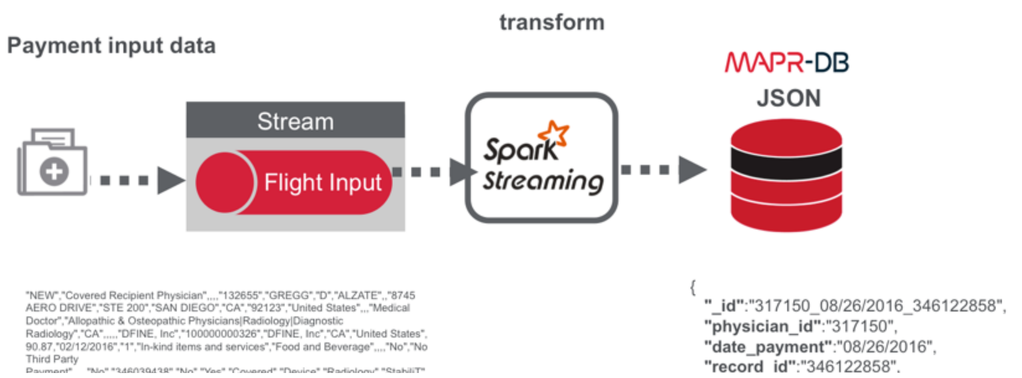
```
"NEW","Covered Recipient Physician",,,,"132655","GREGG","D","ALZATE",,"8745 AE
```

There are a lot of fields in this data that we will not use; we will parse the following fields:

| Provider ID | Record ID | Date | Payer | Specialty | Amount | Payment Nature |
|---|---|---|---|---|---|---|
| 1261770 | 346122858 | 01/11/2016 | Southern Anesthesia & Surgical, Inc | Oral and Maxillofacial Surgery | 117.5 | Food and Beverage |

And transform them into the following JSON document for storing in MapR-DB:

```
{

    "_id" :"317150_08/26/2016_346122858",
    "physician_id" :"317150",
    "date_payment" :"08/26/2016",
    "record_id" :"346122858",
    "payer" :"Mission Pharmacal Company",
    "amount" :9.23,
    "Physician_Specialty" :"Obstetrics & Gynecology",
    "Nature_of_payment" :"Food and Beverage"
}
```



```
"NEW","Covered Recipient Physician",,,,"132655","GREGG","D","ALZATE",,"8745
AERO DRIVE","STE 200","SAN DIEGO","CA","92123","United States",,,"Medical
Doctor","Allopathic & Osteopathic Physicians|Radiology|Diagnostic
Radiology","CA",,,,,"DFINE, Inc","100000000326","DFINE, Inc","CA","United States",
90.87,"02/12/2016","1","In-kind items and services","Food and Beverage",,,,"No","No
Third Party
Payment",,,"No","346039438","No","Yes","Covered","Device","Radiology","StabiliT"
```

```
{
  "_id":"317150_08/26/2016_346122858",
  "physician_id":"317150",
  "date_payment":"08/26/2016",
  "record_id":"346122858",
```

,"Covered","Device","Radiology","STAR Tumor Ablation System",,,,,,,,,,,,,,,,"2016","06/30/2017"

```
"payer":"Mission Pharmacal Company",
"amount":9.23,
"Physician_Specialty":"Obstetrics & Gynecology",
"Nature_of_payment":"Food and Beverage"
}
```

Spark Kafka Consumer Producer Code

(Note code snippets are shown here, you can download the complete code and instructions from the github link at the end of this post)

## Parsing the Data Set Records

A Scala Payment case class defines the schema corresponding to the CSV data that we are interested in. The parsePayment function parses a line of comma separated values into the Payment case class.

```scala
case class Payment(physician_id: String, date_payment: String, record_id: String,
  payer: String, amount: Double, physician_specialty: String,
  nature_of_payment: String) extends Serializable

def parsePayment(str: String): Payment = {
  val td = str.split(",(?=([^\\\"]*\\\"[^\\\"]*\\\")*[^\\\"]*$)")
  Payment(td(5), td(31),td(45), td(27),Try(td(30).toDouble) getOrElse 0.0,td(19),td(34))
}
```

A PaymentwId class defines the JSON document schema for MapR-DB.

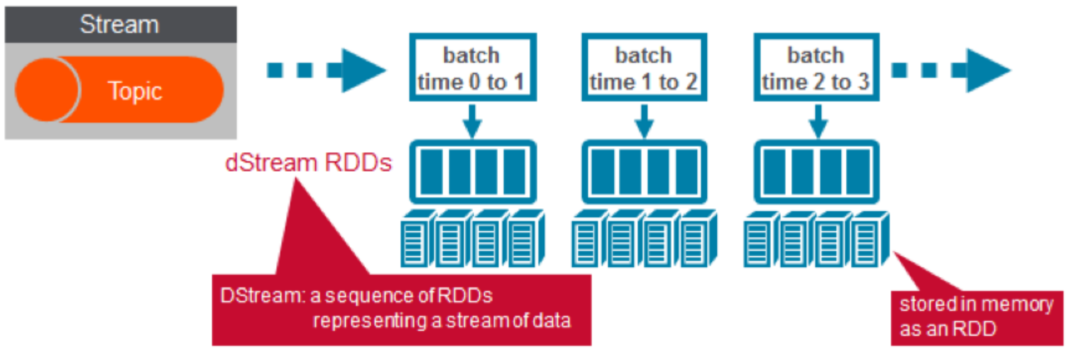
In order to save the JSON objects to MapR-DB, we need to define the_id field, which is the row key and primary index for MapR-DB. The parsePaymentwID function creates an object with the id equal to a combination of the physician id, the date, and the record id. Since MapR-DB row keys are partitioned and sorted by row key when inserted, the payment documents will be grouped by physician and date in MapR-DB. This will give really fast queries, aggregations and sorting by physician id and date. We will also look at secondary indexes later in this post.

```scala
case class PaymentwId(_id: String, physician_id: String, date_payment: String,
  payer: String, amount: Double, physician_specialty: String,
  nature_of_payment: String) extends Serializable

def parsePaymentwID(str: String): PaymentwId = {
  val pa = parsePayment(str)
  val id = pa.physician_id + '_' + pa.date_payment + '_' + pa.record_id
  PaymentwId(id, pa.physician_id, pa.date_payment, pa.payer, pa.amount,
    pa.physician_specialty, pa.nature_of_payment)
}
```

