Glossary

Spark

APIs

Low-level APIs

RDD. Immutable distributed collection of objects with lineage dependency (i.e., how the RDD is constructed), partitioned across nodes in the cluster that can be operated in parallel.

- The RDD API is not structured.
- Computation expressed in high-level structured APIs (Dataframe and Dataset) => low-level optimized RDD operations => Scala bytecode for executors' JVMs.

Row. A generic untyped JVM object type in Spark, holding a collection of different types of fields that can be accessed using an index.

Structured APIs

DataFrame. A structured collection of generic objects; each record/row is a Row object, so a DataFrame is also a DatasetRow.

- A DataFrame is an untyped view of Dataset.
- Python and R only support the untyped DataFrame API.
- Can let Spark infer schema.

Dataset. A structured collection of strongly-typed JVM objects; the type of each record/row is defined by a Scala case class or JavaBean.

- Only supported in Scala and Java.
- Schema must be supplied.
- Dataset and Dataframe are unified as structured APIs in Spark 2.

As stated earlier in this chapter, Spark 2.0 unified the DataFrame and Dataset APIs as Structured APIs with similar interfaces so that developers would only have to learn a single set of APIs. Datasets take on two characteristics: *typed* and *untyped* APIs, as shown in Figure 3-1.

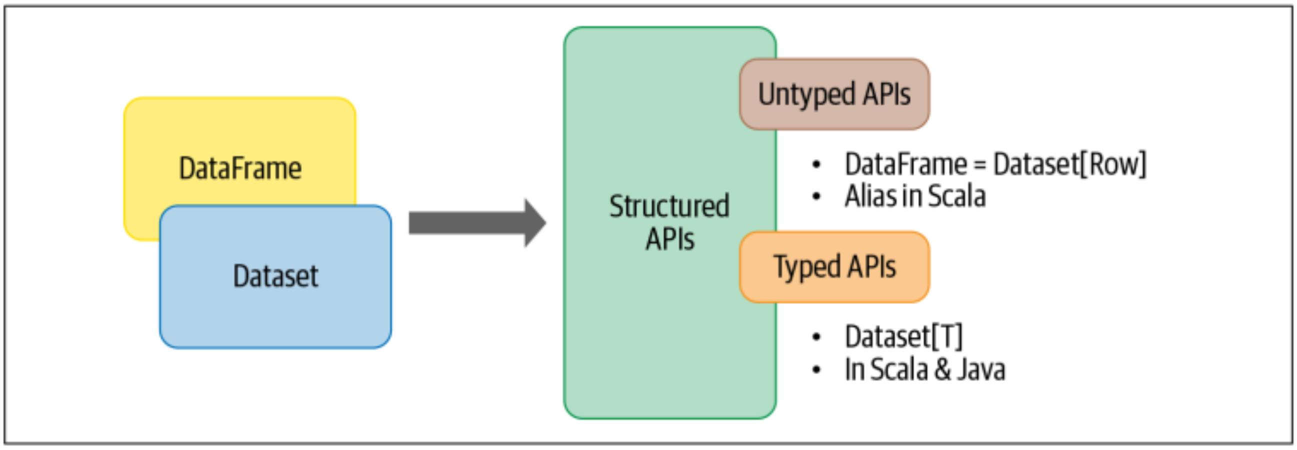

Figure 3-1. Structured APIs in Apache Spark

- If you want errors caught during compilation rather than at runtime, choose the appropriate API as depicted in Figure 3-2.

