SPARK

1) What's Spark? Why do you require Apache Spark?

A. - Apache Spark is an open source, fast, general-purpose, in-memory processing engine for big data processing.

- It was developed in AMP Lab of University of California Berkeley in 2009
- In 2010 it was open sourced.
- It's written in Scala.
- It provides high level application programming Interfaces (APIs) in Java, Scala, Python & R.

* Apache Spark has following pros:

1. It provides unified framework to perform different tasks like batch, real-time processing that previously required different engines.
2. It supports both batch processing and real-time processing.
3. It's 100 times faster than Hadoop MapReduce when run in memory and 10 times faster

than Hadoop MapReduce when run on disk.

4. It provides high-level operators (Eg: map, filter) to process data unlike Hadoop MapReduce

5. It provides interactive shell which is useful in learning about & exploring data.

6. Its not bundled with storage system. So you can use it with storage system of your choice like:
   - Local file system
   - HDFS
   - Cassandra
   - S3 etc.,

2) Differentiate between Spark and Hadoop MapReduce.

A.

Apache Spark	Hadoop Map Reduce
Intro 1. Open source big data processing framework Faster & general purpose processing engine	1. Open source framework to process structured & unstructured data stored in HDFS.
2. Unified framework to process various tasks like batch, interactive & iterative processing.	2. Processes data only in batch mode

Speed	3. Runs → 100 times faster → in memory → 10 times faster → on disk than Hadoop MR which is made possible by reducing number of read/write cycles to disk and storing intermediate data in memory.	3. Hadoop MapReduce reads & writes from disk which slows down processing speed.
Difficulty	4. It provides high-level operators (map, filter etc.,) which makes developer's job easy.	4. Developers need to hand code each operation
Real time processing	5. It supports both batch & real-time processing	5. It supports only batch processing.
Interactive mode	6. Provides interactive shell to learn and explore data	6. It doesn't provide an interactive shell.

3) List the areas where Spark and Hadoop MapReduce are good.

A. (I) Hadoop MapReduce is good for:

LE (i) Linear processing of huge datasets.

- Allows parallel processing by
  - breaking large chunks into smaller ones.
  - Processes them separately on different datanodes
  - Automatically gathers results and returns as single unit.

- In case, the resulting dataset's larger than available RAM, it outperforms Spark.

(ii) Economical solution :
- If speed of processing's not critical i.e., no immediate results expected, MapReduce's a good solution. [Going datasets in next section(vi)]

(II) Apache Spark's good for : (JF MINA)

(i) Fast data processing :
- In-memory processing makes it faster than Hadoop MapReduce - upto 100 times for data in RAM upto 10 times for data in storage.

(ii) Interactive processing :
- If task's to process data again & again, Sparks Resilient Distributed Datasets (RDDs) enable multiple map operations in memory
- while HadoopMapReduce has to write interim results on disk.

(iii) Near real-time processing :
- If a business need immediate processing/insights, then they should go for Spark.

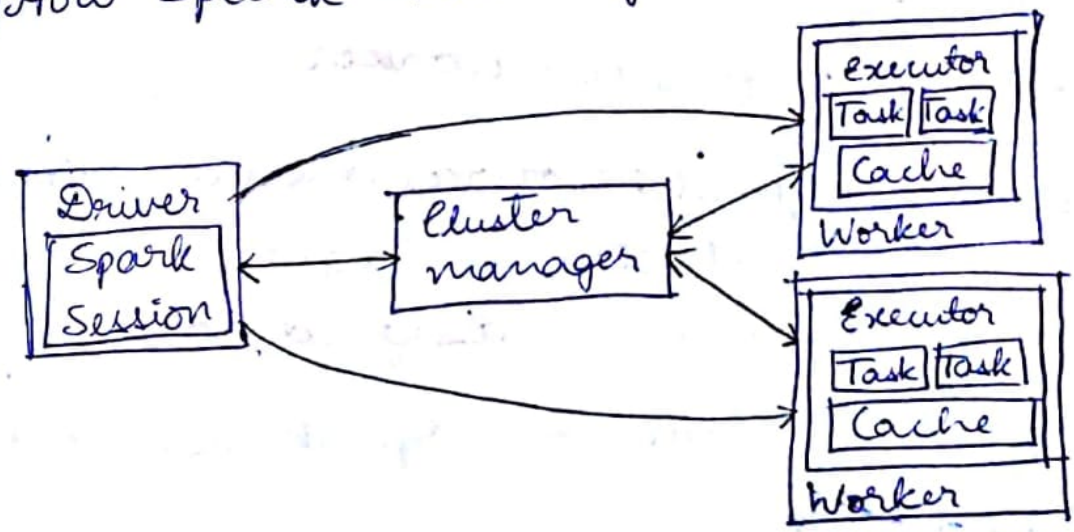
(iv) Graph processing :
- It has GraphX - an API for graph computation
- It's computational model's good for iterative

