CS4225
AY23/24 S1
github.com/SeekSaveServe

Lectures

L1: Introduction

Four Vs of Data Science

- Volume
- Variety
- Velocity
- Veracity - uncertainty of data

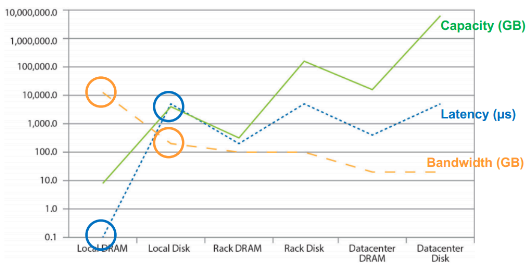
Storage Hierarchy

- Volume: Server le Rack le Cluster
- Speed: Server ge Rack ge Cluster

Bandwidth vs Latency

- **Throughput** Actual rate at which data is transmitted across the network over a period of time
- **Bandwidth** Maximum (capacity) amount of data that can be transmitted per unit time
- **Latency** Time taken for 1 data packet to go from source to destination (or both ways)
- Latency does not matter when transmitting a large amount of data
- Bandwith does not matter when transmitting a small amount of data

Cost of moving data



- Bandwidth drops and latency increases as we move up the data hierarchy
- Disk reads are also much more expensive

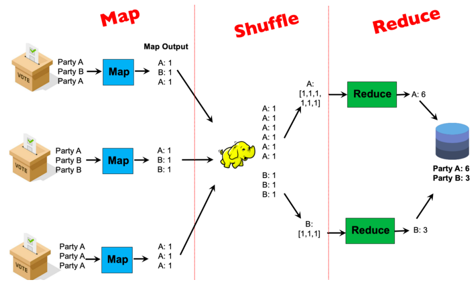
Big ideas of data processing

- Horizontal scaling is cheaper than vertical scaling
- Move data processing to the machine with the data since data clusters have limited bandwidth
- Process data sequentially and avoid random access to reduce total seek time
- Seamless scalability $\rightarrow$ use more machines to reduce time taken to process data

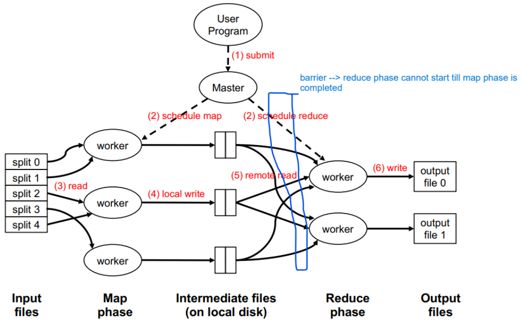
Challenges

- Machine failures
- Synchronisation
- Programming difficulty

L2 Map reduce



- Map: extract something of interest from each. Emits a key value pair
- Shuffle: Shuffle intermediate results by key value pairs
- Reduce: Aggregate intermediate results
- Each of these three processes can occur concurently across different machines



Map Reduce Implementation

1. **Submit:** User submits mapreduce program and configuration (e.g. no. of workers) to Master node
2. **Schedule:** Master schedules resource for map and reduce tasks (master does not handle actual data)
3. **Read:** Input files are separated into splits of 128MB. Each split corresponds to one map task. Each worker executes map tasks 1 ata a time
4. **Map phase:** Each worker iterates over each key,value tuple and applies the map function
5. **Local write:** each worker writes the output of map to intermiediate files on its local disk.These filees are partitioned by key
6. **Remote read:** each reduce worker is responsible for $\geq$ key. For each key, it reads the data it needs from the corresponding partitioon of each mapper's local disk
7. **Write:** output of the reduce function is written (usually to a distributed file system such as HDFS)

Interface

- map(k,v) $\rightarrow$ list(k',v')
- reduce

L3: No SQL Overview

NoSQL

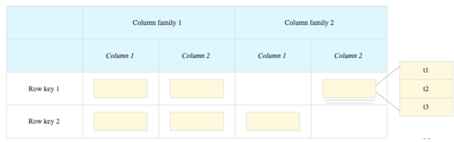
- Not Only SQL: can include sql
- Stores data in a format other than relational DB
- Sql refers to to relational DBMS, not the querying language - NoSQL can have querying lang too
- Used for large volumes of data and data that does not fit in a structured data (e.g. some has image, some don't)

