databricks

# Mastering Apache Spark™ 2.0

Highlights from Databricks Blogs, Spark Summit Talks, and Notebooks

# Mastering Apache Spark™ 2.0

Highlights from Databricks Blogs, Spark Summit Talks, and Notebooks

By Sameer Agarwal, Michael Armbrust, Joseph Bradley, Jules S. Damji, Tathagata Das, Hossein Falaki, Tim Hunter, Davies Liu, Herman von Hovell, Reynold Xin, and Matei Zaharia



4th in a series from Databricks:


Databricks
160 Spear Street, 13th Floor
San Francisco, CA 94105
Contact Us

## About Databricks

Databricks' mission is to accelerate innovation for its customers by unifying Data Science, Engineering and Business. Founded by the team who created Apache Spark™, Databricks provides a Unified Analytics Platform for data science teams to collaborate with data engineering and lines of business to build data products. Users achieve faster time-to-value with Databricks by creating analytic workflows that go from ETL and interactive exploration to production. The company also makes it easier for its users to focus on their data by providing a fully managed, scalable, and secure cloud infrastructure that reduces operational complexity and total cost of ownership. Databricks, venture-backed by Andreessen Horowitz and NEA, has a global customer base that includes CapitalOne, Salesforce, Viacom, Amgen, Shell and HP. For more information, visit www.databricks.com.

# Introduction

Apache Spark 2.0, released in July, was more than just an increase in its numerical notation from 1.x to 2.0: It was a monumental shift in ease of use, higher performance, and smarter unification of APIs across Spark components; and it laid the foundation for a unified API interface for Structured Streaming. It also defined the course for subsequent releases in how these unified APIs across Spark's components will be developed, providing developers expressive ways to write their computations on structured data sets.

Since inception, Databricks' mission has been to make big data simple and accessible for everyone—for organizations of all sizes and across all industries. And we have not deviated from that mission. Over the last couple of years, we have learned how the community of developers use Spark and how organizations use it to build sophisticated applications. We have incorporated, along with the community contributions, much of their requirements in Spark 2.0, focusing on what users love and fixing what users lament.

In this ebook, we curate technical blogs and related assets specific to Spark 2.0, written and presented by leading Spark contributors and members of Spark PMC including Matei Zaharia, the creator of Spark; Reynold Xin, chief architect; Michael Armbrust, lead architect behind Spark SQL; Joseph Bradley and Hossein Falaki, the drivers behind Spark MLlib and SparkR; Tathagata Das, the lead developer for Structured Streaming; Tim Hunter, creator of TensorFrames and contributor for MLlib; and many others.

Collectively, the ebook speaks to the Spark 2.0's three themes—easier, faster, and smarter. Whether you're getting started with Spark or already an accomplished developer, this ebook will arm you with the knowledge to employ all of Spark 2.0's benefits.

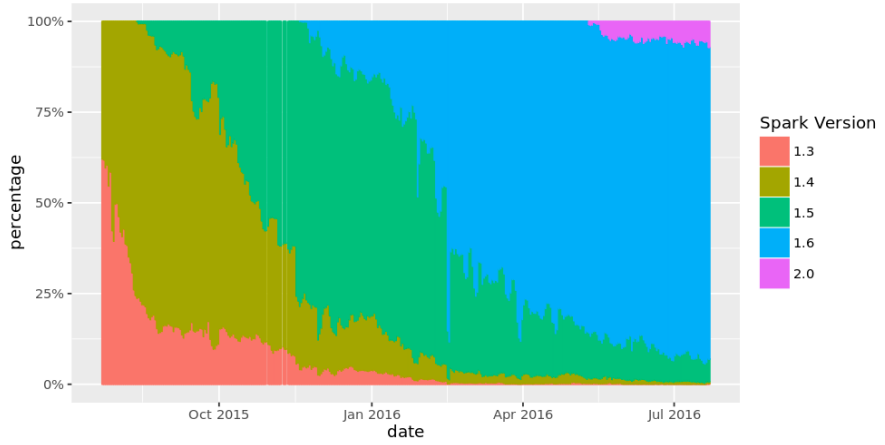
# Section 1:
# An Introduction to Apache Spark 2.0

# Introducing Apache Spark 2.0

July 26, 2016 | by Reynold Xin, Michael Armbrust and Matei Zaharia

Today, we're excited to announce the general availability of Apache Spark 2.0 on Databricks. This release builds on what the community has learned in the past two years, doubling down on what users love and fixing the pain points. This post summarizes the three major themes—easier, faster, and smarter—that comprise Spark 2.0. We also explore many of them in more detail in our anthology of Spark 2.0 content.

Two months ago, we launched a preview release of Apache Spark 2.0 on Databricks. As you can see in the chart below, 10% of our clusters are already using this release, as customers experiment with the new features and give us feedback. Thanks to this experience, we are excited to be the first commercial vendor to support Spark 2.0.




*Apache Spark Usage Over Time By Version*

Now, let's dive into what's new in Apache Spark 2.0.

## Easier: ANSI SQL and Streamlined APIs

One thing we are proud of in Spark is APIs that are simple, intuitive, and expressive. Spark 2.0 continues this tradition, focusing on two areas: (1) standard SQL support and (2) unifying DataFrame/Dataset API.

On the SQL side, we have significantly expanded Spark's SQL support, with the introduction of a new ANSI SQL parser and subqueries. **Spark 2.0 can run all the 99 TPC-DS queries, which require many of the SQL:2003 features.** Because SQL has been one of the primary interfaces to Spark, these extended capabilities drastically reduce the effort of porting legacy applications.

On the programmatic API side, we have streamlined Spark's APIs:

- **Unifying DataFrames and Datasets in Scala/Java:** Starting in Spark 2.0, DataFrame is just a type alias for Dataset of Row. Both the typed methods (e.g. `map, filter, groupByKey`) and the untyped methods (e.g. `select, groupBy`) are available on the Dataset class. Also, this new combined Dataset interface is the abstraction used for Structured Streaming. Since compile-time type-safety is not a feature in Python

and R, the concept of Dataset does not apply to these language APIs. Instead, DataFrame remains the primary interface there, and is analogous to the single-node data frame notion in these languages. Get a peek from this notebook and this blog for the stories behind these APIs.

- **SparkSession:** a new entry point that supersedes SQLContext and HiveContext. For users of the DataFrame API, a common source of confusion for Spark is which "context" to use. Now you can use SparkSession, which subsumes both, as a single entry point, as demonstrated in this notebook. Note that the old SQLContext and HiveContext classes are still kept for backward compatibility.

- **Simpler, more performant Accumulator API:** We have designed a new Accumulator API that has a simpler type hierarchy and support specialization for primitive types. The old Accumulator API has been deprecated but retained for backward compatibility.

- **DataFrame-based Machine Learning API emerges as the primary ML API:** With Spark 2.0, the spark.ml package, with its "pipeline" APIs, will emerge as the primary machine learning API. While the original spark.mllib package is preserved, future development will focus on the DataFrame-based API.

- **Machine learning pipeline persistence:** Users can now save and load machine learning pipelines and models across all programming languages supported by Spark. See this blog post for more details and this notebook for examples.

- **Distributed algorithms in R:** Added support for Generalized Linear Models (GLM), Naive Bayes, Survival Regression, and K-Means in R.

- **User-defined functions (UDFs) in R:** Added support for running partition level UDFs (dapply and gapply) and hyper-parameter tuning (lapply).

## Faster: Apache Spark as a Compiler

According to our 2015 Spark Survey, 91% of users consider performance as the most important aspect of Apache Spark. As a result, performance optimizations have always been a focus in our Spark development. Before we started planning our contributions to Spark 2.0, we asked ourselves a question: **Spark is already pretty fast, but can we push the boundary and make Spark 10X faster?**

This question led us to fundamentally rethink the way we build Spark's physical execution layer. When you look into a modern data engine (e.g. Spark or other MPP databases), majority of the CPU cycles are spent in useless work, such as making virtual function calls or reading/writing intermediate data to CPU cache or memory. Optimizing performance by reducing the amount of CPU cycles wasted in these useless work has been a long time focus of modern compilers.

Spark 2.0 ships with the second generation Tungsten engine. **This engine builds upon ideas from modern compilers and MPP databases and applies them to Spark workloads.** The main idea is to emit optimized code at runtime that collapses the entire query into a single function,

eliminating virtual function calls and leveraging CPU registers for intermediate data. We call this technique "whole-stage code generation."

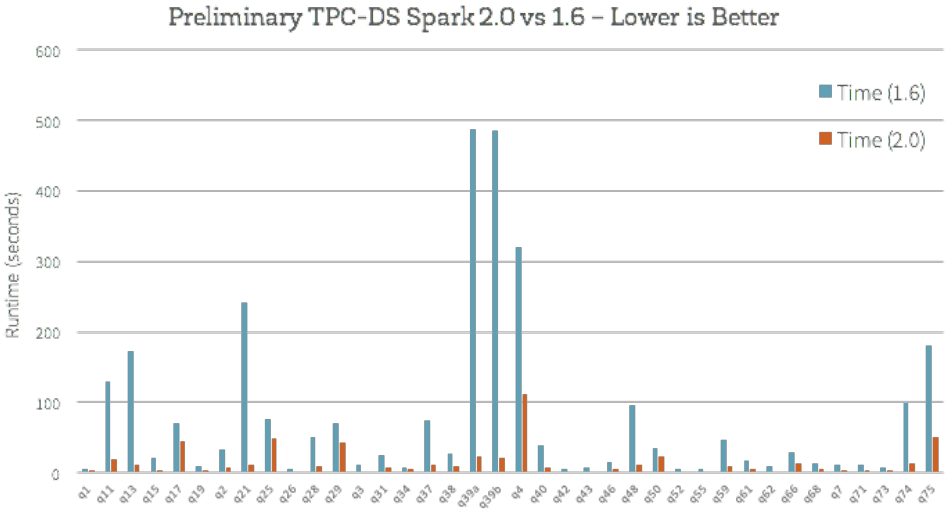
To give you a teaser, we have measured the time (in nanoseconds) it takes to process a row on one core for some of the operators in Spark 1.6 vs. Spark 2.0. The table below shows the improvements in Spark 2.0. Spark 1.6 also included an expression code generation technique that is used in some state-of-the-art commercial databases, but as you can see, many operators became an order of magnitude faster with whole-stage code generation.

You can see the power of whole-stage code generation in action in this notebook, in which we perform aggregations and joins on 1 billion records on a single machine.

***Cost Per Row (in nanoseconds, single thread)***

| primitive | Spark 1.6 | Spark 2.0 |
|---|---|---|
| filter | 15ns | 1.1ns |
| sum w/o group | 14ns | 0.9ns |
| sum w/ group | 79ns | 10.7ns |
| hash join | 115ns | 4.0ns |
| sort (8-bit entropy) | 620ns | 5.3ns |
| sort (64-bit entropy) | 620ns | 40ns |
| sort-merge join | 750ns | 700ns |

How does this new engine work on end-to-end queries? We did some preliminary analysis using TPC-DS queries to compare Spark 1.6 and Spark 2.0:




Beyond whole-stage code generation to improve performance, a lot of work has also gone into improving the Catalyst optimizer for general query optimizations such as nullability propagation, as well as a new vectorized Parquet decoder that improved Parquet scan throughput by 3X. Read this blog post for more detail on the optimizations in Spark 2.0.

## Smarter: Structured Streaming

Spark Streaming has long led the big data space as one of the first systems unifying batch and streaming computation. When its streaming API, called DStreams, was introduced in Spark 0.7, it offered developers with several powerful properties: exactly-once semantics, fault-tolerance at scale, strong consistency guarantees and high throughput.

However, after working with hundreds of real-world deployments of Spark Streaming, we found that applications that need to make decisions in real-time often require **more than just a streaming engine**. They require deep integration of the batch stack and the streaming stack, interaction with external storage systems, as well as the ability to cope with changes in business logic. As a result, enterprises want more than just a streaming engine; instead they need a full stack that enables them to develop end-to-end **"continuous applications."**

Spark 2.0 tackles these use cases through a new API called Structured Streaming. Compared to existing streaming systems, Structured Streaming makes three key improvements:

1. **Integrated API with batch jobs.** To run a streaming computation, developers simply write a batch computation against the DataFrame / Dataset API, and Spark automatically incrementalizes the computation to run it in a streaming fashion (i.e. update the result as data comes in). This powerful design means that developers don't have to manually manage state, failures, or keep the application in sync with batch jobs. Instead, the streaming job always gives the same answer as a batch job on the same data.

2. **Transactional interaction with storage systems.** Structured Streaming handles fault tolerance and consistency holistically across the engine and storage systems, making it easy to write applications that update a live database used for serving, join in static data, or move data reliably between storage systems.

3. **Rich integration with the rest of Spark.** Structured Streaming supports interactive queries on streaming data through Spark SQL, joins against static data, and many libraries that already use DataFrames, letting developers build complete applications instead of just streaming pipelines. In the future, expect more integrations with MLlib and other libraries.

Spark 2.0 ships with an initial, alpha version of Structured Streaming, as a (surprisingly small!) extension to the DataFrame/Dataset API. This makes it easy to adopt for existing Spark users that want to answer new questions in real-time. Other key features include support for event-time based processing, out-of-order/delayed data, interactive queries, and interaction with non-streaming data sources and sinks.

We also updated the Databricks workspace to support Structured Streaming. For example, when launching a streaming query, the notebook UI will automatically display its status.


Streaming is clearly a broad topic, so stay tuned for a series of blog posts with more details on Structured Streaming in Apache Spark 2.0.

## Conclusion

Spark users initially came to Apache Spark for its ease-of-use and performance. Spark 2.0 doubles down on these while extending it to support an even wider range of workloads. Enjoy the new release on Databricks.

## Read More

You can also import the following notebooks and try them on Databricks Community Edition with Spark 2.0.

SparkSession: A new entry point

Datasets: A more streamlined API

Performance of whole-stage code generation

Machine learning pipeline persistence

# Apache Spark as a Compiler: Joining a Billion Rows on your Laptop

May 23, 2016 | by Sameer Agarwal, Davies Liu and Reynold Xin

*Try this notebook in Databricks*

When our team at Databricks planned our contributions to the upcoming Apache Spark 2.0 release, we set out with an ambitious goal by asking ourselves: **Apache Spark is already pretty fast, but can we make it 10x faster?**

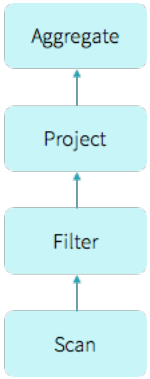
This question led us to fundamentally rethink the way we built Spark's physical execution layer. When you look into a modern data engine (e.g. Spark or other MPP databases), a majority of the CPU cycles are spent in useless work, such as making virtual function calls or reading or writing intermediate data to CPU cache or memory. Optimizing performance by reducing the amount of CPU cycles wasted in this useless work has been a long-time focus of modern compilers.