computations typical in graph processing.

(v) Machine learning :
- It has MLib - a built in machine learning library which has out-of-box algorithms that also run in memory.
- Hadoop needs a third-party provider.

(vi) Joining datasets :
- Due to its speed, it can create combinations faster.
- Hadoop's better if datasets to join is very large, which requires lot of shuffling and sorting.

4) How Spark manages cluster and resources?



A. - Spark's a distributed system designed to process large volumes of data efficiently & quickly
- This distributed system is deployed onto a collection of machines, viz known as Spark cluster
- A cluster can be → small with only few machines
→ large with 1000s of machines

- To efficiently & intelligently manage the collection of machine we use resource management systems like - Apache YARN, Apache Mesos.

- Main components of resource management systems are → Cluster Manager
  → Worker.

- Cluster manager knows :
  - where the workers are located.
  - how much memory they've.
  - number of CPU cores each has.
- Cluster manager has to orchestrate the work by assigning it to each worker
- Each worker: • offers resources (memory, CPU etc.) to cluster manager
  • performs assigned work.

5) Write a short note on Spark Drivers and Executors.

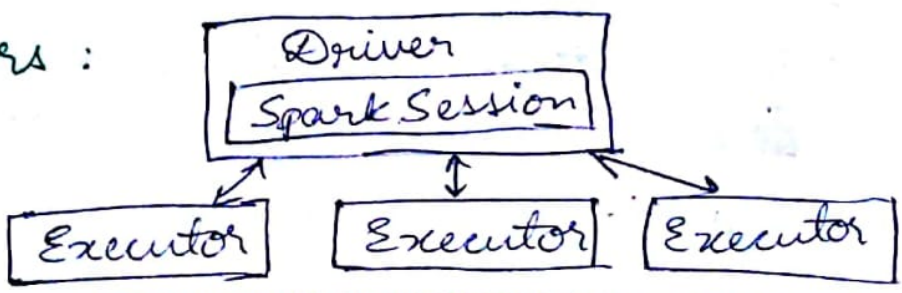
A (I) Spark Driver :

- Spark application has two parts :
  - application data processing logic expressed using Spark APIs
  - Spark driver.

- Application data processing logic can be:
  - as simple as few lines of code that perform few data processing operations.
  - as complex as training large ML model which requires many iterations and could run for hours.

- (I) Spark driver's central co-ordinator & (i) interacts with cluster manager to select machines to run data processing logic on. (ii) For each of these machines, it requests cluster manager to launch Spark executor process.
  (iii) manages & distributes Spark tasks onto each executor
  (iv) collects and merges results from each Spark executor if user display of results required
- The entry point into Spark application's through a class called SparkSession. It provides facilities:
  - For setting up configurations
  - for API to express logic.

(II) Spark executors:

