8 Steps for a Developer to Learn Apache Spark™ with Delta Lake

From the original creators of Apache Spark and Delta Lake

databricks

8 Steps for a Developer to Learn Apache Spark™ with Delta Lake

From the original creators of Apache Spark and Delta Lake

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.

Table of Contents

Introduction 4

Step 1: Why Apache Spark 5

Step 2: Apache Spark Concepts, Key Terms and Keywords 7

Step 3: Advanced Apache Spark Internals and Core 11

Step 4: DataFames, Datasets and Spark SQL Essentials 13

Step 5: Graph Processing with GraphFrames 17

Step 6: Continuous Applications with Structured Streaming 21

Step 7: Machine Learning for Humans 27

Step 8: Reliable Data Lakes & Data Pipelines 30

Conclusion 35

Introduction

Since inception, Databricks' mission has been to make Big Data simple and accessible to everyone through the unification of data and analytics — 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.

In this ebook, we expand, augment and curate on concepts initially published on KDnuggets. In addition, we augment the ebook with technical blogs and related assets specific to Delta Lake and Apache Spark 2.x, 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 and Structured Streaming; Joseph Bradley, one of the drivers behind Spark MLlib and SparkR; and Tathagata Das, lead developer for Structured Streaming.

Delta Lake is an open source storage layer that sits on top of your existing data lake file storage, such AWS S3, Azure Data Lake Storage, or HDFS. Delta Lake brings reliability, performance, and lifecycle management to data lakes. No more malformed data ingestion, difficulty deleting data for compliance, or issues modifying data for change data capture. Accelerate the velocity that high quality data can get into your data lake, and the rate that teams can leverage that data, with a secure and scalable cloud service. As an open source project supported by the Linux Foundation, Delta Lake allows data to be read by any compatible reader and is compatible with Apache Spark.

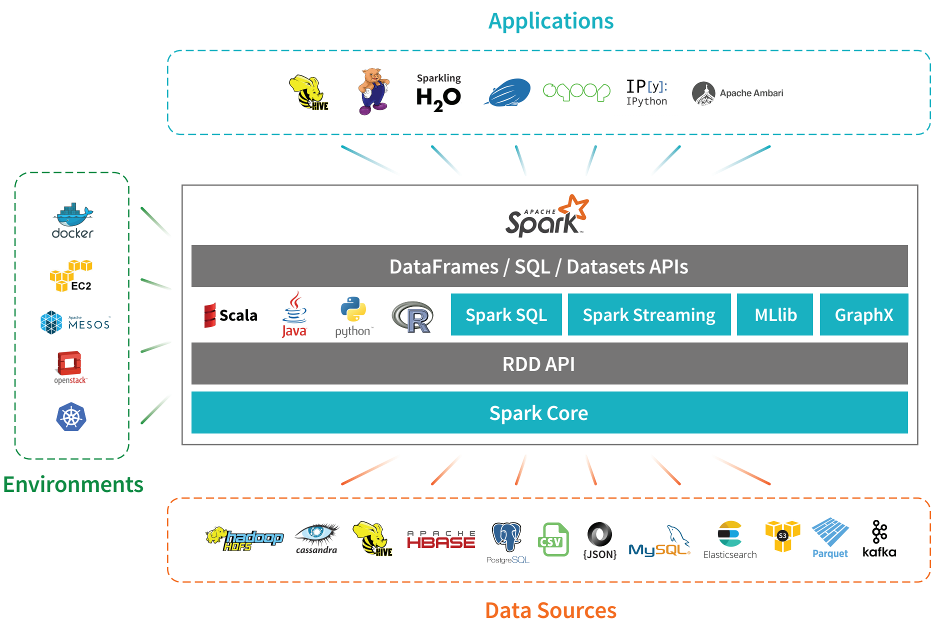
Collectively, the ebook introduces steps for a developer to understand Delta Lake and Apache Spark, at a deeper level. Whether you're getting started with Delta Lake and Apache Spark or already an accomplished developer, this ebook will arm you with the knowledge to employ all of Delta Lake's and Apache Spark's benefits.

Jules S. Damji
Apache Spark Community Evangelist

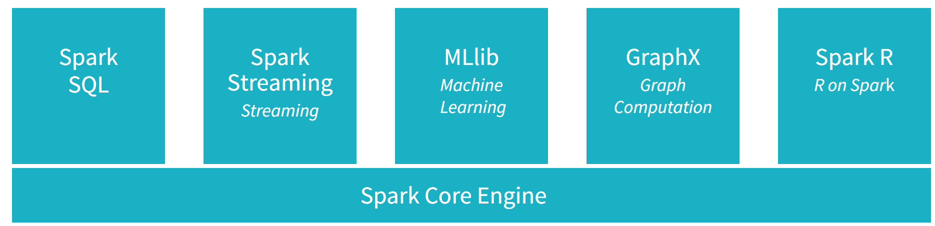
Step 1:
Why Apache Spark?

Why Apache Spark?

For one, Apache Spark is the most active open source data processing engine built for speed, ease of use, and advanced analytics, with over 1000+ contributors from over 250 organizations and a growing community of developers and adopters and users. Second, as a general purpose fast compute engine designed for distributed data processing at scale, Spark supports multiple workloads through a unified engine comprised of Spark components as libraries accessible via unified APIs in popular programing languages, including Scala, Java, Python, and R. And finally, it can be deployed in different environments, read data from various data sources, and interact with myriad applications.



All together, this unified compute engine makes Spark an ideal environment for diverse workloads—traditional and streaming ETL, interactive or ad-hoc queries (Spark SQL), advanced analytics (Machine Learning), graph processing (GraphX/GraphFrames), and Streaming (Structured Streaming)—all running within the same engine.



In the subsequent steps, you will get an introduction to some of these components, from a developer's perspective, but first let's capture key concepts and key terms.

Step 2:
Apache Spark Concepts, Key Terms and Keywords

Apache Spark Architectural Concepts, Key Terms and Keywords

In June 2016, KDnuggets published *Apache Spark Key Terms Explained*, which is a fitting introduction here. Add to this conceptual vocabulary the following Spark's architectural terms, as they are referenced in this article.

Spark Cluster

A collection of machines or nodes in the public cloud or on-premise in a private data center on which Spark is installed. Among those machines are Spark workers, a Spark Master (also a cluster manager in a Standalone mode), and at least one Spark Driver.