Propertiess

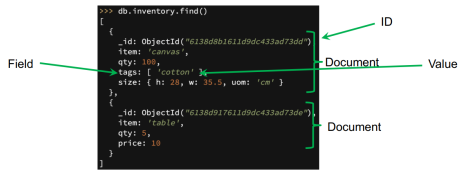
- Horizontal Scalable: easy to partition and distribute across machines
- Replicate and distributed over many servers
- Simple call interface
- Often weaker concurrency model than RDBMS
- Efficient use of distributed indexes and RAM
- Flexible schema

Major NoSQL DB

- Key-value stores
  - Stores mapping (associations) bewteen keys and values
  - Keys are usually primitives (int,str,raw bytes etc) that can be easily queried
  - Values can be primitive or complex; usually cannot be easily queried (lists, JSON, HTML, BLOB)
  - Eventually consistent
  - **Operations**
    - Get - fetch value with key
    - Put - set value with key
    - Multi-Get, multi-put, range queries (must be comparable, e.g. int, str)
  - **Suitable for**
    - Small continuous read and writes
    - Storing basic information or no clear schema
    - Complex queries are rarely required
    - Improves scalability and efficiency of read and writes
    - Eventually consistent, so the data might be stale
  - E.g. Storing user sesions, caches, user data that is often processed individually
  - **Implementation**
    - Non-persistent: Just a big in memory hash table (E.g. redis, memcached) that needs to be regularly backed up to disk
    - Persistent: data is stored persistently to disk (E.g. RocksDB, Dynamo, Riak)
- Wide-column databases - stored sparsely

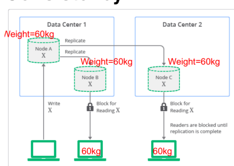


  - Rows describe entities
  - Related groups of columns are grouped as column families (similar to separate tables, except they share the same row)
  - Sparsity: If a cloumn is not used for a row, it doesn't use space (saves space for sparse data)
- Document stores

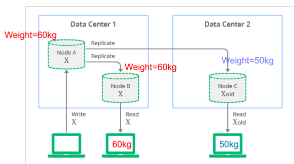


  - no schema (flexible schema)
  - A data base can have multiple collections
  - A collection (tables) can have multiple documents (rows)
  - A document is a JSON-like object with field (columns) and values
  - Different documents can have different field and can be nested
  - Flexible Schema: accommodates data with different characteristics
  - **Querying**
    - Unlike key val stores, doc stores allow querying based on the content
    - If the field does not exist on the doc, we just skip it when doing CRUD
- Graph databases
  - Need to store information about the nodes and edges
  - Edges: relationship between data (nodes)
  - Good at modelling and querying complexed relationships between entities
  - Good for modelling data as graphh problems (traversing relationships, shortest path, social networks etc)
- Vector Databases
  - Store vectors (each row is a point in d dimensions)
  - Usually dense, numerical, and high-dimensional (data with many features)
  - Allow fast similarity search via locality sensitive hashing (LSH), similar to min-hashing
  - Scalable, real-time updates, replication
  - Good for LLM and vision models as they need to be converted to vectors, and search, recommendation, clustering can be easily added
  - Good for contetn based similarity matching
  - E.g. Milvus, Radis, MongoDB, Atlas, Weaviate

Consistency



All readers read new value (60kg)



Readers may read old value (50kg)

- Strong: Any reads on all observers immediately read the same result after update (uses locks, higher latency)
- Eventual: If the system is working and we wait long enough, eventually all reads will produce the same value (correctness affected)

BASE

- **B**asically **A**vailable - basic writing and reading operations are available most of the time
- **S**oft state: without guarantees, we only have some probability of knowing the state at any time
- **E**ventually consistent
- Eventual consistency offers better availability at the cost of weaker consistency
- NoSQL allows for weaker consistency guarantees, and can be tuned to be stronger (tunable consistency)
- Suitable for statistical queries and social media feed but not suited for financial transactions

Duplication / Denormalization

- Motivation: Support join statments $\rightarrow$ how do we join 2 tables to form 1 new table

