BITS Pilani
Pilani | Dubai | Goa | Hyderabad

DSECL ZG 522: Big Data Systems

Session 16: Spark -Part 4

Janardhanan PS

Professor

janardhanan.ps@wilp.bits-pilani.ac.in

Topics for today

- **Spark performance**
- Spark ML performance
- Stream processing
- Spark Streaming

Shuffle

reduceByKey() Example

- The reduceByKey operation generates a new RDD
- For this:
  - ✓ all values for a single key are combined into a tuple
  - ✓ by executing a reduce function against all values associated with that key

- For example,
  - ✓ Let pair RDD is { (1,2), (3,4), (3, 6) }
  - ✓ Apply reduceByKey() on rdd ---→ rdd. reduceByKey((x,y) => x + y)
  - ✓ Result will be

```
{
        (1, 2)
        (3, 10)
}
```

- The challenge is that not all values for a single key necessarily reside
  - ✓ on the same partition
  - ✓ or even the same machine
  - ✓ all keys don't have same number of values
- but they must be co-located to compute the result

Shuffle Operations

- Certain operations within Spark trigger an event known as the shuffle
- The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions
- This typically involves copying data across executors and machines, making the shuffle a complex and costly operation
- In Spark, data is generally not distributed across partitions to be in the necessary place for a specific operation
- During computations, a single task will operate on a single partition - thus, to organize all the data for a single reduceByKey reduce task to execute, Spark needs to perform an all-to-all operation
- It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key - this is called the shuffle.
- Additional operations which can cause a shuffle include
  - ✓ repartition operations like repartition and coalesce
  - ✓ 'ByKey operations (except for counting) like groupByKey and reduceByKey
  - ✓ join operations like co-group and join.

RDD Persistence

- One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations

- When you persist an RDD
  - ✓ every node stores any partitions of it that it computes in memory
  - ✓ reuses them in other actions on that dataset (or datasets derived from it)
  - ✓ allows future actions to be much faster (often by more than 10x)

- Caching is a key tool for iterative algorithms and fast interactive use.

- Can mark an RDD to be persisted using the persist() or cache() methods on it
  - ✓ The first time it is computed in an action, it will be kept in memory on the nodes
  - ✓ Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

RDD *storage level*

- Each persisted RDD can be stored using a different storage level, allowing you,
  - ✓ to persist the dataset on disk
  - ✓ to persist it in memory but as serialized Java objects (to save space)
  - ✓ to replicate it across nodes

- These levels are set by passing a StorageLevel object (Scala, Java, Python) to persist()
  - ✓ The cache() method is a shorthand for using the default storage level, StorageLevel.MEMORY_ONLY

RDD *storage level (2)*

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

Which Storage Level to Choose?

- Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency.

- Recommended process
  - ✓ If your RDDs fit comfortably with the default storage level (MEMORY_ONLY), leave them that way.
    - Most CPU-efficient option, allowing operations on the RDDs to run as fast as possible
  - ✓ If not, try using MEMORY_ONLY_SER and selecting a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access. (Java and Scala)
  - ✓ Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data
  - ✓ Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application)
    - All the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.

Removing Data

- Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion.

- If you would like to manually remove an RDD instead of waiting for it to fall out of the cache
  - ✓ use the RDD.unpersist() method.

Topics for today

- Spark performance
- **Spark ML performance**
- Stream processing
- Spark Streaming

MLlib: Tips and Performance Considerations

- Preparing Features
- Configuring Algorithms
- Caching RDDs to reuse
- Recognizing Sparsity
- Level of Parallelism

Preparing Features

- Usually too much emphasis given on the algorithms but features are more critical
- Each algorithm is only as good as the features put into it!
- Feature preparation is the most important step to large-scale machine learning!

- Improvements into results can be obtained by
  - ✓ Adding new more informative features
  - ✓ Converting available features to suitable vector representations

- Common tips:
  - ✓ Scale your features
  - ✓ Featurize text correctly (can think of use of external libraries like NLTK)
  - ✓ Label classes correctly

Configuring Algorithms

- Most algorithms perform better in terms of accuracy with regularization when its applicable and available
  - ✓Models often learn random irrelevant behaviour from test data
  - ✓Regularization avoid overfitting by making models simpler, penalising large coefficients etc.
- Though default values works well but
  - ✓ Try changing the values from default to see if improves the accuracy
  - ✓ Make sure evaluating these changes on test data as well

Other Tips

- Caching RDDs to reuse
  - ✓ Most algorithms are iterative in nature, needs to pass over data sets multiple times
  - ✓ Important to cache the dataset before passing to MLLib
  - ✓ If does not fit in memory, persist on disk

- Recognizing Sparsity
  - ✓ Check whether the feature vector contains mostly zeros
  - ✓ Storing in the sparse format can save space as well as time for processing

- Level of Parallelism
  - ✓ Have at least as many partitions in input RDD as the number of cores on cluster
  - ✓ Can specify the number of partitions while reading data
  - ✓ Can use repartition method on RDD to partition it into more pieces
  - ✓ Be careful – adding more partitions increases communication overhead!

