CS4225

AY24/25 SEM 1

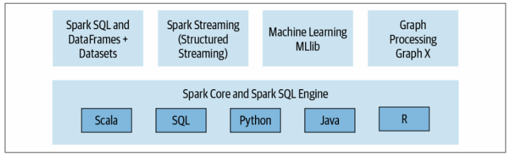
github/Gavino3o

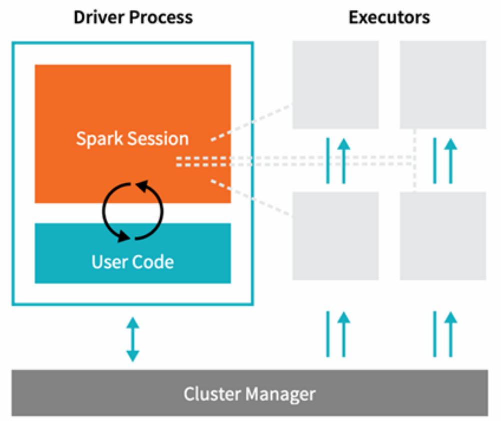
Apache Spark

Hadoop vs Spark

- **Hadoop**: Disk-based, MapReduce, HDFS. Not suitable for iterative algorithms as it incurs network and disk I/O overhead for intermediate data.
- **Spark**: In-memory, DAG, RDDs. In the event memory is insufficient, Spark spills data from memory to disk.

Spark Architecture and APIs


Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R



- **Driver Process**: Manages the execution of the Spark job. Responds to user inputs. Distribute work to the executors.
- **Cluster Manager**: Manages the resources of the cluster. Eg. YARN, Mesos, Kubernetes.
- **Worker Node**: Runs the executors.
- **Executor**: Runs tasks and keeps data in memory or disk storage across them.
- **RDDs**: Resilient Distributed Datasets. A collection of JVM objects. Functional operators (map, filter, etc) are applied to RDDs to transform them.
- **DataFrames**: A distributed collection of data organized into named columns. Similar to a table in a relational database. Expression-based transformation operations. Logical plans and optimisers.
- **Datasets**: A distributed collection of data with a known schema. Combines the benefits of RDDs and DataFrames. Internally rows, externally JVM objects. Typs safe and fast.

RDDs

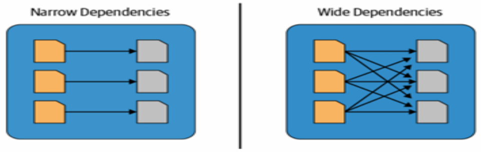
- **Resilient Distributed Datasets**
- Fault-tolerant collection of elements that can be operated on in parallel
- Immutable, partitioned collection of records
- Distributed across a cluster of machines.

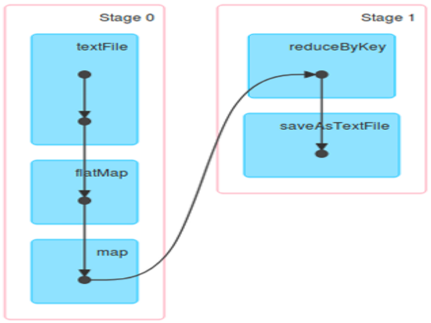
Transformations and Actions

- Transformations are operations that create a new RDD from an existing one.
- Lazy evaluation: Transformations are not executed immediately. They are only executed when an action is called. This allows Spark to optimise the execution plan.
- Example: `map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`, `join`, `order`, `select`
- Actions are operations that return a value to the driver program after running a computation on the dataset.
- Example: `collect`, `count`, `reduce`, `saveAsTextFile`, `foreach`, `take`, `show`
- Execution of transformations and actions are executed in parallel accross different worker machines as RDDs are distributed across different worker machines. Results are returned to the driver program in the final step.
- Caching: Persisting RDDs in memory across operations. Useful for iterative algorithms.
- `cache()` is a transformation that persists the RDD in memory.
- `persist(options)` is an action that allows for more control over the persistence of the RDD.
- `unpersist()` is an action that removes the RDD from memory.
- We should cache RDDs that are used multiple times in the computation or when it is expensive to recompute the RDD.
- If we did not cache the RDD, Spark will recompute the RDD each time it is used in an action.
- When worker nodes have insufficient memory, Spark may evict LRU RDDs from memory to disk.

Directed Acyclic Graph (DAG)

- A DAG is a graph with directed edges and no cycles.
- In Spark, the DAG is a logical representation of the computation.
- Transformations construct the DAG; actions execute the DAG.
- Spark optimises the DAG by combining operations and minimising data shuffling.
- Narrow dependencies: Each partition of the parent RDD is used by at most one partition of the child RDD. Example: `map`, `filter`, `flatMap`, `contains`
- Wide dependencies: Each partition of the parent RDD may be used by multiple partitions of the child RDD. Example: `groupByKey`, `join`, `reduceByKey`, `sortByKey`, `orderByKey`
- In the DAG, consecutive narrow transformations are combined into a single stage and executed on the same machines. Wide transformations are separated into different stages.
- Across stages, data is shuffled across the network, which involves writing intermediate data to disk.
- Spark tries to minimise the number of stages and the amount of data shuffled.
- **Lineage**: The sequence of transformations that lead to an RDD.
- If a worker node fails, Spark can recompute the lost partitions of an RDD using the lineage. Note: we only need to recompute the lost partitions, not the entire RDD.