Apache Spark 2.0 will ship with the second generation Tungsten engine. Built upon ideas from modern compilers and MPP databases and applied to data processing queries, Tungsten emits (SPARK-12795) optimized bytecode at runtime that collapses the entire query into a single function, eliminating virtual function calls and leveraging CPU registers for intermediate data. As a result of this streamlined strategy, called "whole-stage code generation," we significantly improve CPU efficiency and gain performance.

## The Past: Volcano Iterator Model

Before we dive into the details of whole-stage code generation, let us revisit how Spark (and most database systems) work currently. Let us illustrate this with a simple query that scans a single table and counts the number of elements with a given attribute value:

```
select count(*) from store_sales
where ss_item_sk = 1000
```




To evaluate this query, older versions (1.x) of Spark leveraged a popular classic query evaluation strategy based on an iterator model (commonly referred to as the Volcano model). In this model, a query consists of multiple operators, and each operator presents an interface, `next()`, that returns a tuple at a time to the next operator in the tree. For instance, the Filter operator in the above query roughly translates into the code below:

```
class Filter(child: Operator, predicate: (Row => Boolean))
 extends Operator {
 def next(): Row = {
   var current = child.next()
   while (current == null || predicate(current)) {
     current = child.next()
   }
   return current
 }
}
```

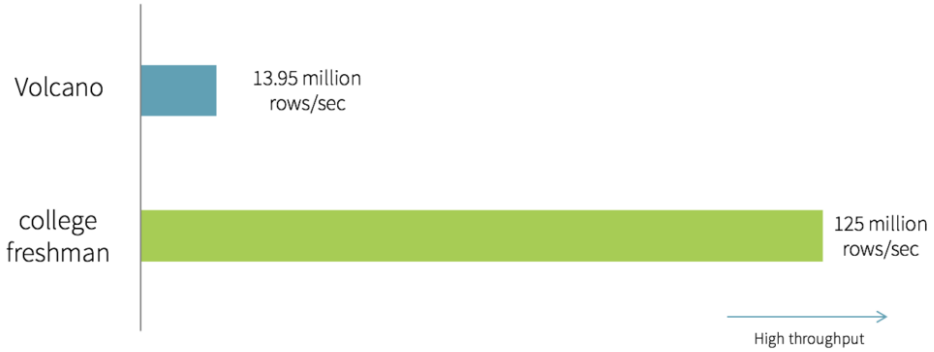
Having each operator implement an iterator interface allowed query execution engines to elegantly compose arbitrary combinations of operators without having to worry about what opaque data type each operator provides. As a result, the Volcano model became the standard for database systems in the last two decades, and is also the architecture used in Spark.

## Volcano vs Hand-written Code

To digress a little, what if we ask a college freshman and give her 10 minutes to implement the above query in Java? It's quite likely she'd come up with iterative code that loops over the input, evaluates the predicate and counts the rows:

```
var count = 0
for (ss_item_sk in store_sales) {
 if (ss_item_sk == 1000) {
   count += 1
 }
}
```

The above code was written specifically to answer a given query, and is obviously not "composable." But how would the two—Volcano generated and hand-written code—compare in performance? On one side, we have the architecture chosen for composability by Spark and majority of the database systems. On the other, we have a simple program written by a novice in 10 minutes. We ran a simple benchmark that compared the "college freshman" version of the program and a Spark program executing the above query using a single thread against Parquet data on disk:




As you can see, the "college freshman" hand-written version is an order of magnitude faster than the Volcano model. It turns out that the 6 lines of Java code are optimized, for the following reasons:

1. **No virtual function dispatches:** In the Volcano model, to process a tuple would require calling the `next()` function at least once. These function calls are implemented by the compiler as virtual function dispatches (via vtable). The hand-written code, on the other hand,

does not have a single function call. Although virtual function dispatching has been an area of focused optimization in modern computer architecture, it still costs multiple CPU instructions and can be quite slow, especially when dispatching billions of times.

2. **Intermediate data in memory vs CPU registers:** In the Volcano model, each time an operator passes a tuple to another operator, it requires putting the tuple in memory (function call stack). In the hand-written version, by contrast, the compiler (JVM JIT in this case) actually places the intermediate data in CPU registers. Again, the number of cycles it takes the CPU to access data in memory is orders of magnitude larger than in registers.

3. **Loop unrolling and SIMD:** Modern compilers and CPUs are incredibly efficient when compiling and executing simple for loops. Compilers can often unroll simple loops automatically, and even generate SIMD instructions to process multiple tuples per CPU instruction. CPUs include features such as pipelining, prefetching, and instruction reordering that make executing simple loops efficient. These compilers and CPUs, however, are not great with optimizing complex function call graphs, which the Volcano model relies on.

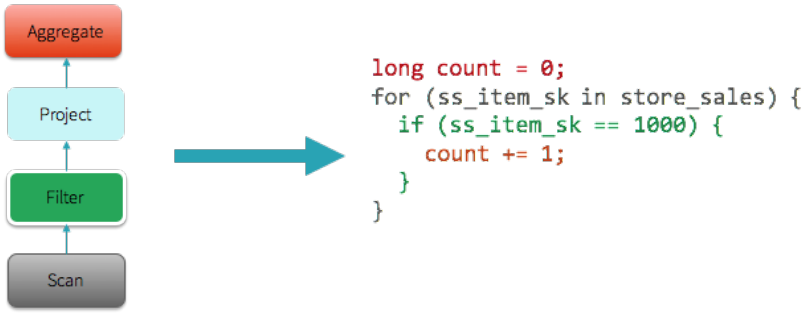
The key take-away here is that the **hand-written code is written specifically to run that query and nothing else, and as a result it can take advantage of all the information that is known,** leading to optimized code that eliminates virtual function dispatches, keeps intermediate data in CPU registers, and can be optimized by the underlying hardware.

## The Future: Whole-stage Code Generation

From the above observation, a natural next step for us was to explore the possibility of automatically generating this *handwritten* code at runtime, which we are calling "whole-stage code generation." This idea is inspired by Thomas Neumann's seminal VLDB 2011 paper on Efficiently Compiling *Efficient Query Plans for Modern Hardware*. For more details on the paper, Adrian Colyer has coordinated with us to publish a review on The Morning Paper blog today.

The goal is to leverage whole-stage code generation so **the engine can achieve the performance of hand-written code, yet provide the functionality of a general purpose engine.** Rather than relying on operators for processing data at runtime, these operators together generate code at runtime and collapse each fragment of the query, where possible, into a single function and execute that generated code instead.

For instance, in the query above, the entire query is a single stage, and Spark would generate the the following JVM bytecode (in the form of Java code illustrated here). More complicated queries would result in multiple stages and thus multiple different functions generated by Spark.




The `explain()` function in the expression below has been extended for whole-stage code generation. In the explain output, when an operator has a star around it (*), whole-stage code generation is enabled. In the following case, Range, Filter, and the two Aggregates are both running with whole-stage code generation. Exchange, however, does not implement whole-stage code generation because it is sending data across the network.

```
spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()
== Physical Plan ==
*Aggregate(functions=[sum(id#201L)])
+- Exchange SinglePartition, None
  +- *Aggregate(functions=[sum(id#201L)])
     +- *Filter (id#201L > 100)
        +- *Range 0, 1, 3, 1000, [id#201L]
```

Those of you that have been following Spark's development closely might ask the following question: "I've heard about code generation since Apache Spark 1.1 in this blog post. How is it different this time?" In the past, similar to other MPP query engines, Spark only applied code generation to expression evaluation and was limited to a small number of operators (e.g. Project, Filter). That is, code generation in the past only sped up the evaluation of expressions such as "1 + a", whereas today whole-stage code generation actually generates code for the entire query plan.

## Vectorization

Whole-stage code-generation techniques work particularly well for a large spectrum of queries that perform simple, predictable operations over large datasets. There are, however, cases where it is infeasible to generate code to fuse the entire query into a single function. Operations might be too complex (e.g. CSV parsing or Parquet decoding), or there might be cases when we're integrating with third party components that can't integrate their code into our generated code (examples can range from calling out to Python/R to offloading computation to the GPU).

To improve performance in these cases, we employ another technique called "vectorization." The idea here is that instead of processing data one row at a time, the engine batches multiples rows together in a columnar format, and each operator uses simple loops to iterate over data within a batch. Each `next()` call would thus return a batch of tuples, amortizing the cost of virtual function dispatches. These simple loops would also enable compilers and CPUs to execute more efficiently with the benefits mentioned earlier.

As an example, for a table with three columns (id, name, score), the following illustrates the memory layout in row-oriented format and column-oriented format.



This style of processing, invented by columnar database systems such as MonetDB and C-Store, would achieve two of the three points mentioned earlier (almost no virtual function dispatches and automatic loop unrolling/SIMD). It, however, still requires putting intermediate data in-memory rather than keeping them in CPU registers. As a result, we use vectorization only when it is not possible to do whole-stage code generation.

For example, we have implemented a new vectorized Parquet reader that does decompression and decoding in column batches. When decoding integer columns (on disk), this new reader is roughly 9 times faster than the non-vectorized one:

In the future, we plan to use vectorization in more code paths such as UDF support in Python/R.

## Performance Benchmarks

We have measured the amount of time (in nanoseconds) it would take to process a tuple on one core for some of the operators in Apache Spark 1.6 vs. Apache Spark 2.0, and the table below is a comparison that demonstrates the power of the new Tungsten engine. Spark 1.6 includes expression code generation technique that is also in use in some state-of-the-art commercial databases today.

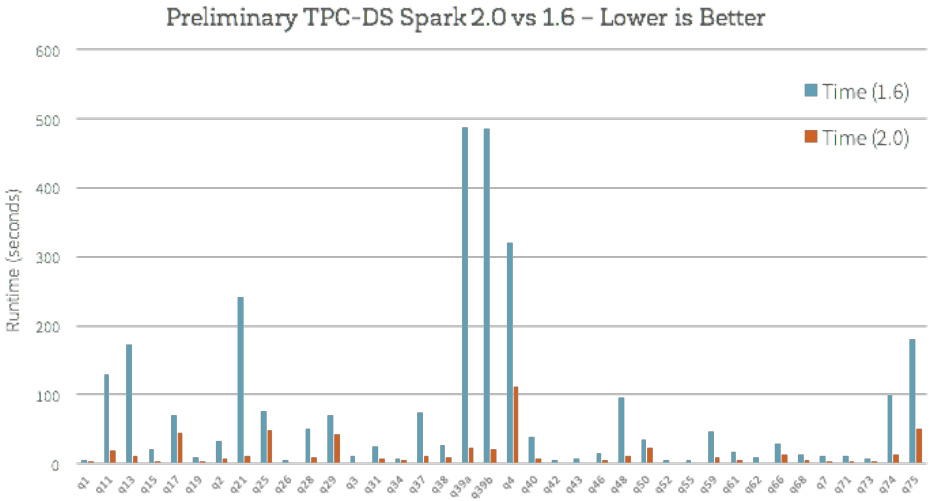
***Cost Per Row (in nanoseconds, single thread)***

| primitive | Spark 1.6 | Spark 2.0 |
|---|---|---|
| filter | 15ns | 1.1ns |
| sum w/o group | 14ns | 0.9ns |
| sum w/ group | 79ns | 10.7ns |
| hash join | 115ns | 4.0ns |
| sort (8-bit entropy) | 620ns | 5.3ns |
| sort (64-bit entropy) | 620ns | 40ns |
| sort-merge join | 750ns | 700ns |
| Parquet decoding (single int column) | 120 ns | 13 ns |

We have surveyed our customers' workloads and implemented whole-stage code generation for the most frequently used operators, such as filter, aggregate, and hash joins. As you can see, many of the core operators are an order of magnitude faster with whole-stage code generation. Some operators such as sort-merge join, however, are inherently slower and more difficult to optimize.

You can see the power of whole-stage code generation in action in this notebook, in which we perform aggregations and joins on 1 billion records on a single machine. It takes less than one second to perform the hash join operation on 1 billion tuples on both the Databricks platform (with Intel Haswell processor 3 cores) as well as on a 2013 Macbook Pro (with mobile Intel Haswell i7).

How does this new engine work on end-to-end queries? Beyond whole-stage code generation and vectorization, a lot of work has also gone into improving the Catalyst optimizer for general query optimizations such as nullability propagation. We did some preliminary analysis using TPC-DS queries to compare Spark 1.6 and the upcoming Spark 2.0:




Does this mean your workload will magically become ten times faster once you upgrade to Spark 2.0? Not necessarily. While we believe the new Tungsten engine implements the best architecture for performance engineering in data processing, it is important to understand that not all workloads can benefit to the same degree. For example, variable-length data types such as strings are naturally more expensive to operate on, and some workloads are bounded by other factors ranging from I/O

throughput to metadata operations. Workloads that were previously bounded by CPU efficiency would observe the largest gains, and shift towards more I/O bound, whereas workloads that were previously I/O bound are less likely to observe gains.

## Conclusion

Most of the work described in this blog post has been committed into Apache Spark's code base and is slotted for the upcoming Spark 2.0 release. The JIRA ticket for whole-stage code generation can be found in SPARK-12795, while the ticket for vectorization can be found in SPARK-12992.

To recap, this blog post described the second generation Tungsten execution engine. Through a technique called whole-stage code generation, the engine will (1) eliminate virtual function dispatches (2) move intermediate data from memory to CPU registers and (3) exploit modern CPU features through loop unrolling and SIMD. Through a technique called vectorization, the engine will also speed up operations that are too complex for code generation. For many core operators in data processing, the new engine is orders of magnitude faster. In the future, given the efficiency of the execution engine, bulk of our performance work will shift towards optimizing I/O efficiency and better query planning.

We are excited about the progress made, and hope you will enjoy the improvements. To try some of these out for free, sign up for an account on Databricks Community Edition.

## Further Reading

- Watch Webinar: Apache Spark 2.0: Easier, Faster, and Smarter
- Technical Preview of Apache Spark 2.0 Now on Databricks
- Approximate Algorithms in Apache Spark: HyperLogLog and Quantiles

# Approximate Algorithms in Apache Spark: HyperLogLog Quantiles

May 19, 2016 | by Tim Hunter, Hossein Falaki and Joseph Bradley

*Try this notebook in Databricks*

## Introduction

Apache Spark is fast, but applications such as preliminary data exploration need to be even faster and are willing to sacrifice some accuracy for a faster result. Since version 1.6, Spark implements approximate algorithms for some common tasks: counting the number of distinct elements in a set, finding if an element belongs to a set, computing some basic statistical information for a large set of numbers. Eugene Zhulenev, from Collective, has already blogged in these pages about the use of approximate counting in the advertising business.