Spark Master

As the name suggests, a Spark Master JVM acts as a cluster manager in a Standalone deployment mode to which Spark workers register themselves as part of a quorum. Depending on the deployment mode, it acts as a resource manager and decides where and how many Executors to launch, and on what Spark workers in the cluster.

Spark Worker

Upon receiving instructions from Spark Master, the Spark worker JVM launches Executors on the worker on behalf of the Spark Driver. Spark applications, decomposed into units of tasks, are executed on each worker's Executor. In short, the worker's job is to only launch an Executor on behalf of the master.

Spark Executor

A Spark Executor is a JVM container with an allocated amount of cores and memory on which Spark runs its tasks. Each worker node launches its own Spark Executor, with a configurable number of cores (or threads). Besides executing Spark tasks, an Executor also stores and caches all data partitions in its memory.

Spark Driver

Once it gets information from the Spark Master of all the workers in the cluster and where they are, the driver program distributes Spark tasks to each worker's Executor. The driver also receives computed results from each Executor's tasks.

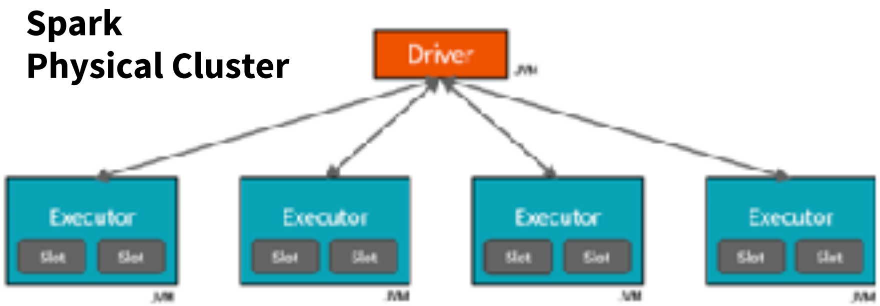


Fig 1. Spark Cluster

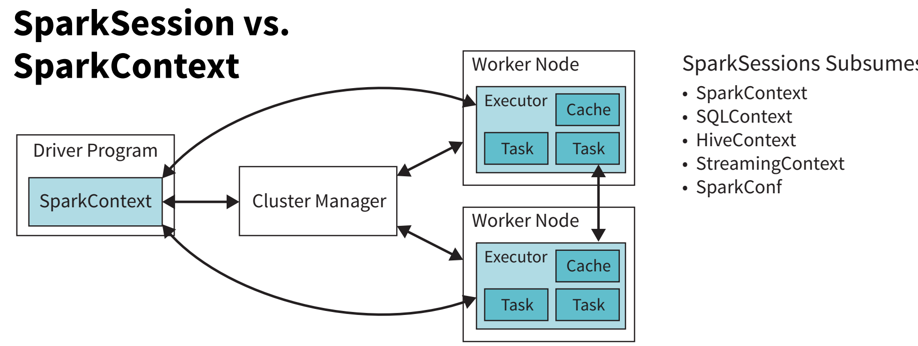


Fig 2. SparkContext and its Interaction with Spark Components

SparkSession and SparkContext

As shown in Fig 2., 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, and submit Spark jobs. It allows you to programmatically adjust Spark configuration parameters. And through SparkContext, the driver can instantiate other contexts such as SQLContext, HiveContext, and StreamingContext to program Spark.

However, with Apache Spark 2.0, SparkSession can access all of Spark's functionality through a single-unified point of entry. As well as making it simpler to access Spark functionality, such as DataFrames and Datasets, Catalogues, and Spark Configuration, it also subsumes the underlying contexts to manipulate data.

A blog post on How to Use SparkSessions in Apache Spark 2.0 explains this in detail, and its accompanying notebooks give you examples in how to use SparkSession programming interface.

Spark Deployment Modes Cheat Sheet

Spark supports four cluster deployment modes, each with its own characteristics with respect to where Spark's components run within a Spark cluster. Of all modes, the local mode, running on a single host, is by far the simplest—to learn and experiment with.

As a beginner or intermediate developer, you don't need to know this elaborate matrix right away. It's here for your reference, and the links provide additional information. Furthermore, Step 3 is a deep dive into all aspects of Spark architecture from a devops point of view.

MODE	DRIVER	WORKER	EXECUTOR	MASTER
Local	Runs on a single JVM	Runs on the same JVM as the driver	Runs on the same JVM as the driver	Runs on a single host
Standalone	Can run on any node in the cluster	Runs on its own JVM on each node	Each worker in the cluster will launch its own JVM	Can be allocated arbitrarily where the master is started
Yarn (client)	On a client, not part of the cluster	YARN NodeManager	YARN's NodeManager's Container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for Executors.
YARN (cluster)	Runs within the YARN's Application Master	Same as YARN client mode	Same as YARN client mode	Same as YARN client mode
Mesos (client)	Runs on a client machine, not part of Mesos cluster	Runs on Mesos Slave	Container within Mesos Slave	Mesos' master
Mesos (cluster)	Runs within one of Mesos' master	Same as client mode	Same as client mode	Mesos' master

Table 1. Cheat Sheet Depicting Deployment Modes And Where Each Spark Component Runs

Spark Apps, Jobs, Stages and Tasks

An anatomy of a Spark application usually comprises of Spark operations, which can be either transformations or actions on your data sets using Spark's RDDs, DataFrames or Datasets APIs. For example, in your Spark app, if you invoke an action, such as *collect()* or *take()* on your DataFrame or Dataset, the action will create a job. A job will then be decomposed into single or multiple stages; stages are further divided into individual tasks; and tasks are units of execution that the Spark driver's scheduler ships to Spark Executors on the Spark worker nodes to execute in your cluster. Often multiple tasks will run in parallel on the same executor, each processing its unit of partitioned dataset in its memory.

In this informative part of the video, Sameer Farooqui elaborates each of the distinct stages in vivid details. He illustrates how Spark jobs, when submitted, get broken down into stages, some multiple stages, followed by tasks, scheduled to be distributed among executors on Spark workers.