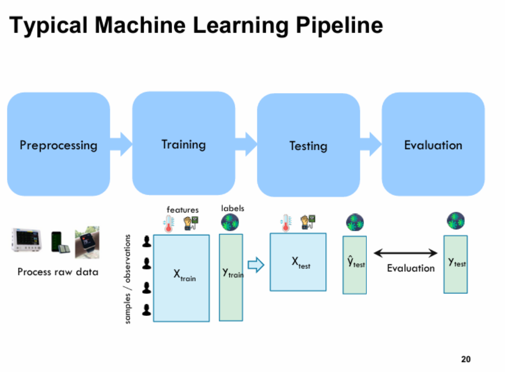
DataFrames

- A distributed collection of data organised into named columns.
- Similar to a table in a relational database.
- Easier to use than RDDs as it has a higher-level API.
- All Dataframe operations are still ultimately compiled down to RDD operation by Spark.
- Generally, transformation functions take in either strings or column objects.
- Transformations are still lazyly evaluated.

Spark SQL

- Spark SQL is a module for working with structured data.
- It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine.
- Spark SQL provides a domain-specific language for working with structured data.
- It allows running SQL queries on existing RDDs and DataFrames.
- Catalyst optimiser is the Spark SQL query optimiser. It takes a computational query and converts it into an optimised logical plan. Four Phases: Analysis, Logical Optimisation, Physical Planning, Code Generation (Project Tungsten).
- Multiple physical plans can be generated for a single logical plan. The optimiser chooses the best physical plan based on cost estimation.
- Project Tungsten is the Spark SQL execution engine. It aims to improve performance by optimising memory usage and CPU utilisation.
- Tungsten optimises memory usage by using binary processing, cache-aware computation, and code generation.
- **Unified API**: Spark SQL can be used with Java, Scala, Python, SQL, and R. It has one engine for all types of data processing.

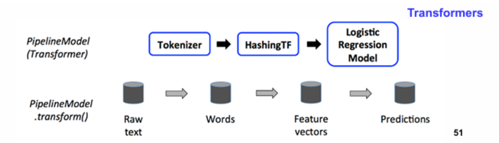
Machine Learning with Spark



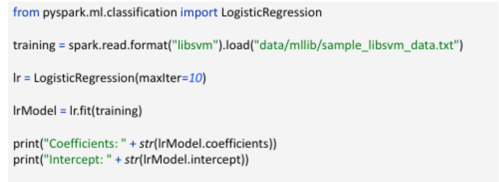
Typical Machine Learning Pipeline

- Data Quality: Missing values (impute, drop, add column indicating it is missing or not)
- Categorical Encoding: Convert categorical variables to numerical variables. Numerical values are often assigned in a way that represents the ordinal relationship between the categories or inherent order among the categories.
- One Hot Encoding: Convert categorical variables to binary vectors. Each category is represented by a binary vector. Useful when there is no ordinal relationship between the categories, and to ensure that the categorical variable does not imply any numerical relationship.
- Normalization: Scale the features to a standard range. Useful for algorithms that are sensitive to the scale of the input features. Example: clipping, log transform, standard scalerm, min-max scaler.
- Logistic Regression: A linear model for binary classification. It models the probability that the output is 1 given the input features. Ultilses the sigmoid function. $\sigma(x) = \frac{1}{1+e^{-x}}$
- $\hat{y} = \sigma(x \cdot w + b)$
- Cross Entroypy Loss: Measures the difference between two probability distributions. It is used as the loss function for logistic regression. $L(y, \hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$
- Gradient Descent: An optimisation algorithm that minimises the loss function. It iteratively updates the weights and biases in the direction of the negative gradient of the loss function.
- $w_{t+1} = w_t - \alpha \nabla_w L(w_t)$
- Evaluation Metrics: Accuracy, Precision, Recall, F1 Score, ROC Curve, AUC.
- Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$
- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F1 Score: $2 \times \frac{Precision \times Recall}{Precision + Recall}$
- Errors: Mean Squared Error, Mean Absolute Error, Root Mean Squared Error.
- Mean Squared Error: $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- Mean Absolute Error: $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- Root Mean Squared Error: $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$
- R Squared Value (0 to 1): Measures the proportion of the variance in the dependent variable that is predictable from the independent variable. The higher the better.