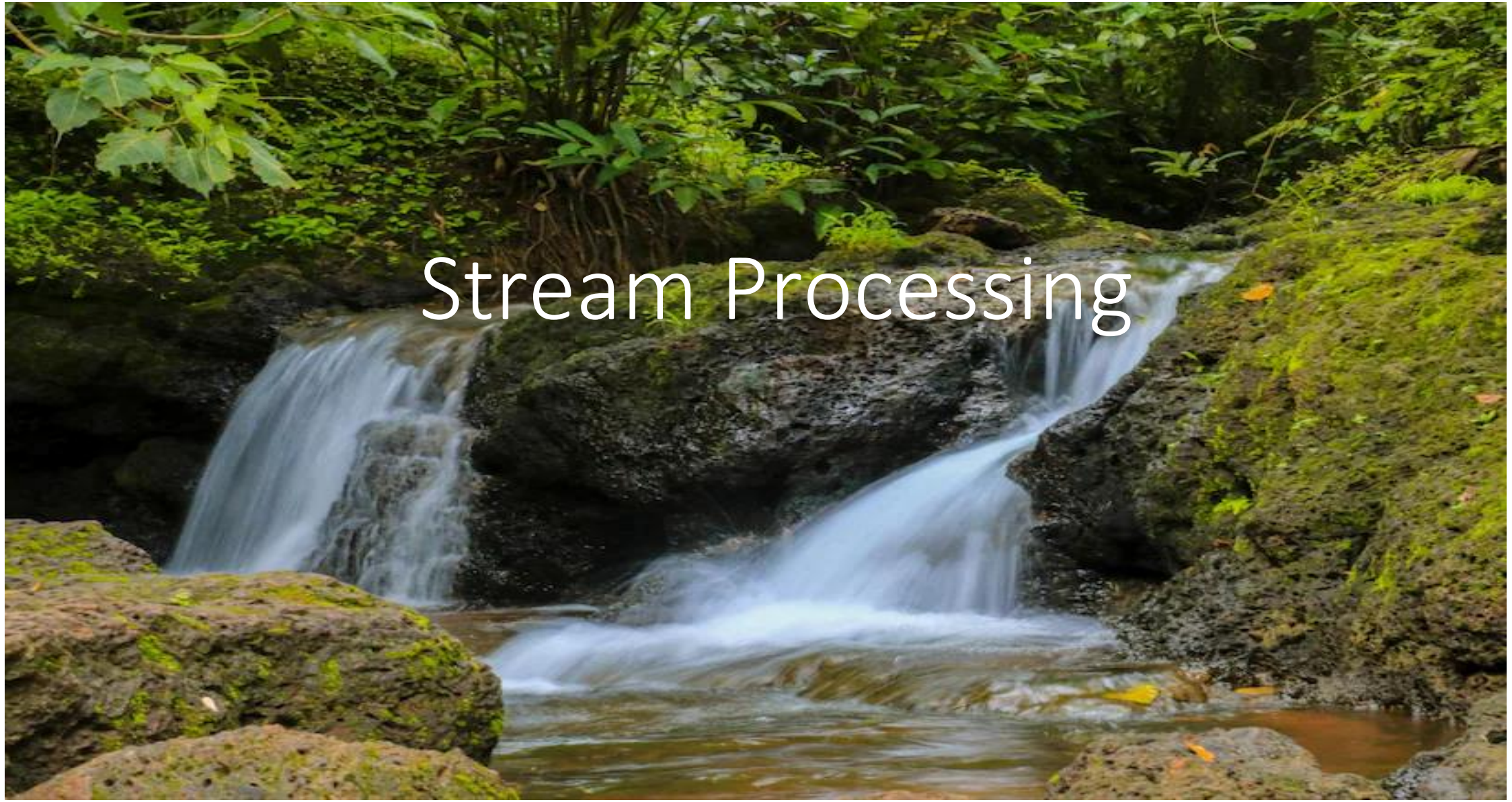
Hardware provisioning tips

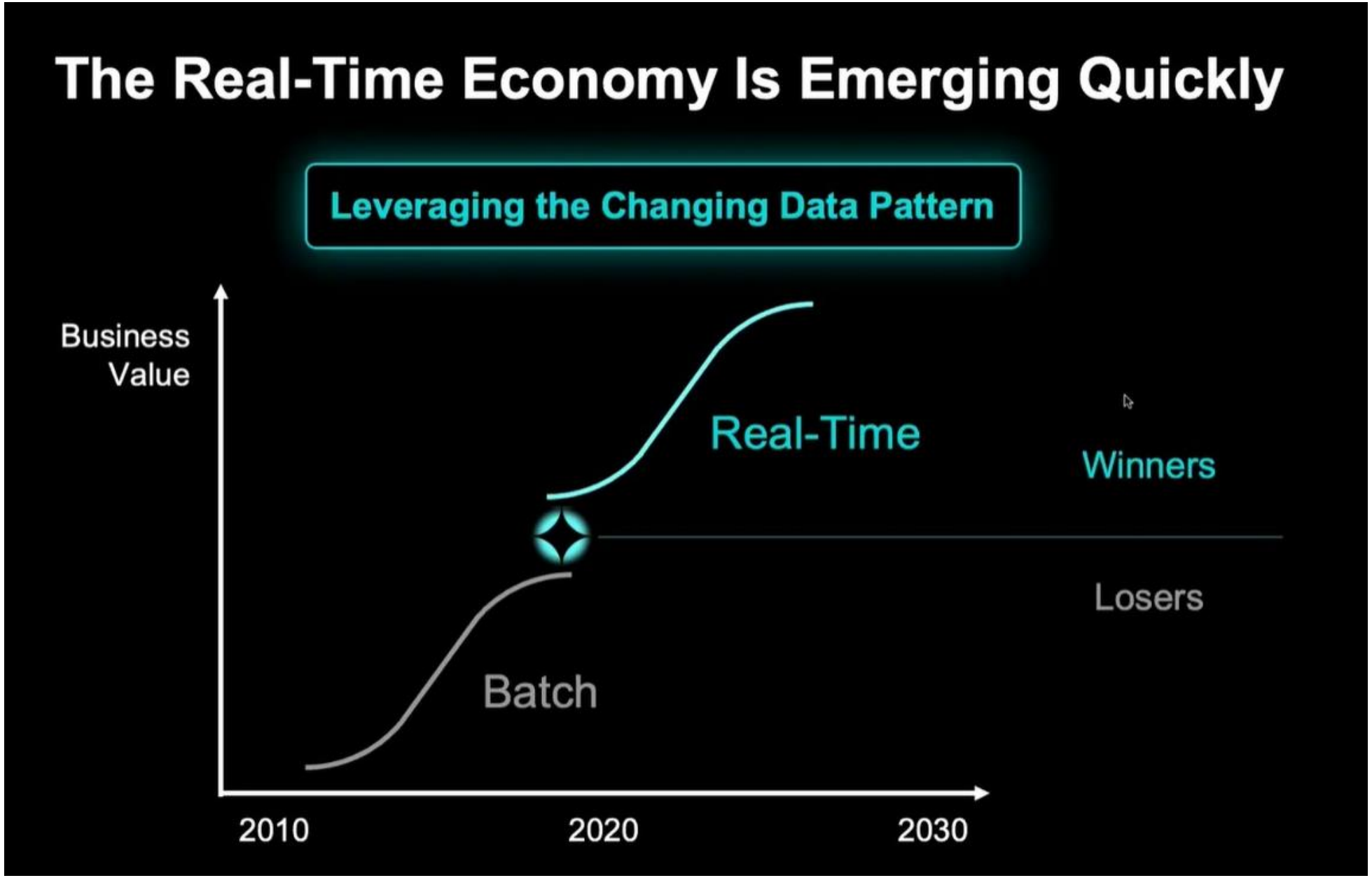
- Run closer to data: If you are using HDFS data then make it is on the same network or run Spark on same nodes, preferably using YARN.
- Storage: Use 4-8 disks per node with separate mount points with no RAID. configure spark.local.dir with these mount points. If using HDFS then use the same disks.
- Memory: Configure 8GB+ on nodes and 25% is typically taken up by OS and buffer cache.
  - ✓ How much memory is needed ? Load typical datasets you use into RDD and check memory taken up using spark UI (on http:<driver machine>: 4040)
  - ✓ Typically JVM memory should be < 200GB. If machine has more then run multiple executors.
- Network: With shuffle / distributed reduce operations need 10GB+ network.
- CPU cores: At least 8-16 cores per machine because typically threads will run in parallel CPU-bound once data is in memory. So CPU and network are heavily used. Observe utilisation and can scale cores as well as add more nodes.