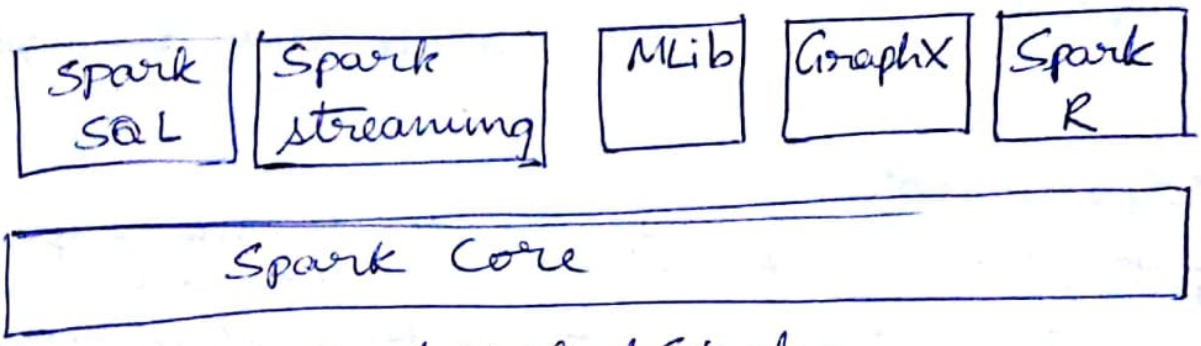


JLMIS
- Spark Executors a JVM process and is exclusively allocated to specific Spark application.
- This is to avoid sharing Spark Executors between multiple Spark applications
- This is to isolate them which prevents one badly behaving application to affect others.
- Lifetime of Spark executors the duration of Spark application.
- Spark applications are running on separate Spark Executors so, sharing data requires external storage system.
- Spark employs master [One Spark driver] - slave [many spark executors] architecture
- Each of these components run as independent processes on a spark cluster.

ESE
- Spark Executor :
  - executes data processing logic in form of tasks.
  - Each task is executed on separate CPU core.
- This is to speed up large scale data processing done using parallel processing.
  - caches a portion of data in memory or disk when told by application logic.

6) Write the Spark Unified Stack and explain Spark Core.

A.



Spark Unified Stack

- Spark Stack's a unified data processing engine built on top of strong foundation called Spark Core.
- Spark core provides functionalities to:
  - manage & run distributed applications like
    - → scheduling
    - → coordination
    - → fault-tolerance
  - powerful & generic programming abstraction for data processing called Resilient Distributed Datasets (RDDs)
- On top of this are collection of components each of which is designed for specific data processing workload:

component	workload
1. Spark SQL	→ batch & interactive data processing
2. Spark streaming	→ real time stream data processing
3. GraphX	→ graph processing

Component	workload
4. MLib	→ machine learning
5. SparkR	→ running machine learning tasks using R shell

*Spark core

- It provides functionalities to → manage, run distributed applications like → scheduling, coordination, fault-tolerance
- It is the bedrock of Spark distributed data processing engine.
- It consists of two parts → distributed computing infrastructure, RDD programming abstraction.

(i) Distributed computing infrastructure:

- It's responsible for (i) dos → distribution, coordination and scheduling of tasks across machines in cluster.
- This enables ability to perfor. parallel data processing of large volume of data efficiently and quickly

(ii) handling computing task failures.

(iii) efficiently moving data across machines i.e., data shuffling.

- Advanced users need to have knowledge Spark distributed computing infrastructure

(ii) RDDs

- They're fip → immutable
  → fault tolerant
  → parallel data structures

mic
- They let users
  - persist intermediate results in memory.
  - control their partitioning to optimize data placement.
  - manipulate them with rich operators.

e
- It provides set of APIs for developers to easily & efficiently perform large-scale data processing without worrying where data resides or dealing with machine failures.

- The RDD APIs → are exposed in multiple programming languages.
  → allow users to pass local functions to run on cluster viz powerful & unique.

7. What're RDDs ? Explain the properties of RDD.

A. (RDDs represent → idea of representation of large dataset
  → abstraction of working with it.

- They're → immutable
→ fault-tolerant
→ parallel data structures

(- It allows users to
- explicitly persist intermediate results in m
- control their partitioning to optimize ~~th~~ data placement.
- manipulate them using rich set of operators.) instd write last para

- Properties of RDDs : $F^2I^2R\ P$

→ Immutable:
- It means you can't specifically modify a particular row in dataset represented by
- You can call RDD operators to manipulate i but the operation 'll return new dataset
- Immutability requires RDD to've lineage information which enables fault tolerance capability

→ Fault tolerant:
- Processing multiple datasets in parallel requires cluster of machines to host & execute computational logic.
- If one or more of these machines fail becomes slow, Spark handles failure by

rebuilding the failed portion using the lineage information.