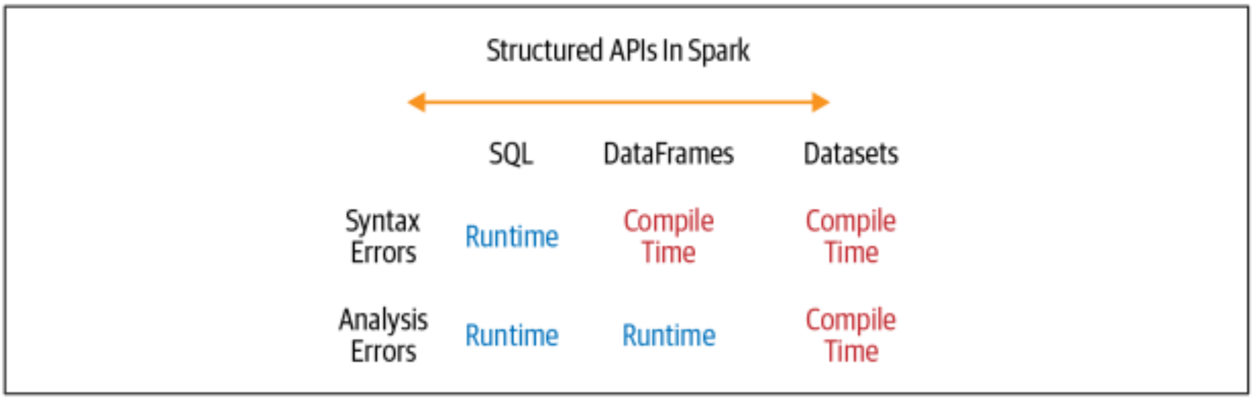

Figure 3-2. When errors are detected using the Structured APIs

Storage

Parquet. A column-oriented data storage format of the Apache Hadoop ecosystem.

> To write the DataFrame into an external data source in your format of choice, you can use the `DataFrameWriter` interface. Like `DataFrameReader`, it supports multiple data sources. Parquet, a popular columnar format, is the default format; it uses snappy compression to compress the data. If the DataFrame is written as Parquet, the schema is preserved as part of the Parquet metadata. In this case, subsequent reads back into a DataFrame do not require you to manually supply a schema.

snappy.parquet. Data stored in parquet. compressed using the [snappy compression](#).

Partition

Disk partition. Partitions of data in hive table stored in HDFS. E.g., `df.partitionBy($"country")` will generate

```
/country=a/
/country=b/
/country=c/
```

folders by country, see [here](#). When reading data, can filter by partition to skip unneeded data.

Memory partition. Partitions of data in memory used by computation in Spark. E.g.,

- Changing memory partitions:`df.repartition(100)` will split data to 100 partitions for Spark to work on, and `df.coalesce(10)` will collapse data to 10 partitions (if original number of partitions is larger than `n`, then `df.coalesce(n)` won't change the partitions).
- Writing partitions to disk
  - `df.repartition(200).write.parquet("xxx")` will generate 200 parquet files on disk, data in each file is distributed randomly.
  - `df.repartition(5).write.partitionBy($"country").parquet("xxx")` will first generate 5 partitions of randomly distributed data and then split each partition based on country. That is, for each of the 5 partitions, there will be 1 separate parquet file for each country to put under `/country=a,b,c`, etc. The result is 1 folder for each unique country, which contains up to 5 parquet files (if a partition has no record, no parquet file will be generated for it), so the maximum number of parquet files is 5 * number of countries.
  - `df.repartition($"country").write.option("maxRecordsPerFile", 10).partitionBy($"country").parquet("xxx")` will create 1 folder of parquet files for each country, and each parquet files has <= 10 records.
- Reading partitions from disk: Partitions of data read from `spark.read.parquet` are determined by (see [here](#))
  - spark.default.parallelism (default is the total number of executor cores of the

Spark application, see here)
    - spark.sql.files.maxPartitionBytes (default 128MB)
    - spark.sql.files.openCostInBytes (default 4MB)

Bucketing

A mechanism to pre-partition data according to how data will be used later, so that expensive shuffles may be avoided.

bucketBy(numberOfBuckets, columnToBucketBy). Distributes data into a fixed number of buckets, and data with the same bucket ID will be in one partition.

- bucketing: the number of unique values is not limited
- partitioning: the number of unique values is limited

Components

Worker node. A member of the cluster. It can be a virtual machine (JVM) or a physical computer.

Executor. A process launched for a Spark application on a worker node.

Shuffle operation. Operations in Spark that trigger shuffle events. E.g., `repartition`, `coalesce`, `reduceBy`, `groupBy`, `join`. https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations

Configurations

spark.sql.shuffle.partitions. The number of partitions to create after shuffling triggered by wide transformations. Large datasets might need higher values, while small datasets need a smaller number to save time by avoiding creating empty shuffle partitions that are not used: https://www.coursera.org/lecture/spark-sql/shuffle-partitions-Y2TDV

Spark SQL

Narrow transformation. Any transformation where a single output partition can be computed from a single input partition. E.g., `select`, `filter` that can be done to each partition in parallel, since performing the operation in a single partition does not require information from other partitions.