Pipelines

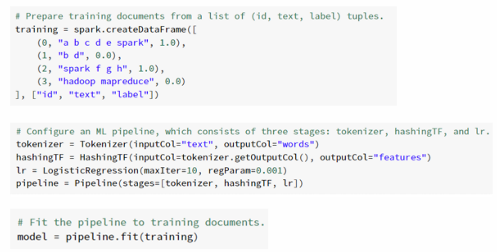

Transformers
PipelineModel (Transformer)
Tokenizer
HashingTF
Logistic Regression Model
PipelineModel .transform()
Raw text
Words
Feature vectors
Predictions
51

- Benefits: Better code reuse, Easier to perform cross validation, Easier to tune hyperparameters, Easier to productionise the model.
- Transformers are the building blocks of a pipeline.
- A transformer has a transform() method that takes in a DataFrame and returns a new DataFrame.
- Example: VectorAssembler, StringIndexer, OneHotEncoder, StandardScaler, LogisticRegression
- Generally, these transformers output a new DataFrame which append their result to the original DataFrame.
- A fitted model (e.g LogisticRegressionModel) is also a transformer. It transforms Dataframe by adding a prediction column.
- Estimators is an algorithm that takes in data, and outputs a fitted model. Example: A learnign algorithm like LogisticRegression can be fit to data, producing the trained logistic regression model.
- Estimators have a `fit()` method that takes in a DataFrame and returns a Transformer.



```
from pyspark.ml.classification import LogisticRegression

training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

lr = LogisticRegression(maxIter=10)

lrModel = lr.fit(training)

print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

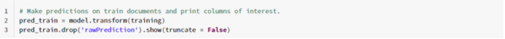
- Pipelines are a sequence of stages. Each stage is either a Transformer or an Estimator.
- Pipeline itself is an Estimator. It has a `fit()` method that takes in a DataFrame and returns a PipelineModel.
- When `fit()` is called on a Pipeline, the stages are executed in order. For Transformers, the `transform()` method is called. For Estimators, the `fit()` method is called.
- The output of `Pipeline.fii()` is a PipelineModel, which is a Transformer, and consists of a series of Transformers.
- The `transform()` method of the PipelineModel applies the fitted model to the input DataFrame.



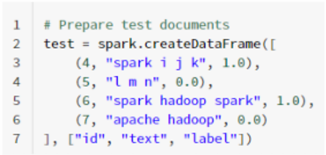
```
# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])
```

```
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

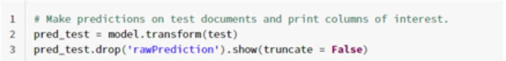
```
# Fit the pipeline to training documents.
model = pipeline.fit(training)
```



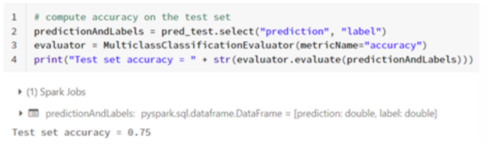
```
# Make predictions on train documents and print columns of interest.
pred_train = model.transform(training)
pred_train.drop('rawPrediction').show(truncate = False)
```



```
# Prepare test documents
test = spark.createDataFrame([
    (4, "spark i j k", 1.0),
    (5, "l m n", 0.0),
    (6, "spark hadoop spark", 1.0),
    (7, "apache hadoop", 0.0)
], ["id", "text", "label"])
```



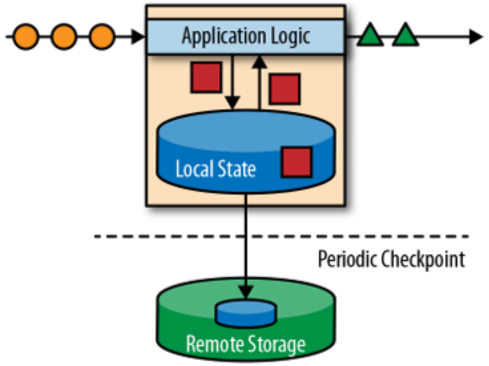
```
# Make predictions on test documents and print columns of interest.
pred_test = model.transform(test)
pred_test.drop('rawPrediction').show(truncate = False)
```



```
# compute accuracy on the test set
predictionAndLabels = pred_test.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

