

CLAIRVOYANT



design



engineer



deliver

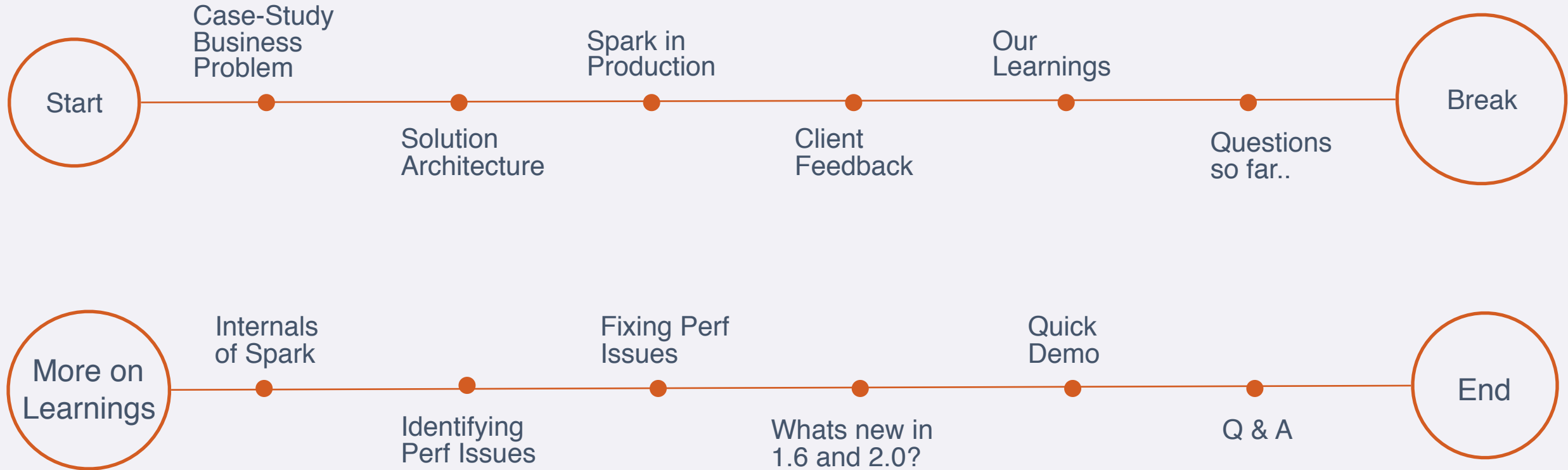


Case Study & Performance Tuning

Shantanu Mirajkar, Vijay Datla
Clairvoyant India Pvt. Ltd.

CLAIRVOYANT

Presentation Agenda



Business



4.5+ TB data monthly 1 billion+ Bids 4 million placement 1.5 million Web-sites 345 million Users