- Some optimizations in SQL may not be possible in NoSQL
- **Denoormalization:**
  - Storage is cheap! Duplicate data to boost efficiency
  - Tables are designed around potential join queries (pre-create the join tables)
  - Good if the queries types are fixed
  - What if a field is updated? → changes need to be propagated to multiple table

Pros	Cons
+ **Flexible / dynamic schema**: suitable for less well-structured data	- **No declarative query language**: query logic (e.g. joins) may have to be handled on the application side, which can add additional programming
+ **Horizontal scalability**: we will discuss this more next week	- **Weaker consistency guarantees**: application may receive stale data that may need to be handled on the application side
+ **High performance and availability**: due to their relaxed consistency model and fast reads / writes	

- Depends on:
1. if denormalization is suitable
2. importance of consistency
3. complexity of queries (joins Vs read/write)

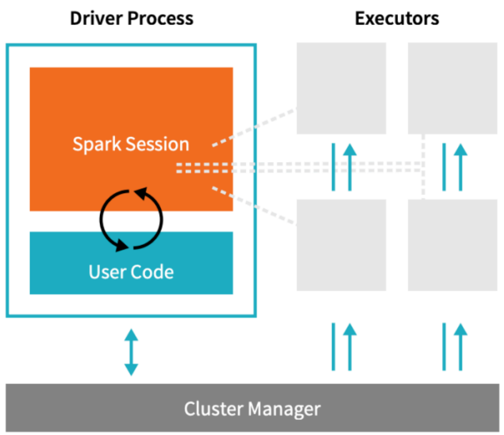
L6: Sparks Basics I

Hadoop Vs Spark

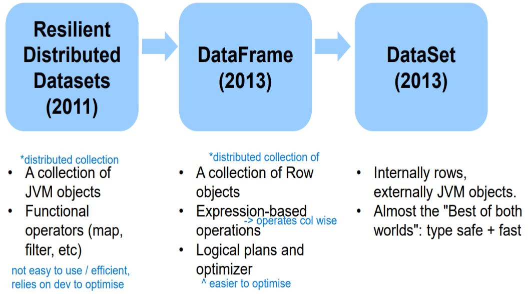
- Spark stores most of intermediate results in memory, making it faster for iterative processing (spillover to disk still happens if memory runs out)
- Hadoop writes intermediate results to local machine / disk. This is not efficient for iterative processing and ML
- Sparks has ease of computability
- Spark combines batch processing, streaming, ML, graph processing

Spark Architecture

- Driver process: respond to user input and distributes work to executors
- Executors: executes code and send result back to the driver



Evolution of Spark APIs



Lineage Approach

- Using replication is expensive since Spark uses memory
- A faulty node is replaced and recomputed using the DAG from the lost partition (E.g narrow transformations)

Resilient Distributed Datasets (RDD)

- Resilient: Fault tolerence through lineage
- Each node executes over 1 partition of data (data parellelism), a RDD is a collection of nodes and the driver
- DDs: colleciton of objects distributed over machines
- Immutable

```
# Create an RDD of names, distributed over 3 partitions
dataRDD = sc.parallelize(["Alice", "Bob", "Carol", "Daniel"], 3)
```

- **Transformations**
  - Transform RDDs into RDDs
  - Lazily evaluated, which allows optimisations to be applied over a series of transformations
  - E.g.: map, order, groupby, filter, join, select
- **Actions**
  - Triggers spark to compute result from a series of transformations

```
dataRDD = sc.parallelize(["Alice", "Bob", "Carol", "Daniel"], 3)
nameLen = dataRDD.map(lambda s: len(s))
nameLen.collect()
```

  - Retrieve all RDD to the driver node
  - E.g.: count, collect, show, save
- Spark actions and transformations are calculated in parallel across distributed workers
- RDDs are objects. Completed RDDs are stored in memory and can be flushed out
- Note: transformation work on files in the worker node, not the driver