```
▸ (1) Spark Jobs
▸ predictionAndLabels:  pyspark.sql.dataframe.DataFrame = [prediction: double, label: double]
Test set accuracy = 0.75
```

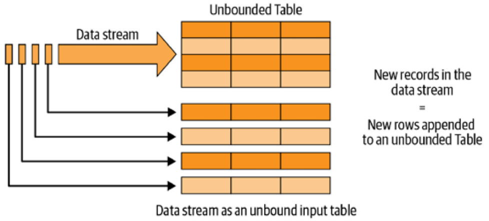
Stream Processing



Goal: Process data in real-time as it is generated. State can be stored and accessed in many different places including program variables, local files, or embedded/external databases.

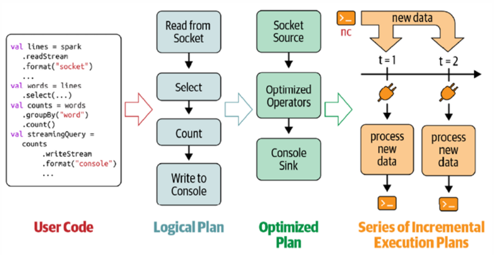
Micro-Batch Stream Processing

- Spark Structured Streaming uses a micro-batch processing model.
- Data from the input stream is divided into micro batches, each of which will be processed in the Spark cluster in a distributed manner.
- Small deterministic tasks generate the output of each micro-batch. Time is divided into small intervals, and data is processed in each interval.
- Advantages: quick; recover from failures efficently; deterministic nature ensures end-to-end exactly-once processing.
- Disadvantages: latency (cannot handle millisecond); micro-batch size affects latency and throughput; micro-batch processing can be less efficient than true stream processing.
- It is sufficient for most use cases.
- In Spark Structured Streaming, the input data is treated as an unbounded table.


Unbounded Table
Data stream
New records in the data stream
=
New rows appended to an unbounded Table
Data stream as an unbound input table

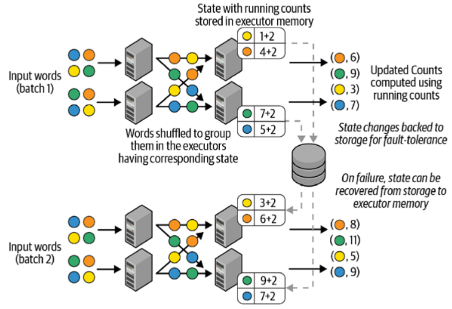
- Five Steps to Define Steaming Query:
  1. Define the input source.
  2. Transform Data.
  3. Define output sink and output mode.
  4. Specifying processing details. (Triggering details, checkpointing, etc)
  5. Start the query.

Incremental execution of streaming queries



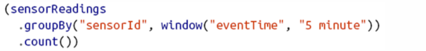
```
val lines = spark
  .readStream
  .format("socket")
  ...
val words = lines
  .select(...)
val counts = words
  .groupBy("word")
  .count()
val streamingQuery =
  counts
    .writeStream
    .format("console")
    ...
```

Distributed state management in Structured Streaming

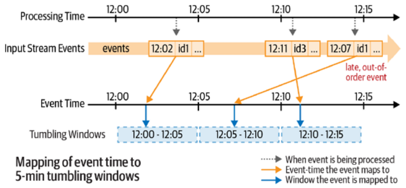

State with running counts stored in executor memory
Input words (batch 1)
1+2
4+2
7+2
5+2
(●, 6)
(●, 9)
(●, 3)
(●, 7)
Updated Counts computed using running counts
Words shuffled to group them in the executors having corresponding state
State changes backed to storage for fault-tolerance
On failure, state can be recovered from storage to executor memory
Input words (batch 2)
3+2
6+2
9+2
7+2
(●, 8)
(●, 11)
(●, 5)
(●, 9)

Data Transformation

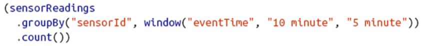
- **Stateless Transformations:**
  - Process each row individually without needing information from previous rows
  - Projection operations: `select()`, `explode()`, `map()`, `flatMap()`
  - Selection operations: `filter()`, `where()`
- **Stateful Transformations:**
  - Process each row based on information from previous rows.
  - Example: `DataFrame.groupBy().count()`
  - In every micro-batch, the incremental plan adds the count of new records to the previous count generated by the previous micro-batch
  - The state is maintained in the memory of the Spark executors and is checkpointed to the configured location to tolerate failures.
- **Stateful Streaming Aggregations:**
  - **Aggregations not based on time windows:**
    - Global aggregations: `groupBy().count()`
    - Grouped aggregations: `groupBy("sensorId").mean("value")`
    - Supported aggregations: `count()`, `sum()`, `avg()`, `min()`, `max()`, `countDistinct()`, `collect_set()`, `approx_count_distinct()`
  - Processing Time: The time at which the data is processed by the system. (Not deterministic, susceptible to system delays)
  - Event Time: The time at which the event occurred in the real world. (Deterministic)
  - Event time decouples the processing speeed from the results. An event time window computation will yield the same result regardless of the processing time.
  - Watermark: A threshold that determines how late the data can be in event time. Data that arrives later than the watermark is considered late data.
  - **Aggregations with Event-Time Windows:**
    - Example: `sensorReadings.groupBy("sensorId", window("eventTime", windowLength, shiftAmt)).count()`



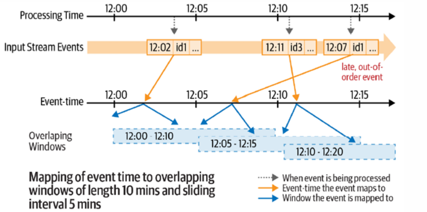
```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "5 minute"))
  .count())
```



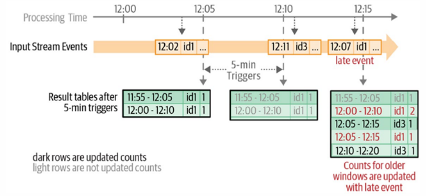
Mapping of event time to 5-min tumbling windows

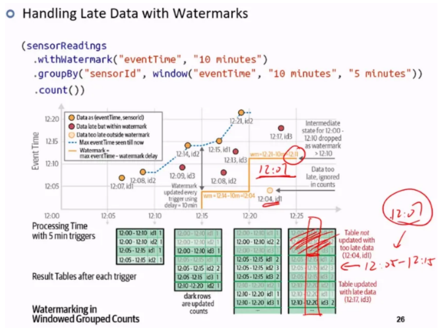