25+ TB data monthly

3-4 billion+ Bids

4

Existing System



Transfer of data from Kafka to 3rd party endpoints using https

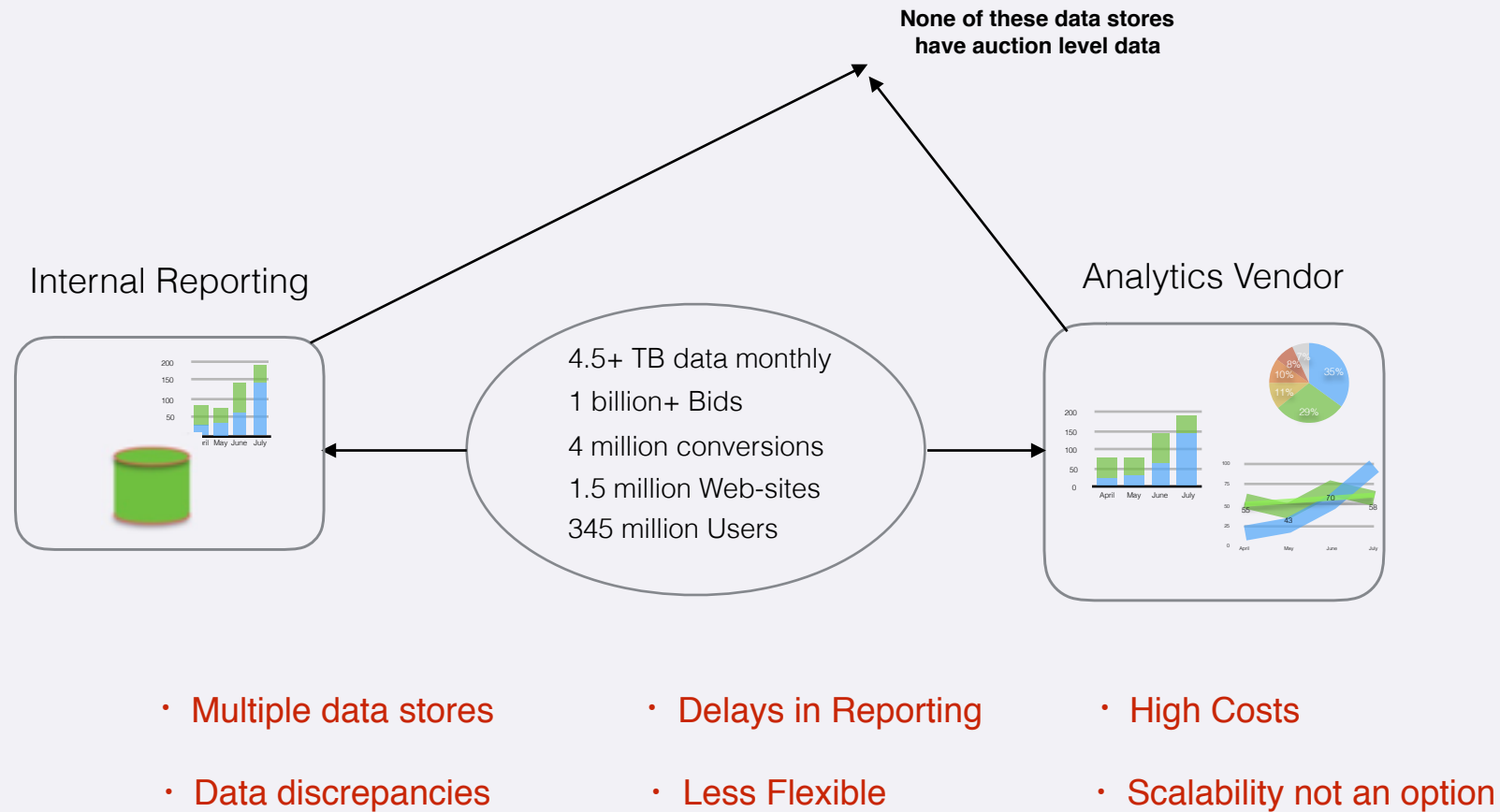
Proprietary jobs transform the data

Proprietary join and aggregate the data incrementally throughout the day

Proprietary jobs daily re-join and aggregate data

Proprietary dashboards

Problem



6

Solution

Goals Explained

- Cut Costs
- Self Service analytics
- Near real time insights into business
- Most Granular Data
- High Volume performance
- Partnership instead of Vendor relationship
- Avoid Data Duplication

Choice made

- Cloudera Ingestion Tools and Data Lake
- ZoomData
- Spark, Impala
- Open Source Technologies
- Cloudera, Zoomdata, Clairvoyant.

7

Initial POC

Hive approach

- Loaded events into a Hive table
- Apply Hive DML statements for any transformation
 - Cleansing, de-duping, and other transactions.
- Load the data into Impala (in Parquet format)

Challenges with Hive

- SLAs for near real time data processing not met
 - 30 mins SLA for new data (Hive took around 2 hours to process the same data set)
 - Failure cases to be handled as soon as possible
 - Resource constraints (Hive reading from and writing to disk)
- Applying event based cleansing, de-dup and transformation logic in separate scripts and testing them and maintenance became a challenge.