The following algorithms have been implemented against DataFrames and Datasets and committed into Apache Spark's branch-2.0, so they will be available in Apache Spark 2.0 for Python, R, and Scala:

- **approxCountDistinct:** returns an estimate of the number of distinct elements
- **approxQuantile:** returns approximate percentiles of numerical data

Researchers have looked at such algorithms for a long time. Spark strives at implementing approximate algorithms that are deterministic (they do not depend on random numbers to work) and that have proven theoretical error bounds: for each algorithm, the user can specify a target error bound, and the result is guaranteed to be within this bound, either exactly (deterministic error bounds) or with very high confidence (probabilistic error bounds). Also, it is important that this algorithm works well for the wealth of use cases seen in the Spark community.

In this blog, we are going to present details on the implementation of **approxCountDistinct** and **approxQuantile** algorithms and showcase its implementation in a Databricks notebook.

## Approximate count of distinct elements

In ancient times, imagine Cyrus the Great, emperor of Persia and Babylon, having just completed a census of all his empire, fancied to know how many different first names were used throughout his empire, and he put his vizier to the task. The vizier knew that his lord was impatient and wanted an answer fast, even if just an approximate.

There was an issue, though; some names such as Darius, Atusa or Ardumanish were very popular and appeared often on the census

records. Simply counting how many people were living within the empire would give a poor answer, and the emperor would not be fooled.

However, the vizier had some modern and profound knowledge of mathematics. He assembled all the servants of the palace, and said: "Servants, each of you will take a clay tablet from the census record. For each first name that is inscribed on the tablet, you will take the first 3 letters of the name, called l1, l2 and l3, and compute the following number:

```
N = l1 + 31 * l2 + 961 * l3
```

For example, for Darius (D = 3, A = 0, R = 17), you will get N = 16340.

This will give you a number for each name of the tablet. For each number, you will count the number of zeros that end this number. In the case of Hossein (N=17739), this will give you no zero. After each of you does that for each name on his or her tablet, you will convene and you will tell me what is the greatest number of zeros you have observed. Now proceed with great haste and make no calculation mistake, lest you want to endure my wrath!"

At the end of the morning, one servant came back, and said they had found a number with four zeros, and that was the largest they all observed across all the census records. The vizier then announced to his master that he was the master of a population with about 1.3 * 10^4 = 13000 different names. The emperor was highly impressed and he asked the vizier how he had accomplished this feat. To which the vizier uttered one word: "hyper-log-log".

The HyperLogLog algorithm (and its variant HyperLogLog++ implemented in Spark) relies on a clever observation: if the numbers are spread uniformly across a range, then the count of distinct elements can be approximated from the largest number of leading zeros in the binary representation of the numbers. For example, if we observe a number whose digits in binary form are of the form 0…(k times)…01…1, then we can estimate that there are in the order of 2^k elements in the set. This is a very crude estimate but it can be refined to great precision with a sketching algorithm. A thorough explanation of the mechanics behind this algorithm can be found in the original paper.

From the example above with the vizier and his servants, this algorithm does not need to perform shuffling, just map (each servant works on a tablet) and combine (the servants can make pairs and decide which one has the greatest number, until there is only one servant). There is no need move data around, only small statistics about each block of data, which makes it very useful in a large dataset setting such as Spark.

Now, in modern times, how well does this technique work, where datasets are much larger and when servants are replaced with a Spark cluster? We considered a dataset of 25 millions online reviews from an online retail vendor, and we set out to approximate the number of customers behind these reviews. Since customers write multiple reviews, it is a good fit for approximate distinct counting.

Here is how to get an approximate count of users in PySpark, within 1% of the true value and with high probability:

```
# users: DataFrame[user: string]
users.select(approxCountDistinct("user", rsd = 0.01)).show()
```

This plot (fig. 1) shows how the number of distinct customers varies by the error margin. As expected, the answer becomes more and more precise as the requested error margin decreases.

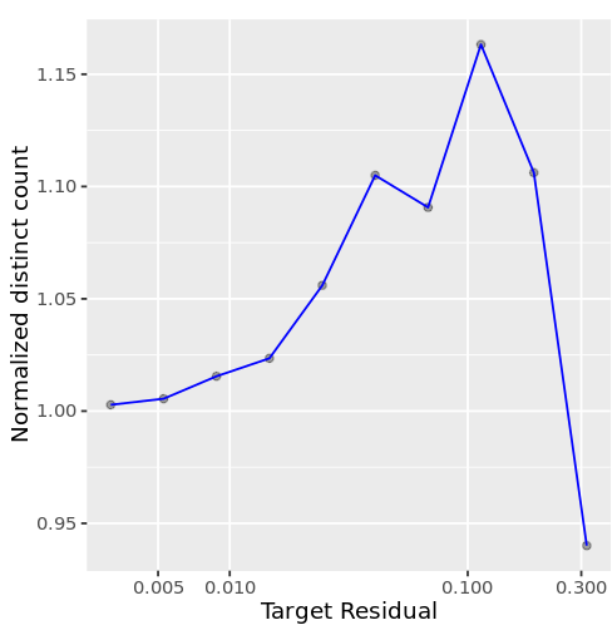



*Figure 1*

How long does it take to compute? For the analysis above, this plot (fig 2.) presents the running time of the approximate counting against the requested precision. For errors above 1%, the running time is just a minute fraction of computing the exact answer. For precise answers, however, the running time increases very fast and it is better to directly compute the exact answer.

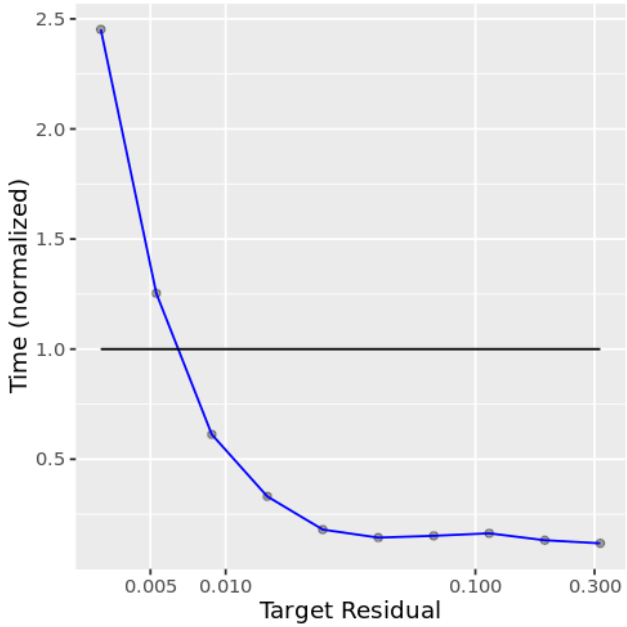



*Figure 2*

As a conclusion, when using approxCountDistinct, you should keep in mind the following:

When the requested error on the result is high (> 1%), approximate distinct counting is very fast and returns results for a fraction of the cost of computing the exact result. In fact, the performance is more or less the same for a target error of 20% or 1%.

For higher precisions, the algorithm hits a wall and starts to take more time than exact counting.

## Approximate quantiles

Quantiles (percentiles) are useful in a lot of contexts. For example, when a web service is performing a large number of requests, it is important to have performance insights such as the latency of the requests. More generally, when faced with a large quantity of numbers, one is often interested in some aggregate information such as the mean, the variance, the min, the max, and the percentiles. Also, it is useful to just have the extreme quantiles: the top 1%, 0.1%, 0.01%, and so on.

Spark implements a robust, well-known algorithm that originated in the streaming database community. Like HyperLogLog, it computes some statistics in each node and then aggregates them on the Spark driver. The current algorithm in Spark can be adjusted to trade accuracy against computation time and memory. Based on the same example as before, we look at the length of the text in each review. Most reviewers express their opinions in a few words, but some customers are prolific writers: the longest review in the dataset is more than 1500 words, while there are several thousand 1-word reviews with various degrees of grammatical freedom.

We plot (fig 3.) here the median length of a review (the 50th percentile) as well as more extreme percentiles. This graph shows that there are few very long reviews and that most of them are below 300 characters.

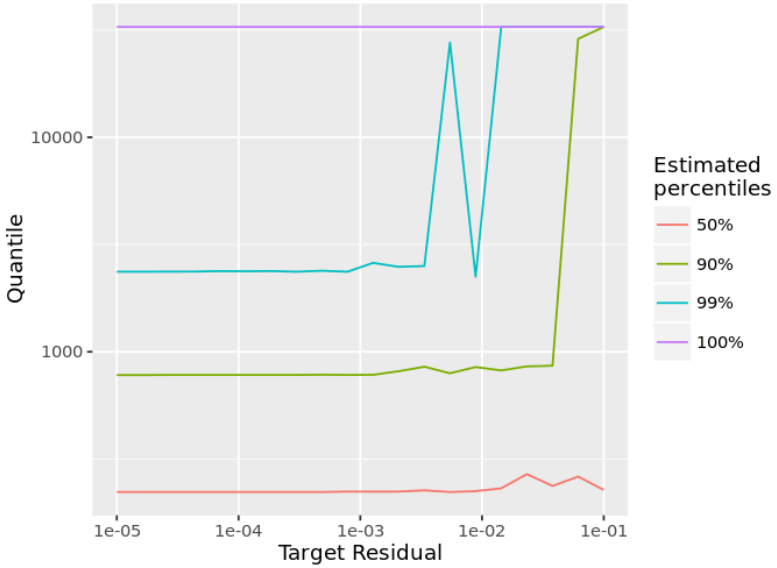



*Figure 3*

The behavior of approximate quantiles is the same as HyperLogLog: when asking for a rough estimate within a few percent of the exact answer, the algorithm is much faster than an exact computation (fig 4.). For a more precise answer, an exact computation is necessary.

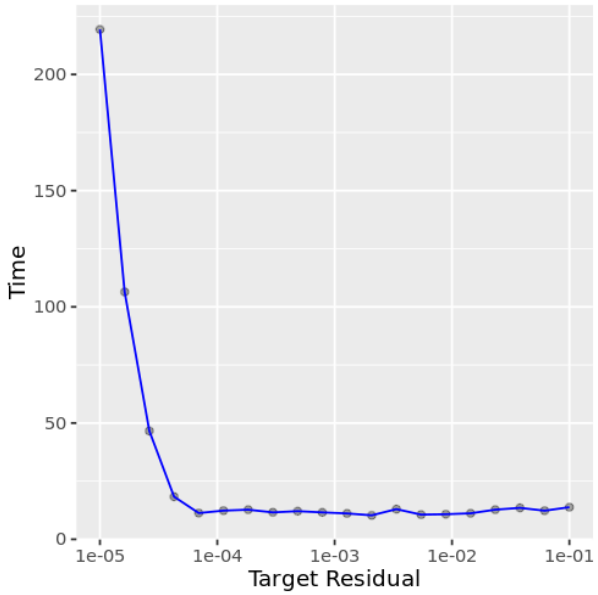



*Figure 4*

## Conclusion

We demonstrated details on the implementation of **approxCountDistinct** and **approxQuantile** algorithms. Though Spark is lighting-fast, sometimes exploratory data applications need even faster results at the expense of sacrificing accuracy. And these two algorithms achieve faster execution.

Apache Spark 2.0 will include some state-of-the art approximation algorithms for even faster results. Users will be able to pick between fast, inexact answers and slower, exact answers. Are there some other approximate algorithms you would like to see? Let us know.

These algorithms are now implemented in a Databricks notebook. To try it out yourself, sign up for an account with Databricks here.

## Further Reading

Interactive Audience Analytics with Spark and HyperLogLog

HyperLogLog: the analysis of the near-optimal cardinality estimation algorithm

Approximate Quantiles in Apache Spark notebook

# Apache Spark 2.0 : Machine Learning Model Persistence

*An ability to save and load models across languages*

May 31, 2016 | By Joseph Bradley

*Try this notebook in Databricks*

## Introduction

Consider these Machine Learning (ML) use cases:

- A data scientist produces an ML model and hands it over to an engineering team for deployment in a production environment.
- A data engineer integrates a model training workflow in Python with a model serving workflow in Java.
- A data scientist creates jobs to train many ML models, to be saved and evaluated later.

All of these use cases are easier with model persistence, the ability to save and load models. With the upcoming release of Apache Spark 2.0, Spark's Machine Learning library MLlib will include near-complete support for ML persistence in the DataFrame-based API. This blog post gives an early overview, code examples, and a few details of MLlib's persistence API.

Key features of ML persistence include:

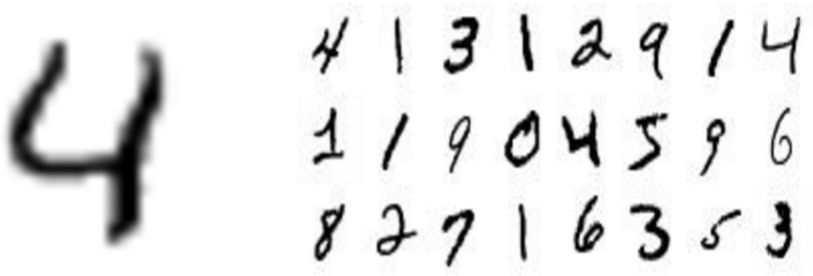
- Support for all language APIs in Spark: Scala, Java, Python & R
- Support for nearly all ML algorithms in the DataFrame-based API
- Support for single models and full Pipelines, both unfitted (a "recipe") and fitted (a result)
- Distributed storage using an exchangeable format

Thanks to all of the community contributors who helped make this big leap forward in MLlib! See the JIRAs for Scala/Java, Python, and R for full lists of contributors.

## Learn the API

In Apache Spark 2.0, the DataFrame-based API for MLlib is taking the front seat for ML on Spark. (See this previous blog post for an introduction to this API and the "Pipelines" concept it introduces.) This DataFrame-based API for MLlib provides functionality for saving and loading models that mimics the familiar Spark Data Source API.

We will demonstrate saving and loading models in several languages using the popular MNIST dataset for handwritten digit recognition (LeCun et al., 1998; available from the LibSVM dataset page). This dataset contains handwritten digits 0–9, plus the ground truth labels. Here are some examples:



Our goal will be to take new images of handwritten digits and identify the digit. See this notebook for the full example code to load this data, fit the models, and save and load them.