```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```



Mapping of event time to overlapping windows of length 10 mins and sliding interval 5 mins





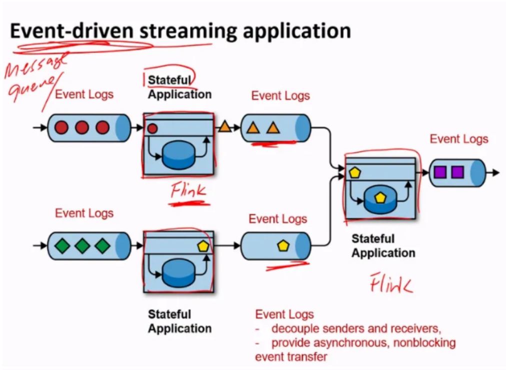
- Handling Late Data with Watermarks

```
(sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
  .count())
```

Event Time
12:20
12:15
12:10
12:05
12:00
12:05
12:10
12:15
12:20
12:25
Data as (eventTime, sensorId)
Data late but within watermark
Data too late outside watermark
Max eventTime seen till now
Watermark = max eventTime - watermark delay
12:07, id1
12:08, id2
12:14, id2
12:09, id3
12:15, id1
12:13, id3
12:08, id2
12:21, id2
12:17, id3
12:04, id1
Watermark updated every trigger using delay = 10 min
wm = 12:14-10m = 12:04
wm=12:21-10m=12:11
12:07
Intermediate state for 12:00 - 12:10 dropped as watermark > 12:10
Data too late, ignored in counts
Processing Time with 5 min triggers
Result Tables after each trigger
Watermarking in Windowed Grouped Counts
12:00 - 12:10 id1 1
12:00 - 12:10 id2 1
12:05 - 12:15 id1 1
12:05 - 12:15 id2 1
12:00 - 12:10 id3 1
12:05 - 12:15 id2 2
12:05 - 12:15 id3 1
12:10 - 12:20 id2 1
dark rows are updated counts
12:00 - 12:10 id2 2
12:05 - 12:15 id1 2
12:05 - 12:15 id2 3
12:05 - 12:15 id3 2
12:10 - 12:20 id1 1
12:10 - 12:20 id3 1
12:10 - 12:20 id3 2
Table not updated with too late data (12:04, id1)
Table updated with late data (12:17, id3)
12:07
12:05-12:15
26

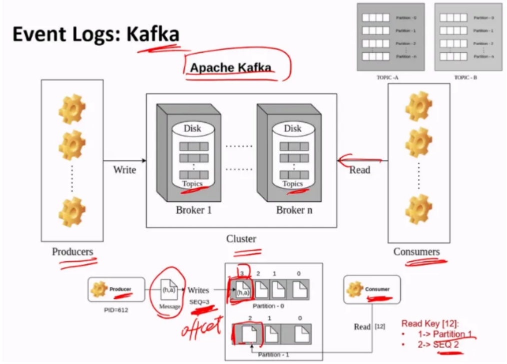
- Take the latest **event time** and minus the watermark time. Let go of the rows in the result window with end interval time lower than calculated time.
- Watermark just determines which window records in the table to drop off. If it was not dropped, it may still be updated even though the late data event time is before the watermark time.
- Data with event time of 12:07 is still updated as the event window [12:05 — 12:15] is not dropped as the window end interval time is not before the watermark time.
- **Performance Tuning:**
  - Cluster resource provisioning appropriately to run 24/7
  - Number of partitions for shuffles to be set much lower than batch queries
  - Setting source rate limits for stability
  - Multiple streaming queries in the same Spark application
  - Tuning Spark SQL Engine

Flink



Event-driven streaming application

Event Logs
- decouple senders and receivers,
- provide asynchronous, nonblocking event transfer

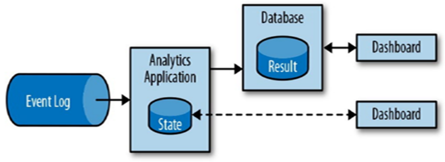


Event Logs: Kafka

Read Key [12]:
- 1-> Partition 1
- 2-> SEQ 2

A Streaming Analytics Application

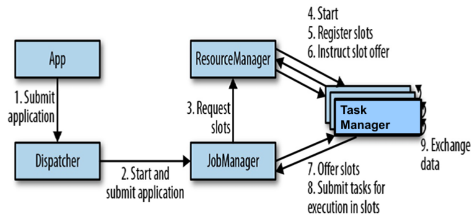
a stream processor takes care of all the processing steps, including event ingestion, continuous computation including state maintenance, and updating the results.



Much shorter time needed for an event to be incorporated into an analytics result !

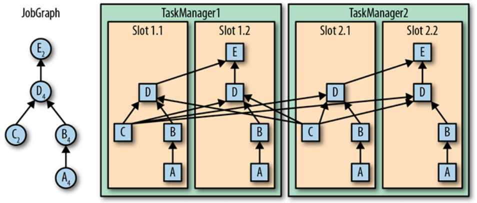
System Architecture

- Flink is a distributed system for stateful parallel data stream processing.