Spark Core with SQL

Immediate goals:

Meets SLAs due to In-memory distributed data processing

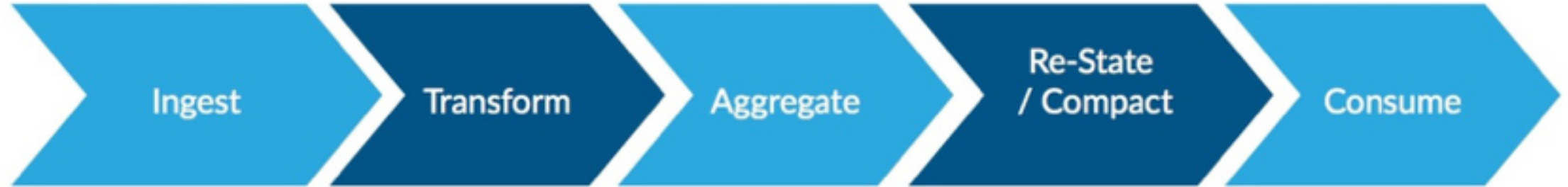
Meets the SLA (around 10 mins)

Strategic Goals:

Streaming requirements

Possible ML application

Architecture



Events data written to HDFS in raw format via Kafka, Flume

Lookup data: FTP periodically from S3

Spark Jobs incrementally cleanse, de-dup, transform and enrich data

Join all the events to load the target tables with partitioning on Date and Time into Impala

Daily re-aggregation and compaction to meet the data consistency and performance needs

ZoomData powered Dashboards & Analytics

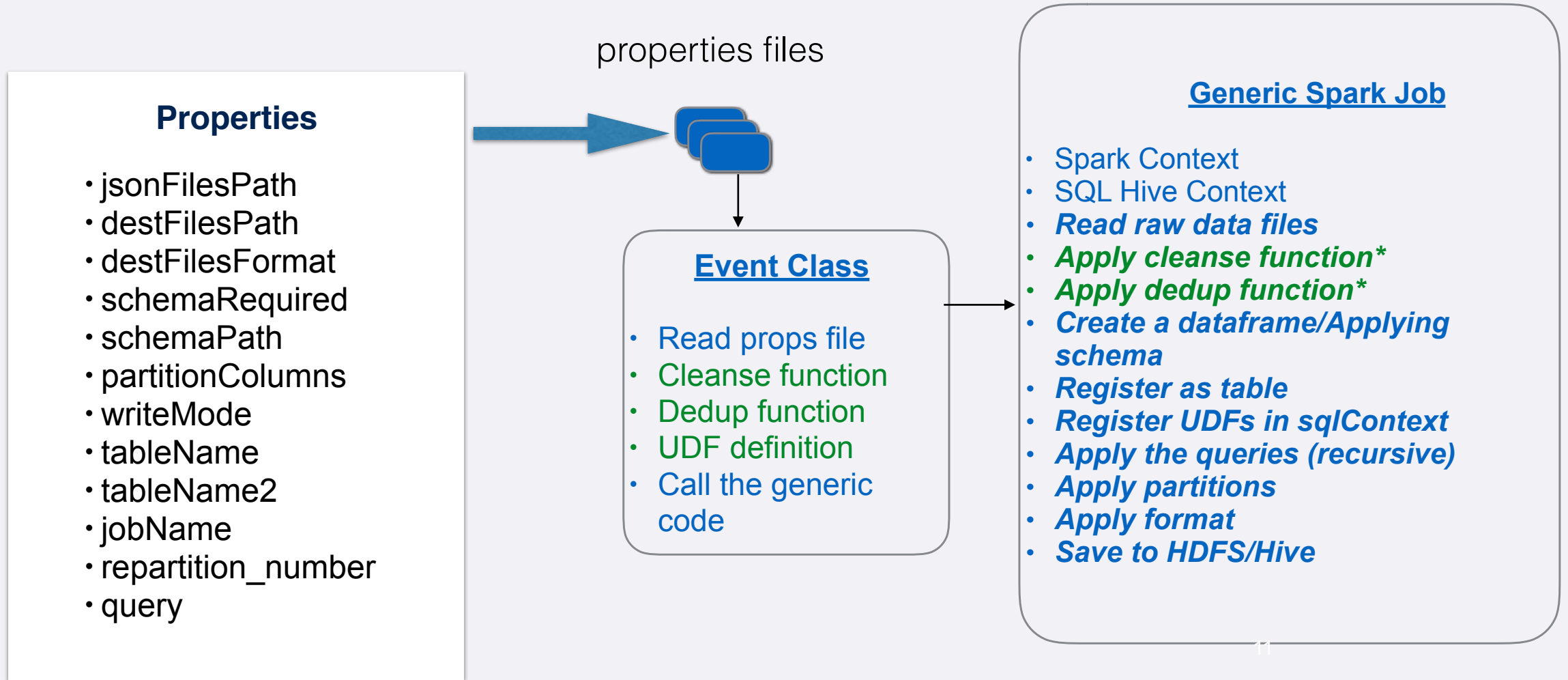
Architecture

The challenge again is, in 15 minutes:

- Gather new input files list
- Clean and Deduplicate using Spark
- Load to Staging Event Tables
- Update Hive Metastore and Invalidate Metadata
- Join the staged events to generate final table
- Move the staging data to their respective target tables
- Clean the staging
- Re-state the final table ever day



Spark for Data Processing



Constraints in Production

- 45 node cluster (6 for Kafka and 6 for Flume)
 - 600 vcores
 - 3.8TB of RAM
- Cluster highly utilised by
 - Custom Hive Queries
 - Custom Spark jobs
 - Impala queries
 - other daily maintenance jobs
- Allotting more number of executors and memory to Jobs is not an option.

Only 30 % of the resources will be available for all the jobs.

12

Learnings

Input size of files should not be un-even:

There could be performance issues with the jobs in case the input files are not evenly sized.

Read the data into DataFrames instead of RDDs wherever possible:

Transformations/Actions performed on bare RDD may take more time than if performed on DataFrames Registered SQL tables .. Reason?? Tungsten and Catalyst

Beware of running complex sql queries in Spark SQL (Need thorough testing):

When running complex inner queries on Registered tables in SQL, you need to be careful about the behaviour or Spark. We saw some in-consistency in the way Spark returned the results in 1.6. Not sure if things have been fixed or we have goofed up something. ;)

Try using the filenames to read the data into Spark to avoid staging the files:

Most of the times, people tend to copy the files to be processed into a staging folder and then process them. However, if the files are already in HDFS, just maintaining a list of new files as a dictionary or List and reading them directly via spark should save the time to copy the data to staging and removing from it.

Learnings (cont..)

Exploding Json is easy using HiveContext:

Using HiveContext where ever possible is recommended. Especially with Json data sets. Ex: Explode function can be applied on data sets using HiveContext but not SQLContext

Caching alone may not help improve the performance.

Opt for Kryo Serialisation when possible:

Serialisation is another thing that was creating issues. Going with Kryo made the situation better.

More resources allocation is not always good:

Assigning the resources was something that we thought would improve the performance a lot.

However, thats not true. Unless the basic RDD tuning is done correctly, giving more resources will create more problems than reducing few.

YARN, User level resource restrictions may make the applications starve and run slow.

Again, finding that sweet spot is crucial

Learnings (cont..)

Use Scala if you can afford to undergo the learning curve:

From a development perspective, we felt that using Scala is much more easier than using Java. Reasons being, less code, more readability and passing functions as args to functions.

Keep the partitions to the optimal number at all stages:

We saw that there are around 200 output files written into HDFS. This is causing a lot of small files to be written which forced us to come up with compaction process.

```
spark.shuffle.sort.bypassMergeThreshold
```

The more the number of partitions to work on, the higher the chances of a task going into hanging stage. So partition wisely.

Few tasks hang during the execution, go for speculation properties:

One way to deal with this issue is to use the configuration property “spark.speculation.interval”

Spark Applications: Performance Tuning

Quick basics

Reasons for slowness of a system

Application:

In-efficient algorithms

One a single computer

CPU, Memory, IO, Context switching

Distributed computing

All the above

Code/Data movement in network

Coordination amongst processes

Focus of this talk

Performance can be tuned at different levels:

Optimal algorithm to give the required output. Lets call it A

Optimal use of APIs of the Framework on which A is supposed to run. Lets call it B1.

Optimal Configurations for the Framework on which A is supposed to run. Lets call it B2.