→ Parallel data structures:

- Take the example where you've to find how many statements contain the word "exception" in a 1TB log file.
- slow solution: → Iterate through entire log file
  → execute logic to determine if statement contains ~~to~~ word exception.

→ Faster solution:

→ divide 1TB file into chunks i.e., collection of rows
→ Execute the required logic on each chunk in a parallelized manner.

- The collection rows is the data structure that can hold set of rows and provide the ability to iterate through them
- These chunks are processed in parallel which leads to the phrase parallel data structures.

→ In-memory computation:

- ML algorithms need to go through many iterations to arrive at the optimal state.

- This is where distributed in-memory computation can help in reducing the completion time from days to hours
- Interactive data mining hugely benefits from this property. It requires multiple ad hoc queries to be performed on the same dataset [subset of data]. If this is kept in memory, it'll need only seconds to complete.

→ Rich set of operations :

- Operations provided by RDDs can perform:
  - data transformation: ☺ FACES
  - grouping
  - filtering
  - joining
  - aggregation
  - sorting
  - counting
- These operations operate in coarse grained level i.e., they're applied to many rows & not any specific row.

- RDD is represented as an abstraction and is defined by the following information

. A set of partitions i.e., the chunks that make up the dataset.
. A set of dependencies on parent RDDs.
. A function for computing all rows in the dataset.
. Metadata about the partitioning scheme [optional].
. Where the data lives in a cluster (optional)

8) What're transformation and Action in Spark? Give suitable examples.

A. - RDD operations're classified into two types:
  • transformations
  • actions

- RDDs can't be changed after they're created i.e., they're immutable. To change a DataFrame, you need to instruct Spark how to modify it.
- These instructions're called transformations.
- Transformation operations're lazily evaluated i.e., they're merely record the

transformation logic and will apply it at a later point.

- Invoking an action operation will trigger evaluation of all the transformations preceding it.
- It will either → return result to driver
  [or] → write data to storage system

- Examples:

~~(i) mapfunc(I)~~ Transformation examples

(i) Defining a function to convert all characters in a string to uppercase

```
> def toUpperCase(line:String) : String = {line.toUpperCase
> stringRDD.map(l => toUpperCase(l)).collect.foreach
                                        (println).
```

→ By abstracting the complex logic in a function it improves readability, maintainability and ease of testing.

→ The second statement collects all the rows in stringRDD and transfer them to driver side where they'll be printed out one per line.

(ii) Using flatMap transformation to transform lines into words.

```
> val wordRDD = stringRDD.flatMap(line =>
    line.split(" "))
> wordRDD.collect().foreach(println)
```

. flatMaps output flattens Map() transformations output into a single array.

(iii) Filtering for lines that contain the word "Awesome".

```
> val awesomeLineRDD =
    stringRDD.filter(line => line.contains("awesome"))
> awesomeLineRDD.collect.
```

(iv) Combining rows from two RDDs using union() transformation

```
> val rdd1 = spark.sparkContext.parallelize(Array(1,2))
> val rdd2 = spark.sparkContext.parallelize(Array(3,8))
> val rdd3 = rdd1.union(rdd2)
> rdd3.collect()
```

· union() doesn't remove duplicates but appends/combines the rows of both RDDs

(v) Removing stopWords using the subtract transformation.

```
>val words = spark.sparkContext.parallelize(
  List("The amazing thing about spark is that its very simple to learn")).flatMap(l=>l.split(" "))
  map(w=>w.toLowerCase)

>val stopWords = spark.sparkContext.parallelize(
  List("the it is to that")).flatMap(l=>l.split(" "))

>val realWords = words.subtract(stopWords)

>realWords.collect()
```

(vi) Removing duplicates using distinct transformation.

```
>val duplicateRDD = spark.sparkContext.parallelize
  (List("one", 1, "two", 2, "one", 2)

>duplicateRDD.distinct().collect.
```

(II) Action examples:

(i) Using collect action to see rows in small RDD

```
>val numRDD = spark.sparkContext.parallelize(
  List(1,2,3,4,5,6,7,8,9,10),2)
```

> numRDD.collect()

(ii) Counting number of rows in RDD

> numRDD.count()

(iii) Getting first row in RDD

> numRDD.first()

(IV) Getting first n rows in RDD [here n = 6]

> numRDD.take(6)

- This action collects rows from 1 partition and then moves to the next partition.