https://spark.apache.org/docs/latest/hardware-provisioning.html

Topics for today

- Spark Performance
- Spark ML Performance
- **Stream processing**
- Spark Streaming


Stream Processing


The Real-Time Economy Is Emerging Quickly
Leveraging the Changing Data Pattern
Business Value
Real-Time
Winners
Losers
Batch
2010
2020
2030

Stream Processing

- ✓ Insights are more valuable immediately after events happen
  - (Near) real-time: from milliseconds to seconds, or minutes
- ✓ It allows faster reaction
  - Detecting patterns, setting alerts
- ✓ Some data is naturally unbounded (e.g. sensor data)
- ✓ Resource constraints (storage and compute)
  - Process large large volumes of data arriving at high velocities
  - Retain only what is useful
- ✓ Continuous processing

Applications of Stream processing

- Computing
  - ✓Log analysis,
  - ✓Detection of DoS attacks,
  - ✓Scaling service capacities
- Real-time monitoring
  - ✓Fraud detection (credit cards),
  - ✓Intrusion detection (surveillance)
- Sensor data processing
  - ✓Weather,
  - ✓Transportation
  - ✓Traffic
  - ✓Patient health
- Social media
  - ✓Trend analysis
- Industry
  - ✓Process optimization
  - ✓Predictive maintenance
  - ✓Logistics
- Advertising and promotions
  - ✓Contextualized to user behavior or geolocation
- Financial trading
  - ✓Algorithmic trading
  - ✓Risk analysis

Streaming Data Systems

- Layered systems that rely on several loosely coupled systems to process continuously collected data
  - Helps in achieving high availability
  - Helps in managing the system
  - Helps in maintaining the cost under control

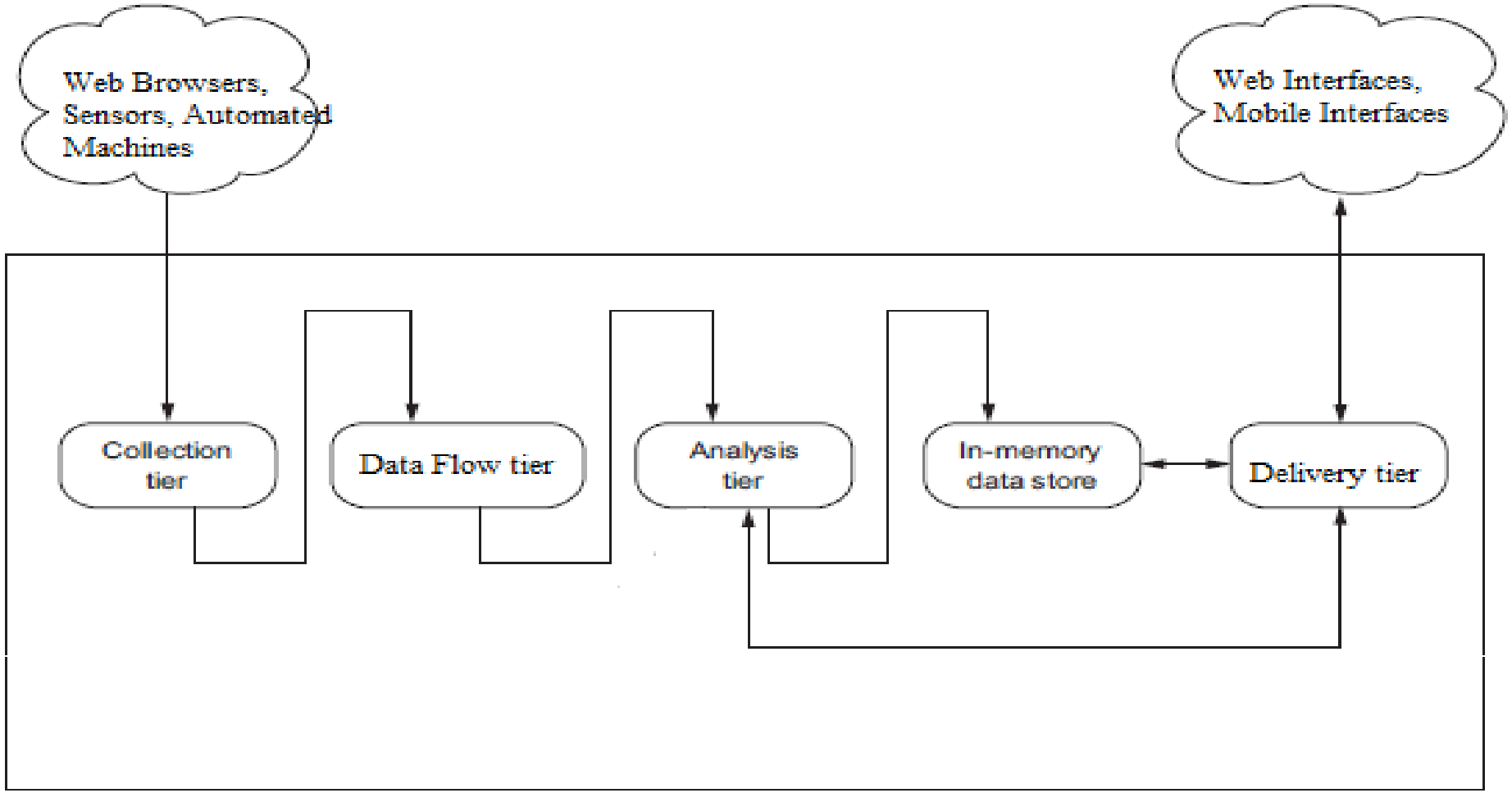
- All subsystems / components can reside on individual physical servers or can be co hosted on the single or more than one servers
- Not all components to be present in every system