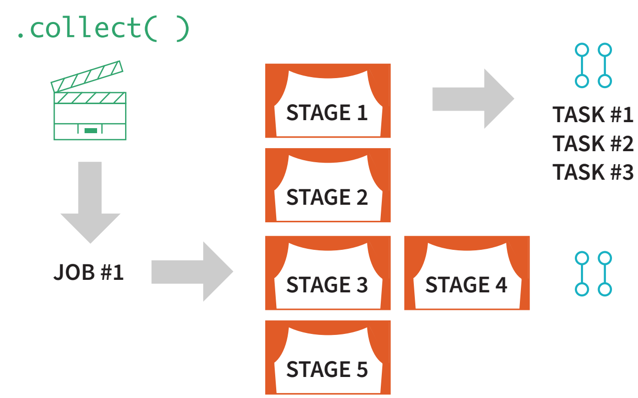
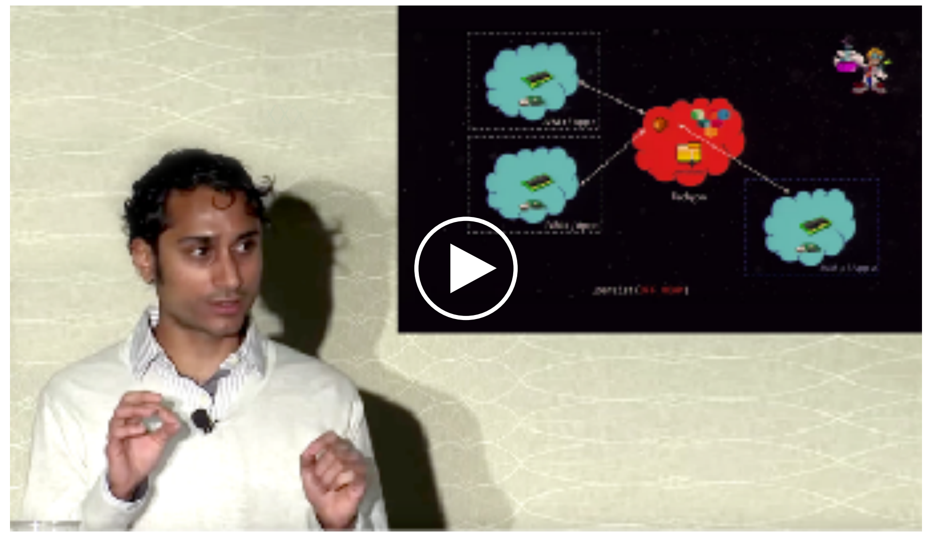


Fig 3. Anatomy of a Spark Application

Step 3:
Advanced Apache Spark Internals and Core

Advanced Apache Spark Internals and Spark Core

To understand how all of the Spark components interact—and to be proficient in programming Spark—it's essential to grasp Spark's core architecture in details. All the key terms and concepts defined in Step 2 come to life when you hear them explained. No better place to see it explained than in this Spark Summit training video; you can immerse yourself and take the journey into Spark's core.



Besides the core architecture, you will also learn the following:

- How the data are partitioned, transformed, and transferred across Spark worker nodes within a Spark cluster during network transfers called "shuffle"
- How jobs are decomposed into stages and tasks.
- How stages are constructed as a Directed Acyclic Graph (DAGs).
- How tasks are then scheduled for distributed execution.

Step 4:
DataFrames, Datasets, and Spark SQL Essentials

DataFrames, Datasets and Spark SQL Essentials

In Steps 2 and 3, you might have learned about Resilient Distributed Datasets (RDDs)—if you watched the linked videos—because they form the core data abstraction concept in Spark and underpin all other higher-level data abstractions and APIs, including DataFrames and Datasets.

In Apache Spark 2.0, DataFrames and Datasets, built upon RDDs and Spark SQL engine, form the core high-level and structured distributed data abstraction. They are merged to provide a uniform API across libraries and components in Spark.

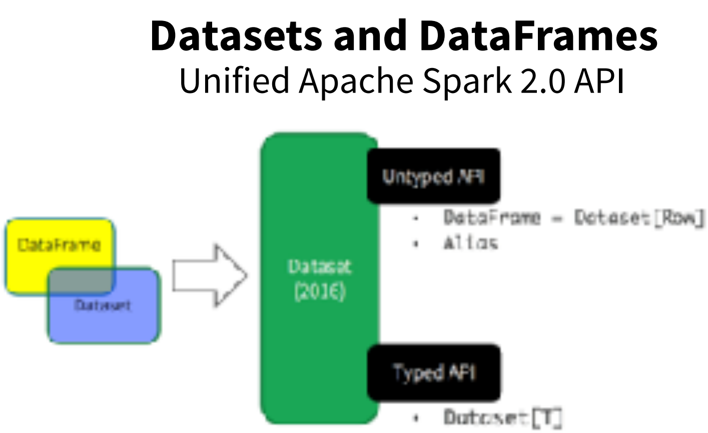


Fig 3. Unified APIs across Apache Spark

DataFrames are named data columns in Spark and they impose a structure and schema in how your data is organized. This organization dictates how to process data, express a computation, or issue a query. For example, your data may be distributed across four RDD partitions, each partition with three named columns: "Time," "Site," and "Req." As such, it provides a natural and intuitive way to access data by their named columns.

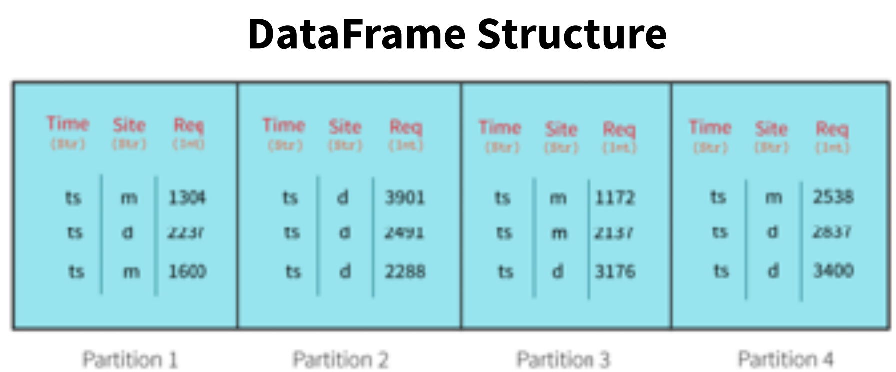


Fig 4. A sample DataFrame with named columns

Datasets, on the other hand, go one step further to provide you strict compile-time type safety, so certain type of errors are caught at compile time rather than runtime.