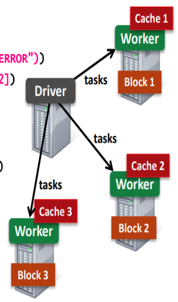
Caching

- **Caching**: sometimes we want to reuse RDDs to avoid recomputation

```
lines = sc.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

start from here

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



- cache is also a transformation!

- It is lazily done. So it only takes effect after an action
- Cache store an RDD to memory of each worker node
- persist()" store RDD to memory or disk or off-heap memory
- RDDs are evicted on a LRU basis so cached RDDs can be evicted

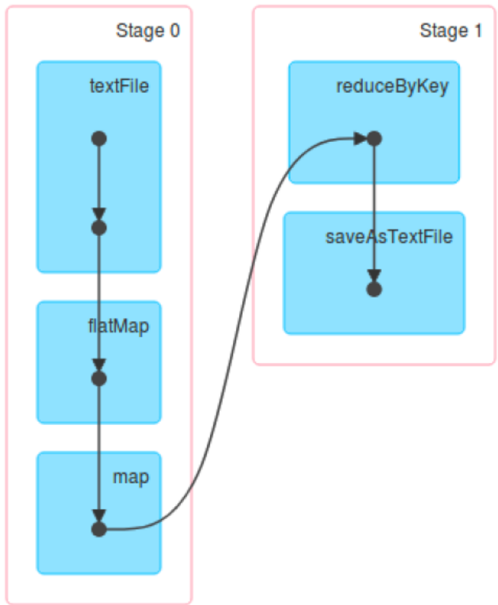
```
lines = spark.textFile("hdfs://log")
errors = lines.filter(lambda s: s.startswith("INFO"))
info = errors.map(lambda s: s.split("\t")[2])

info.cache()

info.filter(lambda s: "hadoop" in s).count()
info.filter(lambda s: "spark" in s).count()
```

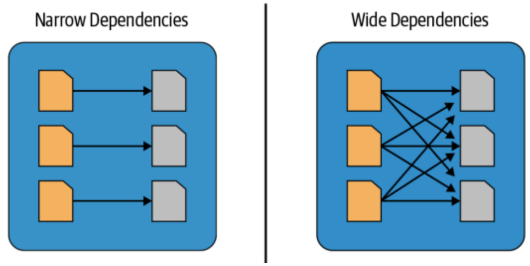
DAG

- Represents all RDD objects and order of transformation
- RDDs are functional operations
- Operations here happens in parallel



Narrow and wide dependencies transformation

- Narrow can be linked together
- Wide dependencies are across stage
- Wide: implict synchronisation effect
- Narrow: each partition of the parent RDD is used by at most 1 partition of child RDD
- Narrow: map, flatmap, filter, contains
- Wide: partition of parent RDD is used by multiple partitions of the child RDD (other worker nodes)
- Wide: reduceByKey, groupBy, orderBy



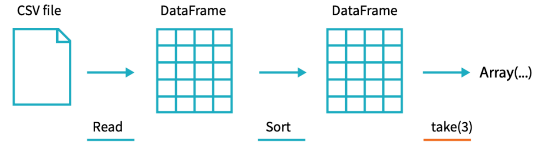
- Consecutive narrows are grouped as "stages" in DAG
- Within stages: spark computes consecutive transformations on the same machine (pipelined, parallelized)
- Across stage: data needs to be shuffled and intermediary results needs to be written to disk
- Minimize shuffling (across stage)

Lineage and fault tolerance

- Does not use replication (unlike hadoop) since memory is limited
- Lineage: if a worker is down, we replace it, and use DAG to recompute the data in the lost partition. Lost partition will be recomputed from the RDDs
- The DAG of each RDD has to be stored
- When flushed, the node can start where it stopped within the wide stage later on
- If the job is passed to a new node, the RDD job starts from the start of the stage

Dataframe

- column based (applied for each attribute)
- Dataframe represents a table of data, similar to tables in sql
- This is a higher level interface that is easier to use
- Implemented with RDDs
- Expression based operation



- Spark can use sql queries for dataframes which is similar to:

```
from pyspark.sql.functions import desc
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.collect()
```

- Expression based, does not specify the order of functions, hence leaving room for optimisation

Datasets

- Type safe version of data frame

- Datasets are not available in python and R ssince they are dynamically typed
- Each row is an object of a user defined class