(V) Getting top k i.e. largest k values.

> numRDD.top(4)

- Returns 10, 9, 8, 7

9) What is lazy evaluation in Spark? Explain with an examples.

A. - Lazy evaluation means spark 'll wait until the last moment to execute the graph of computation instructions.

- That is, in Spark instead of modifying data immediately you build up a plan of

transformations - By waiting until last moment, Spark will produce a streamlined physical plan that'll run as efficiently as possible across the cluster.

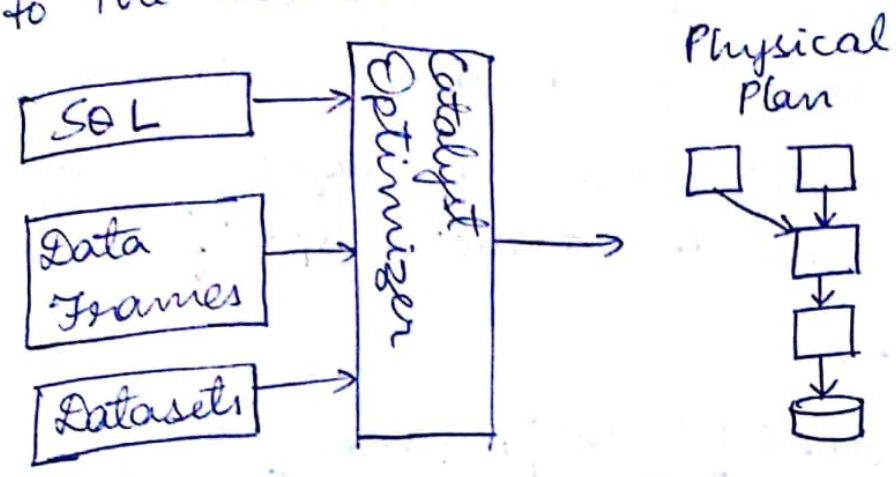
- This has immense benefits as Spark can optimize entire dataflow from end to end.

- Example: Predicate pushdown on dataframes.
  - If we build a large Spark job
  - specify filter at the end which requires us to fetch only one row from source data.
  - Spark will optimize this for us by pushing the filter down automatically.

10) With Logical & Physical plans explain how structured APIs will be executed in Spark.

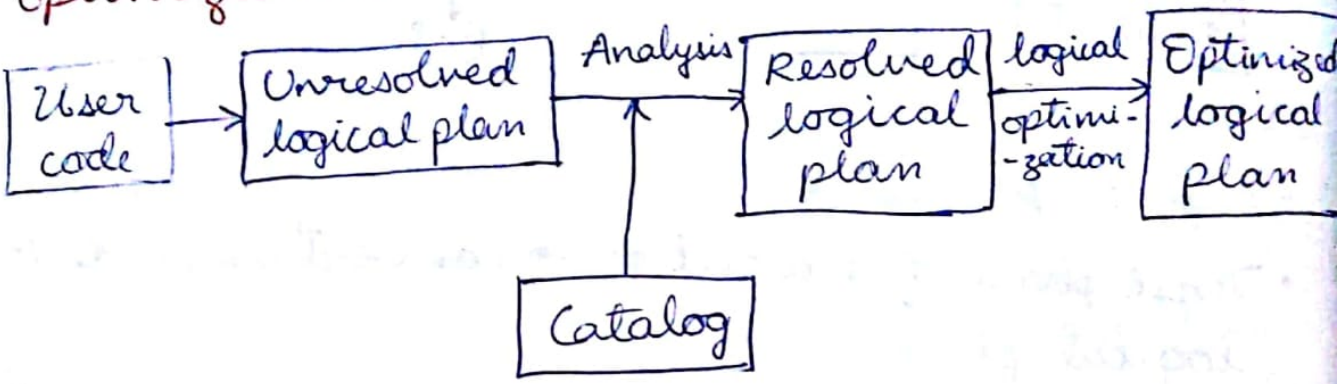
A. - Code to be executed is written
- This is submitted to spark → through console or via submitted job
- The code is passed through Catalyst Optimizer which decides how the code should be executed and lays out a plan

- The code's then run and results returned to the user.