## Spark Streaming Code

We use the KafkaUtils createDirectStream method with Kafka configuration parameters to create an input stream from a MapR-ES topic. This creates a DStream that represents the stream of incoming data, where each message is a key value pair. We use the DStream map transformation to create a DStream with the message values, and then another map transformation with the parsePaymentwID function to create a DStream of PaymentwID objects.

```
val messagesDStream = KafkaUtils.createDirectStream[String, String](
   ssc, LocationStrategies.PreferConsistent, consumerStrategy
)
val valuesDStream:DStream[String]=messagesDStream.map(_.value())
val paymentDStream:DStream[PaymentwId]=valuesDStream.map(parsePaymentwID)
paymentDStream.print(3)
```

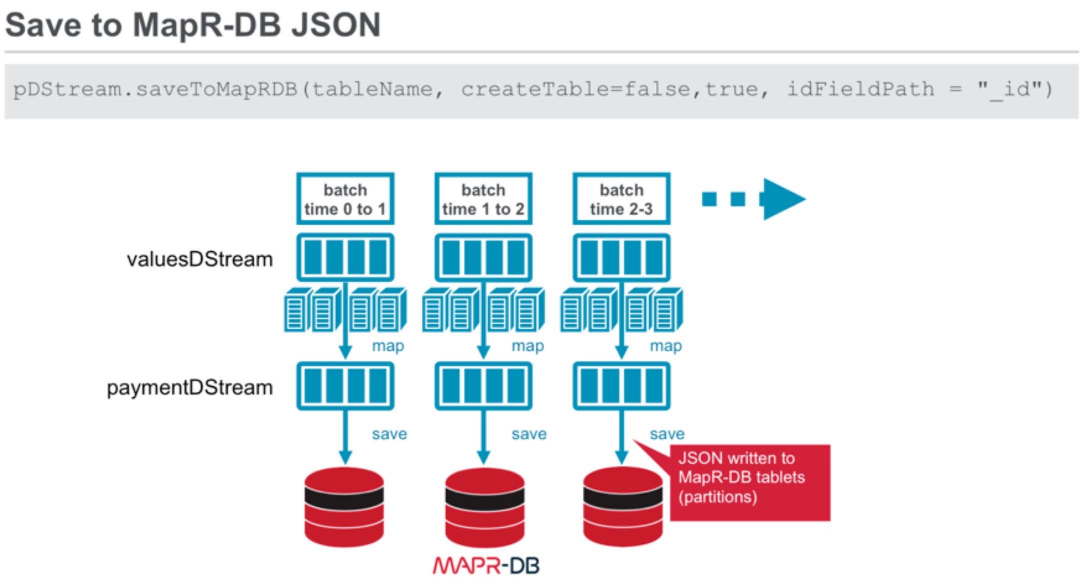

Stream
Topic
batch time 0 to 1
batch time 1 to 2
batch time 2 to 3
dStream RDDs
DStream: a sequence of RDDs representing a stream of data
stored in memory as an RDD

The output of the paymentDStream.print(3) is shown below

```
-------------------------------------------
Time: 1515544322000 ms
-------------------------------------------
PaymentwId(1212024_01/14/2016_346146118,1212024,01/14/2016,Braintree Labs,18.53,Internal
Medicine,Food and Beverage)
PaymentwId(1212024_02/03/2016_346146120,1212024,02/03/2016,Braintree Labs,15.34,Internal
Medicine,Food and Beverage)
PaymentwId(1212024_03/31/2016_346146122,1212024,03/31/2016,Braintree Labs,10.63,Internal
Medicine,Food and Beverage)
```

For storing lots of streaming data, we need a data store that supports fast writes and scales. The MapR-DB Spark Connector DStream saveToMapRDB method performs a parallel partitioned bulk insert of JSON PaymentwID objects into MapR-DB:

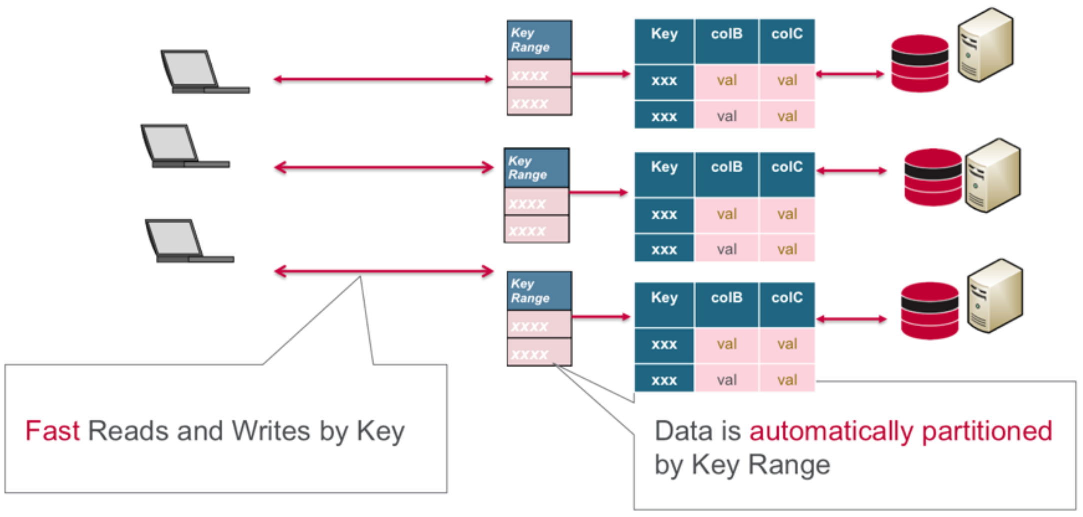
```
paymentDStream.saveToMapRDB(tableName,
   createTable= false, bulkInsert=true, idFieldPath = "_id")
```