Modifying the Framework to overcome the inherent limitations. Lets call it C

B1+B2 will be our focus

Fixing C will make a spark committer. :) Lets leave it for the upcoming meet ups. :)

Problem that Spark attempted to solve

MapReduce is inefficient for *multi-pass* applications that require low-latency data sharing across multiple parallel operations

Iterative algorithms

Interactive data mining

Streaming applications

Internals of Spark

Spark Basics

- RDD
- Spark Application execution model
 - Resource allocation
 - Data distribution for processing
- Data processing through various stages
- Data Locality

RDD

RDD is an interface that declares the below functions

Lineage

Partitions

Dependencies

Computation (functional Paradigm)

fundamental data structure

immutable distributed collection

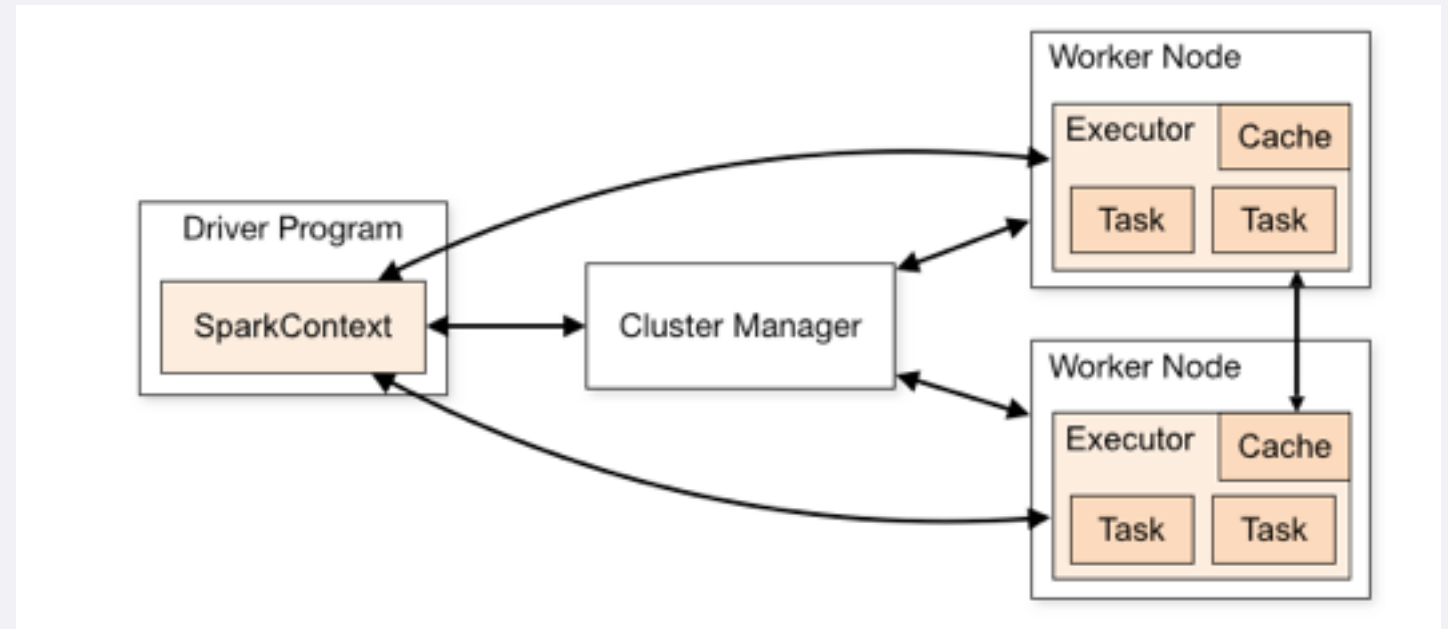
Optimisation

Partitioner

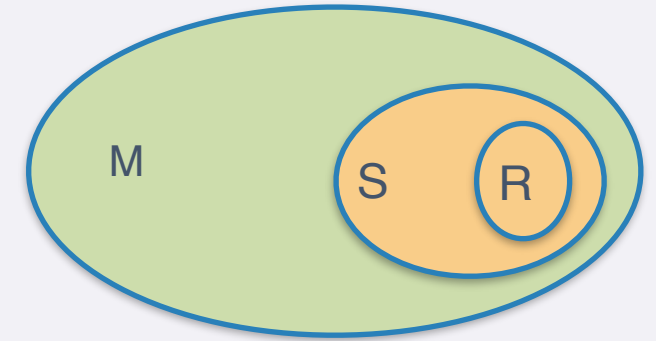
Preferred Location

Execution Model

- Resource allocation
 - Executors, Cores, Memory
- Data distribution for processing
 - Partitioning of Data
- Processing data via various stages
 - Consolidation of operations into stages.