## Save & load single models

We first show how to save and load single models to share between languages. We will fit a Random Forest Classifier using Python, save it, and then load the same model back using Scala.

```
training = sqlContext.read...  # data: features, label
rf = RandomForestClassifier(numTrees=20)
model = rf.fit(training)
```

We can simply call the save method to save this model, and the load method to load it right back:

```
model.save("myModelPath")
sameModel = RandomForestClassificationModel.load("myModelPath")
```

We could also load that same model (which we saved in Python) into a Scala or Java application:

```
// Load the model in Scala
val sameModel = RandomForestClassificationModel.load("myModelPath")
```

This works for both small, local models such as K-Means models (for clustering) and large, distributed models such as ALS models (for recommendation). The loaded model has the same parameter settings and data, so it will return the same predictions even if loaded on an entirely different Spark deployment.

## Save & load full Pipelines

So far, we have only looked at saving and loading a single ML model. In practice, ML workflows consist of many stages, from feature extraction and transformation to model fitting and tuning. MLlib provides Pipelines to help users construct these workflows. (See this notebook for a tutorial on ML Pipelines analyzing a bike sharing dataset.)

MLlib allows users to save and load entire Pipelines. Let's look at how this is done on an example Pipeline with these steps:

- Feature extraction: Binarizer to convert images to black and white
- Model fitting: Random Forest Classifier to take images and predict digits 0–9
- Tuning: Cross-Validation to tune the depth of the trees in the forest

Here is a snippet from our notebook to build this Pipeline:

```
// Construct the Pipeline: Binarizer + Random Forest
val pipeline = new Pipeline().setStages(Array(binarizer, rf))
// Wrap the Pipeline in CrossValidator to do model tuning.
val cv = new CrossValidator().setEstimator(pipeline) …
```

Before we fit this Pipeline, we will show that we can save entire workflows (before fitting). This workflow could be loaded later to run on another dataset, on another Spark cluster, etc.

```
cv.save("myCVPath")
val sameCV = CrossValidator.load("myCVPath")
```

Finally, we can fit the Pipeline, save it, and load it back later. This saves the feature extraction step, the Random Forest model tuned by Cross-Validation, and the statistics from model tuning.

```
val cvModel = cv.fit(training)
cvModel.save("myCVModelPath")
val sameCVModel =
CrossValidatorModel.load("myCVModelPath")
```

# Learn the details

## Python tuning

The one missing item in Spark 2.0 is Python tuning. Python does not yet support saving and loading CrossValidator and TrainValidationSplit, which are used to tune model hyperparameters; this issue is targeted for Spark 2.1 (SPARK-13786). However, it is still possible to save the results from CrossValidator and TrainValidationSplit from Python. For example, let's use Cross-Validation to tune a Random Forest and then save the best model found during tuning.

```
# Define the workflow
rf = RandomForestClassifier()
cv = CrossValidator(estimator=rf, ...)
# Fit the model, running Cross-Validation
cvModel = cv.fit(trainingData)
# Extract the results, i.e., the best Random Forest model
bestModel = cvModel.bestModel
# Save the RandomForest model
bestModel.save("rfModelPath")
```

See the notebook for the full code.

## Exchangeable storage format

Internally, we save the model metadata and parameters as JSON and the data as Parquet. These storage formats are exchangeable and can be read using other libraries. Parquet allows us to store both small models (such as Naive Bayes for classification) and large, distributed models (such as ALS for recommendation). The storage path can be any URI

supported by Dataset/DataFrame save and load, including paths to S3, local storage, etc.

### Language cross-compatibility

Models can be easily saved and loaded across Scala, Java, and Python. R has two limitations. First, not all MLlib models are supported from R, so not all models trained in other languages can be loaded into R. Second, the current R model format stores extra data specific to R, making it a bit hacky to use other languages to load models trained and saved in R. (See the accompanying notebook for the hack.) Better cross-language support for R will be added in the near future.

## Conclusion

With the upcoming 2.0 release, the DataFrame-based MLlib API will provide near-complete coverage for persisting models and Pipelines. Persistence is critical for sharing models between teams, creating multi-language ML workflows, and moving models to production. This feature was a final piece in preparing the DataFrame-based MLlib API to become the primary API for Machine Learning in Apache Spark.

## What's next?

High-priority items include complete persistence coverage, including Python model tuning algorithms, as well as improved compatibility between R and the other language APIs.

**Get started** with this tutorial notebook in Scala and Python. You can also just update your current MLlib workflows to use save and load.

**Experiment with** this API using an Apache Spark branch-2.0 preview in Databricks Community Edition.

## Read More

- Read the notebook with the full code referenced in this blog post.
- Learn about the DataFrame-based API for MLlib & ML Pipelines:
  - Notebook introducing ML Pipelines: tutorial analyzing a bike sharing dataset
  - Original blog post on ML Pipelines

# SQL Subqueries in Apache Spark 2.0

*Hands-on examples of scalar and predicate type of subqueries*

June 17, 2016 | By Davies Liu and Herman van Hovell

*Try this notebook in Databricks*

---

In the upcoming Apache Spark 2.0 release, we have substantially expanded the SQL standard capabilities. In this brief blog post, we will introduce subqueries in Apache Spark 2.0, including their limitations, potential pitfalls and future expansions, and through a notebook, we will explore both the scalar and predicate type of subqueries, with short examples that you can try yourself.

A subquery is a query that is nested inside of another query. A subquery as a source (inside a `SQL FROM` clause) is technically also a subquery, but it is beyond the scope of this post. There are basically two kinds of subqueries: scalar and predicate subqueries. And within scalar and predicate queries, there are uncorrelated scalar and correlated scalar queries and nested predicate queries respectively.

For brevity, we will let you jump and explore the notebook, which is more an interactive experience rather than an exposition here in the blog. Click on this diagram below to view and explore the subquery notebook with Apache Spark 2.0 on Databricks.

```
> %sql
  SELECT  A.dep_id,
          A.employee_id,
          A.age,
          B.max_age
  FROM    employee A
          LEFT OUTER JOIN (SELECT   dep_id,
                                    MAX(age) max_age
                           FROM     employee B
                           GROUP BY dep_id) B
            ON B.dep_id = A.dep_id
  ORDER BY 1,2
```

| dep_id | employee_id | age | max_age |
|---|---|---|---|
| 0 | 6 | 28 | 59 |
| 0 | 7 | 52 | 59 |
| 0 | 8 | 59 | 59 |
| 1 | 0 | 55 | 55 |
| 1 | 1 | 42 | 55 |
| 1 | 2 | 50 | 55 |
| 1 | 3 | 27 | 55 |
| 1 | 4 | 29 | 55 |
| 2 | 5 | 56 | 56 |
| 2 | 9 | 36 | 56 |

## What's Next

Subquery support in Apache Spark 2.0 provides a solid solution for the most common subquery usage scenarios. However, there is room for improvement in the areas noted in detail at the end of the notebook.

To try this notebook on Databricks, sign up now.

# Section 2: Unification of APIs and Structuring Spark: Spark Sessions, DataFrames, Datasets and Streaming

# Structuring Spark: DataFrames, Datasets, and Streaming

June 7, 2016 | by Michael Armbrust

As Spark becomes more widely adopted, we have focused on creating higher-level APIs that provide increased opportunities for automatic optimization. In this Spark Summit talk, Armbrust gives an overview of some of the exciting new API's available in Spark 2.0, namely Datasets and Streaming DataFrames/Datasets. Datasets provide an evolution of the RDD API by allowing users to express computation as type-safe lambda functions on domain objects, while still leveraging the powerful optimizations supplied by the Catalyst optimizer and Tungsten execution engine.


He describes the high-level concepts as well as dive into the details of the internal code generation that enable us to provide good performance automatically. Streaming DataFrames/Datasets let developers seamlessly turn their existing structured pipelines into real-time incremental processing engines. He demonstrates this new API's capabilities and discusses future directions including easy sessionization and event-time-based windowing.

Finally, Michael also convinces us why structuring Spark facilitates these high-level, expressive APIs.

# A Tale of Three Apache Spark APIs: RDDs, DataFrames and Datasets

July 14, 2016 | by Jules S. Damji


*Try this notebook in Databricks*

Of all the developers' delight, a set of APIs that makes them productive, that are easy to use, and that are intuitive and expressive is the most attractive delight. One of Apache Spark's appeal to developers has been its easy-to-use APIs, for operating on large datasets, across languages: Scala, Java, Python, and R.

In this blog, I explore three sets of APIs—RDDs, DataFrames, and Datasets—available in Apache Spark 2.0; why and when you should use each set; outline their performance and optimization benefits; and enumerate scenarios when to use DataFrames and Datasets instead of RDDs. Mostly, I will focus on DataFrames and Datasets, because in Apache Spark 2.0, these two APIs are unified.

Our primary motivation behind this unification is our quest to simplify Spark by limiting the number of concepts that you have to learn and by offering ways to process structured data. And through structure, Spark can offer higher-level abstraction and APIs as domain specific language constructs.

## Resilient Distributed Dataset (RDD)

RDD was the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

## When to use RDDs?

Consider these scenarios or common use cases for using RDDs when:

- you want low-level transformation and actions and control on your dataset;
- your data is unstructured, such as media streams or streams of text;
- you want to manipulate your data with functional programming constructs than domain specific expressions;
- you don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
- you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

## What happens to RDDs in Apache Spark 2.0?

You may ask: Are RDDs being relegated as second class citizens? Are they being deprecated?

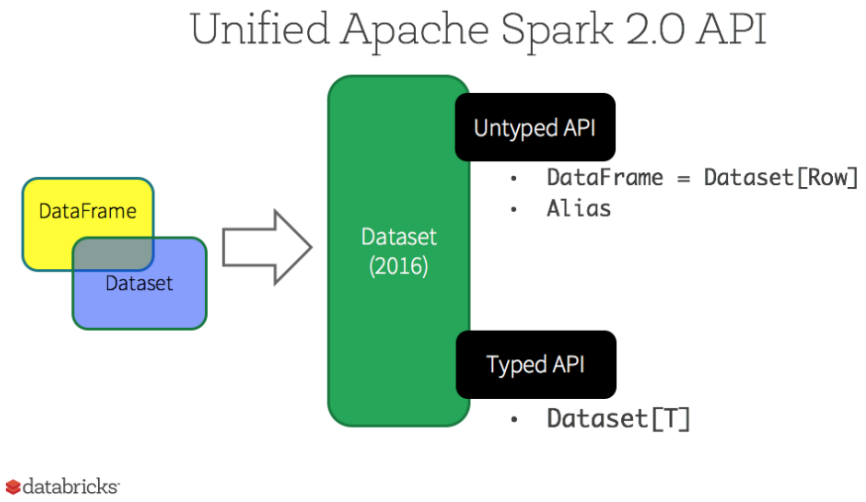
The answer is a resounding **NO!**

What's more, as you will note below, you can seamlessly move between DataFrame or Dataset and RDDs at will—by simple API method calls—and DataFrames and Datasets are built on top of RDDs.

## DataFrames

Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction; it provides a domain specific language API to manipulate your distributed data; and makes Spark accessible to a wider audience, beyond specialized data engineers.

In our preview of Apache Spark 2.0 webinar and subsequent blog, we mentioned that in Spark 2.0, DataFrame APIs will merge with Datasets APIs, unifying data processing capabilities across libraries. Because of this unification, developers now have fewer concepts to learn or remember, and work with a single high-level and type-safe API called Dataset.




## Datasets

Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a ***strongly-typed API*** and an untyped API, as shown in the table below. Conceptually, consider DataFrame as an *alias* for a collection of generic objects *Dataset[Row]*, where a *Row* is a generic ***untyped*** JVM object. Dataset, by contrast, is a collection of ***strongly-typed*** JVM objects, dictated by a case class you define in Scala or a class in Java.

## Typed and Un-typed APIs

| Language | Main Abstraction |
|---|---|
| Scala | Dataset[T] & DataFrame (alias for Dataset[Row]) |
| Java | Dataset[T] |
| Python* | DataFrame |
| R* | DataFrame |

***Note:*** *Since Python and R have no compile-time type-safety, we only have untyped APIs, namely DataFrames.*

## Benefits of Dataset APIs

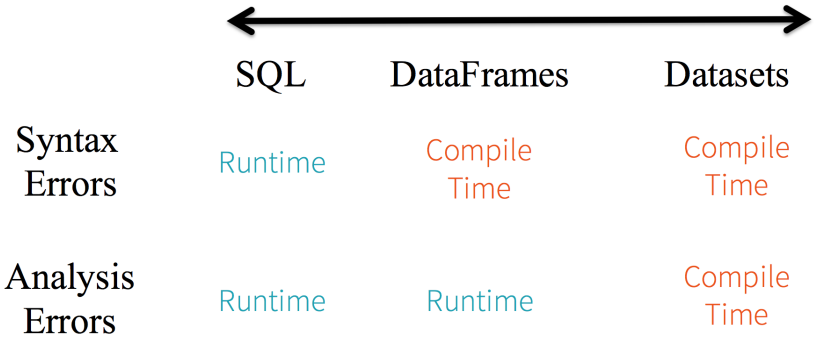
As a Spark developer, you benefit with the DataFrame and Dataset unified APIs in Spark 2.0 in a number of ways.

### 1. Static-typing and runtime type-safety

Consider static-typing and runtime safety as a spectrum, with SQL least restrictive to Dataset most restrictive. For instance, in your Spark SQL string queries, you won't know a syntax error until runtime (which could be costly), whereas in DataFrames and Datasets you can catch errors at compile time (which saves developer-time and costs). That is, if you invoke a function in DataFrame that is not part of the API, the compiler will catch it. However, it won't detect a non-existing column name until runtime.

At the far end of the spectrum is Dataset, most restrictive. Since Dataset APIs are all expressed as lambda functions and JVM typed objects, any mismatch of typed-parameters will be detected at compile time. Also, your analysis error can be detected at compile time too, when using Datasets, hence saving developer-time and costs.

All this translates to is a spectrum of type-safety along syntax and analysis error in your Spark code, with Datasets as most restrictive yet productive for a developer.