Save to MapR-DB JSON

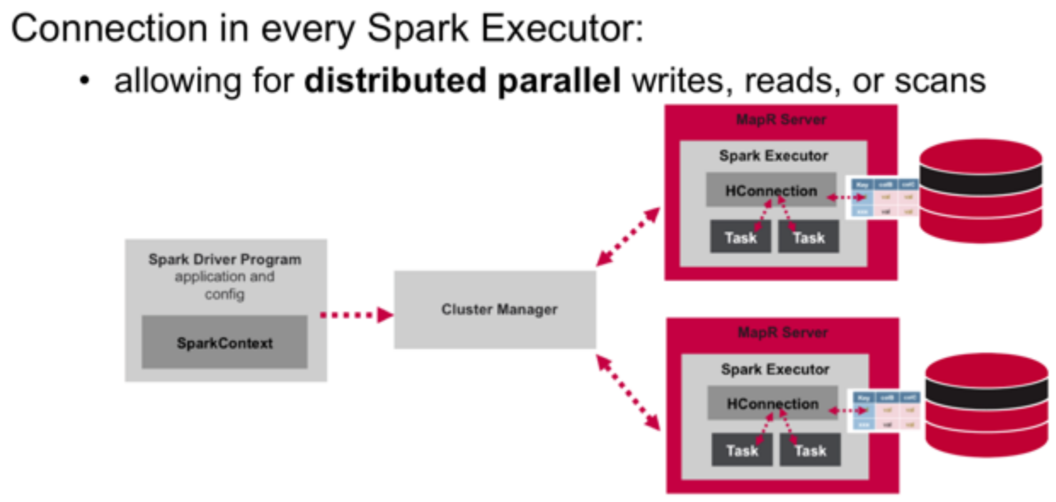
```
pDStream.saveToMapRDB(tableName, createTable=false,true, idFieldPath = "_id")
```

## MapR-DB

One of the challenges when you are processing lots of data is where do you want to store it? With MapR-DB (HBase API or JSON API), a table is automatically partitioned into tablets across a cluster by key range, providing for scalable and fast reads and writes by row key.

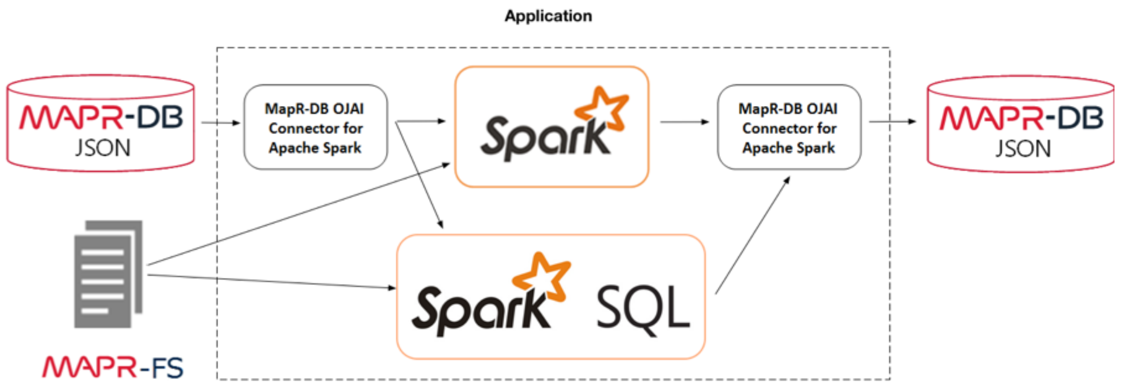

Key Range
xxxx
Key
colB
colC
xxx
val
Fast Reads and Writes by Key
Data is automatically partitioned by Key Range

The Spark MapR-DB Connector leverages the Spark DataSource API. The connector architecture has a connection object in every Spark Executor, allowing for distributed parallel writes, reads, or scans with MapR-DB tablets.

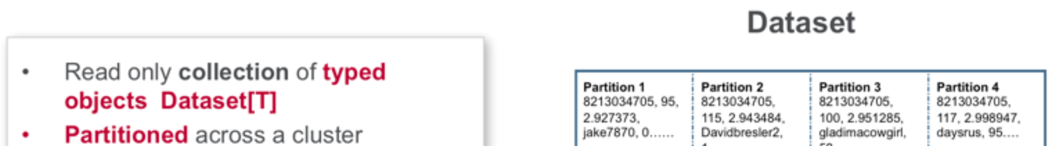


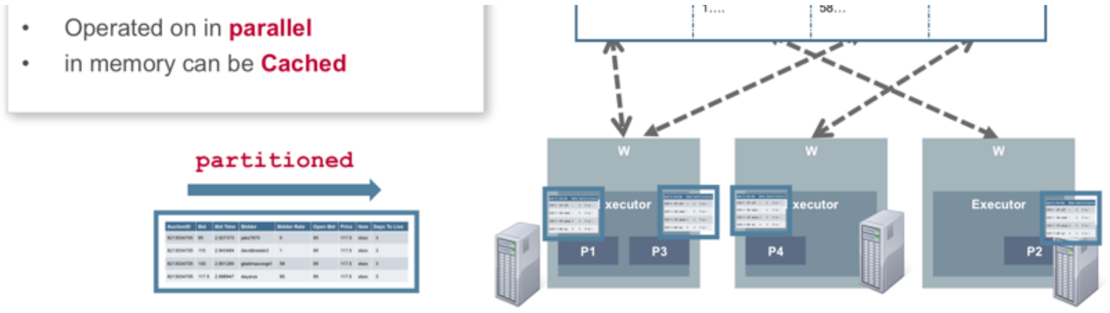
## Querying MapR-DB JSON with Spark SQL

The Spark MapR-DB Connector enables users to perform complex SQL queries and updates on top of MapR-DB using a Spark Dataset, while applying critical techniques such as Projection and filter pushdown, custom partitioning, and data locality.