Contents adapted from Real-Time Analytics , Byron Ellis

Streaming Data System Components

- Streaming Data System Architecture Components
  - Collection
  - Data Flow
  - Processing
  - Storage
  - Delivery

Generalized Architecture



Altered version, original concept : Andrew G. Psaltis

Architecture Components (1)

Collection System

- Mostly communication over TCP/IP network using HTTP
- Websites log data was the initial days use case
- W3C standard log data format was used
- Newer formats like JSON, AVRO, Thrift are available now

- Collection happens at specialized servers called edge servers
- Collection process is usually application specific
- New servers integrates directly with data flow systems
- Old servers may or may not integrate directly with data flow systems

Architecture Components (2)

Data Flow Tier

- Separation between collection tier and processing layer is required
  - Rates at which these systems works are different
  - What if one of system is not able to cope with another system?

- Required intermediate layer that takes responsibility of
  - accepting messages / events from collection layer
  - providing those messages / events to processing layer

- Real time interface to data layer for both producers and consumers of data
- Helps in guaranteeing the “at least once” semantics

Architecture Components (3)

Processing / Analytics Tier

- Based on “data locality” principle
- Move the software / code to the location of data
- Rely on distributed processing of data
- Framework does most of the heavy lifting of data partitioning, job scheduling, job managing
- Available Frameworks
  - Apache Storm
  - Apache Spark (Streaming)
  - Apache Kafka Streaming etc

Architecture Components (4)

Storage Tier

- In memory or permanent
- Usually in memory as data is processed once
- But can have use cases where events / outcomes needs to be persisted as well

- NoSQL databases becoming popular choice for permanent storage
  - MongoDB
  - Cassandra

- But usage varies as per the use case, still no database that fits all use cases

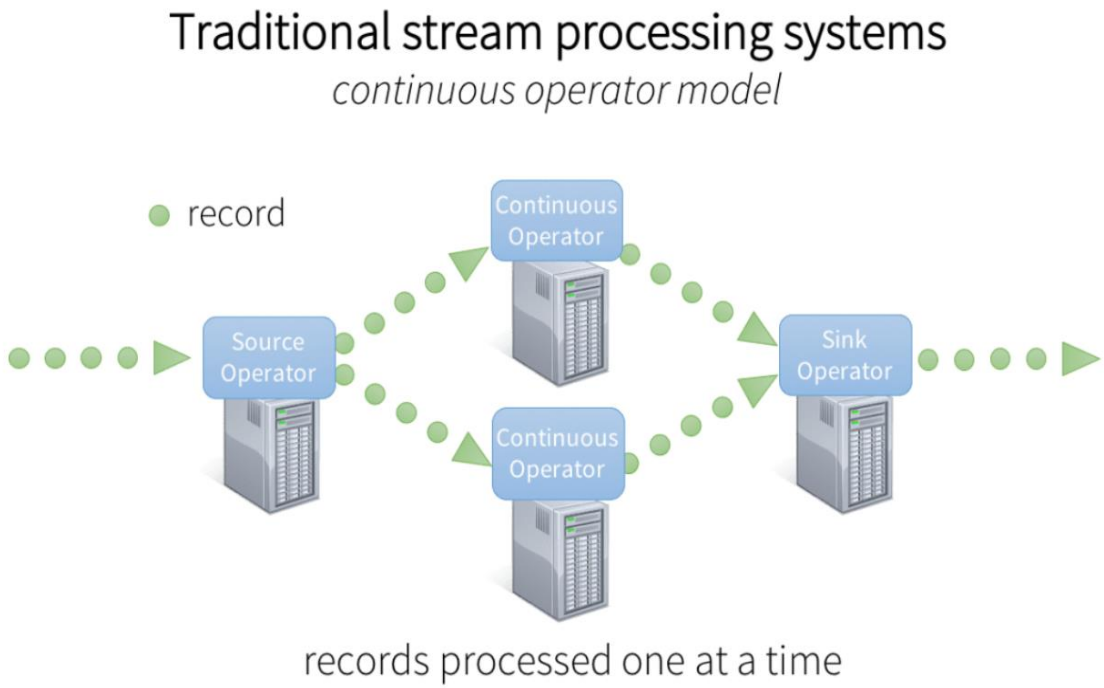
Architecture Components (5)