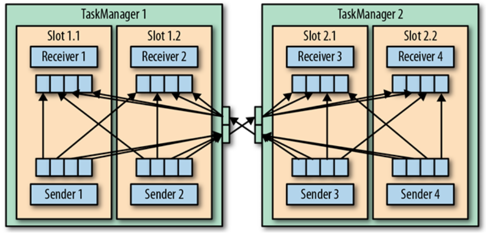


Task Execution

- A Task Manager can run multiple tasks concurrently
  - Tasks of the same operator (data parallelism)
  - Tasks of different operators (task parallelism)
  - Tasks of different applications (job parallelism)
- Task Managers offers certain number of processing slots to control the number of tasks it is able to concurrently execute. Think of slots as CPU cores.
- A processing slot can execute one slice of an application, i.e one parallel task of each operator of the application



- The tasks of a running application are continuously exchanging data.
- Task Managers take care of shipping data from sending tasks to receiving tasks.
- The network component of a Task Manager collects records in buffers before they are shipped.



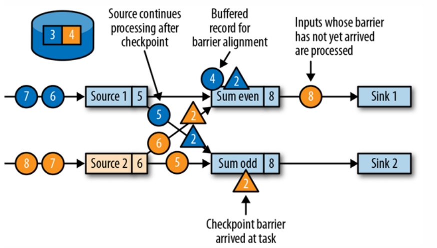
Event-Time Processing in Flink

- Flink allows for event-time processing by allowing the user to assign **timestamps** to events/records.
- Flink can handle out-of-order events by using watermarks.
- **Watermarks** are used to derive the current event time at each task in an event-time application.
- In Flink, watermarks are implemented as special records holding a timestamp as a Long value. Wtermarks flow in a stream of regular records with annotated timestamps.
- Watermarks in Flink is not directly invovled in late data handling. It is used to determine when to emit results of windowed operations.

Statem Management

- Stateful Stream Processing Task: All data maintained by a task and used to compute the results of a function belong to the state of the task.
- **State Backend:**
  - Local State Management: A task of a stateful operator reads and updates its state for each incoming record.
  - Each parallel task locally maintains its state in memory to ensure fast state accesses.
- **Checkpointing:**
  - A Task Manager process may fail at any point, hence its storage must be considered volatile

- Checkpointing the state of a task to a remote and persistent storage
- The remote storage for checkpointing could be a distributed filesystem or a database system
- *Consistent Checkpoints* (similar to Spark's micro-batch checkpointing, Stop the world, not used by Flink)
  1. Pause the ingestion of all input streams
  2. Wait for all in-flight data to be completely processed, meaning all tasks have processed all their input data
  3. Take a checkpoint by copying the state of each task to a remote, persistent storage. The checkpoint is complete when all tasks have finished their copies.
  4. Resume the ingestion of all streams.
- To execute failure recovery from consistent checkpointing. Simply restart the application, reset the state of all staetful tasks to latest checkpoint, and resume tasks.
- **Flink Checkpointing Algorithm**
  - Chandy-Lamport Algorithm: A distributed algorithm for recording a consistent global snapshot of a distributed system.
  - Does not pause the application but decouples checkpointing from processing
  - Some tasks contninue processing while others persist their state
  - Uses **checkpoint barrier**, a special record that signals the tasks to persist their state
  - Checkpoint barriers are injected by source operators into the regular stream of records and cannot overtake or be passed by other records
  - A checkpoint barrier carries a checkpoint ID to identify the checkpoint it belongs to and logically splits a stream into two parts
  - All state modifications due to records that precede a barrier are included in the barrier's checkpoint and all modifications due to records that follow the barrier are included in a later checkpoint.

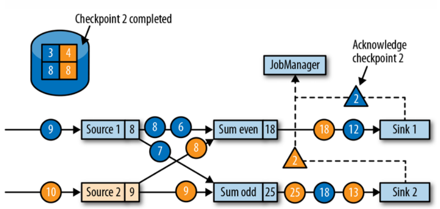


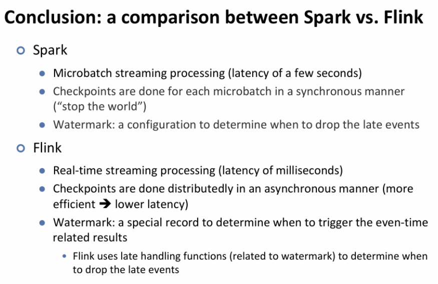
3 4
Source continues processing after checkpoint
Buffered record for barrier alignment
Inputs whose barrier has not yet arrived are processed
7
6
Source 1
5
4
2
Sum even
8
8
Sink 1
5
2
6
2
8
7
Source 2
6
5
Sum odd
8
Sink 2
2
Checkpoint barrier arrived at task