## 2. High-level abstraction and custom view into structured and semi-structured data

DataFrames as a collection of Datasets[Row] render a structured custom view into your semi-structured data. For instance, let's say, you have a huge IoT device event dataset, expressed as JSON. Since JSON is a semi-structured format, it lends itself well to employing Dataset as a collection of strongly typed-specific Dataset[DeviceIoTData].

```
{"device_id": 198164, "device_name": "sensor-
pad-198164owomcJZ", "ip": "80.55.20.25", "cca2": "PL",
"cca3": "POL", "cn": "Poland", "latitude": 53.080000,
"longitude": 18.620000, "scale": "Celsius", "temp": 21,
"humidity": 65, "battery_level": 8, "c02_level": 1408,
"lcd": "red", "timestamp" :1458081226051}
```

You could express each JSON entry as DeviceIoTData, a custom object,

```
case class DeviceIoTData (battery_level: Long, c02_level:
Long, cca2: String, cca3: String, cn: String, device_id:
Long, device_name: String, humidity: Long, ip: String,
latitude: Double, lcd: String, longitude: Double,
scale:String, temp: Long, timestamp: Long)
```

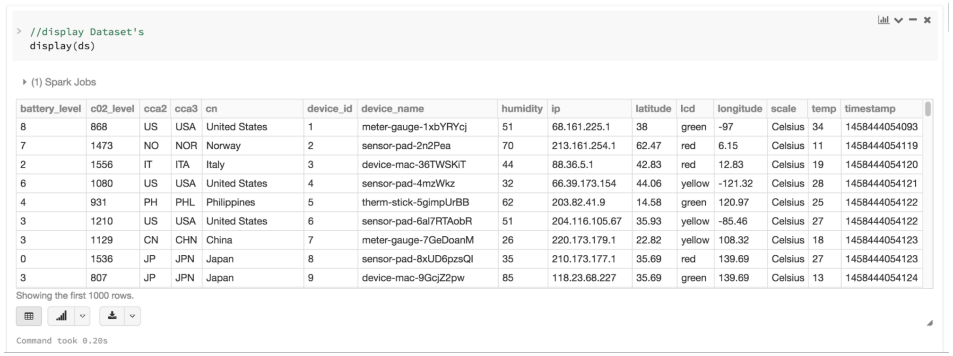
with a Scala case class.

```
// read the json file and create the dataset from the
// case class DeviceIoTData
// ds is now a collection of JVM Scala objects DeviceIoTData
val ds = spark.read.json("/databricks-public-datasets/data/
iot/iot_devices.json").as[DeviceIoTData]
```

Next, we can read the data from a JSON file.

Three things happen here under the hood in the code above:

1. Spark reads the JSON, infers the schema, and creates a collection of DataFrames.
2. At this point, Spark converts your data into *DataFrame = Dataset[Row]*, a collection of generic Row object, since it does not know the exact type.
3. Now, Spark converts the *Dataset[Row] -> Dataset[DeviceIoTData]* **type-specific** Scala JVM object, as dictated by the **class** *DeviceIoTData.*



```
> //display Dataset's
  display(ds)
```

▸ (1) Spark Jobs

| battery_level | c02_level | cca2 | cca3 | cn | device_id | device_name | humidity | ip | latitude | lcd | longitude | scale | temp | timestamp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 868 | US | USA | United States | 1 | meter-gauge-1xbYRYcj | 51 | 68.161.225.1 | 38 | green | -97 | Celsius | 34 | 1458444054093 |
| 7 | 1473 | NO | NOR | Norway | 2 | sensor-pad-2n2Pea | 70 | 213.161.254.1 | 62.47 | red | 6.15 | Celsius | 11 | 1458444054119 |
| 2 | 1556 | IT | ITA | Italy | 3 | device-mac-36TWSKiT | 44 | 88.36.5.1 | 42.83 | red | 12.83 | Celsius | 19 | 1458444054120 |
| 6 | 1080 | US | USA | United States | 4 | sensor-pad-4mzWkz | 32 | 66.39.173.154 | 44.06 | yellow | -121.32 | Celsius | 28 | 1458444054121 |
| 4 | 931 | PH | PHL | Philippines | 5 | therm-stick-5gimpUrBB | 62 | 203.82.41.9 | 14.58 | green | 120.97 | Celsius | 25 | 1458444054122 |
| 3 | 1210 | US | USA | United States | 6 | sensor-pad-6al7RTAobR | 51 | 204.116.105.67 | 35.93 | yellow | -85.46 | Celsius | 27 | 1458444054122 |
| 3 | 1129 | CN | CHN | China | 7 | meter-gauge-7GeDoanM | 26 | 220.173.179.1 | 22.82 | yellow | 108.32 | Celsius | 18 | 1458444054123 |
| 0 | 1536 | JP | JPN | Japan | 8 | sensor-pad-8xUD6pzsQI | 35 | 210.173.177.1 | 35.69 | red | 139.69 | Celsius | 27 | 1458444054123 |
| 3 | 807 | JP | JPN | Japan | 9 | device-mac-9GcjZ2pw | 85 | 118.23.68.227 | 35.69 | green | 139.69 | Celsius | 13 | 1458444054124 |

Showing the first 1000 rows.

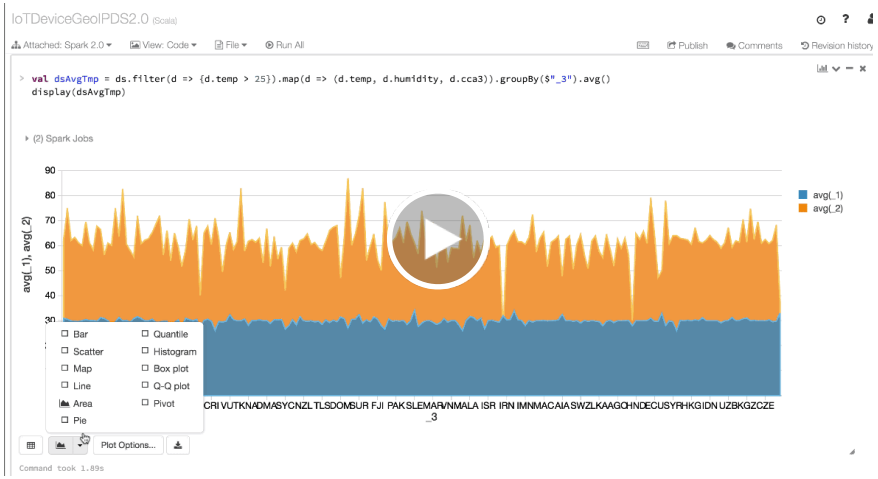
Command took 0.20s

## 3. Ease-of-use of APIs with structure

Although structure may limit control in what your Spark program can do with data, it introduces rich semantics and an easy set of domain specific operations that can be expressed as high-level constructs. Most computations, however, can be accomplished with Dataset's high-level APIs. For example, it's much simpler to perform **`agg, select, sum, avg, map, filter, or groupBy`** operations by accessing a Dataset typed object's *DeviceIoTData* than using RDD rows' data fields.

Expressing your computation in a domain specific API is far simpler and easier than with relation algebra type expressions (in RDDs). For instance, the code below will **`filter()`** and **`map()`** create another immutable Dataset.

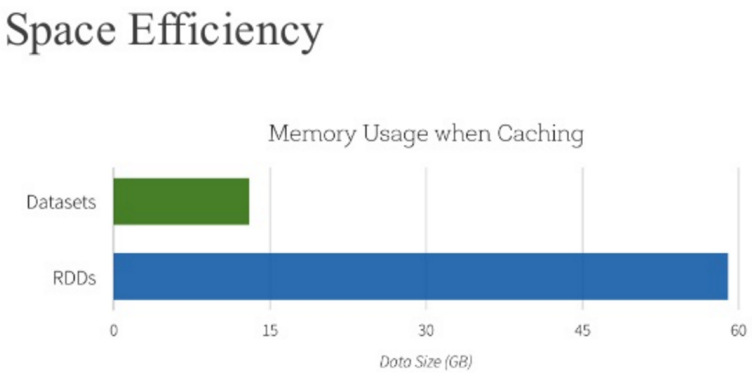
```
// Use filter(), map(), groupBy() country, and compute avg()
// for temperatures and humidity. This operation results in
// another immutable Dataset. The query is simpler to read,
// and expressive

val dsAvgTmp = ds.filter(d => {d.temp > 25}).map(d => (d.temp,
d.humidity, d.cca3)).groupBy($"_3").avg()

//display the resulting dataset
display(dsAvgTmp)
```




## 4. Performance and Optimization

Along with all the above benefits, you cannot overlook the space efficiency and performance gains in using DataFrames and Dataset APIs for two reasons.

First, because DataFrame and Dataset APIs are built on top of the Spark SQL engine, it uses Catalyst to generate an optimized logical and physical query plan. Across R, Java, Scala, or Python DataFrame/Dataset APIs, all relation type queries undergo the same code optimizer, providing the space and speed efficiency. Whereas the Dataset[T] typed API is optimized for data engineering tasks, the untyped Dataset[Row] (an alias of DataFrame) is even faster and suitable for interactive analysis.




Second, since Spark as a compiler understands your Dataset type JVM object, it maps your type-specific JVM object to Tungsten's internal memory representation using Encoders. As a result, Tungsten Encoders can efficiently serialize/deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds.

## When should I use DataFrames or Datasets?

- If you want rich semantics, high-level abstractions, and domain specific APIs, use DataFrame or Dataset.
- If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.
- If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset.
- If you want unification and simplification of APIs across Spark Libraries, use DataFrame or Dataset.
- If you are a R user, use DataFrames.
- If you are a Python user, use DataFrames and resort back to RDDs if you need more control.

Note that you can always seamlessly interoperate or convert from DataFrame and/or Dataset to an RDD, by simple method call `.rdd`. For instance:

```
// select specific fields from the Dataset, apply a predicate
// using the where() method, convert to an RDD, and show first 10
// RDD rows
val deviceEventsDS = ds.select($"device_name", $"cca3",
$"c02_level").where($"c02_level" > 1300)
// convert to RDDs and take the first 10 rows
val eventsRDD = deviceEventsDS.rdd.take(10)
```

```
> val deviceEventsDS = ds.select($"device_name",$"cca3", $"c02_level").where($"c02_level" > 1300)
  // concert to RDDs
  val eventsRDD = deviceEventsDS.rdd.take(10)
```

▸ (1) Spark Jobs

```
deviceEventsDS: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [device_name: string, cca3: string ... 1 more field]
eventsRDD: Array[org.apache.spark.sql.Row] = Array([sensor-pad-2n2Pea,NOR,1473], [device-mac-36TWSKiT,ITA,1556], [sensor-pad-8xUD6pzsQI,JPN,1536], [sens
or-pad-10BsywSYUF,USA,1470], [meter-gauge-11dlMTZty,ITA,1544], [sensor-pad-14QL93sBR0j,NOR,1346], [sensor-pad-16aXmIJZtdO,USA,1425], [meter-gauge-17zb8F
ghhl,USA,1466], [meter-gauge-19eg1BpfCO,USA,1531], [sensor-pad-22oWV2D,JPN,1522])
Command took 0.34s
```

## Bringing It All Together

In summation, the choice of when to use RDD or DataFrame and/or Dataset seems obvious. While the former offers you low-level functionality and control, the latter allows custom view and structure, offers high-level and domain specific operations, saves space, and executes at superior speeds.

As we examined the lessons we learned from early releases of Spark—how to simplify Spark for developers, how to optimize and make it performant—we decided to elevate the low-level RDD APIs to a high-level abstraction as DataFrame and Dataset and to build this unified data abstraction across  libraries atop Catalyst optimizer and Tungsten.

Pick one—DataFrames and/or Dataset or RDDs APIs—that meets your needs and use-case, but I would not be surprised if you fall into the camp of most developers who work with structure and semi-structured data.

## What's Next?

You can try Apache Spark 2.0 on Databricks and run this accompanying notebook. If you haven't signed up yet, try Databricks now.

# How to Use SparkSessions in Apache Spark 2.0: A unified entry point for manipulating data with Spark

August 15, 2016 | By Jules S. Damji

*Try this notebook in Databricks*

Generally, a session is an interaction between two or more entities. In computer parlance, its usage is prominent in the realm of networked computers on the internet. First with TCP session, then with login session, followed by HTTP and user session, so no surprise that we now have *SparkSession*, introduced in Apache Spark 2.0.

Beyond a time-bounded interaction, *SparkSession* provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with DataFrame and Dataset APIs. Most importantly, it curbs the number of concepts and constructs a developer has to juggle while interacting with Spark.

In this blog and its accompanying Databricks notebook, we will explore SparkSession functionality in Spark 2.0.

## Exploring SparkSession's Unified Functionality

First, we will examine a Spark application, SparkSessionZipsExample, that reads zip codes from a JSON file and do some analytics using DataFrames APIs, followed by issuing Spark SQL queries, without accessing SparkContext, SQLContext or HiveContext.

## Creating a SparkSession

In previous versions of Spark, you had to create a SparkConf and SparkContext to interact with Spark, as shown here:

```
//set up the spark configuration and create contexts
val sparkConf = new
SparkConf().setAppName("SparkSessionZipsExample").setMaster("local")
// your handle to SparkContext to access other context like SQLContext
val sc = new SparkContext(sparkConf).set("spark.some.config.option",
"some-value")
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Whereas in Spark 2.0 the same effects can be achieved through SparkSession, without expliciting creating SparkConf, SparkContext or SQLContext, as they're encapsulated within the SparkSession. Using a builder design pattern, it instantiates a SparkSession object if one does not already exist, along with its associated underlying contexts.

// Create a SparkSession. No need to create SparkContext

```
// Create a SparkSession. No need to create SparkContext
// You automatically get it as part of the SparkSession
val warehouseLocation = "file:${system:user.dir}/spark-warehouse"
val spark = SparkSession
   .builder()
   .appName("SparkSessionZipsExample")
   .config("spark.sql.warehouse.dir", warehouseLocation)
   .enableHiveSupport()
   .getOrCreate()
```

At this point you can use the spark variable as your instance object to access its public methods and instances for the duration of your Spark job.

## Configuring Spark's Runtime Properties

Once the SparkSession is instantiated, you can configure Spark's runtime config properties. For example, in this code snippet, we can alter the existing runtime config options. Since configMap is a collection, you can use all of Scala's iterable methods to access the data.

```
//set new runtime options
spark.conf.set("spark.sql.shuffle.partitions", 6)
spark.conf.set("spark.executor.memory", "2g")
//get all settings
val configMap:Map[String, String] = spark.conf.getAll()
```

## Accessing Catalog Metadata

Often, you may want to access and peruse the underlying catalog metadata. SparkSession exposes "catalog" as a public instance that contains methods that work with the metastore (i.e data catalog). Since these methods return a Dataset, you can use Dataset API to access or view data. In this snippet, we access table names and list of databases.