```
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)

val flightsDF = spark.read.parquet("/mnt/defg/flight-data/parquet/2010-summary.parquet/")

val flights = flightsDF.as[Flight]

flights.collect()
```

L8: Streams

Motivation

- Data arrives overtime (online) via message queue, file stream etc
- System cannot store the entire stream, so we have to process the data as they arrive
- E.g. search, online activity data, sensor data, financial data
- Cannot wait till all the data is recevied to decide

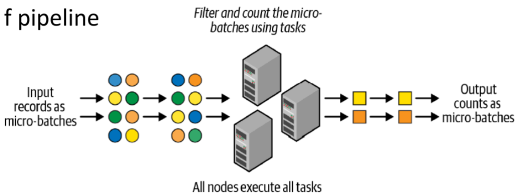
Fault Tolerance

- Need to be able to store and access intermediate data
- Non-stateful stream processing is not accurate

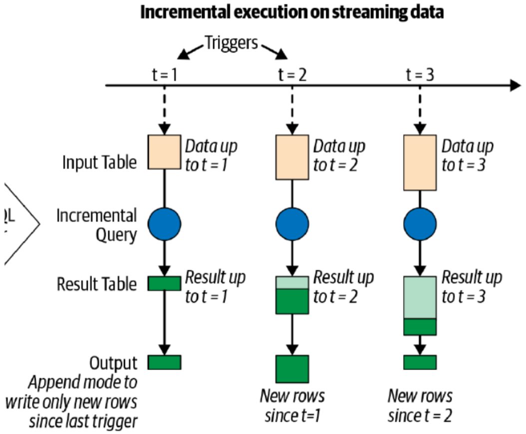
Spark stream processing - structued streaming

- Micro Batch model: Divides data from input stream into micro batches
- Each batch is processed in a distributed manner
- Small, deterministic task generates the output to batches
- **Advantages**
  - Quick and efficiently recover from failures (process the failed batch again, rollover)
  - Determinisitc nature: end-to-end exactly once processing is guaranteed
- **Disadvantages: High throughput, high latency**
  - Latency of a few seconds - need to wait for all records in the microbatch to be completed
  - Application may experience higher delay in other parts of the pipeline
  - The latency might be too high for some

f pipeline

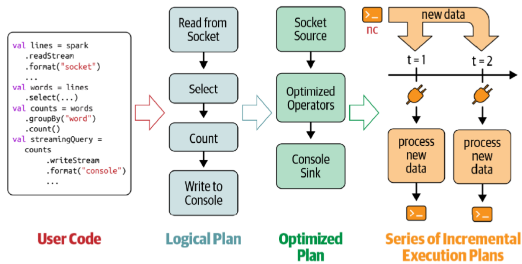


- Treat the table as unbounded, with new rows streaming in
- Data flows in incrementally, new "rows" are processed are the result is appended to the output table as new rows as well



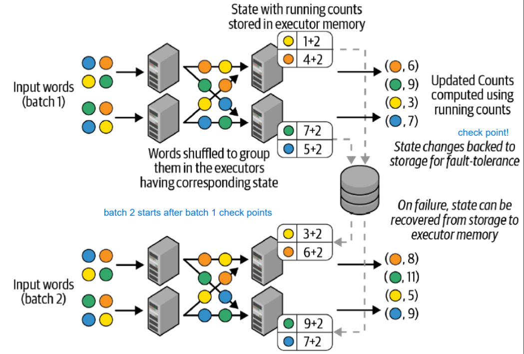
- **Defining a structured query**

1. Define input source(s)
2. Transform data
3. Define output sink and output mode
   - output writing details (where and how)
   - processing details (how to process and recover from failure)
4. Specify processing details
   - Triggering details: When to trigger the discovery and processing of newly available steaming data
   - Check point location: store streaming info for recovery
5. start query



Data Transformation

- **Stateless transformation**
  - Process each row without info from prev rows
  - Projection: select(), explode(), map(), flatmap()
  - Selection: filter(), where()
- **Stateful transformation**
  - E.g. df.groupBy().count()
  - partial count is stored somewhere and passed to the next batch
  - It is important to have exact-once even with potential failure and recovery so that the final count is accurate

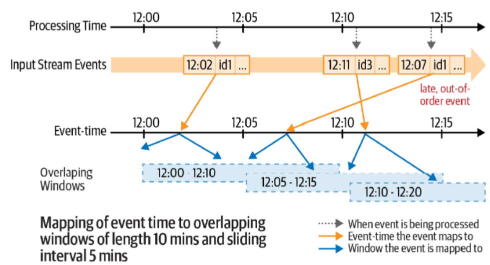


Stateful streaming aggregation

- **Aggregation not based on time**
- Global: running count = sensorReadings.groupBy().count()