Memory Allocation



If M is not being used S can occupy the whole memory. Vice versa.

If M requires more memory, M can evict data from S till certain threshold (R)

However, S can not evict M

M → Execution

S → Storage

spark.memory.fraction (0.75) → decides the fraction of JVM heap that's allowed to M

spark.memory.storageFraction (0.5) → fraction of M for R

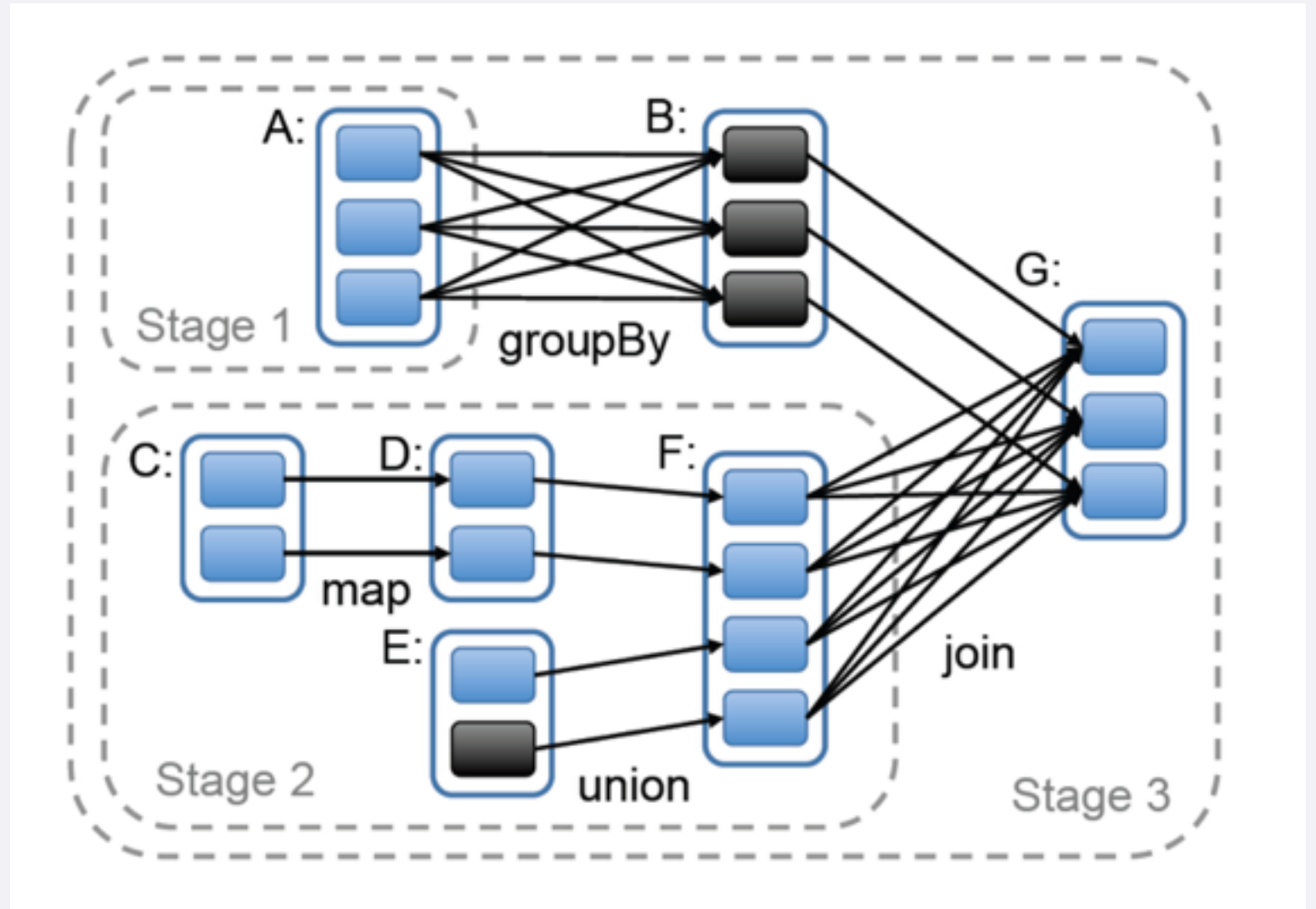
Ex: With 4 GB Heap memory for executor,

300 → reserved memory

949MB → User memory,

2847MB → spark memory (shared by M and S)