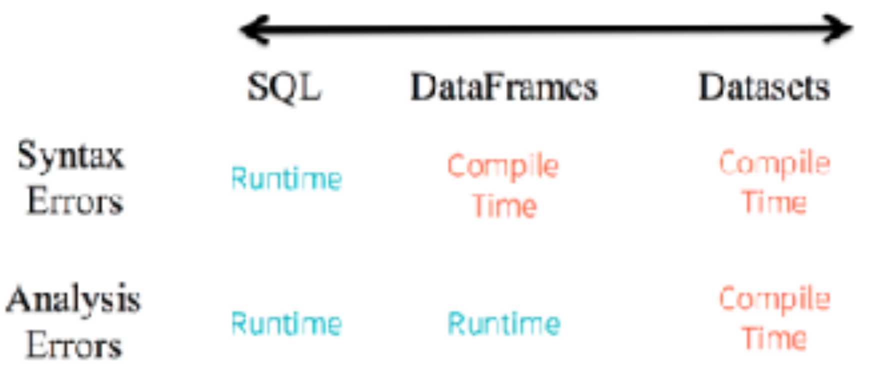


Fig 5. Spectrum of Errors Types detected for DataFrames & Datasets

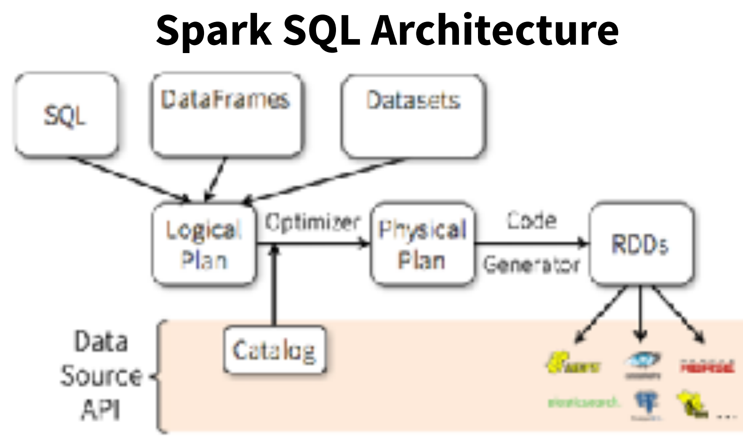


Fig 6. Journey undertaken by a high-level computation in DataFrame, Dataset or SQL

Because of structure in your data and type of data, Spark can understand how you would express your computation, what particular typed-columns or typed-named fields you would access in your data, and what domain specific operations you may use. By parsing your high-level or compute operations on the structured and typed-specific data, represented as DataSets, Spark will optimize your code, through Spark 2.0's Catalyst optimizer, and generate efficient bytecode through Project Tungsten.

DataFrames and Datasets offer high-level domain specific language APIs, making your code expressive by allowing high-level operators like *filter, sum, count, avg, min, max* etc. Whether you express your computations in Spark SQL, Python, Java, Scala, or R Dataset/ Dataframe APIs, the underlying code generated is identical because all execution planning undergoes the same Catalyst optimization as shown in Fig 6.

For example, this high-level domain specific code in Scala or its equivalent relational query in SQL will generate identical code. Consider a Dataset Scala object called Person and an SQL table "person."

```
// a dataset object Person with field names fname, lname, age, weight
// access using object notation
val seniorDS = peopleDS.filter(p=>p.age > 55)
// a dataframe with structure with named columns fname, lname, age, weight
// access using col name notation
val seniorDF = peopleDF.where(peopleDF("age") > 55)
// equivalent Spark SQL code
val seniorDF = spark.sql("SELECT age from person where age > 35")
```

To get a whirlwind introduction of why structuring data in Spark is important and why DataFrames, Datasets, and Spark SQL provide an efficient way to program Spark, we urge you to watch this Spark Summit talk by Michael Armbrust, Spark PMC and committer, in which he articulates the motivations and merits behind structure in Spark.

In addition, these technical assets discuss DataFrames and Datasets, and how to use them in processing structured data like JSON files and issuing Spark SQL queries.

1. Introduction to Datasets in Apache Spark
2. A tale of Three APIS: RDDs, DataFrames, and Datasets
3. Datasets and DataFrame Notebooks

Step 5:
Graph Processing with GraphFrames

Graph Processing with GraphFrames

Even though Spark has a general purpose RDD-based graph processing library named GraphX, which is optimized for distributed computing and supports graph algorithms, it has some challenges. It has no Java or Python APIs, and it's based on low-level RDD APIs. Because of these constraints, it cannot take advantage of recent performance and optimizations introduced in DataFrames through Project Tungsten and Catalyst Optimizer.

By contrast, the DataFrame-based GraphFrames address all these constraints: It provides an analogous library to GraphX but with high-level, expressive and declarative APIs, in Java, Scala and Python; an ability to issue powerful SQL like queries using DataFrames APIs; saving and loading graphs; and takes advantage of underlying performance and query optimizations in Apache Spark 2.0. Moreover, it integrates well with GraphX. That is, you can seamlessly convert a GraphFrame into an equivalent GraphX representation.

Consider a simple example of cities and airports. In the Graph diagram in Fig 7, representing airport codes in their cities, all of the vertices can be represented as rows of DataFrames; and all of the edges can be represented as rows of DataFrames, with their respective named and typed columns.

Collectively, these DataFrames of vertices and edges comprise a GraphFrame.