- Group: baseline values = sensorReadings.groupBy("sensorID").mean("value")
- sum(), mean(), count(), stddev(), countDistinct(), collect_set(), approx_count_distinct()
- **Aggregation based on time**
- Groups are based on processing-time window or event-time window
- Processing-time windows may not always contain the same events due to network latency, congestion etc – result not consistent
- Event-time window is persistent and ensures exact-once semantics – preferred!
- Event-time decouples processing speed from results
- sensorReadings.groupBy("sensorId", window("eventTime", "5 minutes")).count()

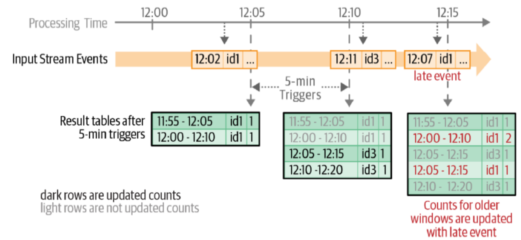
Tumbling Window

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



- We tag an id to the event based on the event time
- The id refers to the corresponding tumbling window
- **Overlapping Window**

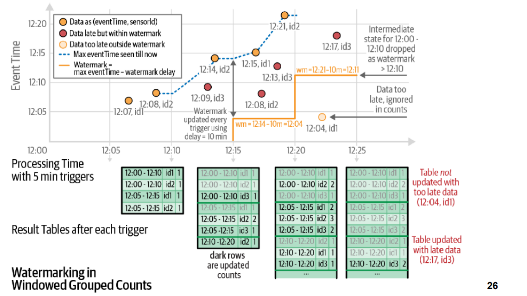
Updated counts in the result table after each five-minute trigger



- Consecutvie windows share some data with adjacent windows
- This makes data more continuous and smooth
- Increases data utilization
- Reduces edge effect, where the values at the edge of a event window weights less than the events in the centre
- Prevent loss of information, particularly the data at the edge
- A data that arrives will update =1 window

Watermark

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

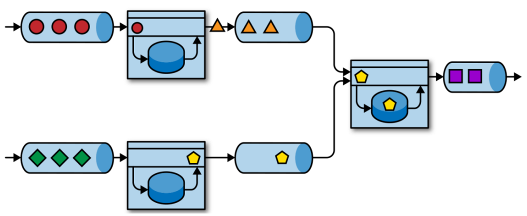


- Water mark: we only track records that are within highest current time - water mark duration: highest current time

Performance Tuning

- Cluster resources appropriately to avoid running 247
- Set partitions for shuffling to be lower than batch queries for streaming data, so that the data in each node is large enough for the batch queries
- Setting source rate limits for stability – prevent incoming stream from breaking spark
- Multiple batch,streaming queries, ML can have at once

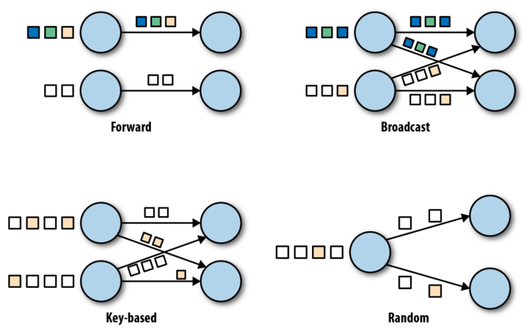
Flink Overview



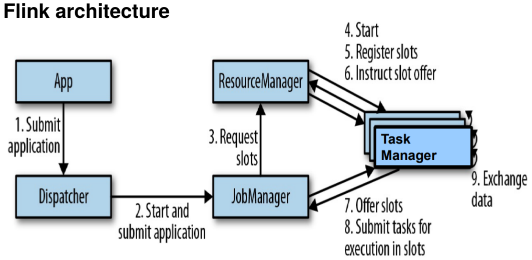
- distributed system for stateful parallel data stream
- Treats stream as a stream
- Achieves microsecond latency

- Event driven, message queue and event logs
- Also has logical plan and physical plan, similar to spark

Dataflow model

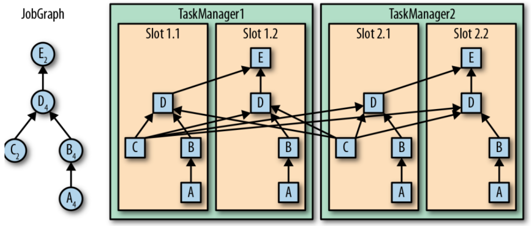


Flink architecture



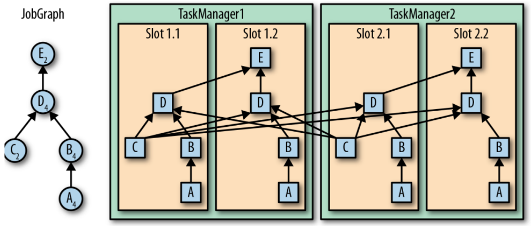
- Resouce manager is responsible for scheduling tasks to resources

Task exectution



- Task manager manages slots that processes tasks
- Task manager can execute tasks from diferent operators and different applications
- C → B, some network shuffling is done

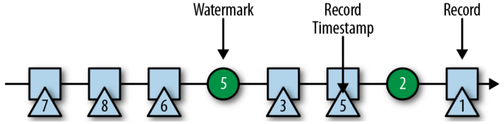