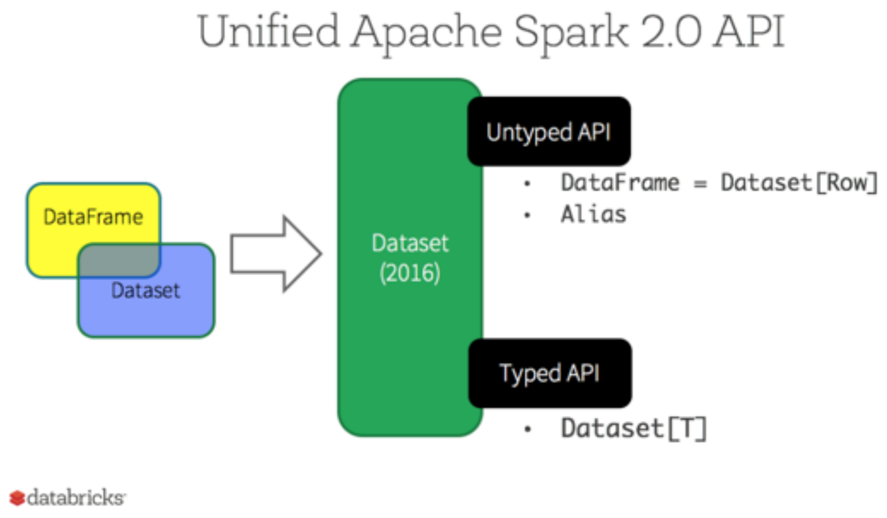

Application
MAPR-DB JSON
MapR-DB OJAI Connector for Apache Spark
Spark
MapR-DB OJAI Connector for Apache Spark
MAPR-DB JSON
Spark SQL
MAPR-FS

A Spark Dataset is a distributed collection of data. Dataset is a newer interface, which provides the benefits of strong typing, the ability to use powerful lambda functions, efficient object serialization/deserialization , combined with the benefits of Spark SQL's optimized execution engine.


• Read only **collection** of **typed objects Dataset[T]**
• **Partitioned** across a cluster
Dataset
Partition 1
8213034705, 95, 2.927373, jake7870, 0......
Partition 2
8213034705, 115, 2.943484, Davidbresler2,
Partition 3
8213034705, 100, 2.951285, gladimacowgirl,
Partition 4
8213034705, 117, 2.998947, daysrus, 95....

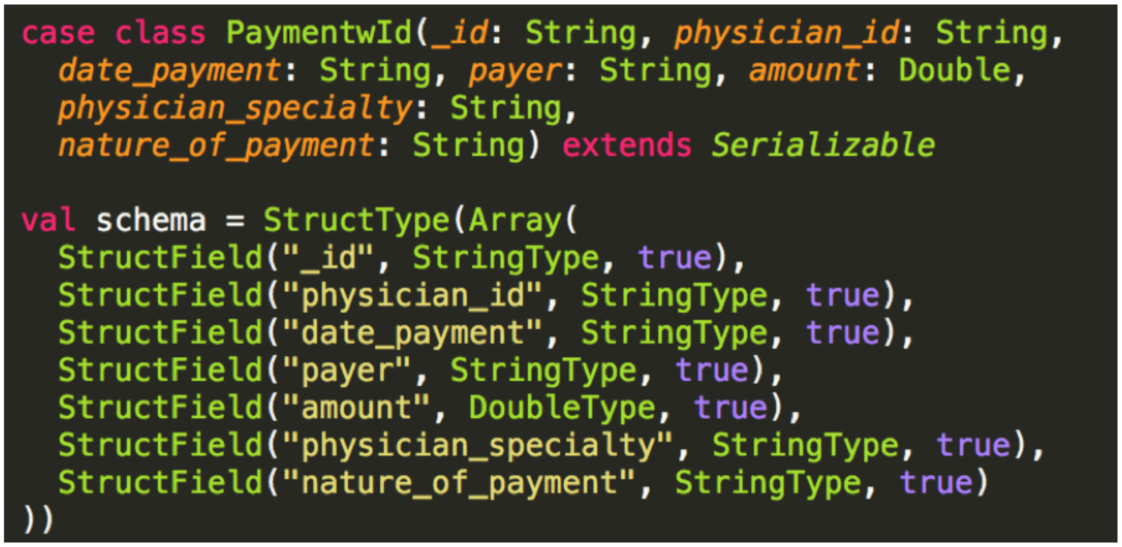


A DataFrame is a Dataset organized into named columns Dataset[Row]. (In Spark 2.0, the DataFrame APIs merged with Datasets APIs.)



## Loading data from MapR-DB into a Spark Dataset

To load data from a MapR-DB JSON table into an Apache Spark Dataset we first define the Scala class and Spark StructType matching the structure of the JSON objects in the MapR-DB table.



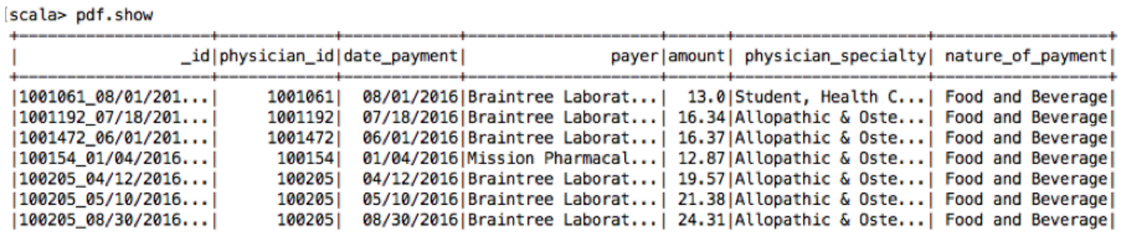
```
case class PaymentwId(_id: String, physician_id: String,
  date_payment: String, payer: String, amount: Double,
  physician_specialty: String,
  nature_of_payment: String) extends Serializable

val schema = StructType(Array(
  StructField("_id", StringType, true),
  StructField("physician_id", StringType, true),
  StructField("date_payment", StringType, true),
  StructField("payer", StringType, true),
  StructField("amount", DoubleType, true),
  StructField("physician_specialty", StringType, true),
  StructField("nature_of_payment", StringType, true)
))
```