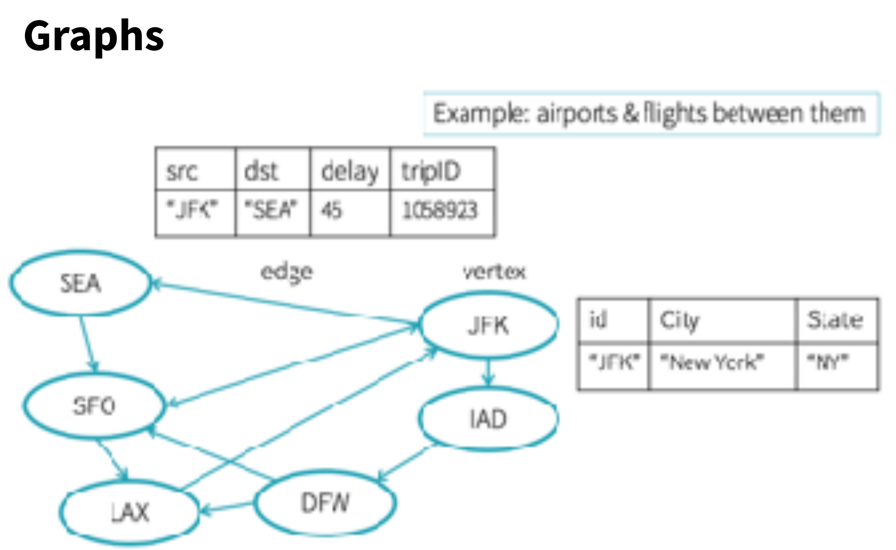


Fig 7. A Graph Of Cities Represented As Graphframe

If you were to represent this above picture programmatically, you would write as follows:

```
// create a Vertices DataFrame
val vertices = spark.createDataFrame(List(("JFK", "New York",
"NY"))).toDF("id", "city", "state")
// create a Edges DataFrame
val edges = spark.createDataFrame(List(("JFK", "SEA", 45,
1058923))).toDF("src", "dst", "delay", "tripID”)
// create a GraphFrame and use its APIs
val airportGF = GraphFrame(vertices, edges)
// filter all vertices from the GraphFrame with delays greater an 30 mins
val delayDF = airportGF.edges.filter(“delay > 30”)
// Using PageRank algorithm, determine the Airport ranking of importance
val pageRanksGF =
airportGF.pageRank.resetProbability(0.15).maxIter(5).run()
display(pageRanksGF.vertices.orderBy(desc("pagerank")))
```

With GraphFrames you can express three kinds of powerful queries. First, simple SQL-type queries on vertices and edges such as what trips are likely to have major delays. Second, graph-type queries such as how many vertices have incoming and outgoing edges. And finally, motif queries, by providing a structural pattern or path of vertices and edges and then finding those patterns in your graph's dataset.

Additionally, GraphFrames easily support all of the graph algorithms supported in GraphX. For example, find important vertices using PageRank, determine the shortest path from source to destination, or perform a Breadth First Search (BFS). You can also determine strongly connected vertices for exploring social connections.

In the webinar *GraphFrames: DataFrame-based graphs for Apache Spark*, Joseph Bradley, Spark Committer, gives an illuminative introduction to graph processing with GraphFrames, its motivations and ease of use, and the benefits of its DataFrame-based API. And through a demonstrated notebook as part of the webinar, you'll learn the ease with which you can use GraphFrames and issue all of the aforementioned types of queries and types of algorithms.

Complementing the above webinar, the two instructive blogs with accompanying notebooks below offer an introductory and hands-on experience with DataFrame-based GraphFrames.

1. Introduction to GraphFrames
2. On-time Flight Performance with GraphFrames for Apache Spark

With Apache Spark 2.0 and beyond, many Spark components, including Machine Learning MLlib and Streaming, are increasingly moving towards offering equivalent DataFrames APIs, because of performance gains, ease of use, and high-level abstraction and structure. Where necessary or appropriate for your use case, you may elect to use GraphFrames instead of GraphX. Below is a succinct summary and comparison between GraphX and GraphFrames.

GraphFrames vs. GraphX

	GraphFrames	GraphX
Built on	DataFrames	RDDs
Languages	Scala, Java, Python	Scala
Use cases	Queries & algorithms	Algorithms
Vertex IDs	Any type (in Catalyst)	Long
Vertex/edge attributes	Any number of DataFrame columns	Any type (VD, ED)
Return types	GraphFrame or DataFrame	Graph[VD, ED], or RDD[Long, VD]

Fig 8. Comparison Cheat Sheet Chart

Finally, GraphFrames continues to get faster, and a Spark Summit talk by Ankur Dave shows specific optimizations. A newer version of the GraphFrames package compatible with Spark 2.0 is available as a Spark package.

Step 6:
Continuous Applications with Structured Streaming

Continuous Applications with Structured Streaming

For much of Spark's short history, Spark streaming has continued to evolve, to simplify writing streaming applications. Today, developers need more than just a streaming programming model to transform elements in a stream. Instead, they need a streaming model that supports end-to-end applications that continuously react to data in real-time. We call them continuous applications that react to data in real-time.

Continuous applications have many facets. Examples include interacting with both batch and real-time data; performing streaming ETL; serving data to a dashboard from batch and stream; and doing online machine learning by combining static datasets with real-time data. Currently, such facets are handled by separate applications rather than a single one.

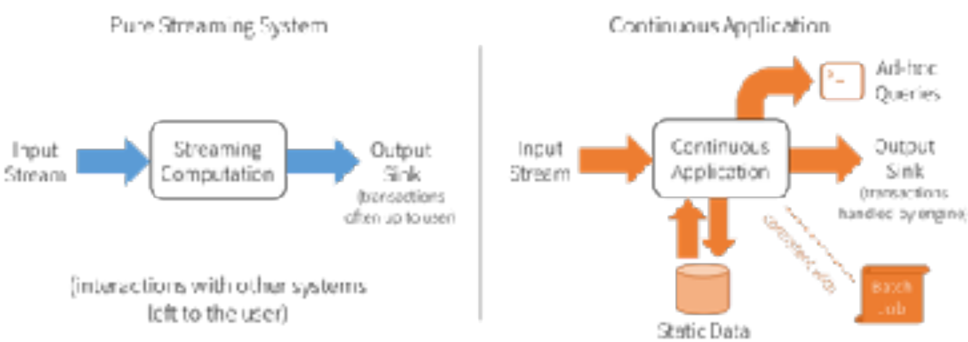


Fig 9. Traditional Streaming Vs Structured Streaming

Apache Spark 2.0 laid the foundational steps for a new higher-level API, Structured Streaming, for building continuous applications. Apache Spark 2.1 extended support for data sources and data sinks, and buttressed streaming operations, including event-time processing watermarking, and checkpointing.