```
//fetch metadata data from the catalog
spark.catalog.listDatabases.show(false)
spark.catalog.listTables.show(false)
```

```
scala> spark.catalog.listTables.show(false)
+------------+--------+-----------+---------+-----------+
|name        |database|description|tableType|isTemporary|
+------------+--------+-----------+---------+-----------+
|hive_numbers|default |null       |MANAGED  |false      |
|numbers     |null    |null       |TEMPORARY|true       |
+------------+--------+-----------+---------+-----------+


scala> spark.catalog.listDatabases.show(false)
+-------+--------------------+----------------------------------------+
|name   |description         |locationUri                             |
+-------+--------------------+----------------------------------------+
|default|Default Hive database|file:/Users/jules/spark2.0/spark-warehouse|
+-------+--------------------+----------------------------------------+
```

*Fig 1. Datasets Returned From Catalog*

# Creating Datasets and Dataframes

There are a number of ways to create DataFrames and Datasets using SparkSession APIs. One quick way to generate a Dataset is by using the spark.range method. When learning to manipulate Dataset with its API, this quick method proves useful. For example:

```
/create a Dataset using spark.range starting from 5 to 100, with
increments of 5
val numDS = spark.range(5, 100, 5)
// reverse the order and display first 5 items
numDS.orderBy(desc("id")).show(5)
//compute descriptive stats and display them
numDs.describe().show()
// create a DataFrame using spark.createDataFrame from a List or Seq
val langPercentDF = spark.createDataFrame(List(("Scala", 35),
("Python", 30), ("R", 15), ("Java", 20)))
//rename the columns
val lpDF = langPercentDF.withColumnRenamed("_1",
"language").withColumnRenamed("_2", "percent")
//order the DataFrame in descending order of percentage
lpDF.orderBy(desc("percent")).show(false)
```

```
scala> numDS.orderBy(desc("id")).show(5)
+---+
| id|
+---+
| 95|
| 90|
| 85|
| 80|
| 75|
+---+
only showing top 5 rows


scala> lpDF.orderBy(desc("percent")).show()
+--------+-------+
|language|percent|
+--------+-------+
|   Scala|     35|
|  Python|     30|
|    Java|     20|
|       R|     15|
+--------+-------+
```

*Fig 2. Dataframe & Dataset Output*

## Reading JSON Data with SparkSession API

Like any Scala object you can use spark, the SparkSession object, to access its public methods and instance fields. I can read JSON or CVS or TXT file, or I can read a parquet table. For example, in this code snippet, we will read a JSON file of zip codes, which returns a DataFrame, a collection of generic Rows.

```
// read the json file and create the dataframe
val jsonFile = args(0)
val zipsDF = spark.read.json(jsonFile)
//filter all cities whose population > 40K
zipsDF.filter(zipsDF.col("pop") > 40000).show(10)
```

## Using Spark SQL with SparkSession

Through SparkSession, you can access all of the Spark SQL functionality as you would through SQLContext. In the code sample below, we create a table against which we issue SQL queries.

```
// Now create an SQL table and issue SQL queries against it without
// using the sqlContext but through the SparkSession object.
// Creates a temporary view of the DataFrame
zipsDF.createOrReplaceTempView("zips_table")
zipsDF.cache()
val resultsDF = spark.sql("SELECT city, pop, state, zip FROM
zips_table")
resultsDF.show(10)
```


*Fig. 3 Partial Output From The Spark Job Run*

## Saving and Reading from Hive table with SparkSession

Next, we are going to create a Hive table and issue queries against it using SparkSession object as you would with a HiveContext.

```
//drop the table if exists to get around existing table error
spark.sql("DROP TABLE IF EXISTS zips_hive_table")
//save as a hive table
spark.table("zips_table").write.saveAsTable("zips_hive_table")
//make a similar query against the hive table
val resultsHiveDF = spark.sql("SELECT city, pop, state, zip FROM
zips_hive_table WHERE pop > 40000")
resultsHiveDF.show(10)
```

*Fig 4. Output From The Hive Table*

As you can observe, the results in the output runs from using the DataFrame API, Spark SQL and Hive queries are identical. You can access all sources and data, and how to run this example, from my github repo.

Second, let's turn our attention to two Spark developer environments where the SparkSession is automatically created for you.

## SparkSession in Spark REPL and Databricks Notebook

First, as in previous versions of Spark, the spark-shell created a SparkContext (sc), so in Spark 2.0, the spark-shell creates a SparkSession (spark). In this spark-shell, you can see spark already exists, and you can view all its attributes.

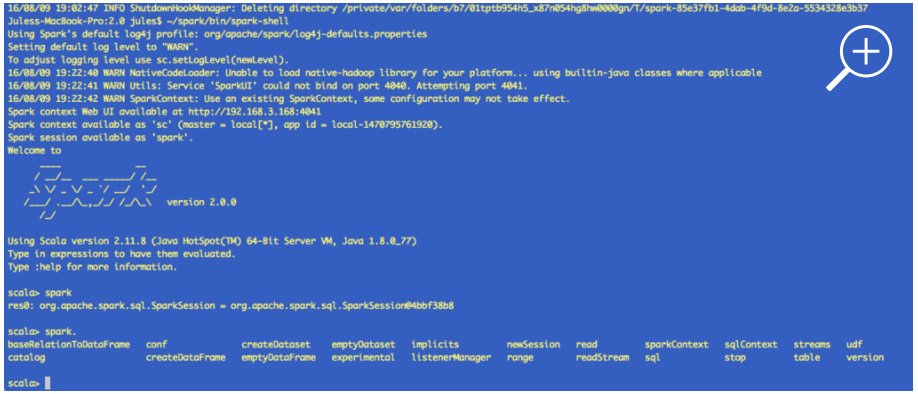


*Fig 5. Sparksession In Spark-Shell*

Second, in the Databricks notebook, when you create a cluster, the SparkSession is created for you. In both cases it's accessible through a variable called spark. And through this variable you can access all its public fields and methods. Rather than repeating the same functionality here, I defer you to examine the notebook, since each section explores SparkSession's functionality—and more.

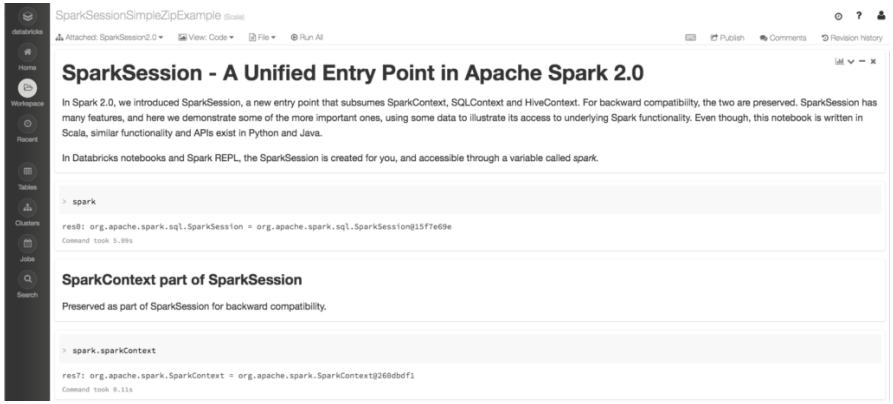



*Fig 6. Sparksession In Databricks Notebook*

You can explore an extended version of the above example in the Databricks notebook SparkSessionZipsExample, by doing some basic analytics on zip code data. Unlike our above Spark application example, we don't create a SparkSession—since one is created for us—yet employ all its exposed Spark functionality. To try this notebook, import it in Databricks.

## SparkSession Encapsulates SparkContext

Lastly, for historical context, let's briefly understand the SparkContext's underlying functionality.

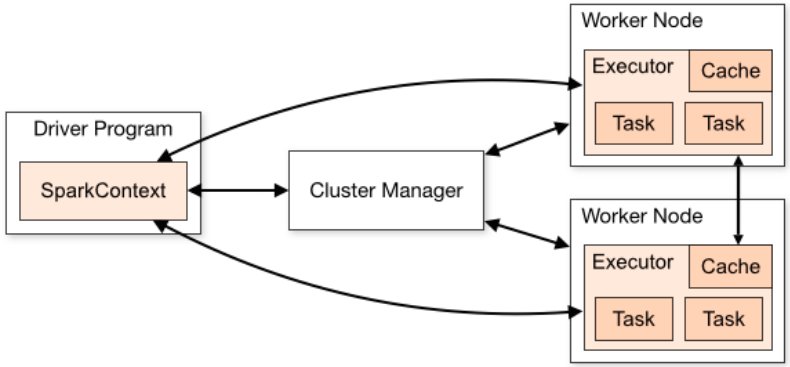



*Fig 7. Sparkcontext As It Relates To Driver And Cluster Manager*

As shown in the diagram, a SparkContext is a conduit to access all Spark functionality; only a single SparkContext exists per JVM. The Spark driver program uses it to connect to the cluster manager to communicate, submit Spark jobs and knows what resource manager (YARN, Mesos or Standalone) to communicate to. It allows you to configure Spark configuration parameters. And through SparkContext, the driver can access other contexts such as SQLContext, HiveContext, and StreamingContext to program Spark.

However, with Spark 2.0, SparkSession can access all aforementioned Spark's functionality through a single-unified point of entry. As well as making it simpler to access DataFrame and Dataset APIs, it also subsumes the underlying contexts to manipulate data.

In summation, what I demonstrated in this blog is that all functionality previously available through SparkContext, SQLContext or HiveContext in early versions of Spark are now available via SparkSession. In essence, SparkSession is a single-unified entry point to manipulate data with Spark, minimizing number of concepts to remember or construct. Hence, if you have fewer programming constructs to juggle, you're more likely to make fewer mistakes and your code is likely to be less cluttered.

## What's Next?

This is the first in the series of how-to blog posts on new features and functionality introduced in Spark 2.0 and how you can use them on the Databricks just-time-data platform. Stay tuned for other how-to blogs in the coming weeks.

- Try the accompanying SparkSessionZipsExample Notebook
- Try the corresponding Spark application on my github repo
- Try an additional SparkSession Notebook
- Import these notebooks today in Databricks for free

# Section 3:
# Evolution of Spark Streaming

# Continuous Applications: Evolving Streaming in Apache Spark 2.0

July 28, 2016 | by Matei Zaharia

Since its release, Spark Streaming has become one of the most widely used distributed streaming engines, thanks to its high-level API and exactly-once semantics. Nonetheless, as these types of engines became common, we've noticed that developers often need more than just a streaming programming model to build real-time applications. At Databricks, we've worked with thousands of users to understand how to simplify real-time applications. In this post, we present the resulting idea, continuous applications, which we have started to implement through the Structured Streaming API in Apache Spark 2.0.

Most streaming engines focus on performing computations on a stream: for example, one can map a stream to run a function on each record, reduce it to aggregate events by time, etc. However, as we worked with users, we found that **virtually no use case of streaming engines only involved performing computations on a stream.** Instead, stream processing happens as part of a larger application, which we'll call a continuous application. Here are some examples:

1. **Updating data that will be served in real-time.** For instance, developers might want to update a summary table that users will query through a web application. In this case, much of the complexity is in the interaction between the streaming engine and the serving system: for example, can you run queries on the table while the streaming engine is updating it? The "complete" application is a real-time serving system, not a map or reduce on a stream.

2. **Extract, transform and load (ETL).** One common use case is continuously moving and transforming data from one storage system to another (e.g. JSON logs to an Apache Hive table). This requires careful interaction with both storage systems to ensure no data is duplicated or lost — much of the logic is in this coordination work.

3. **Creating a real-time version of an existing batch job.** This is hard because many streaming systems don't guarantee their result will match a batch job. For example, we've seen companies that built live dashboards using a streaming engine and daily reporting using batch jobs, only to have customers complain that their daily report (or worse, their bill!) did not match the live metrics.

4. **Online machine learning.** These continuous applications often combine large static datasets, processed using batch jobs, with real-time data and live prediction serving.

These examples show that streaming computations are part of larger applications that include serving, storage, or batch jobs. Unfortunately, in

current systems, streaming computations run on their own, in an engine focused just on streaming. This leaves developers responsible for the complex tasks of interacting with external systems (e.g. managing transactions) and making their result consistent with the the rest of the application (e.g., batch jobs). This is what we'd like to solve with continuous applications.

## Continuous Applications

We define a continuous application as an end-to-end application that reacts to data in real-time. In particular, we'd like developers to use a single programming interface to support the facets of continuous applications that are currently handled in separate systems, such as query serving or interaction with batch jobs. For example, here is how we would handle the use cases above:

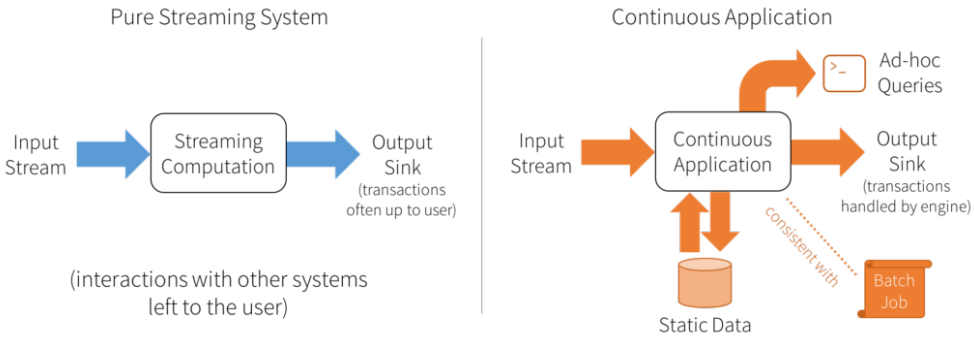
1. **Updating data that will be served in real time.** The developer would write a single Spark application that handles both updates and serving (e.g. through Spark's JDBC server), or would use an API that automatically performs transactional updates on a serving system like MySQL, Redis or Apache Cassandra.

2. **Extract, transform and load (ETL).** The developer would simply list the transformations required as in a batch job, and the streaming system would handle coordination with both storage systems to ensure exactly-once processing.

3. **Creating a real-time version of an existing batch job.** The streaming system would guarantee results are always consistent with a batch job on the same data.

4. **Online machine learning.** The machine learning library would be designed to combine real-time training, periodic batch training, and prediction serving behind the same API.

The figure below shows which concerns are usually handled in streaming engines, and which would be needed in continuous applications:




## Structured Streaming

Structured Streaming is a new high-level API we have contributed to Apache Spark 2.0 to support continuous applications. It is, first, a higher-level API than Spark Streaming, bringing in ideas from the other structured APIs in Spark (DataFrames and Datasets)—most notably, a way to perform database-like query optimizations. More importantly, however, Structured Streaming also incorporates the idea of continuous

applications to provide a number of features that no other streaming engines offer.

1. **Strong guarantees about consistency with batch jobs.** Users specify a streaming computation by writing a batch computation (using Spark's DataFrame/Dataset API), and the engine automatically *incrementalizes* this computation (runs it continuously). At any point, the output of the Structured Streaming job is the same as running the batch job on a prefix of the input data. Most current streaming systems (e.g. Apache Storm, Kafka Streams, Google Dataflow and Apache Flink) **do not** provide this "prefix integrity" property.

2. **Transactional integration with storage systems.** We have taken care in the internal design to process data exactly once and update output sinks transactionally, so that serving applications always see a consistent snapshot of the data. While the Spark 2.0 release only supports a few data sources (HDFS and S3), we plan to add more in future versions. Transactional updates were one of the top pain points for users of Spark and other streaming systems, requiring manual work, so we are excited to make these part of the core API.

3. **Tight integration with the rest of Spark.** Structured Streaming supports serving interactive queries on streaming state with Spark SQL and JDBC, and integrates with MLlib. These integrations are only beginning in Spark 2.0, but will grow in future releases. Because Structured Streaming builds on DataFrames, many other libraries of Spark will naturally run over it (e.g., all feature transformations in MLlib are written against DataFrames).

Apart from these unique characteristics, Structured Streaming has other new features to simplify streaming, such as explicit support for "event time" to aggregate out of order data, and richer support for windowing and sessions. Achieving its consistency semantics in a fault-tolerant manner is also not easy—see our sister blog post about the API and execution model.

Structured Streaming is still in alpha in Spark 2.0, but we hope you try it out and send feedback. Our team and many other community members will be expanding it in the next few releases.

## An Example

As a simple example of Structured Streaming, the code below shows an Extract, Transform and Load (ETL) job that converts data from JSON into Apache Parquet. Note how Structured Streaming simply uses the DataFrame API, so the code is nearly identical to a batch version.

### Streaming Version