Next we invoke the loadFromMapRDB method on a SparkSession object, providing the tableName , schema and case class. This will return a Dataset of PaymentwId objects:

```
val pdf: Dataset[PaymentwId] = spark.sparkSession
  .loadFromMapRDB[PaymentwId](tableName, schema).as[PaymentwId]
```

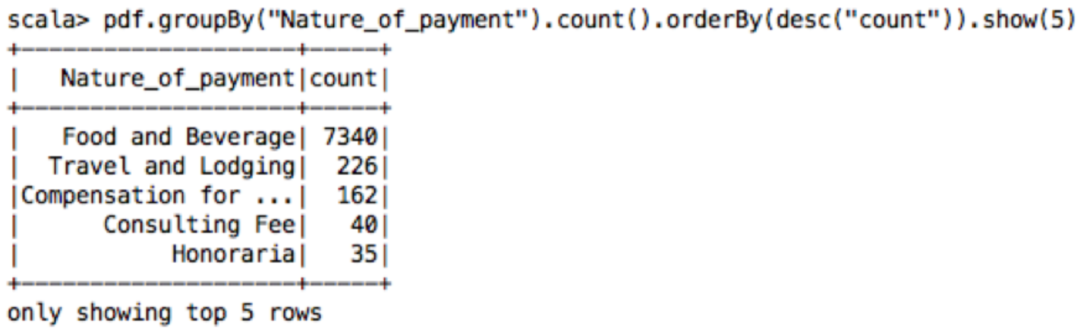
Explore and query the Payment data with Spark SQL

Datasets provide a domain-specific language for structured data manipulation in Scala, Java, and Python. Below are some examples in scala. The Dataset show() action displays the top 20 rows in a tabular form.



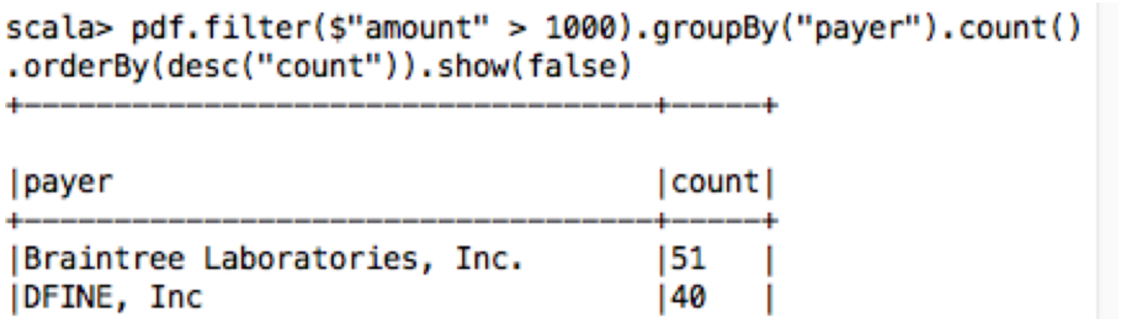
```
scala> pdf.show
+--------------------+------------+------------+--------------------+------+--------------------+------------------+
|                 _id|physician_id|date_payment|               payer|amount| physician_specialty| nature_of_payment|
+--------------------+------------+------------+--------------------+------+--------------------+------------------+
|1001061_08/01/201...|     1001061|  08/01/2016|Braintree Laborat...|  13.0|Student, Health C...| Food and Beverage|
|1001192_07/18/201...|     1001192|  07/18/2016|Braintree Laborat...| 16.34|Allopathic & Oste...| Food and Beverage|
|1001472_06/01/201...|     1001472|  06/01/2016|Braintree Laborat...| 16.37|Allopathic & Oste...| Food and Beverage|
|100154_01/04/2016...|      100154|  01/04/2016|Mission Pharmacal...| 12.87|Allopathic & Oste...| Food and Beverage|
|100205_04/12/2016...|      100205|  04/12/2016|Braintree Laborat...| 19.57|Allopathic & Oste...| Food and Beverage|
|100205_05/10/2016...|      100205|  05/10/2016|Braintree Laborat...| 21.38|Allopathic & Oste...| Food and Beverage|
|100205_08/30/2016...|      100205|  08/30/2016|Braintree Laborat...| 24.31|Allopathic & Oste...| Food and Beverage|
```

What are the top 5 nature of payments by count?

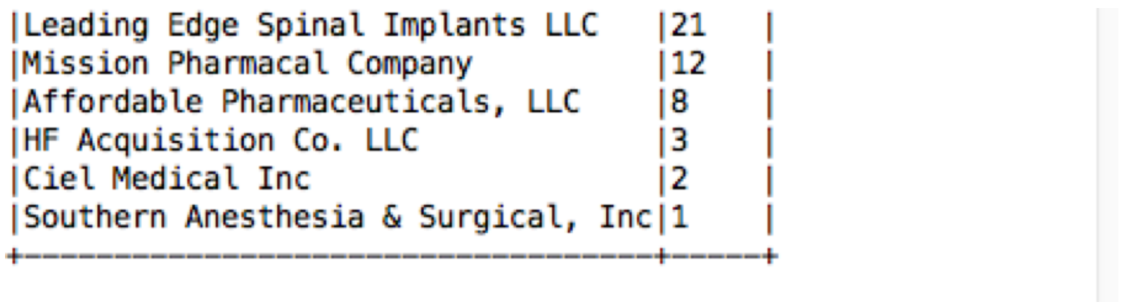


```
scala> pdf.groupBy("Nature_of_payment").count().orderBy(desc("count")).show(5)
+--------------------+-----+
|   Nature_of_payment|count|
+--------------------+-----+
|   Food and Beverage| 7340|
|  Travel and Lodging|  226|
|Compensation for ...|  162|
|      Consulting Fee|   40|
|           Honoraria|   35|
+--------------------+-----+
only showing top 5 rows
```

What is the count of Payers with payment amounts > $1000?