When Is a Stream not a Stream

Central to Structured Streaming is the notion that you treat a stream of data *not* as a stream but as an unbounded table. As new data arrives from the stream, new rows of DataFrames are appended to an unbounded table:

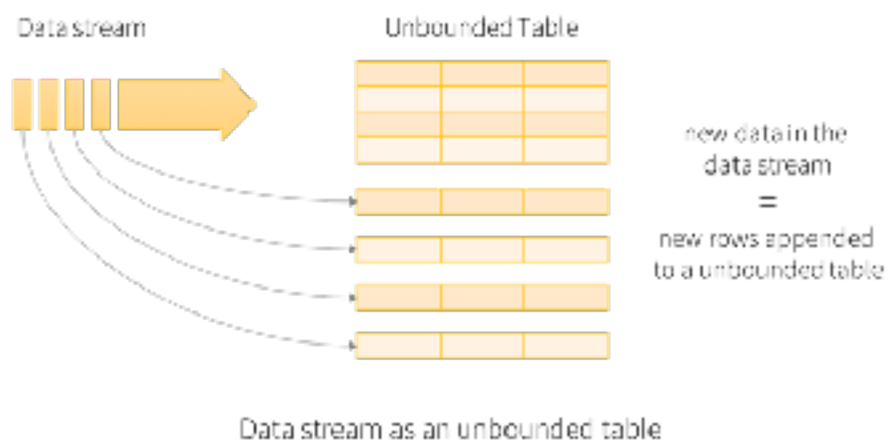
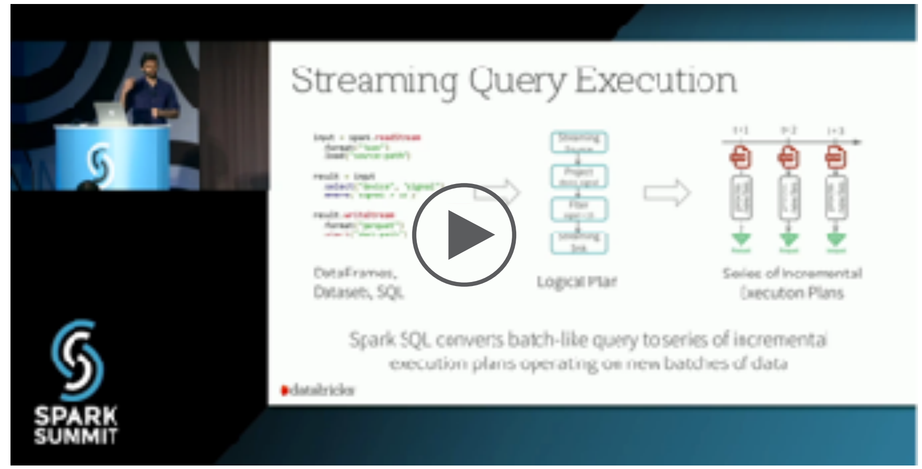


Fig 10. Stream as an Unbounded Table of DataFrames

You can then perform computations or issue SQL-type query operations on your unbounded table as you would on a static table. In this scenario, developers can express their streaming computations just like batch computations, and Spark will automatically execute it incrementally as data arrives in the stream. This is powerful!

Fig 11. Similar Code for Streaming and Batch

Based on the DataFrames/Datasets API, a benefit of using the Structured Streaming API is that your DataFrame/SQL based query for a batch DataFrame is similar to a streaming query, as you can see in the code in Fig 11., with a minor change. In the batch version, we read a static bounded log file, whereas in the streaming version, we read off an unbounded stream. Though the code looks deceptively simple, all the complexity is hidden from a developer and handled by the underlying model and execution engine, which undertakes the burden of fault-tolerance, incremental query execution, idempotency, end-to-end guarantees of exactly-once semantics, out-of-order data, and watermarking events. All of this orchestration under the cover is explained in this technical talk by Tathagata Das at Spark Summit. More importantly, he makes the case by demonstrating how streaming ETL, with Structured Streaming, obviates traditional ETL.



Data Sources

Data sources within the Structure Streaming nomenclature refer to entities from which data can emerge or read. Spark 2.x supports three built-in data sources.

File Source

Directories or files serve as data streams on a local drive, HDFS, or S3 bucket. Implementing the *DataStreamReader* interface, this source supports popular data formats such as avro, JSON, text, or CSV. Since the sources continue to evolve with each release, check the most recent docs for additional data formats and options.

Apache Kafka Source

Compliant with Apache Kafka 0.10.0 and higher, this source allows structured streaming APIs to poll or read data from subscribed topics, adhering to all Kafka semantics. This Kafka integration guide offers further details in how to use this source with structured streaming APIs.

Network Socket Source

An extremely useful source for debugging and testing, you can read UTF-8 text data from a socket connection. Because it's used for testing only, this source does not guarantee any end-to-end fault-tolerance as the other two sources do.

To see a simple code example, check the Structured Streaming guide section for Data Sources and Sinks.

Data Sinks

Data Sinks are destinations where your processed and transformed data can be written to. Since Spark 2.1, three built-in sinks are supported, while a user defined sink can be implemented using a *foreach* interface.

File Sinks

File Sinks, as the name suggests, are directories or files within a specified directory on a local file system, HDFS or S3 bucket can serve as repositories where your processed or transformed data can land.

Foreach Sinks

Implemented by the application developer, a *ForeachWriter* interface allows you to write your processed or transformed data to the destination of your choice. For example, you may wish to write to a NoSQL or JDBC sink, or to write to a listening socket or invoke a REST call to an external service. As a developer you implement its three methods: *open()*, *process()* and *close()*.

Console Sink

Used mainly for debugging purposes, it dumps the output to console/stdout, each time a trigger in your streaming application is invoked. Use it only for debugging purposes on low data volumes, as it incurs heavy memory usage on the driver side.

Memory Sink

Like Console Sinks, it serves solely for debugging purposes, where data is stored as an in-memory table. Where possible use this sink only on low data volumes.

Streaming Operations on DataFrames and Datasets

Most operations with respect to selection, projection, and aggregation on DataFrames and Datasets are supported by the Structured Streaming API, except for few unsupported ones.

For example, a simple Python code performing these operations, after reading a stream of device data into a DataFrame, may look as follows:

```
devicesDF = ...  # streaming DataFrame with IOT device data with schema
{ device: string, type: string, signal: double, time: DateType }
# Select the devices which have signal more than 10
devicesDF.select("device").where("signal > 10")
# Running count of the number of updates for each device type
devicesDF.groupBy("type").count()
```

Event Time Aggregations and WaterMarking

An important operation that did not exist in DStreams is now available in Structured Streaming. Windowing operations over time line allows you to process data not by the *time* data record was received by the system, but by the *time* the event occurred inside its data record. As such, you can perform windowing operations just as you would perform *groupBy* operations, since windowing is classified just as another *groupBy* operation. A short excerpt from the guide illustrates this:

```
import spark.implicits._
val words = ... // streaming DataFrame of schema { timestamp: Timestamp,
word: String }
// Group the data by window and word and compute the count of each group
val deviceCounts = devices.groupBy( window($"timestamp", "10 minutes", "5
minutes"), $"type"
).count()
```

An ability to handle out-of-order or late data is a vital functionality, especially with streaming data and its associated latencies, because data not always arrives serially. What if data records arrive too late or out of order, or what if they continue to accumulate past a certain time threshold?