Wide transformation. Transformations like `join`, `group by` that require data shuffling among partitions. E.g., the group by operation cannot be performed in each partition in parallel because all records associated with a group by key might reside in more than one partitions.

Stateless transformation. Operations like `select` , `filter` that do not require information from previous rows to process the next row.

Stateful transformation. Operations like `count` , `join` , `group by` that require maintaining state to combine data across multiple rows.

Managed table. A table created in Spark whose data and metadata is managed by Spark. E.g., a table whose data is stored in HDFS.

Unmanaged table. A table created in Spark whose metadata is managed by Spark, but not its data. E.g., a table whose data is stored in an external data source like Cassandra.

View. Temporary view created from tables that don't actually hold the data and disappear after the Spark application terminates. They may be global (accessible to all Spark sessions in cluster) or session-scoped.

Catalyst optimizer. A core component in the Spark SQL engine that takes a computational query and converts it into an execution plan. It has four stages: Analysis (resolve references), logical optimization, physical planning, and code generation.

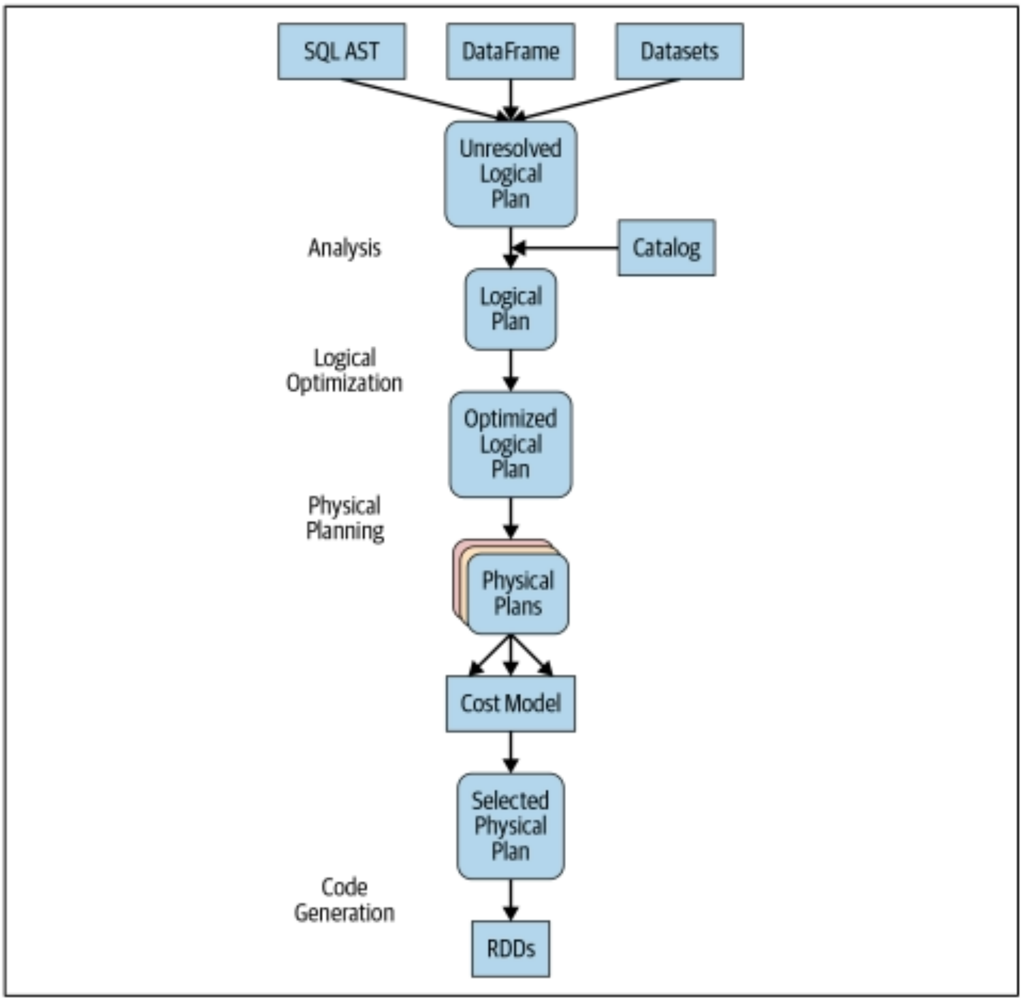

Figure 3-4. A Spark computation's four-phase journey

Whole-Stage CodeGen. Spark SQL engine's mechanism to generate compact Java bytecode from queries by collapsing the query plan into a single function and leveraging CPU registers instead of memory, thereby avoiding overheads caused by virtual function dispatches.

https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

Tungsten. The other core component in the Spark SQL engine that acts as a compiler to generate efficient and compact Java code to run on each machine in the cluster (whole-stage code generation).

Tungsten binary format. Spark's internal binary format for storing objects in Java's off-heap memory using pointer arithmetic and offsets.