Stages in a spark Job



Tungsten (Spark Core Optimisation)

- **Memory Management and Binary Processing**

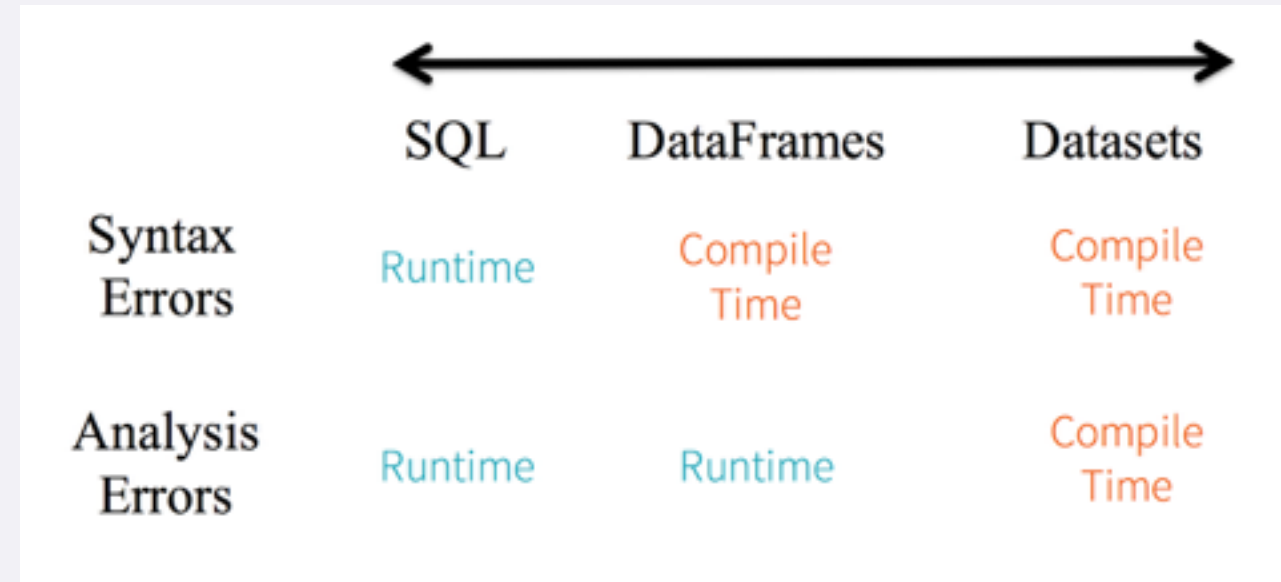
- Spark is constrained by CPU/Memory?
 - Disk IO and network are not an issue:
 - Serialisation operations are taking time which are CPU bound.
 - GC becomes a bottleneck.

- **Cache-aware Computation**

- Effective use of L1/ L2/L3 CPU caches

Compiler Optimisations

- **Catalyst**
 - General library for representing **trees**
 - Applying **rules** to manipulate them
- **DataFrame and Dataset APIs**
 - built on top of the Spark SQL engine
 - Catalyst optimises query execution plan



Debugging

Debugging the spark applications:

- > Spark application UI
- > Understanding the time taken by tasks
- > Understanding the memory occupied by the tasks
- > Identify the skewed Data partitions

Important Debug Tips

<i>Mode</i>	<i>Advantage</i>
Single threaded	<i>sequential execution allows easier debugging of program logic</i>
Multi-threaded	<i>concurrent execution leverages parallelism and allows debugging of coordination</i>
Pseudo-distributed cluster	<i>distributed execution allows debugging of communication and I/O</i>

Compile Time Debugging

```
val textFile = sc.textFile("hdfs:///user/vijaydatla/demo/*")
```

```
val counts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
```

See the stages:

```
counts.toDebugString
```

```
res0: String =
```

```
(2) ShuffledRDD[4] at reduceByKey at <console>:23 []
```


```
+- (2) MapPartitionsRDD[3] at map at <console>:23 []
```

```
  | MapPartitionsRDD[2] at flatMap at <console>:23 []
```

```
  | MapPartitionsRDD[1] at textFile at <console>:21 []
```

```
  | hdfs:///user/vijaydatla/demo/* HadoopRDD[0] at textFile at <console>:21 []
```

Spark UI (Jobs)

1.5.0-cdh5.5.0

JobsStagesStorageEnvironmentExecutorsSQL

Spark shell application UI

Spark Jobs (?)