Data transfer in flink



- Task of an app is continuously exchanging data
- Task manager takes care ofsending and echanging data
- Network component of task manager buffers the records before sending
- Send and receive has their own buffer to reduce network traffic, so send and receive are done async (unlike micro batch)

Event time processing (Flink)

- Every record has an accompanying time stamp

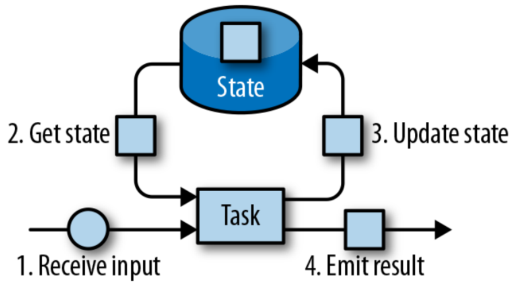


- **Watermark (flink)**
  - more like a trigger mark (spark triggers after every micro batch, so latency is the size of the micro batch) – watermark in flink determines how freqeuntly calculations are triggered!
  - Watermark is represented as special records holding a timestamp
  - Water mark flows in a stream of regular records
  - Heuristic watermark: Results trigger after watermark, late records processed again later
  - Perfect watermark: Late records included, trigger happens after

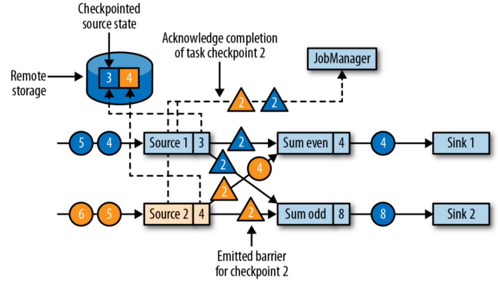
```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(AfterWatermark()
                            .withEarlyFirings(AlignedDelay(ONE_MINUTE))
                            .withLateFirings(AfterCount(1))))
  .apply(Sum.integersPerKey());
```

- $withLateFiring$ determines when to drop late records

State Management in Flink



- **Operator State**
  - Scoped to an operator task, cannot be accessed by other operators
  - All records processed by the same parallel task have acess to the same state
- **Consistent Checkpointing**
  - Similar to micro batch checkpoint
  - Pause the ingestion of all input streams
  - Wait for in-flight data to be completely processed (all tasks processed their inputs)
  - Like a barrier? Need to finish the whole "microbatch"
  - Copy and store state to a persistent storage
  - Need to reset from latest checkpoint – cannot achieve milisecond delay!
- **Chandy-lamport algorithm**



- Distributed checkpoints
- Decouples checkpoiting from processing, does not pause the entire app, only some tasks pauses
- Sets up a checkpoint barrier, all tasks to perform this barrier in a distributed way
- After receiving the checkpoint message, the source will save their state and broadcast checkpoint barrier to the receiving nodes
- Tasks will buffer the records for barrier alignment (only process once they receive the barrier msg from all sources)
- Only the tasks receiving the barrier will buffer
- After receiving all checkpoint barriers, the task will save their state
- Then the barrier is emitted to the next level (sink nodes)
- The jobManager is notified once all the tasks have completed the checkpoint

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