- off-heap: Unhindered by JVM's garbage collection.
- Pointer arithmetic: Encoders can quickly serialize objects by traversing across memory.

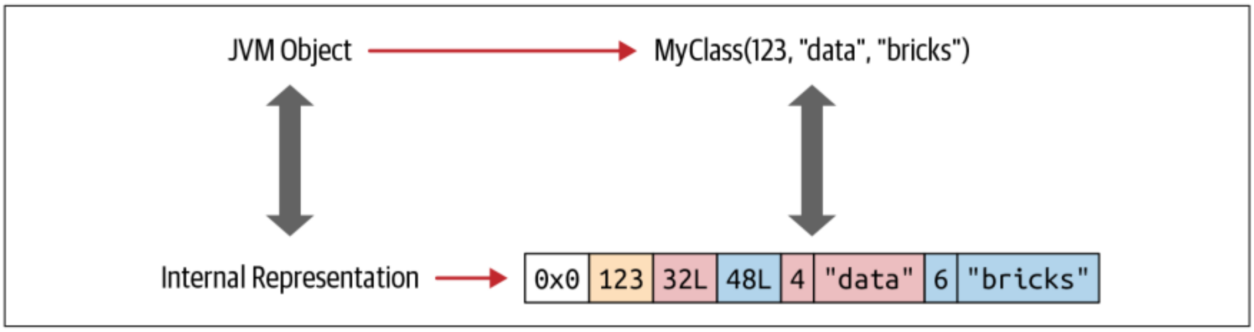

Figure 6-1. JVM object stored in contiguous off-heap Java memory managed by Spark

Encoder. Maps the Scala/Java/Python/R types to Spark's internal type system. It directs an efficient mechanism (better than Java's own serializer operating on heap memory) for serializing and deserializing between JVM objects and Spark's internal Tungsten binary format.

- With Dataset, Encoder helps serializing Scala/Java objects to case class/JavaBean
- With DataFrame, Encoder helps serializing Python/R objects to Row

Catalog. A high-level abstraction in Spark SQL for storing metadata. Accessible via `spark.catalog`.

Joins

Broadcast hash join. A join strategy for joining small table to large table. The smaller dataset is broadcasted by the driver to all executors and subsequently joined with the larger dataset on each executor.The smaller dataset should fit in driver's and executor's memory. Spark uses broadcast hash join if the smaller data set is less than spark.sql.autoBroadcastJoinThreshold (default is 10mb).

Sort merge join. A join strategy for joining two large tables. In sort phase, both datasets are shuffled based on the join key and sorted so that the rows with the same key are in the same partition. The merge phase iterates over the two tables and merge two rows if they have the same join key.

Structured Streaming

Stream processing. Continuous processing of endless streams of data.

Record-at-a-time processing model. Traditional distributed stream processing.

- Directed graph of nodes.
- Low latency of milliseconds.
- Not efficient at recovering from failures and straggler nodes.
- Same API between DStreams and RDDs, but different APIs between batch and streaming.
- No separation between logical and physical plans => no automatic optimization.
- Window operations only based on processing time (the time each record is received by Spark Streaming), not event time (time when each record is generated).