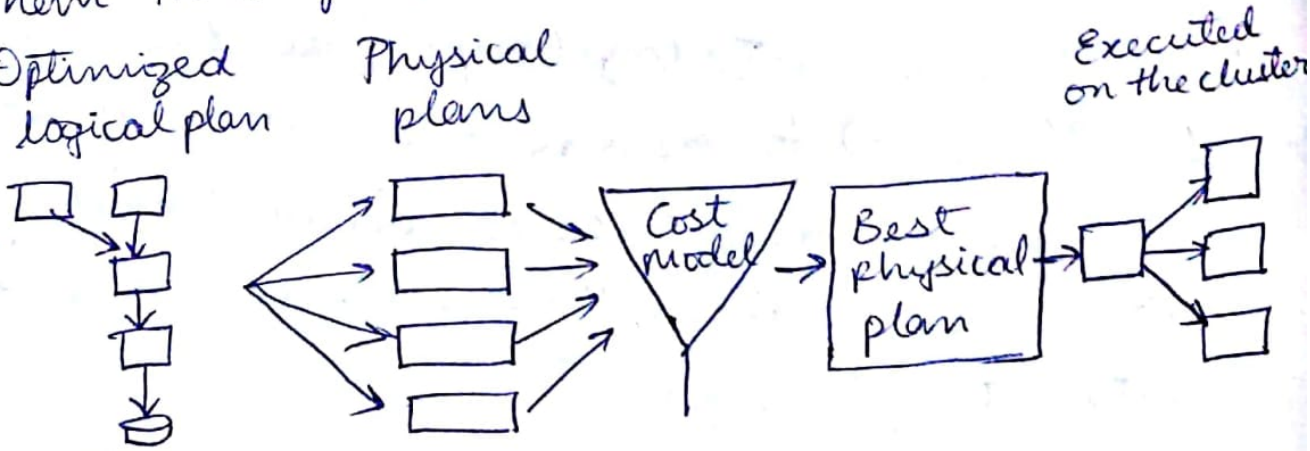


- Logical planning:
  - First phase of execution → convert user code to a logical plan.
  - Logical plan represents a set of abstract transformations that don't refer to executors or drivers, it's purely to convert users set of instructions to most optimized version. Which done by converting user code into unresolved logical plan (the tables or colums that it refers to may not exist)
  - Spark uses catalog (repo of all tables and DataFrames) to resolve them in analyzer.
  - The analyzer → rejects the unresolved plan if they're not in catalog if it can resolve them, the results passed through Catalyst Optimizer.

- Catalyst Optimizer is a collection of rules that optimize the logical plan by pushing down predicates or selections.
- Packages can extend the catalyst to include their own rules for domain specific optimizations.



- Physical planning:
  - After logical planning, Spark begins the physical planning process (Also called Spark plan)
  - It specifies how the logical plan will execute on cluster by generating different physical execution strategies and comparing them through a cost model.



11) Explain Stream processing.

A. - Stream processing is an act of continuously incorporating new data to compute result.

- Input is unbounded → it has no predetermined beginning or end.
- It's just a series of events that arrive at the stream processing system.

Eg: • Credit card transactions
• Clicks on websites.
• Sensor readings from IoT devices.

- User applications then compute queries over this stream of events.

Eg: Tracking running count of each type of events.

- The application will output multiple versions of result as it runs or keep it upto date in an external "sink" system such as key-value store.

- Now comparing it to batch processing in which computation runs on fixed input dataset (which has all the historical events)

Eg: • all website visits in the past month.
• Sensor readings for past month.

- It takes a query to compute and gives result only once.
- Both streaming and batch processing have to work together in practice.

Eg: • Streaming applications need to join input data against a dataset written periodically by a batch job.

• The output of streaming jobs is often files or tables that're queried ~~by~~ in batch jobs.

- Any business logic in your applications need to work consistently across streaming and batch execution.

Eg: Your custom code to compute user's billing amount if run ~~or~~ in streaming or batch should give the same results.