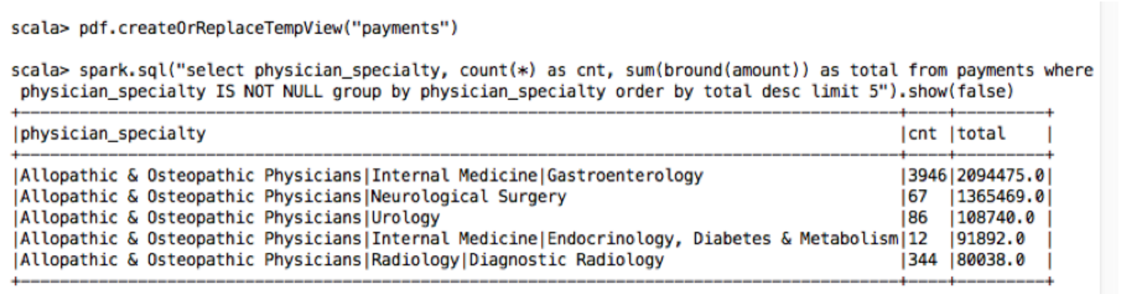
```
scala> pdf.filter($"amount" > 1000).groupBy("payer").count()
.orderBy(desc("count")).show(false)
+------------------------------------+-----+

|payer                               |count|
+------------------------------------+-----+
|Braintree Laboratories, Inc.        |51   |
|DFINE, Inc                          |40   |
```



```
|Leading Edge Spinal Implants LLC   |21   |
|Mission Pharmacal Company          |12   |
|Affordable Pharmaceuticals, LLC    |8    |
|HF Acquisition Co. LLC             |3    |
|Ciel Medical Inc                   |2    |
|Southern Anesthesia & Surgical, Inc|1    |
+-----------------------------------+-----+
```

You can register a Dataset as a temporary table using a given name, and then run Spark SQL. Here are some example Spark SQL queries on the payments dataset:

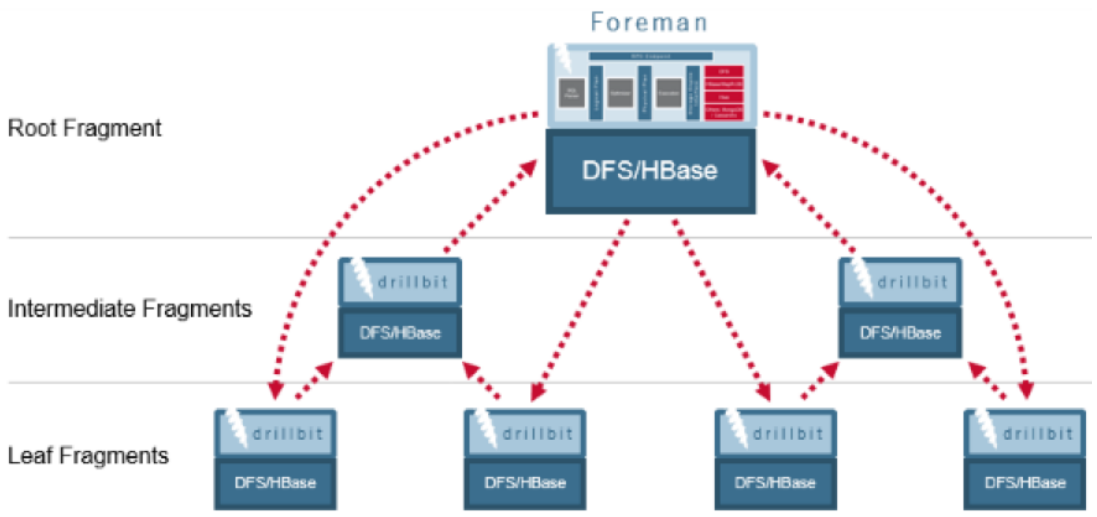
What are the top 5 physician specialties by amount with count?



```
scala> pdf.createOrReplaceTempView("payments")

scala> spark.sql("select physician_specialty, count(*) as cnt, sum(bround(amount)) as total from payments where
 physician_specialty IS NOT NULL group by physician_specialty order by total desc limit 5").show(false)
+----------------------------------------------------------------------------------------+----+---------+
|physician_specialty                                                                     |cnt |total    |
+----------------------------------------------------------------------------------------+----+---------+
|Allopathic & Osteopathic Physicians|Internal Medicine|Gastroenterology                    |3946|2094475.0|
|Allopathic & Osteopathic Physicians|Neurological Surgery                                |67  |1365469.0|
|Allopathic & Osteopathic Physicians|Urology                                             |86  |108740.0 |
|Allopathic & Osteopathic Physicians|Internal Medicine|Endocrinology, Diabetes & Metabolism|12  |91892.0  |
|Allopathic & Osteopathic Physicians|Radiology|Diagnostic Radiology                        |344 |80038.0  |
+----------------------------------------------------------------------------------------+----+---------+
```

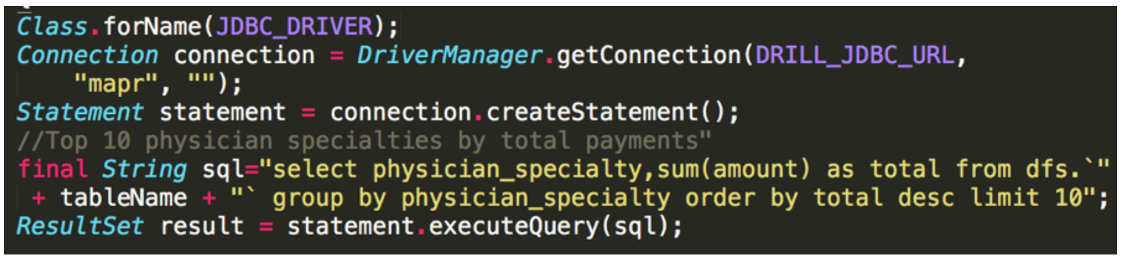
## Querying the Data with Apache Drill

Apache Drill is an open source, low-latency query engine for big data that delivers interactive SQL analytics at petabyte scale. Drill provides a massively parallel processing execution engine, built to perform distributed query processing across the various nodes in a cluster.