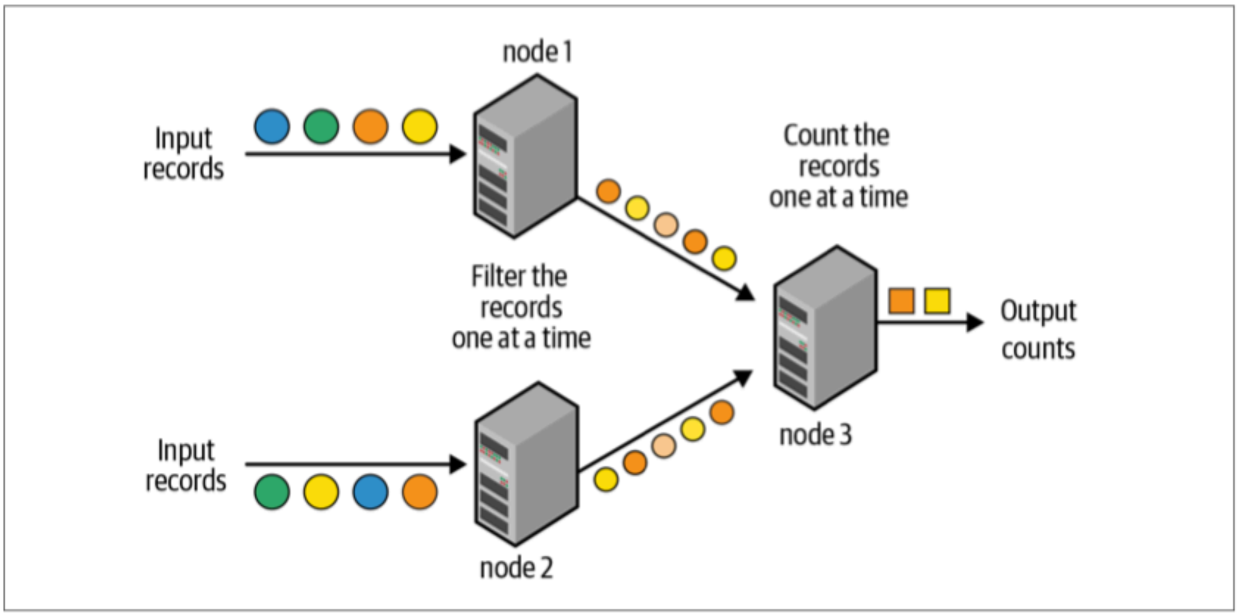

Figure 8-1. Traditional record-at-a-time processing model

Spark Streaming (DStreams). Micro-batch stream processing: Streaming computation is modeled as a continuous series of small, map/reduce-style batch processing jobs on small chunks of the stream data.

- Quickly recover from failures and straggler nodes.
- Deterministic: End-to-end exactly-once processing.
- High latency of seconds.
- Unified API for streaming and batch.
- Works for any application that periodically (e.g., every few hours) to continuously processes data.