  - Job manager initiates checkpoints by sending message to all sources.
  - Sources checkpoint their state and emit a checkpoint barrier.
  - Records from input streams for which a barrier already arrived are buffered.
  - All other records are regularly processed
  - Tasks checkpoint their state once all barriers have been received, then they forward the checkpoint barrier
  - Tasks continue regular processing after the

checkpoint barrier is forwarded

- Sinks acknowledge the reception of a checkpoint barrier to the JobManager
- A checkpoint is complete when all tasks have acknowledged the successful checkpointing of their state



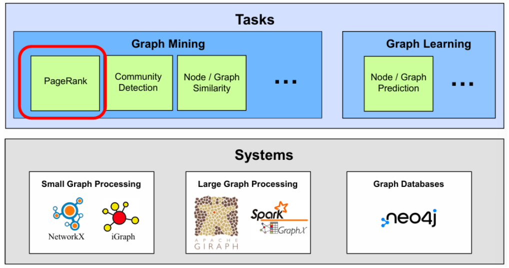


Conclusion: a comparison between Spark vs. Flink

- Spark
  - Microbatch streaming processing (latency of a few seconds)
  - Checkpoints are done for each microbatch in a synchronous manner ("stop the world")
  - Watermark: a configuration to determine when to drop the late events
- Flink
  - Real-time streaming processing (latency of milliseconds)
  - Checkpoints are done distributedly in an asynchronous manner (more efficient ➔ lower latency)
  - Watermark: a special record to determine when to trigger the even-time related results
    - Flink uses late handling functions (related to watermark) to determine when to drop the late events

Graphs

Graph Processing

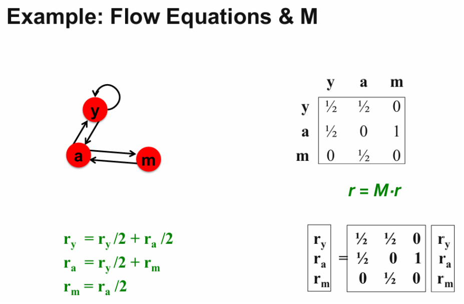


Simple PageRank

- We can visualise the web as a directed graph where each page is a node and each hyperlink is an edge.
- PageRank is an algorithm that measures the importance of a page in a network.
- The importance of a page is determined by the number of incoming links and the importance of the pages that link to it.
- If we assume incoming links are harder to manipulate, we can rank each page based on the number of incoming links.
- Problem: Malicious users can create a large number of pages that link to a target page to increase its rank.
- Solution: Make the rank of a page dependent on the rank of the pages that link to it.
- Therefore, links from important pages count more, this is true recursively.
- PageRank recursively computes the rank of a page based on the ranks of the pages that link to it. [Weighted edges]
- **Voting Formulation:**
  - For each page j, we define its importance as r_j
  - If page j with importance r_j has n outgoing links, each link gets $\frac{r_j}{n}$ importance.
  - Page j's own importance is the sum of the votes on its incoming links.
  - Formally: $r_j = \sum_{i \to j} \frac{r_i}{d_i}$.

    Where d_i is the number of outgoing links from page i.
  - The sum of importances of pages linking to j, each divided by number of outgoing links from that page.
  - In the event that the flow equations have no unique solution, we can add an additional constain to force uniqueness:

    $r_a + r_b + r_c = 1$
- **Matrix Formulation:**
  - Stochastic adjacency matrix, M
  - Let page i has d_i, out-links.
  - If $i \to j$, then $M_{ji} = \frac{1}{d_i}$, else $M_{ji} = 0$
  - M is a column-stochastic matrix, whjen each column sums to 1.
  - Let **Rank Vector** $= r$ and r_i is the rank of page i.
  - $\sum_i r_i = 1$
  - The flow equation can be written as $r = M \cdot r$
  - Column points to row. Each column is a page, each row is a page.



Example: Flow Equations & M

	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$r = M \cdot r$

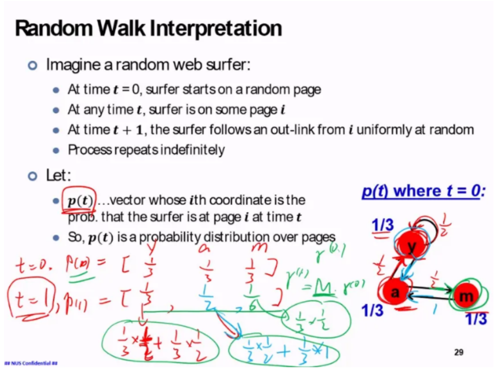
$r_y = r_y/2 + r_a/2$

$r_a = r_y/2 + r_m$

$r_m = r_a/2$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{bmatrix} \begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix}$$

- **Power Iteration Method:**
  - Each node starts with equal importance (of $\frac{1}{N}$). During each step, each node passes its current importance along its outgoing edges, to its neighbors.
    1. Suppose there are N web pages
    2. Initialise: $r^{(0)} = [\frac{1}{N}, ..., \frac{1}{N}]^T$
    3. Iterate: $r^{t+1} = M \cdot r^{(t)}$
    4. Stop when $|r^{t+1} - r^t|_1 < \epsilon$
- **Random Walk Interpretation:**