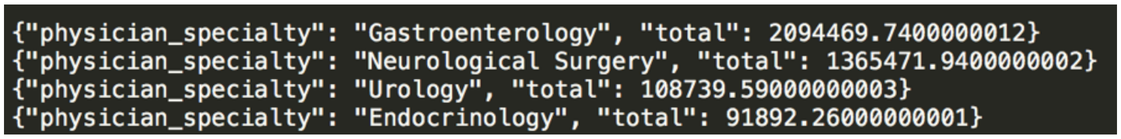
With Drill, you can use SQL to interactively query and join data from files in

JSON, Parquet, or CSV format, Hive, and NoSQL stores, including HBase, MapR-DB, and Mongo, without defining schemas. MapR provides a Drill JDBC driver that you can use to connect Java applications, BI tools, such as SquirreL and Spotfire, to Drill. Below is an snippit of Java code for querying MapR-DB using Drill and JDBC:



```
Class.forName(JDBC_DRIVER);
Connection connection = DriverManager.getConnection(DRILL_JDBC_URL,
    "mapr", "");
Statement statement = connection.createStatement();
//Top 10 physician specialties by total payments"
final String sql="select physician_specialty,sum(amount) as total from dfs.`"
 + tableName + "` group by physician_specialty order by total desc limit 10";
ResultSet result = statement.executeQuery(sql);
```

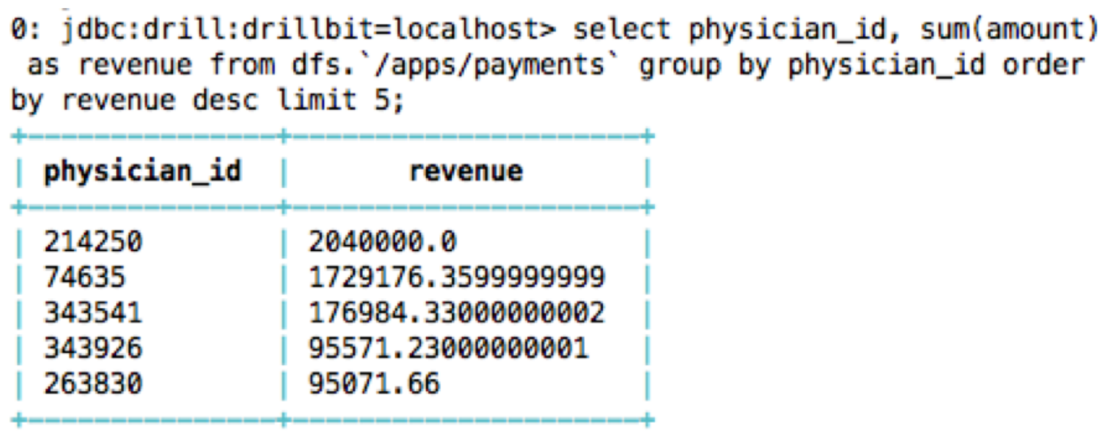
Partial output for this query is shown below:



```
{"physician_specialty": "Gastroenterology", "total": 2094469.7400000012}
{"physician_specialty": "Neurological Surgery", "total": 1365471.9400000002}
{"physician_specialty": "Urology", "total": 108739.59000000003}
{"physician_specialty": "Endocrinology", "total": 91892.26000000001}
```

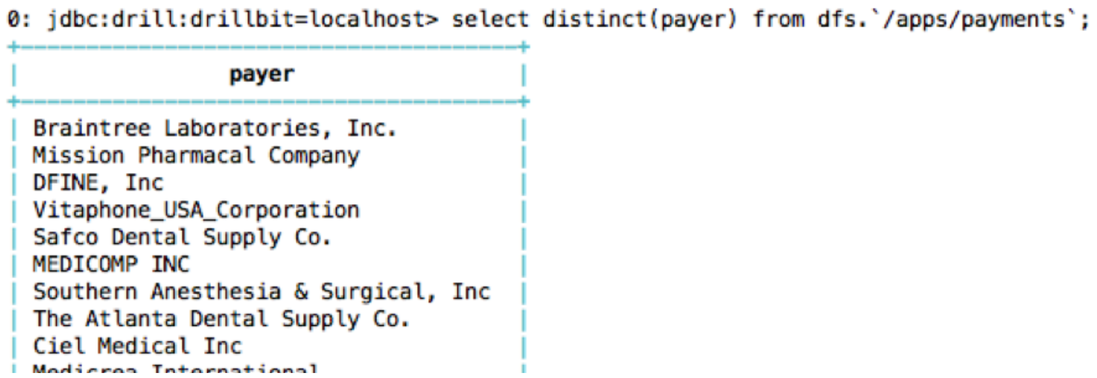
Below are some examples SQL queries using the Drill shell.

What are the top 5 physicians by total amount?

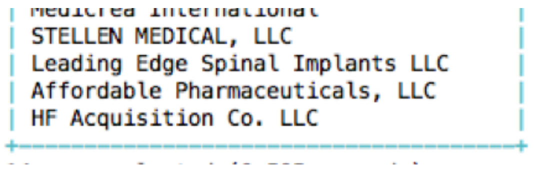


```
0: jdbc:drill:drillbit=localhost> select physician_id, sum(amount)
 as revenue from dfs.`/apps/payments` group by physician_id order
by revenue desc limit 5;
+---------------+-----------------------+
| physician_id  |        revenue        |
+---------------+-----------------------+
| 214250        | 2040000.0             |
| 74635         | 1729176.3599999999    |
| 343541        | 176984.33000000002    |
| 343926        | 95571.23000000001     |
| 263830        | 95071.66              |
+---------------+-----------------------+
```

What are the distinct payers in the Payments table?



```
0: jdbc:drill:drillbit=localhost> select distinct(payer) from dfs.`/apps/payments`;
+------------------------------------------+
|                  payer                   |
+------------------------------------------+
| Braintree Laboratories, Inc.             |
| Mission Pharmacal Company                |
| DFINE, Inc                               |
| Vitaphone_USA_Corporation                |
| Safco Dental Supply Co.                  |
| MEDICOMP INC                             |
| Southern Anesthesia & Surgical, Inc      |
| The Atlanta Dental Supply Co.            |
| Ciel Medical Inc                         |
| Medicrea International                   |
```



```
| Medicrea International               |
| STELLEN MEDICAL, LLC                 |
| Leading Edge Spinal Implants LLC     |
| Affordable Pharmaceuticals, LLC      |
| HF Acquisition Co. LLC               |
+--------------------------------------+
```

Follow the instructions in the github code README to add a secondary index to MapR-DB and try more queries.