- To handle this, structured streaming was designed to interoperate easily with the rest of Spark.
- continuous applications are end-to-end applications that consist of streaming, batch, interactive jobs all working on same data to deliver end products..
- Structured streaming is focused on making

it simple to build.

12) Explain some use cases of stream processing. OU NR²I

A. (i) Notifications and alerting:

- Given some series of events, a notification or alert should be triggered if an event or series of events occur.
- This doesn't imply preprogrammed decision making.
- Alerts can be used to notify some human counterpart to take action.

Eg: Generating alert to find certain item from certain location in the warehouse to an employee at a fulfillment centre

(ii) Real time reporting:

- Many organisations use streaming systems to run real-time dashboards that any employee can look at.

Eg: Running real-time dashboards to monitor total platform usage, system load, uptime and usage of new features in an organization that runs a platform.

(iii) Incremental ETL:

- streaming's also commonly used ~~for~~ to reduce latency caused by information retrieval into data warehouse
- Spark batch jobs're often used for ETL workloads that turn raw data into a structured format to enable efficient queries.
- using structured streaming these jobs can incorporate new data within seconds.
- Its critical that data is processed exactly once in a fault tolerant manner:
  - we don't want to lose input data before it reaches the warehouse.
  - we don't want to load the same data twice
  - To not confuse queries running on it to partially written data, the streaming system has to make updates transactionally to data warehouse.

(iv) Update data to serve in real time:

- They're frequently used to compute data that gets served interactively

Eg: Google Analytics might track page visits and use streaming to keep the count up to date

- When users interact with UI, query's done on latest counts.
- To support this, streaming system must perform incremental updates to a key-value store which also must be transactional to avoid data corruption.

(v) Real-time decision making:
- In a streaming system, it involves analyzing new inputs and responding to them automatically.

Eg: Credit card fraud detection.
- It requires verification of a new transaction as to whether its fraudulent & deny it if its true.
- It has to happen in real-time for each transaction.
- Developers can implement this business logic in a streaming system and run it against a stream of transactions.
- It requires maintainance of state of each user, to track spending patterns and compare this state against new transaction.

(vi) Online machine learning:
- Here you may want to train a model on a combination of streaming and historical data.

from multiple users.

Eg: Instead of hardcoded rules based on one customer's behaviour, it needs to continuously update a model from all customers' behavior.

- Test each transaction against this.

- This is the most challenging use case since it requires:
  - aggregation across multiple customers
  - joins against static datasets.
  - integration with machine learning libraries.
  - low latency response times.

13) List out the challenges of stream processing. JV'S HO'L' T

A. - Processing out-of-order data based on application timestamps.
   - Maintaining large amounts of state.
   - Supporting high-data throughput.
   - Processing each event exactly once despite machine failures.

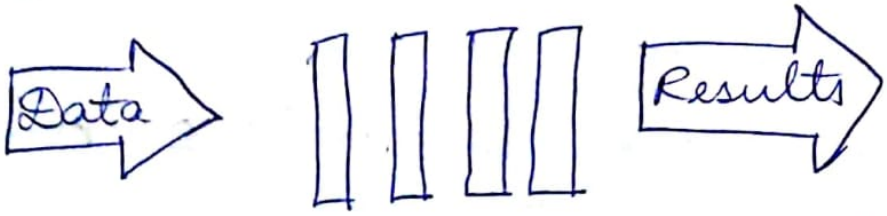
- Handling load imbalance & stragglers
- Responding to events at low latency
- Joining with external data in other storage systems.
- Determining how to update output sinks as new events arrive.
- Writing data transactionally to output streams.
- Updating your application's business logic at runtime.

4) Explain Continuous Stream Processing methods. What're its advantages and disadvantages?

A. - Here each node's continuously listening to messages from other nodes and continuously outputs new updates to child nodes.

Eg: • Application has map reduce computation over many input streams.
- Here, each map node would read records from input source, compute its function on them & send them to reducer.

- Reducer'd update its state whenever it gets a new record.
- Key idea : It happens on each record.