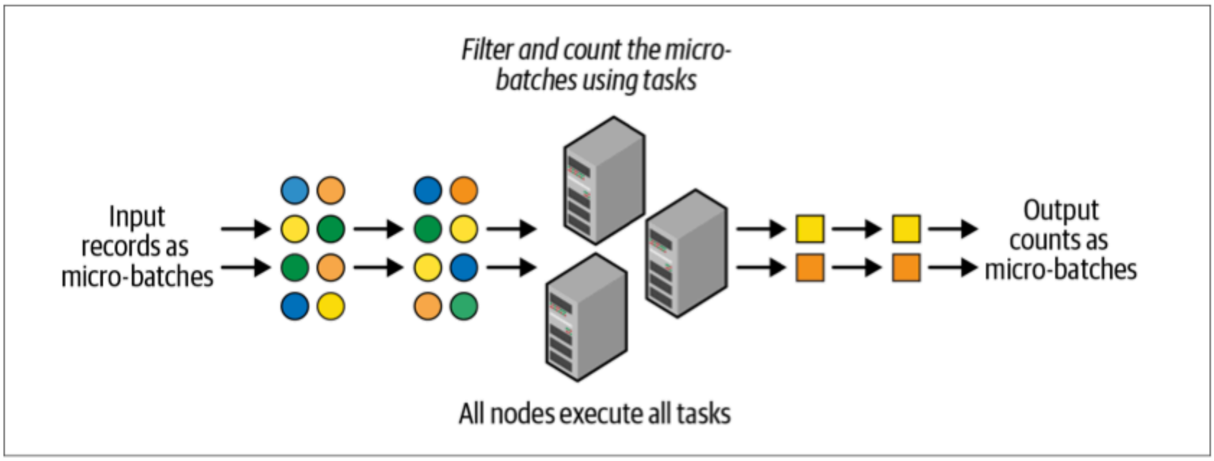

Figure 8-2. Structured Streaming uses a micro-batch processing model

Programming Model

Table. Unbounded, continuously appended table.

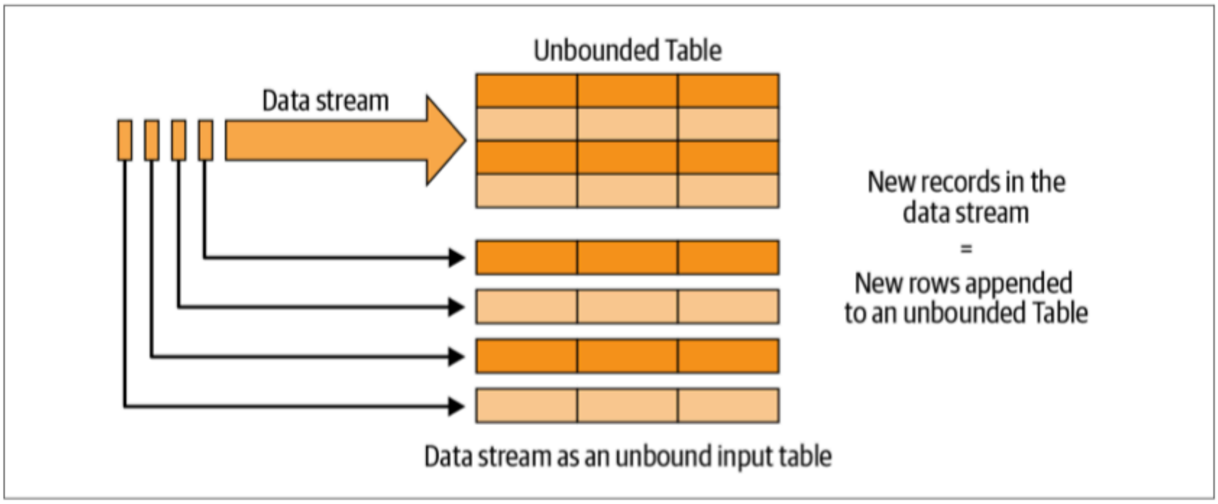

Figure 8-3. The Structured Streaming programming model: data stream as an unbounded table

Incrementalization. User defines batch-like query. Structured Streaming automatically converts the batch-like query to a streaming execution plan.