```
// Read JSON continuously from S3
logsDF = spark.readStream.json("s3://logs")

// Transform with DataFrame API and save
logsDF.select("user", "url", "date")
      .writeStream.parquet("s3://out")
      .start()
```

### Batch Version

```
// Read JSON once from S3
logsDF = spark.read.json("s3://logs")

// Transform with DataFrame API and save
logsDF.select("user", "url", "date")
      .write.parquet("s3://out")
```

While the code looks deceptively simple, Spark does a lot of work under the hood, such as grouping the data into Parquet partitions, ensuring each record appears in the output exactly once, and recovering the job's state if you restart it. Finally, to serve this data *interactively* instead of writing it to Parquet, we could just change writeStream to use the (currently alpha) in-memory sink and connect a JDBC client to Spark to query it.

## Long-Term Vision

Our long-term vision for streaming in Spark is ambitious: we want every library in *Spark to work in an incremental fashion on Structured Streaming*. Although this is a big goal, Apache Spark is well positioned to achieve it. Its libraries are already built on common, narrow APIs (RDDs and DataFrames), and Structured Streaming is designed explicitly to give results consistent with these unified interfaces.

The biggest insight in Spark since its beginning is that developers need **unified interfaces**. For example, batch computation on clusters used to require many disjoint systems (MapReduce for ETL, Hive for SQL, Giraph for graphs, etc), complicating both development and operations. Spark unified these workloads on one engine, greatly simplifying both tasks. The same insight applies to streaming. Because streaming workloads are usually part of a much larger continuous application, which may include serving, storage, and batch jobs, we want to offer a unified API and system for building end-to-end continuous applications.

# Read More

Our Structured Streaming model blog post explores the streaming API and execution model in more detail. We recommend you read this post to get started with Structured Streaming.

In addition, the following resources cover Structured Streaming:

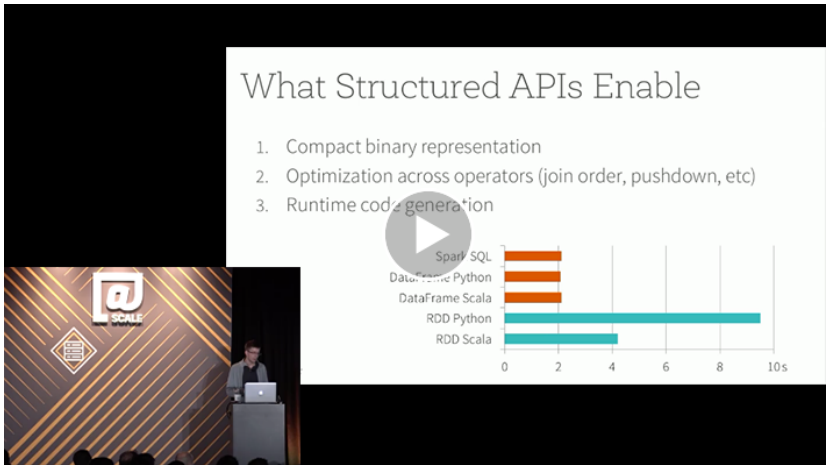
- Spark 2.0 and Structured Streaming
- Future of Real-time Spark
- Structuring Spark: DataFrames, Datasets and Streaming
- A Deep Dive Into Structured Streaming
- Structured Streaming Programming Guide

# Unifying Big Data Workloads in Apache Spark

September 2, 2016 | by Matei Zaharia

In this section, Matei Zaharia laid out the vision where Spark streaming is heading, with continuous applications. Apart from aspiring that all the Spark components work in an incremental fashion as Structured Streaming does, what makes all this possible is a unified interface, built upon DataFrames/DataSets and Spark SQL engine.

Also, what simplifies writing big data applications, Zaharia explains, is this unified interface across big data workloads: such as ETL, Machine Learning, Streaming, Ad-hoc interactive queries and serving or updating dashboards. Hear Zaharia as he makes the case at the @Scale conference.



# Section 4:
# Structured Streaming

# Structured Streaming in Apache Spark 2.0

*A new high-level API for streaming*

July 28, 2016 | Matei Zaharia, Tathagata Das, Michael Armbrust, Reynold Xin

Apache Spark 2.0 adds the first version of a new higher-level API, Structured Streaming, for building continuous applications. The main goal is to make it easier to build end-to-end streaming applications, which integrate with storage, serving systems, and batch jobs in a consistent and fault-tolerant way. In this post, we explain why this is hard to do with current distributed streaming engines, and introduce Structured Streaming.

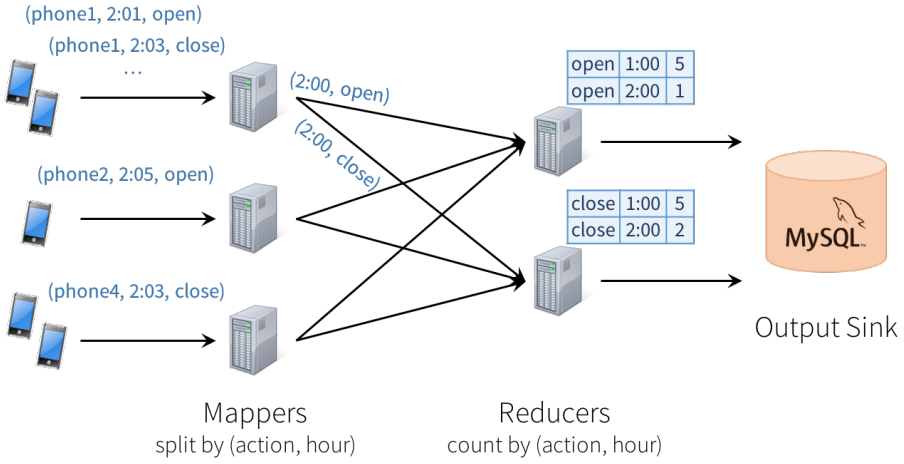
## Why Streaming is Difficult

At first glance, building a distributed streaming engine might seem as simple as launching a set of servers and pushing data between them. Unfortunately, distributed stream processing runs into multiple complications that don't affect simpler computations like batch jobs.

To start, consider a simple application: we receive (phone_id, time, action) events from a mobile app, and want to count how many actions of each type happened each hour, then store the result in MySQL. If we were running this application as a batch job and had a table with all the input events, we could express it as the following SQL query:

```
SELECT action, WINDOW(time, "1 hour"), COUNT(*)
FROM events
GROUP BY action, WINDOW(time, "1 hour")
```

In a distributed streaming engine, we might set up nodes to process the data in a "map-reduce" pattern, as shown below. Each node in the first layer reads a partition of the input data (say, the stream from one set of phones), then hashes the events by (action, hour) to send them to a reducer node, which tracks that group's count and periodically updates MySQL.



Unfortunately, this type of design can introduce quite a few challenges:

1. **Consistency:** This distributed design can cause records to be processed in one part of the system before they're processed in another, leading to nonsensical results. For example, suppose our app sends an "open" event when users open it, and a "close" event when closed. If the reducer node responsible for "open" is slower than the one for "close", we might see a *higher total count of "closes" than "opens" in MySQL*, which would not make sense. The image above actually shows one such example.

2. **Fault tolerance:** What happens if one of the mappers or reducers fails? A reducer should not count an action in MySQL twice, but should somehow know how to request old data from the mappers when it comes up. Streaming engines go through a great deal of trouble to provide strong semantics here, at least *within* the engine. In many engines, however, keeping the result consistent in external storage is left to the user.

3. **Out-of-order data:** In the real world, data from different sources can come out of order: for example, a phone might upload its data hours late if it's out of coverage. Just writing the reducer operators to assume data arrives in order of time fields will not work—they need to be prepared to receive out-of-order data, and to update the results in MySQL accordingly.

In most current streaming systems, some or all of these concerns are left to the user. This is unfortunate because these issues—how the application interacts with the outside world—are some of the hardest to reason about and get right. In particular, there is no easy way to get semantics as simple as the SQL query above.

## Structured Streaming Model

In Structured Streaming, we tackle the issue of semantics head-on by making a strong guarantee about the system: *at any time, the output of the application is equivalent to executing a batch job on a prefix of the data*. For example, in our monitoring application, the result table in MySQL will always be equivalent to taking a prefix of each phone's update stream (whatever data made it to the system so far) and running the SQL query we showed above. There will never be "open" events counted faster than "close" events, duplicate updates on failure, etc. Structured Streaming automatically handles consistency and reliability both within the engine and in interactions with external systems (e.g. updating MySQL transactionally).

This *prefix integrity* guarantee makes it easy to reason about the three challenges we identified. In particular:

1. Output tables are **always consistent** with all the records in a prefix of the data. For example, as long as each phone uploads its data as a sequential stream (e.g., to the same partition in Apache Kafka), we will always process and count its events in order.

2. **Fault tolerance** is handled holistically by Structured Streaming, including in interactions with output sinks. This was a major goal in supporting continuous applications.

3. The effect of **out-of-order data** is clear. We know that the job outputs counts grouped by action and time for a prefix of the stream. If we later receive more data, we might see a time field for an hour in the past, and we will simply update its respective row in MySQL. Structured Streaming also supports APIs for filtering out overly old data if the user wants. But fundamentally, out-of-order data is not a "special case": the query says to group by time field, and seeing an old time is no different than seeing a repeated action.

The last benefit of Structured Streaming is that the API is very easy to use: it is simply Spark's DataFrame and Dataset API. Users just describe the query they want to run, the input and output locations, and optionally a few more details. The system then runs their query incrementally, maintaining enough state to recover from failure, keep the results consistent in external storage, etc. For example, here is how to write our streaming monitoring application:

```
// Read data continuously from an S3 location
val inputDF = spark.readStream.json("s3://logs")

// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
      .writeStream.format("jdbc")
      .start("jdbc:mysql//...")
```

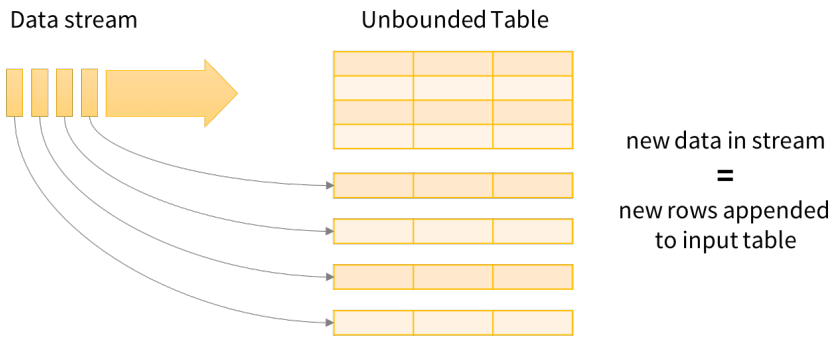
This code is nearly identical to the batch version below—only the "read" and "write" changed:

```
// Read data once from an S3 location
val inputDF = spark.read.json("s3://logs")

// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
      .writeStream.format("jdbc")
      .save("jdbc:mysql//...")
```

The next sections explain the model in more detail, as well as the API.

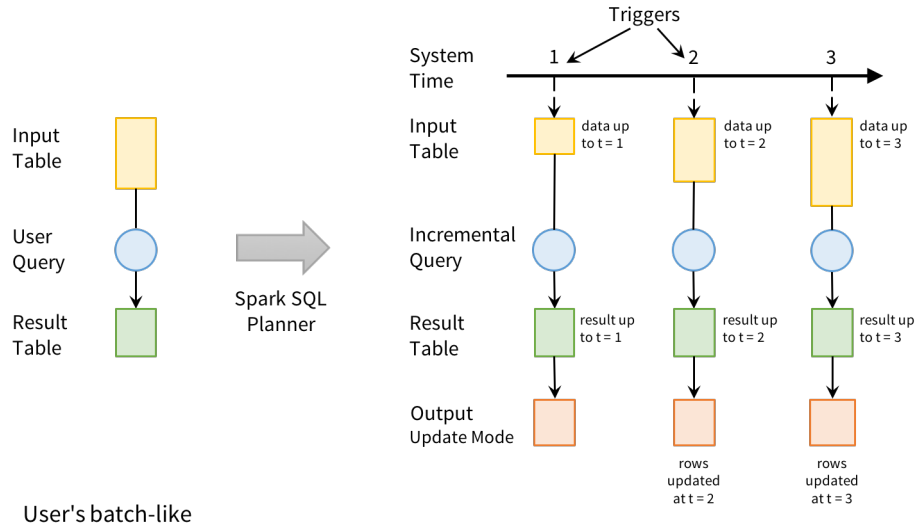
## Model Details

Conceptually, Structured Streaming treats all the data arriving as an unbounded input table. Each new item in the stream is like a row appended to the **input table.** We won't actually retain all the input, but our results will be equivalent to having all of it and running a batch job.




Data stream as an unbounded Input Table

The developer then defines a **query** on this input table, as if it were a static table, to compute a final **result table** that will be written to an output sink. Spark automatically converts this batch-like query to a streaming execution plan. This is called *incrementalization*: Spark figures out what state needs to be maintained to update the result each time a record arrives. Finally, developers specify **triggers** to control when to update the results. Each time a trigger fires, Spark checks for new data (new row in the input table), and incrementally updates the result.




Structured Streaming Processing Model

Users express queries using a batch API; Spark incrementalizes them to run on streams

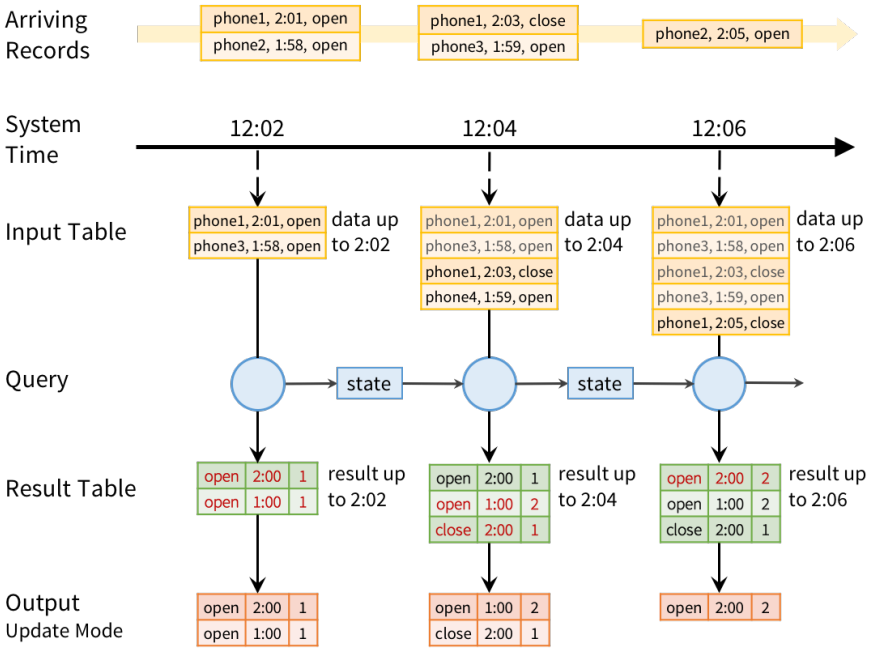
The last part of the model is **output modes**. Each time the result table is updated, the developer wants to write the changes to an external system, such as S3, HDFS, or a database. We usually want to write output incrementally. For this purpose, Structured Streaming provides three output modes:

- **Append:** Only the new rows appended to the result table since the last trigger will be written to the external storage. This is applicable only on queries where existing rows in the result table cannot change (e.g. a map on an input stream).

- **Complete:** The entire updated result table will be written to external storage.

- **Update:** Only the rows that were updated in the result table since the last trigger will be changed in the external storage. This mode works for output sinks that can be updated in place, such as a MySQL table.

Let's see how we can run our mobile monitoring application in this model. Our batch query is to compute a count of actions grouped by (action, hour). To run this query incrementally, Spark will maintain some state with the counts for each pair so far, and update when new records arrive. For each record changed, it will then output data according to its output mode. The figure below shows this execution using the Update output mode:




At every trigger point, we take the previous grouped counts and update them with new data that arrived since the last trigger to get a new result table. We then emit only the changes required by our output mode to the sink—here, we update the records for (action, hour) pairs that changed during that trigger in MySQL (shown in red).

Note that the system also automatically handles late data. In the figure above, the "open" event for phone3, which happened at 1:58 on the phone, only gets to the system at 2:02. Nonetheless, even though it's past 2:00, we update the record for 1:00 in MySQL. However, the prefix integrity guarantee in Structured Streaming ensures that we process the records from each source *in the order they arrive*. For example, because phone1's "close" event arrives after its "open" event, we will always update the "open" count before we update the "close" count.

## Fault Recovery and Storage System Requirements

Structured Streaming keeps its results valid even if machines fail. To do this, it places two requirements on the input sources and output sinks:

1. Input sources must be *replayable*, so that recent data can be re-read if the job crashes. For example, message buses like Amazon Kinesis and Apache Kafka are replayable, as is the file system input source. Only a few minutes' worth of data needs to be retained; Structured Streaming will maintain its own internal state after that.

2. Output sinks must support *transactional updates*, so that the system can make a set of records appear atomically. The current version of Structured Streaming implements this for file sinks, and we also plan to add it for common databases and key-value stores.

We found that most Spark applications already use sinks and sources with these properties, because users want their jobs to be reliable.

Apart from these requirements, Structured Streaming will manage its internal state in a reliable storage system, such as S3 or HDFS, to store data such as the running counts in our example. Given these properties, Structured Streaming will enforce prefix integrity end-to-end.

## Structured Streaming API

Structured Streaming is integrated into Spark's Dataset and DataFrame APIs; in most cases, you only need to add a few method calls to run a streaming computation. It also adds new operators for windowed aggregation and for setting parameters of the execution model (e.g. output modes). In Apache Spark 2.0, we've built an alpha version of the system with the core APIs. More operators, such as sessionization, will come in future releases.

## API Basics

Streams in Structured Streaming are represented as DataFrames or Datasets with the isStreaming property set to true. You can create them using special read methods from various sources. For example, suppose we wanted to read data in our monitoring application from JSON files uploaded to Amazon S3. The code below shows how to do this in Scala:

```
val inputDF = spark.readStream.json("s3://logs")
```

Our resulting DataFrame, inputDF, is our input table, which will be continuously extended with new rows as new files are added to the directory. The table has two columns—time and action. Now you can use the usual DataFrame/Dataset operations to transform the data. In our example, we want to count action types each hour. To do that we have to group the data by action and 1 hours windows of time.

```
val countsDF = inputDF.groupBy($"action", window($"time", "1 hour"))
                      .count()
```

The new DataFrame countsDF is our result table, which has the columns action, window, and count, and will be continuously updated when the query is started. Note that this transformation would give hourly counts even if inputDF was a static table. This allows developers to test their business logic on static datasets and seamless apply them on streaming data without changing the logic.

Finally, we tell the engine to write this table to a sink and start the streaming computation.

```
val query = countsDF.writeStream.format("jdbc").start("jdbc://...")
```

The returned **query** is a StreamingQuery, a handle to the active streaming execution and can be used to manage and monitor the execution. You can run this complete example by importing the following notebooks into Databricks Community Edition.

- Scala Notebook
- Python Notebook

Beyond these basics, there are many more operations that can be done in Structured Streaming.

## Mapping, Filtering and Running Aggregations

Structured Streaming programs can use DataFrame and Dataset's existing methods to transform data, including map, filter, select, and others. In addition, running (or infinite) aggregations, such as a `count` from the beginning of time, are available through the existing APIs. This is what we used in our monitoring application above.

## Windowed Aggregations on Event Time

Streaming applications often need to compute data on various types of *windows*, including *sliding windows*, which overlap with each other (e.g. a 1-hour window that advances every 5 minutes), and tumbling windows, which do not (e.g. just every hour). In Structured Streaming, *windowing is simply represented as a group-by*. Each input event can be mapped to one or more windows, and simply results in updating one or more result table rows.

Windows can be specified using the window function in DataFrames. For example, we could change our monitoring job to count actions by sliding windows as follows:

```
inputDF.groupBy($"action", window($"time", "1 hour", "5 minutes"))
       .count()
```

Whereas our previous application outputted results of the form (hour, action, count), this new one will output results of the form (window, action, count), such as ("1:10-2:10", "open", 17). If a late record arrives, we will update all the corresponding windows in MySQL. And unlike in many other systems, windowing is not just a special operator for streaming computations; we can run the same code in a batch job to group data in the same way.

Windowed aggregation is one area where we will continue to expand Structured Streaming. In particular, in Spark 2.1, we plan to add *watermarks*, a feature for dropping overly old data when sufficient time has passed. Without this type of feature, the system might have to track state for all old windows, which would not scale as the application runs. In addition, we plan to add support for *session-based windows*, i.e. grouping the events from one source into variable-length sessions according to business logic.

## Joining Streams with Static Data

Because Structured Streaming simply uses the DataFrame API, it is straightforward to join a stream against a static DataFrame, such as an Apache Hive table:

```
// Bring in data about each customer from a static "customers" table,
// then join it with a streaming DataFrame
val customersDF = spark.table("customers")
inputDF.join(customersDF, "customer_id")
       .groupBy($"customer_name", hour($"time"))
       .count()
```

Moreover, the static DataFrame could itself be computed using a Spark query, allowing us to mix batch and streaming computations.

## Interactive Queries

Structured Streaming can expose results directly to interactive queries through Spark's JDBC server. In Spark 2.0, there is a rudimentary "memory" output sink for this purpose that is not designed for large data volumes. However, in future releases, this will let you write query results to an in-memory Spark SQL table, and run queries directly against it.

```
// Save our previous counts query to an in-memory table
countsDF.writeStream.format("memory")
 .queryName("counts")
 .outputMode("complete")
 .start()
// Then any thread can query the table using SQL
sql("select sum(count) from counts where action='login'")
```

## Comparison With Other Engines

To show what's unique about Structured Streaming, the next table compares it with several other systems. As we discussed, Structured Streaming's strong guarantee of prefix integrity makes it equivalent to batch jobs and easy to integrate into larger applications. Moreover, building on Spark enables integration with batch and interactive queries.

| Property | Structured Streaming | Spark Streaming | Apache Storm | Apache Flink | Kafka Streams | Google Dataflow |
|---|---|---|---|---|---|---|
| Streaming API | incrementalize batch queries | integrates with batch | separate from batch | separate from batch | separate from batch | integrates with batch |
| Prefix Integrity Guarantee | ✔ | ✔ | ✘ | ✘ | ✘ | ✘ |
| Internal Processing | exactly once | exactly once | at least once | exactly once | at least once | exactly once |
| Transactional Sources/Sinks | ✔ | some | some | some | ✘ | ✘ |
| Interactive Queries | ✔ | ✔ | ✘ | ✘ | ✘ | ✘ |
| Joins with Static Data | ✔ | ✔ | ✘ | ✘ | ✘ | ✘ |

## Conclusion

Structured Streaming promises to be a much simpler model for building end-to-end real-time applications, built on the features that work best in Spark Streaming. Although Structured Streaming is in alpha for Apache Spark 2.0, we hope this post encourages you to try it out.

Long-term, much like the DataFrame API, we expect Structured Streaming to complement Spark Streaming by providing a more restricted but higher-level interface. If you are running Spark Streaming today, don't worry—it will continue to be supported. But we believe that Structured Streaming can open up real-time computation to many more users.

Structured Streaming is also fully supported on Databricks, including in the free Databricks Community Edition. Try out any of our sample notebooks to see it in action:

- Scala notebook for monitoring app
- Python notebook for monitoring app

## Read More

In addition, the following resources cover Structured Streaming:

- Structuring Spark: DataFrames, Datasets and Streaming
- Structured Streaming Programming Guide
- Spark 2.0 and Structured Streaming
- A Deep Dive Into Structured Streaming

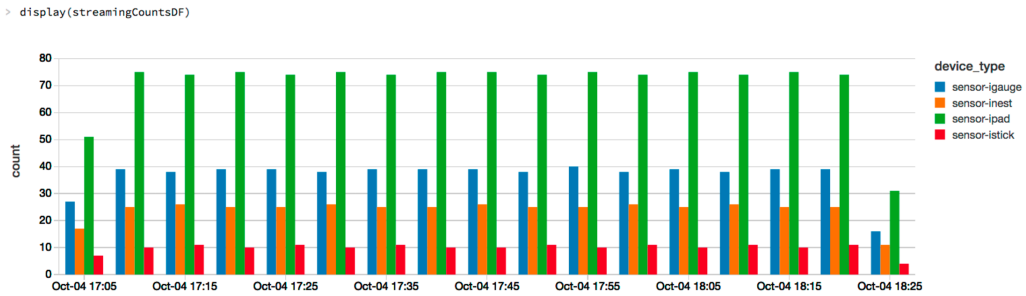
# How to Use Structured Streaming to Analyze IoT Streaming Data

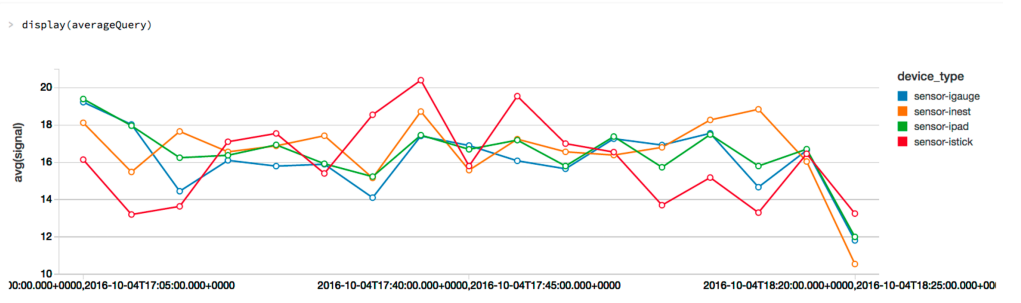
Oct 12, 2016 | by Jules S. Damji

*Try this notebook in Databricks*

## Structured Streaming using Scala DataFrames API

Apache Spark 2.0 adds the first version of a new higher-level stream processing API, Structured Streaming. In this notebook, we are going to take a quick look at how to use DataFrame API to build Structured Streaming queries. We want to compute real-time metrics like running counts and windowed counts on a stream of timestamped device events. These events are randomly generated so there will be unpredictability in the data analysis, since it is computer generated data than real data. But it does not preclude us from showing and illustrating some Structured Streaming APIs and concepts behind issuing equivalent queries on batch as on streaming, with minimal code changes.



# Conclusion

Our mission at Databricks is to dramatically simplify big data processing and data science so organizations can immediately start working on their data problems, in an environment accessible to data scientists, engineers, and business users alike. We hope the collection of blog posts, notebooks, and video tech-talks in this ebook will provide you with the insights and tools to help you solve your biggest data problems.

If you enjoyed the technical content in this ebook, check out the previous books in the series and visit the Databricks Blog for more technical tips, best practices, and case studies from the Apache Spark experts at Databricks.

**Read all the books in this Series:**

Apache Spark Analytics Made Simple
Mastering Advanced Analytics with Apache Spark
Lessons for Large-Scale Machine Learning Deployments on Apache Spark

**To learn more about Databricks, check out some of these resources:**

Databricks Primer
Getting Started with Apache Spark on Databricks
How-To Guide: The Easiest Way to Run Spark Jobs
Solution Brief: Making Data Warehousing Simple
Solution Brief: Making Machine Learning Simple
White Paper: Simplifying Spark Operations with Databricks

To try Databricks yourself, start your free trial today!