One record @ a time

- Advantages :
- When total input rate is low, it offers offers lowest possible latency as each node responds immediately to a new message.
- Disadvantages :
- They generally have lower throughput because they incur significant amount of overhead per-record.

[Eg : calling OS to send a packet to a downstream node]

- They're fixed topology of operators that can't be moved at runtime without stopping the whole system
- This introduces load balancing issues

15) Explain Micro-Batch ~~sysl~~ stream processing method. What're its advantages and disadvantages?

A. - They wait to accumulate small batches of input data, then process each batch in parald using a distributed collection of tasks.

Microbatches of dataframes

• Advantages:
- They often achieve high throughput per node as they leverage same optimizations as batch systems
- They need fewer nodes to process same rate of data
- They can use dynamic load balancing techniques to handle changing workloads.

• Disadvantages:
- It has higher base latency as it waits to accumulate micro-batch.

→ In practice applications're large scale and Sparks traditionally impleted as micro-batch processing

16) Write short notes on DStreaming and structured streaming.

A. - DStream API :

- Its used broadly for stream processing.
- Pros: hide
  - Its widely used due to (i) its high level API interface
  (ii) simple exactly-once semantics
  (iii) Supports interactions with RDD code, such as joins with static data.
  (iv) Its operating difficulty isn't much more than operating a normal Spark cluster.
- Limitations : nop
  (i) Its based on Java/Python objects and functions as opposed to a richer concept of structured tables in Dataframes and datasets. This limits engine's opportunity to perform optimizations.
  (ii) Its purely based on processing time. To handle event-time operations, applications've to implement them on their own.
  (iii) DStreams can operate only in a micro-batch fashion, also exposes micro-batch duration in some parts of API, making it diffi

to support alternative execution modes.

-Structured streaming : H^2E^3

. Its a higher-level streaming API built ground up on Spark's Structured API s

. Its available in all environments where structured processing runs including Scala, Java, Python, R and SQL.

. Its a declarative API based on high-level operations, but since its built on structured data model, it can perform it can perform more types of optimizations automatically.

. It has native support for event time data.

. Its designed to ease the building of end-to-end continuous applications that combine streaming, batch and interactive queries.

. Its a major help as developers don't need to maintain separate streaming version of their batch code and risk the versions falling out of sync.

17) Write short notes on Data Sources and Data Sinks of structured streaming in Spark.

A. -Data sources:

. In batch processing, data sources a static dataset

that resides on some storage system. In structured streaming, they may produce continuous data which may never end and the producing rate may vary overtime.

- Structured streaming provides the following sources: SKaRF

- Kafka source: Most popular data source and requires Apache Kafka with version 0.10 or higher.
- File source: Files are located on either the local file system, HDFS or S3. As new files are dropped into a directory, this data source will pick them up for processing.
- Socket source: This is for testing purposes only and reads UTF-8 data from a socket listening on a certain host and port.
- Rate source: This is for testing and benchmark purposes only. It can be configured to generate number of events per second where each event consists of a timestamp and a monotonically increasing value.

Output modes: Its a way to tell Structured Streaming how the output data should be written to a sink.
- This concept is unique to streaming processing in Spark. There are 3 options.

(i) Append mode:
- Its the default mode
- Only new rows that were appended to result table will be sent to the specified sink.

(ii) Complete mode:
- The entire result table'll be written to the output sink.

(iii) Update mode:
- Only the updated rows in the result table will be written to output sink.

- Trigger types:
- Structured Streaming engine uses the trigger information to determine when to run the provided streaming computation logic
- Trigger types are: 'Fixed-Interval', 'One-Time', 'Continuous'.

- Data sinks : ⊂ᴍ &KarF²

- They're at the opposite end of data sources.
- They're meant for storing output of streaming applications.
- Its important to recognize which sinks can support which output mode & whether they're fault-tolerant.
- Kafka sink · Requires Apache Kafka version 0.10 or higher.
- File sink : Its a destination on a file system, HDFS or S3.
- Foreach sink : This is meant for running arbitrary computations on the rows in the output.
- Console sink : This is for testing & debugging purposes only & when working with low-volume data. The output is printed to the console on every trigger.
- Memory sink : This is for testing and debugging purposes only when working with low volume data. It uses the memory of the driver to store the output.