Watermarking is a scheme whereby you can mark or specify a threshold in your processing timeline interval beyond which any data's event time is deemed useless. Even better, it can be discarded, without ramifications. As such, the streaming engine can effectively and efficiently retain only late data within that time threshold or interval. To read the mechanics and how the Structured Streaming API can be expressed, read the watermarking section of the programming guide. It's as simple as this short snippet API call:

```
val deviceCounts = devices
    .withWatermark("timestamp", "15 minutes")
    .groupBy(window($"timestamp", "10 minutes", "5 minutes"),
        $"type")
    .count()
```

What's Next

After you take a deep dive into Structured Streaming, read the Structure Streaming Programming Model, which elaborates all the under-the-hood complexity of data integrity, fault tolerance, exactly-once semantics, window-based and event-time aggregations, watermarking, and out-of-order data. As a developer or user, you need not worry about these complexities; the underlying streaming engine takes the onus of fault-tolerance, end-to-end reliability, and correctness guarantees.

Learn more about Structured Streaming directly from Spark committer Tathagata Das, and try the accompanying notebook to get some hands-on experience on your first Structured Streaming continuous application. An additional workshop notebook illustrates how to process IoT devices' streaming data using Structured Streaming APIs.
Structured Streaming API in Apache Spark 2.0: A new high-level API for streaming

Similarly, the Structured Streaming Programming Guide offers short examples on how to use supported sinks and sources:

Structured Streaming Programming Guide

Step 7:
Machine Learning for Humans

Machine Learning for Humans

At a human level, machine learning is all about applying statistical learning techniques and algorithms to a large dataset to identify patterns, and from these patterns probabilistic predictions. A simplified view of a model is a mathematical function *f(x)*; with a large dataset as the input, the function *f(x)* is repeatedly applied to the dataset to produce an output with a prediction. A model function, for example, could be any of the various machine learning algorithms: a Linear Regression or Decision Tree.

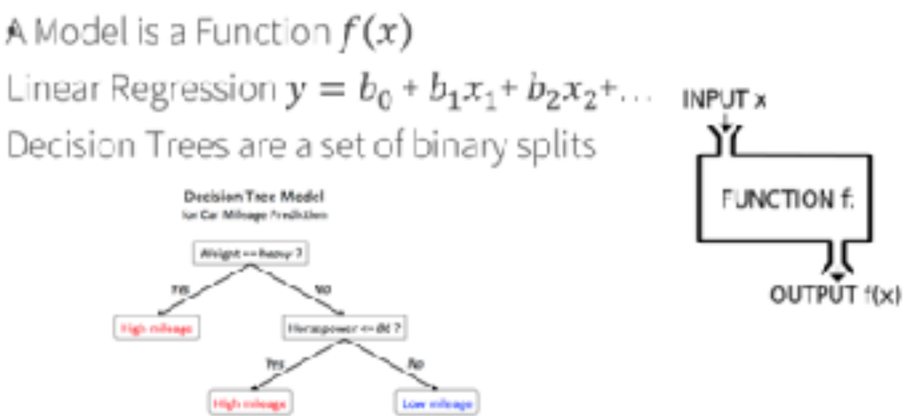


Fig 12. Model as a Mathematical Function

As the core component library in Apache Spark, MLlib offers numerous *supervised* and *unsupervised* learning algorithms, from Logistic Regression to k-means and clustering, from which you can construct these mathematical models.

Key Terms and Machine Learning Algorithms

For introductory key terms of machine learning, Matthew Mayo's Machine Learning Key Terms, Explained is a valuable reference for understanding some concepts discussed in the Databricks webinar on the following page. Also, a hands-on getting started guide, included as a link here, along with documentation on Machine Learning algorithms, buttress the concepts that underpin machine learning, with accompanying code examples in Databricks notebooks.

Machine Learning Pipelines

Apache Spark's DataFrame-based MLlib provides a set of algorithms as models and utilities, allowing data scientists to build machine learning pipelines easily. Borrowed from the scikit-learn project, MLlib pipelines allow developers to combine multiple algorithms into a single pipeline or workflow. Typically, running machine learning algorithms involves a sequence of tasks, including pre-processing, feature extraction, model fitting, and validation stages. In Spark 2.0, this pipeline can be persisted and reloaded again, across languages Spark supports (see the blog link below).

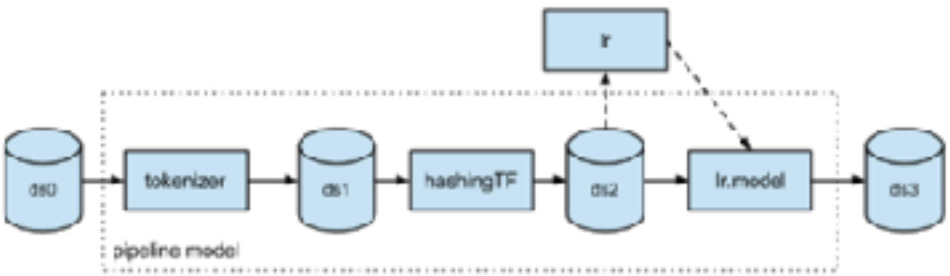


Fig 13.. Machine Learning Pipeline

In the webinar on Apache Spark MLlib, you will get a quick primer on machine learning, Spark MLlib, and an overview of some Spark machine learning use cases, along with how other common data science tools such as Python, pandas, scikit-learn and R integrate with MLlib.

Moreover, two accompanying notebooks for some hands-on experience and a blog on persisting machine learning models will give you insight into why, what and how machine learning plays a crucial role in advanced analytics.

1. Auto-scaling scikit-learn with Apache Spark
2. 2015 Median Home Price by State
3. Population vs. Median Home Prices: Linear Regression with Single Variable
4. Saving and Loading Machine Learning Models in Apache Spark 2.0

If you follow each of these guided steps, watch all the videos, read the blogs, and try out the accompanying notebooks, we believe that you will be on your way as a developer to learn Apache Spark 2.x.