Delivery Layer

- Usually web based interface
  - Now a days mobile interfaces are becoming quite popular
  - Dashboards are built with streaming visualizations that gets continuously updated as underlying events are processed
  - HTML + CSS + Java script + Websockets can be used to create interfaces and update them
  - HTML5 elements can be used to render interfaces
  - SVG, PDF formats used to render the outcomes
- Monitoring / Alerting Use cases
- Feeding data to downstream applications

Continuous operator model

- A simple and natural model
  - ✓ Streaming data is received from data sources (e.g. live logs, system telemetry data, IoT device data, etc.) into some data ingestion system like Apache Kafka, Amazon Kinesis, etc.
  - ✓ The data is then processed in parallel on a cluste
  - ✓ Results are given to downstream systems like HBase, Cassandra, Kafka, etc.
  - ✓ Data is received from ingestion systems via Source operators and given as output to downstream systems via sink operators
  - ✓ Cluster has set of worker nodes, each of which runs one or more continuous operators
  - ✓ Each continuous operator processes the streaming data one record at a time and forwards the records to other operators in the pipeline



Continuous operator model - Challenges

- Fast Failure and Straggler Recovery
  - ✓ System must be able to quickly and automatically recover from failures and stragglers to provide results
  - ✓ Difficult to achieve due to the static allocation of continuous operators to worker nodes

- Load Balancing
  - ✓ The system needs to be able to dynamically adapt the resource allocation based on the workload
  - ✓ Uneven allocation of the processing load between the workers can cause bottlenecks

- Unification of Streaming, Batch and Interactive Workloads
  - ✓ System might have requirements to query the streaming data interactively, or to combine it with static datasets
  - ✓ Hard in continuous operator systems which are not designed for ad-hoc queries
  - ✓ This requires a single engine that can combine batch, streaming and interactive queries

- Advanced Analytics with Machine learning and SQL Queries
  - ✓ Complex workloads require continuously learning and updating data models, or even querying the streaming data with SQL queries.
  - ✓ Having a common abstraction across these analytic tasks makes the developer's job much easier.

Topics for today

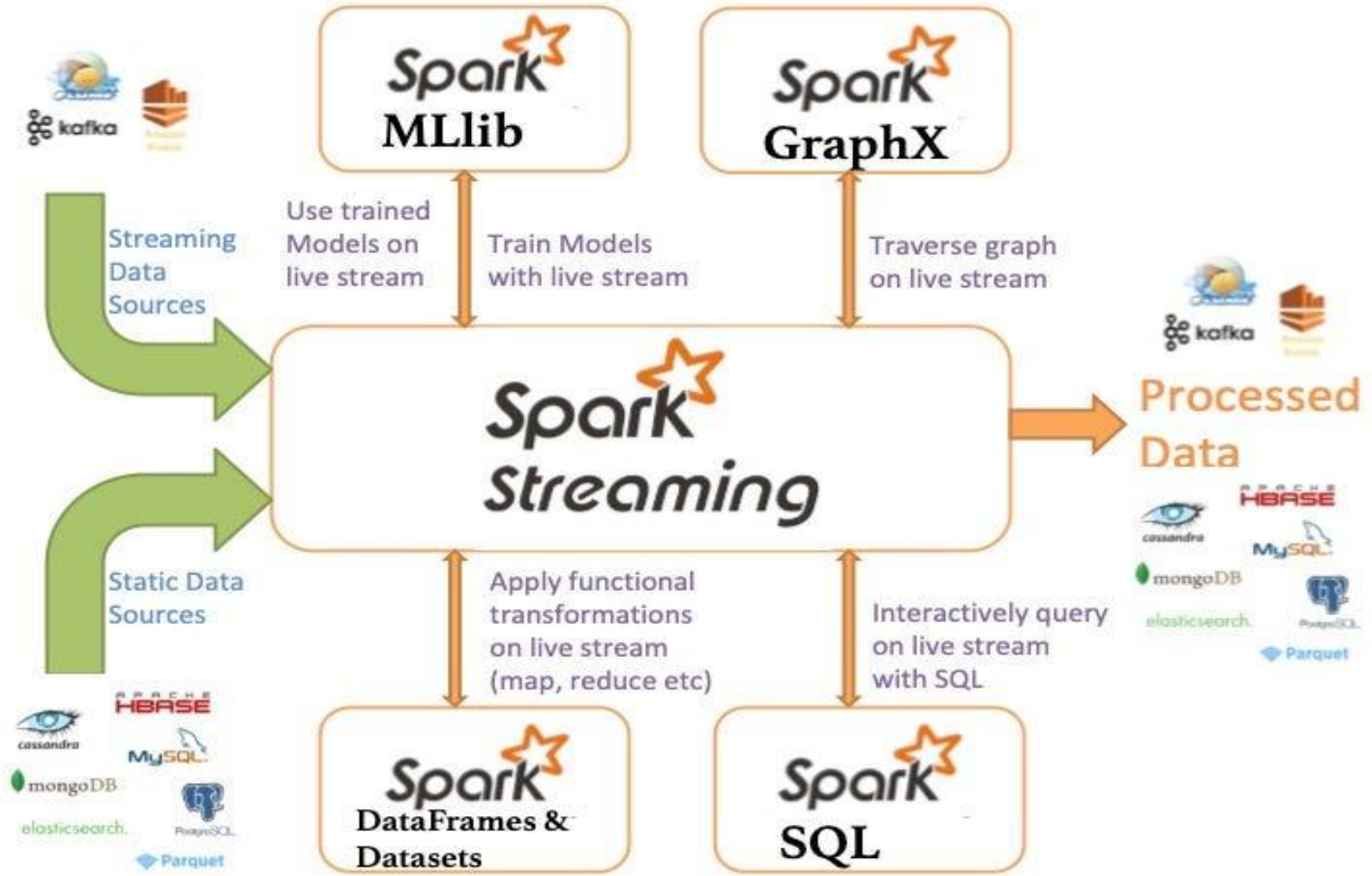
- Spark Performance
- Spark ML performance
- Stream processing
- **Spark Streaming**

Why Spark Streaming?