## Querying with the Open JSON API (OJAI)

Below is a Java example of using the OJAI Query interface to query documents in a MapR-DB JSON table:

```
// Create an OJAI connection to MapR cluster
Connection connection = DriverManager.getConnection(OJAI_CONNECTION_URL);
// Get an instance of OJAI
DocumentStore store = connection.getStore(tableName);
System.out.println("find payments > $10,000");
Query query = connection.newQuery()
        .select("_id", "nature_of_payment", "amount") // projection
        .where(connection.newCondition().is("amount",
        QueryCondition.Op.GREATER_OR_EQUAL, 10000).build()) // condition
        .build();
DocumentStream stream = store.findQuery(query);
for (Document userDocument : stream) {
    // Print the OJAI Document
    System.out.println("\t" + userDocument.asJsonString());
    counter++;
}
```

Partial output for this query is shown below:

```
{"_id":"343541_02/09/2016_346027328","amount":44710.05,"nature_of_payment":"Royalty or License"}
{"_id":"343541_05/09/2016_346027330","amount":35527.72,"nature_of_payment":"Royalty or License"}
{"_id":"343541_08/08/2016_346027332","amount":44737.41,"nature_of_payment":"Royalty or License"}
{"_id":"343541_10/27/2016_346027334","amount":52009.15,"nature_of_payment":"Royalty or License"}
{"_id":"505383_08/15/2016_346084566","amount":26950.5,"nature_of_payment":"Consulting Fee"}
{"_id":"505383_09/15/2016_346084644","amount":47862.5,"nature_of_payment":"Consulting Fee"}
{"_id":"505383_10/13/2016_346084664","amount":11725,"nature_of_payment":"Consulting Fee"}
```

### Summary

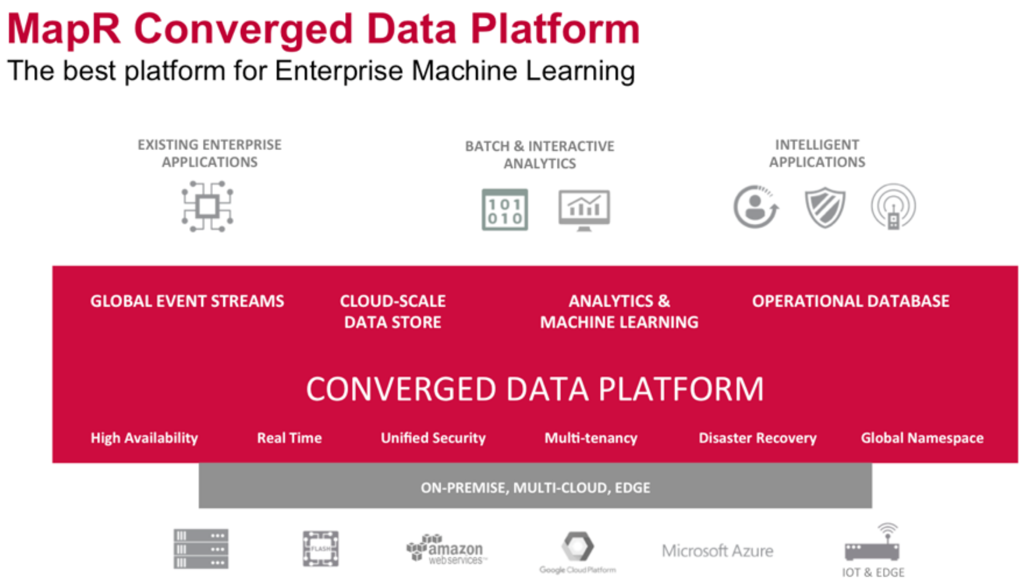
In this blog post, you've learned how to consume streaming Open Payments CSV data, transform to JSON, store in a document database, and explore with SQL using Apache Spark, MapR-ES MapR-DB, OJAI, and Apache Drill

### Code

- You can download the code and data to run these examples from here (refer to the README for complete instructions to run): https://github.com/mapr-demos/mapr-es-db-spark-payment

## Running the Code

All of the components of the use case architecture we just discussed can run on the same cluster with the MapR Converged Data Platform.



This example was developed using the MapR 6.0 container for developers , a docker container that enables you to create a single node MapR cluster. The container is lightweight and designed to run on your laptop. (refer to the code README for instructions on running the code).

You can also look at the following examples:

- mapr-db-60-getting-started to learn Discover how to use DB Shell, Drill and OJAI to query and update documents, but also how to use indexes.
- The MapR Docker Container for Developers
- Youtube video demonstrating how to get started with the MapR 6.0

- Youtube video demonstrating how to get started with the MapR 6.0 developer container
- MapR-ES getting started on MapR 6.0 developer container
- Ojai 2.0 Examples to learn more about OJAI 2.0 features
- MapR-DB Change Data Capture to capture database events such as insert, update, delete and react to this events.

There are several other ways you can get Started with the Converged Data Platform:

- You can download and install MapR on your server easily and quickly with our GUI installer
- MapR can easily be deployed on Azure, Google Cloud Platform, Amazon Web Services as well as on Outscale or CenturyLink.
- MapR Data Science Refinery is an easy-to-deploy and scalable data science toolkit with native access to all platform assets and superior out-of-the-box security.
    - Community for The MapR Data Science Refinery

WANT TO LEARN MORE?

- Spark SQL programming Guide
- Free On Demand Spark Training
- Apache Spark
- MapR-DB Rowkey Design
- MapR-DB OJAI Connector for Apache Spark
- MapR-DB shell
- MapR-DB 6.0 - The Modern Database for Global Data-Intensive Applications
- MapR and Spark
- Integrate Spark with MapR Streams Documentation
- Apache Drill Tutorial
- MapR-DB secondary indexes
- MapR Container for Developers
- Apache Spark Streaming programming guide

**This blog post was published February 27, 2018.**

**Categories**