Step 8:
Reliable Data Lakes & Data Pipelines

Data Reliability Challenges with Data Lakes

Failed Writes

If a production job that is writing data experiences failures which are inevitable in large distributed environments, it can result in data corruption through partial or multiple writes. What is needed is a mechanism that is able to ensure that either a write takes place completely or not at all (and not multiple times, adding spurious data). Failed jobs can impose a considerable burden to recover to a clean state

Lack of Consistency

In a complex big data environment one may be interested in considering a mix of both batch and streaming data. Trying to read data while it is being appended to provides a challenge since on the one hand there is a desire to keep ingesting new data while on the other hand anyone reading the data prefers a consistent view. This is especially an issue when there are multiple readers and writers at work. It is undesirable and impractical, of course, to stop read access while writes complete or stop write access while a reads are in progress.

Schema Mismatch

When ingesting content from multiple sources, typical of large, modern big data environments, it can be difficult to ensure that the same data is encoded in the same way i.e. the schema matches. A similar challenge arises when the formats for data elements are changed without informing the data engineering team. Both can result in low quality, inconsistent data that requires cleaning up to improve its usability. The ability to observe and enforce schema would serve to mitigate this.

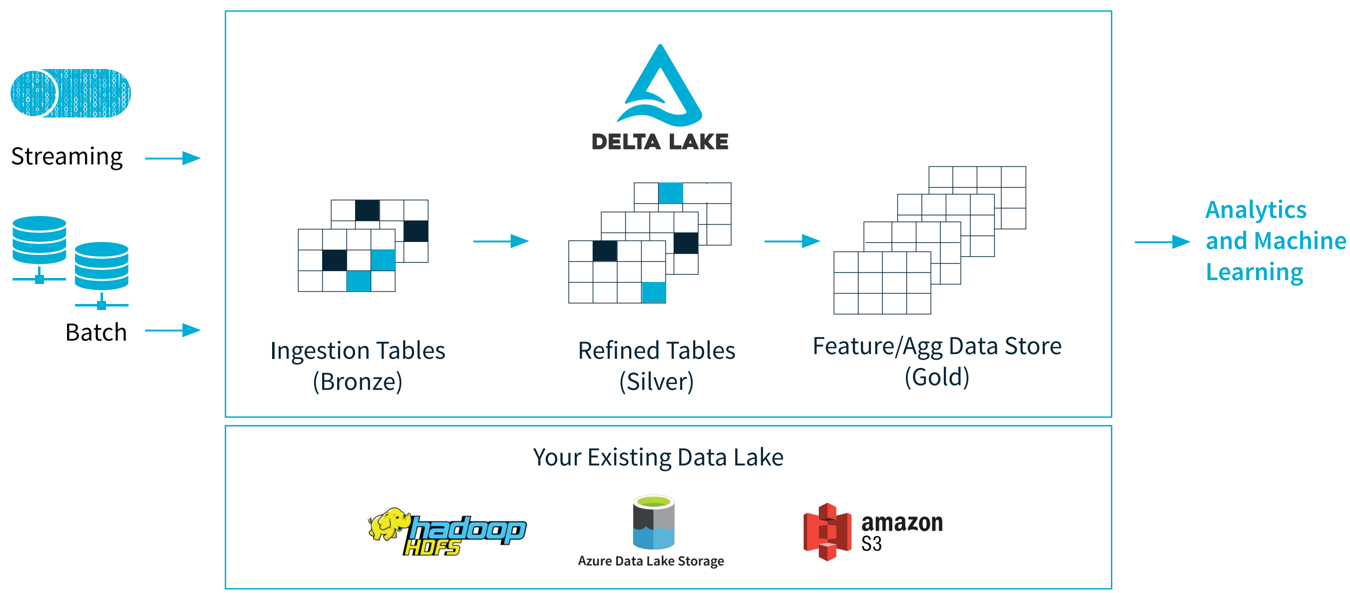
Delta Lake: A New Storage Layer

ACID Transactions

Data lakes typically have multiple data pipelines reading and writing data concurrently, and data engineers have to go through a tedious process to ensure data integrity, due to the lack of transactions. Delta Lake brings ACID transactions to your data lakes. It provides serializability, the strongest level of isolation level.

Scalable Metadata Handling

In big data, even the metadata itself can be "big data". Delta Lake treats metadata just like data, leveraging Spark's distributed processing power to handle all its metadata. As a result, Delta Lake can handle petabyte-scale tables with billions of partitions and files at ease.



Delta Lake: A New Storage Layer

Time Travel (data versioning)

Delta Lake provides snapshots of data enabling developers to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments. For more details on versioning please read this blog Introducing Delta Time Travel for Large Scale Data Lakes.

Open Format

All data in Delta Lake is stored in Apache Parquet format enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet.

Unified Batch and Streaming Source and Sink

A table in Delta Lake is both a batch table, as well as a streaming source and sink. Streaming data ingest, batch historic backfill, and interactive queries all just work out of the box.

Schema Enforcement

Delta Lake provides the ability to specify your schema and enforce it. This helps ensure that the data types are correct and required columns are present, preventing bad data from causing data corruption.

Schema Evolution

Big data is continuously changing. Delta Lake enables you to make changes to a table schema that can be applied automatically, without the need for cumbersome DDL

100% Compatible with Apache Spark API

Developers can use Delta Lake with their existing data pipelines with minimal change as it is fully compatible with Spark, the commonly used big data processing engine.

Getting Started with Delta Lake

Getting started with Delta is easy. Specifically, to create a Delta table simply specify Delta instead of using Parquet.

Instead of parquet...

```
dataframe
.write
.format("parquet")
.save("/data")
```

... simply say delta

```
dataframe
.write
.format("delta")
.save("/data")
```

YOU CAN TRY DELTA LAKE TODAY USING THE QUICKSTART AND EXAMPLE NOTEBOOKS.

The following blogs share examples and news about Delta:

- Introducing Delta Time Travel for Large Scale Data Lakes
- Building a Real-Time Attribution Pipeline with Databricks Delta
- Simplifying Streaming Stock Data Analysis Using Databricks Delta

For more information, please refer to the documentation.

Conclusion

Our mission at Databricks is to unify data analytics 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 and accelerate the velocity that your teams can leverage high quality data. 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 Delta Lake and Apache Spark experts at Databricks.

To try Databricks yourself, start your free trial today!

databricks™