Random Walk Interpretation

- Imagine a random web surfer:
  - At time $t = 0$, surfer starts on a random page
  - At any time t, surfer is on some page i
  - At time $t + 1$, the surfer follows an out-link from i uniformly at random
  - Process repeats indefinitely
- Let:
  - $p(t)$ …vector whose ith coordinate is the prob. that the surfer is at page i at time t
  - So, $p(t)$ is a probability distribution over pages

p(t) where t = 0:

$t=0$, $P(0) = [\frac{1}{3} \quad \frac{1}{3} \quad \frac{1}{3}]$ (y a m)

$t=1$, $p(1) = [\frac{1}{3}, \quad \frac{1}{2}, \quad \frac{1}{6}]$

$r^{(0)}$

$r^{(1)} = M \cdot r^{(0)}$

$\frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2}$

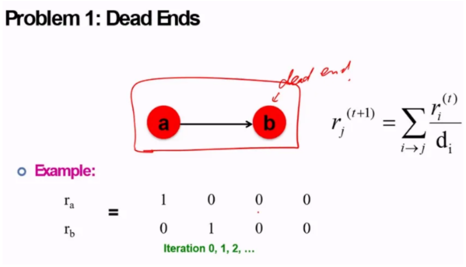
$\frac{1}{3} \times \frac{1}{2} + \frac{1}{3} * 1$

$\frac{1}{3} \times \frac{1}{2}$

29

NUS Confidential

- - Stationary Distribution: as $t \to \inf$, the probability distribution approaches a steady state, representing the long term probability that the random walker is at each node, which is the PageRank scores.
  - Three Questions:
    1. Does the random walk converge? Not always
    2. Does it converge to what we want?
    3. Are results reasonable?
  - Some pages are **Dead Ends**: Pages with no outgoing links. Causes importance to leak out of the network.



Problem 1: Dead Ends

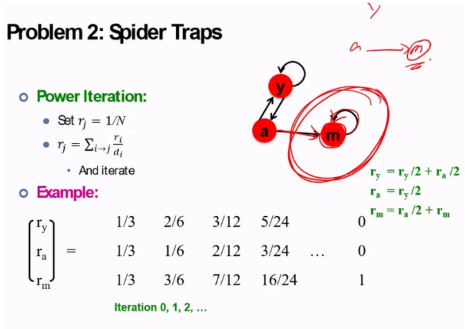
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

- Example:

$$\begin{array}{c} r_a \\ r_b \end{array} = \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

Iteration 0, 1, 2, …

- **Spider Traps**: Pages that link to each other but have no outgoing links. Eventually, all importance will be trapped in the spider trap.



Problem 2: Spider Traps

- Power Iteration:
  - Set $r_j = 1/N$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate
- Example:

$r_y = r_y/2 + r_a/2$

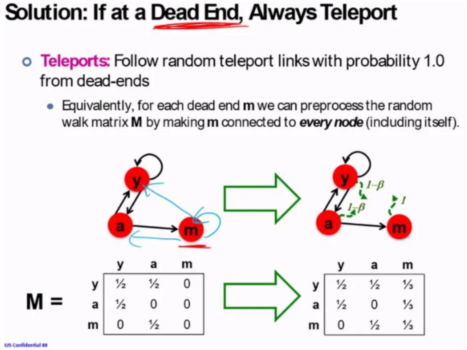
$r_a = r_y/2$

$r_m = r_a/2 + r_m$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{array}{cccccc} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \dots & 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & & 1 \end{array}$$

Iteration 0, 1, 2, …

- To solve deadends and spider traps, we cam introduce teleportation. At each step, the random walker has a small probability of teleporting to a random page. This ensures that the random walker can escape deadends and spider traps.
- PageRank Equation: $r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$
- β is also known as the damping factor. It is the probability that the random walker follows a link, and $1 - \beta$ is the probability that the random walker teleports.
- Google Matrix A: $A = \beta M + (1 - \beta)[\frac{1}{N}]_{N \times N}$
- PagreRank Equation (Matrix Form): $r = A \cdot r$
- In practice, β is usually set to 0.8 to 0.9. Which allows for 5 to 10 steps on average before teleport.
- If at a Dead End, Always Teleport: preprocess random walk matrix M and set each entry in the column of the dead-end page to $\frac{1}{N}$. This makes the matrix column stochastic.



Solution: If at a Dead End, Always Teleport

- Teleports: Follow random teleport links with probability 1.0 from dead-ends
  - Equivalently, for each dead end **m** we can preprocess the random walk matrix **M** by making **m** connected to ***every node*** (including itself).

$$\mathbf{M} = \begin{array}{c|ccc} & y & a & m \\ \hline y & ½ & ½ & 0 \\ a & ½ & 0 & 0 \\ m & 0 & ½ & 0 \end{array} \Rightarrow \begin{array}{c|ccc} & y & a & m \\ \hline y & ½ & ½ & ⅓ \\ a & ½ & 0 & ⅓ \\ m & 0 & ½ & ⅓ \end{array}$$

US Confidential ##

Topic Specific PageRank