Spark provides

- Fast recovery from failures and stragglers
- Better load balancing and resource usage
- Data from streaming sources can combine with a very large range of static data sources available through Apache Spark SQL
- Provides the unification of disparate data processing capabilities, e.g. streaming with interactive, batch etc.
  - Has a unified engine that natively supports both batch and streaming workloads
  - Makes it very easy for developers to use a single framework to satisfy all the processing needs
- Native integration with advanced libraries, such as ML, graph, SQL

Spark Streaming



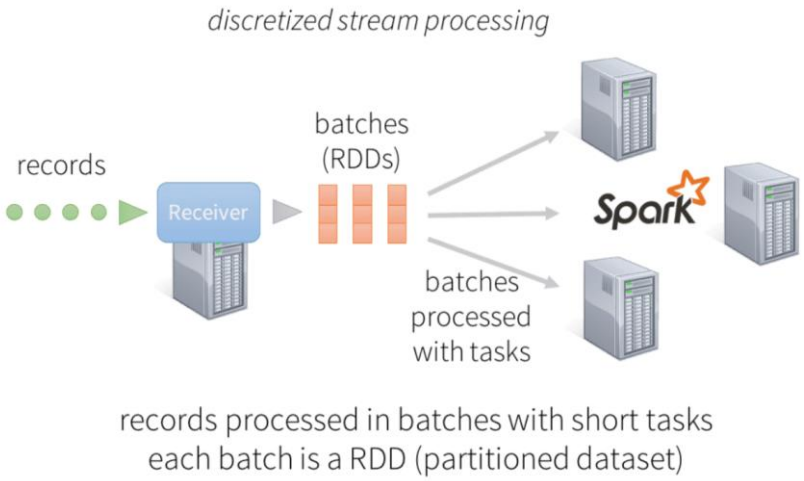
Overview

- An extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
- Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window
- Processed data can be pushed out to filesystems, databases, and live dashboards
- Can apply Spark's machine learning and graph processing algorithms on data streams

https://spark.apache.org/streaming/

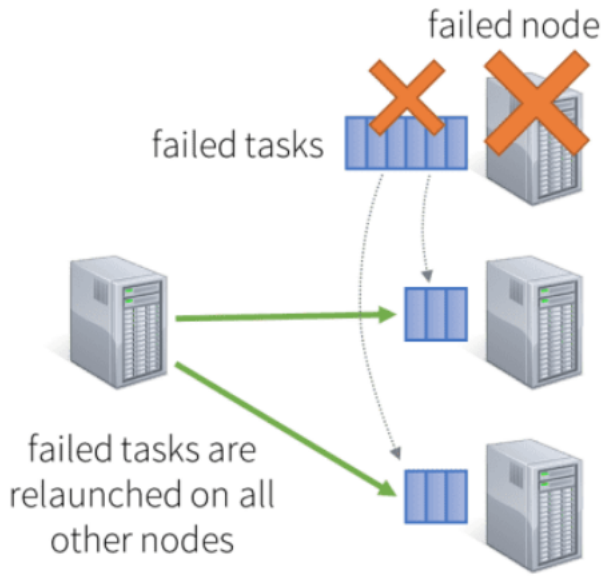
Discretized Streams

- Live input data streams are divided into batches
- These batches are then processed to generate the final stream of results in batches.
- DStreams
  - Provides a high-level abstraction which represents a continuous stream of data
  - Can be created either from input data streams from sources such as Kafka, Flume, and Kinesis or by applying high-level operations on other DStreams
  - Internally, a DStream is represented as a sequence of RDDs

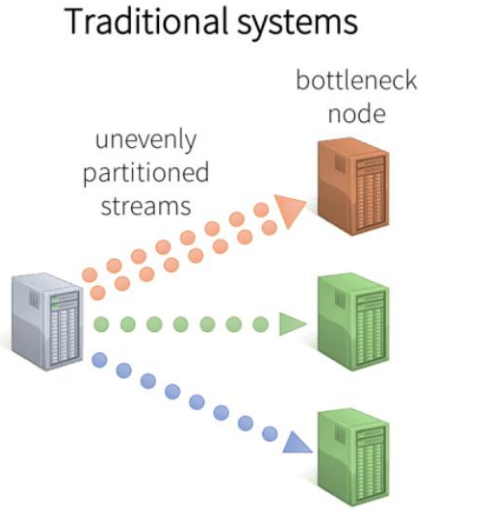


- https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

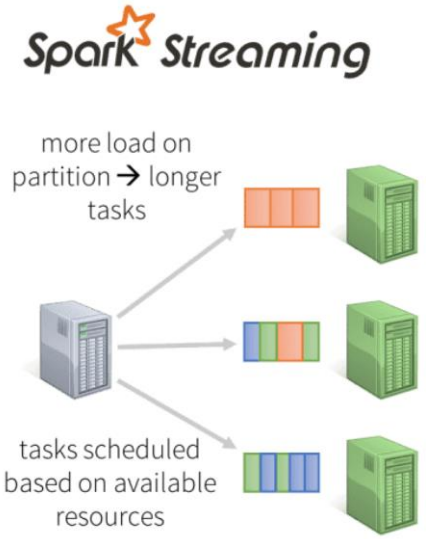
Benefits of batching (1)