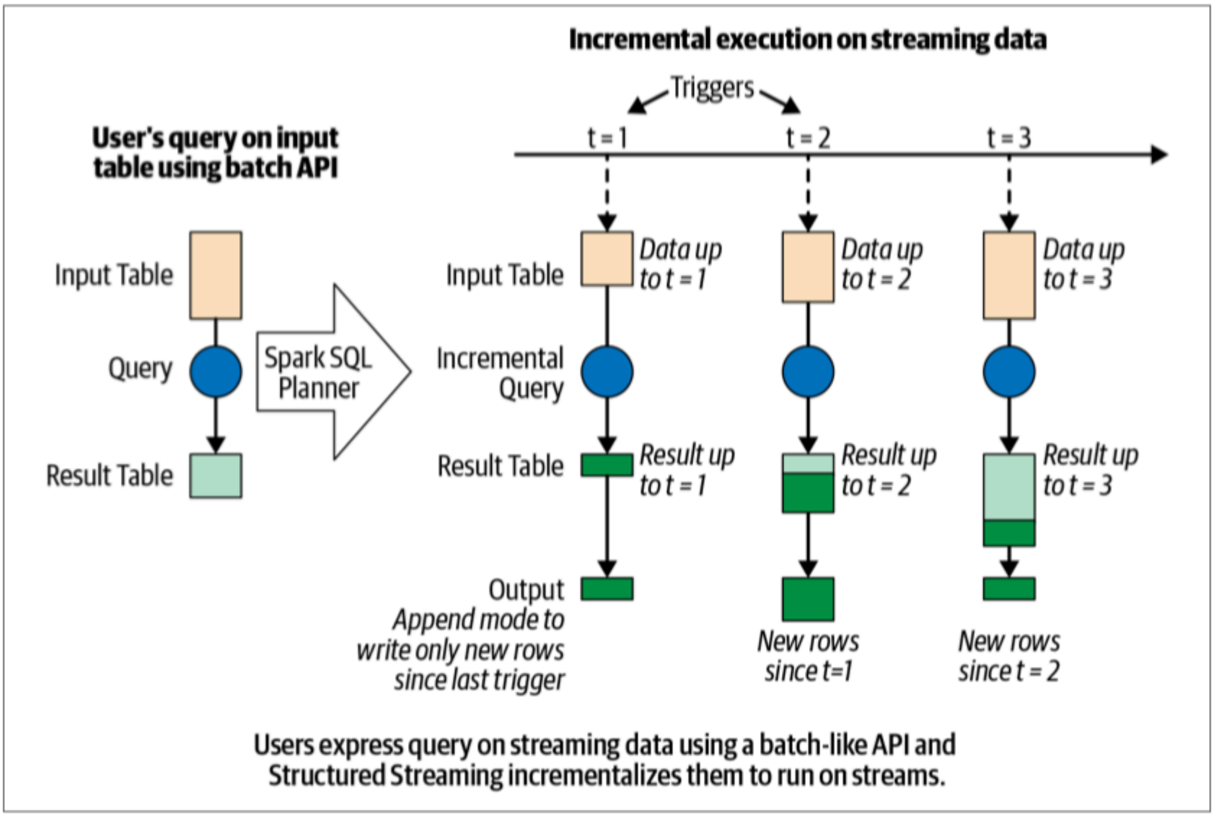

Figure 8-4. The Structured Streaming processing model

Output modes. How to update the streaming output.

- Append mode: Only new rows appended since the last trigger will be written to external storage.
- Update mode: Only rows updated since the last trigger will be changed in external storage.
- Complete mode: Entire updated result table will be written to external storage.

Hadoop Eco-system

Hadoop A framework/suite that enables processing of large data sets in clusters. Made up of several modules that form a eco-system.

HDFS

HDFS Hadoop Distributed File System. Component of the Hadoop eco-system responsible for storing large datasets, structured and unstructured, across various nodes.

Name node Nodes in HDFS that store metadata (data about data, e.g., schema).

Data node Nodes in HDFS that store the actual data.

YARN

YARN Yet Another Resource Negotiator. Performs scheduling and resource allocation for the Hadoop systems.

Resource manager Allocates resource for applications in the cluster.

Nodes manager Allocates resource for CPU, memory for each machine.

Application manager Negotiates resource for an application with resource manager.

MapReduce

MapReduce Data processing paradigm that uses distributed and parallel algorithms to process data in distributed manner.

Map Breaks data into partitions.

Reduce Produces output by aggregating intermediate results in each partition.

PIG

PIG A high-level (MapReduce) platform for creating programs that run on Hadoop. Can execute its Hadoop jobs in MapReduce, Spark, etc.

PIG Latin Language for the PIG platform. Abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of SQL for relational database management systems (RDBMS). Supports UDFs in Java, Python, etc.

HIVE

Hive A data warehouse software project built on top of Hadoop for providing data query and analysis.

Hive Query Language Hive's SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop.

Spark

Spark Engine for distributed computing in cluster. Supports in-memory data processing (faster than the original MapReduce framework which writes intermediate results to disk).

Others

Flume System for collecting large amounts of log data from many different sources to a centralized data store.

Zookeeper Synchronization service in distributed application.

Kafka Distributed streaming platform.

[redacted]