faster recovery by using multiple nodes for recomputations



static scheduling of continuous operators to nodes can cause bottlenecks



dynamic scheduling of tasks ensures even distribution of load

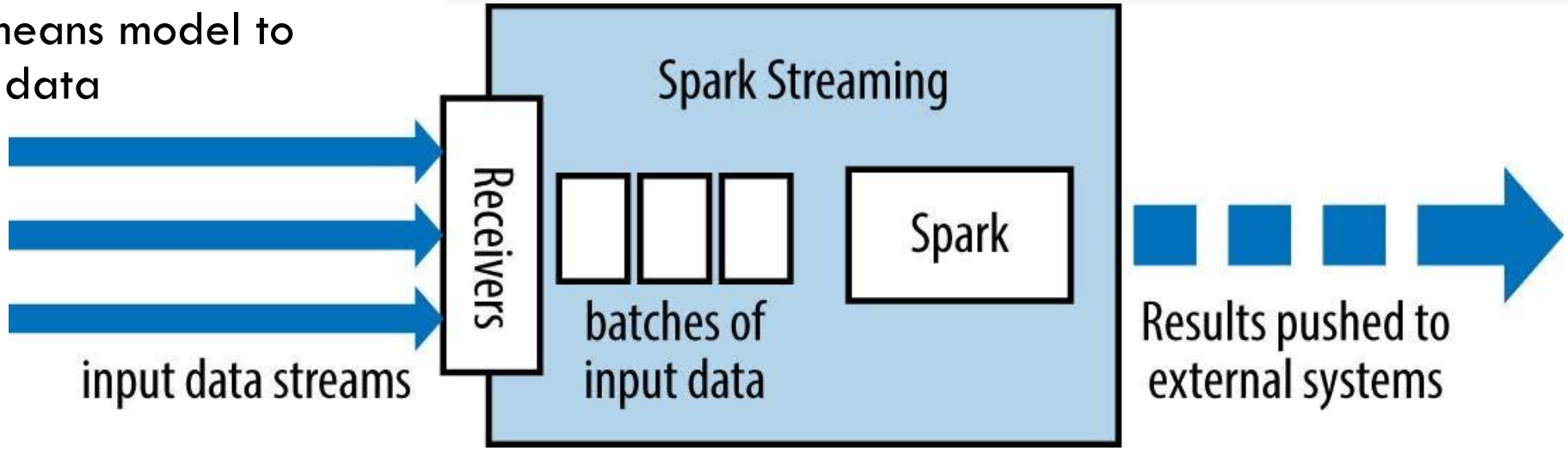
- Faster recovery from failures
- Dynamic scheduling of batches / tasks leads to better load balancing

Benefits of batching (2)

- Easier interoperability of batch, stream and interactive analysis
  - DStream is just a series of RDDs
  - E.g. join a DStream with static RDD or convert into DataFrame and interactively query using SQL
- Can apply MLlib functions on DStream RDDs.
  - E.g. apply a k-means model to label streaming data

```
// Create data set from Hadoop file
val dataset = sparkContext.hadoopFile("file")
// Join each batch in stream with the dataset
kafkaDStream.transform { batchRDD =>
  batchRDD.join(dataset).filter(...)
}
```

```
// Learn model offline
val model = KMeans.train(dataset, ...)
// Apply model online on stream
val kafkaStream = KafkaUtils.createDStream(...)
kafkaStream.map { event => model.predict(featurize(event)) }
```



Spark Streaming Sources & Receivers

- Every input DStream (except file stream) is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing.
- Basic sources
  - ✓ These are the sources directly available in the StreamingContext API.
  - ✓ Examples: file systems, and socket connections
- Advanced sources
  - ✓ Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes
  - ✓ These require linking against extra dependencies
- Reliable Receiver
  - ✓ Is the one that correctly sends an acknowledgment to a source when the data receives and stores in Spark with replication
- Unreliable Receiver
  - ✓ Is the one that does not send an acknowledgment to a source

Spark Streaming Operations

Transformation Operations

- Spark transformations allow modification of the data from the input Dstream

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

Source : https://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

Spark Streaming Operations(2)

Output Operations

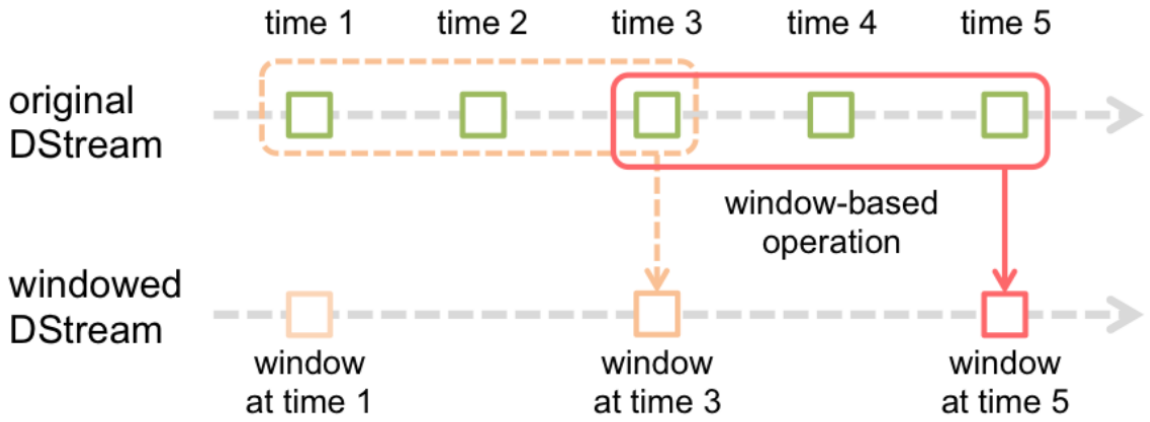
- Output operations allow DStream's data to be pushed out to external systems like a database or a file systems

Output Operation	Meaning
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as `SequenceFiles` of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

Source : https://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations-on-dstreams

Window operations (1)

- Batch interval : Time period taken to batch data into an RDD
- Window size : How much data will be in RDD before processing
  - ✓ e.g. 3 RDDs in this example
- Sliding interval
  - ✓ How much will the window slide, e.g. 2 in this example



Window operations (2)

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel.
reduceByKeyAndWindow(*func*, *windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKeyAndWindow(*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*])	A more efficient version of the above `reduceByKeyAndWindow()` where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable only to "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*). Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument. Note that checkpointing must be enabled for using this operation.
countByValueAndWindow(*windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.

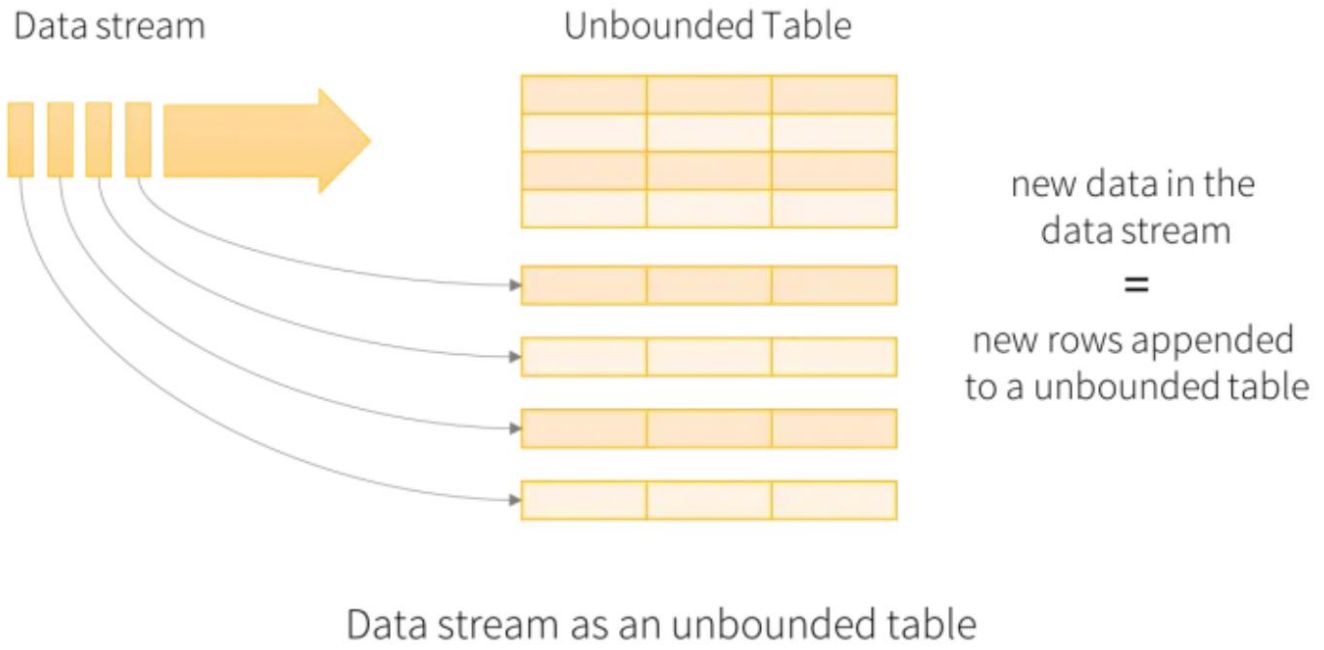
Structured Streaming

- Structured Streaming treats a live data stream as a table that is being continuously appended.
- This leads to a new stream processing model that is very similar to a batch processing model
- Express streaming computation as standard batch-like query as on a static table
- Spark runs it as an incremental query on the unbounded input table
- Structured Streaming does not materialize the entire table.
- It reads the latest available data from the streaming data source
- Processes data incrementally to update the result
- After processing, it discards the source data.
- Spark is responsible for updating the Result Table when there is new data

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Structured Streaming

- Works on DataSets/DataFrames API
- No batching
- Data is processed as rows because it is structured
- Closer to real streaming because rows are smaller than an RDD



Data stream as an unbounded table

Comparison of Spark with Kafka

	Apache Spark	Apache Kafka
Description	General purpose distributed processing system used for big data workloads. It uses in-memory caching and optimized query execution to provide fast analytic queries against data of any size.	Distributed Streaming platform that allows developers to create applications that continuously produce and consume data streams.
Processing Model	Batch processing and streaming	Event Streaming
Throughput & Latency	High throughput, low latency	High throughput, low latency
Scalability	Horizontally scalable as a distributed system, though scaling is expensive due to RAM requirement	Horizontally scalable to support a growing number of users and use cases, meaning that it can handle an increasing amount of data by adding more nodes & partitions to the system
Fault Tolerance	Fault tolerant by nature (RDDs). In case of node failure, RDDs are automatically recreated.	Fault-tolerant as it replicates data across multiple servers and can automatically recover from node failures.

Thank you

Post mid-sem topics review

1. What kind of store / DB to use for a given use case - e.g. what needs eventual consistency, strong consistency, key-value etc.
2. Storage layouts for different NoSQL stores - how would we model a data set in a NoSQL store - what about normalisation, joins etc.
3. Cloud storage solutions - where would you use object store vs block storage vs file storage, architectural concepts behind Dynamo
4. Spark internal architecture - resiliency and distribution, driver and cluster
5. Different types of operations in Spark - you should be aware as you write a Spark program on where shuffles happen and why a DAG is used, memory/storage optimisations discussed in class
6. Write basic Spark ML programs - know the basic flow and which functions solve which use cases