Total Uptime: 9.7 min
Scheduling Mode: FIFO
Completed Jobs: 1

[Event Timeline](#)

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	count at <console>:26	2016/08/08 23:28:44	27 s	2/2	4/4

Spark UI (Stages)

1.5.0-cdh5.5.0

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application UI

Details for Job 0

Status: SUCCEEDED

Completed Stages: 2


▶ Event Timeline

▶ DAG Visualization

Completed Stages (2)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at <console>:26	+details	2016/08/08 23:29:10	0.3 s	2/2			717.0 B	
0	map at <console>:23	+details	2016/08/08 23:28:44	25 s	2/2	1116.0 B			717.0 B

Spark UI (Stages Details)

 1.5.0-cdh5.5.0

JobsStagesStorageEnvironmentExecutorsSQL

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 6 s
Input Size / Records: 49.4 KB / 46
Shuffle Write: 2.3 KB / 264

- ▶ [DAG Visualization](#)
- ▶ [Show Additional Metrics](#)
- ▶ [Event Timeline](#)

Summary Metrics for 3 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	5 s	5 s
Scheduler Delay	95 ms	95 ms	0.1 s	2 s	2 s
GC Time	0 ms	0 ms	0 ms	1 s	1 s
Input Size / Records	744.0 B / 1	744.0 B / 1	16.0 KB / 21	32.7 KB / 24	32.7 KB / 24

Spark UI (Stages Details)

Shuffle Read Size / Records	503.0 B / 60	503.0 B / 60	795.0 B / 87	1018.0 B / 117	1018.0 B / 117
-----------------------------	--------------	--------------	--------------	----------------	----------------


Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	CANNOT FIND ADDRESS	0.6 s	3	0	3	2.3 KB / 264

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	GC Time	Shuffle Read Size / Records	Errors
0	3	0	SUCCESS	PROCESS_LOCAL	1 / quickstart.cloudera	2016/08/08 23:55:37	0.2 s	91 ms		503.0 B / 60	
1	4	0	SUCCESS	PROCESS_LOCAL	1 / quickstart.cloudera	2016/08/08 23:55:37	67 ms	40 ms		1018.0 B / 117	
2	5	0	SUCCESS	PROCESS_LOCAL	1 / quickstart.cloudera	2016/08/08 23:55:37	75 ms	40 ms		795.0 B / 87	

Spark UI (Cached)

 1.5.0-edh5.5.0

Jobs | Stages | Storage | Environment | Executors | SQL

Spark shell application UI


Spark Jobs (?)

Total Uptime: 20 min
Scheduling Mode: FIFO
Completed Jobs: 4
Event Timeline

Completed Jobs (4)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	count at <console> 20	2016/08/08 23:39:12	0.3 s	1/1 (1 skipped)	2/2 (2 skipped)
2	count at <console> 20	2016/08/08 23:38:41	22 s	1/1 (1 skipped)	2/2 (2 skipped)
1	count at <console> 20	2016/08/08 23:35:30	18 s	1/1 (1 skipped)	2/2 (2 skipped)
0	count at <console> 20	2016/08/08 23:28:44	27 s	2/2	4/4

Spark UI (Storage Details)

 1.5.0-cdh5.5.0

Jobs

Stages

Storage

Environment

Executors

SQL


Spark sl

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
ShuffledRDD	Memory Deserialized 1x Replicated	3	100%	7.3 KB	0.0 B	0.0 B

Spark UI (Executors Info)

1.5.0-cdh5.5.0

Jobs

Stages

Storage

Environment

Executors


SQL

Executors (2)

Memory: 7.3 KB Used (1069.1 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
2	quickstart.cloudera:39835	3	7.3 KB / 534.5 MB	0.0 B	0	0	3	3	6.2 s	0.0 B	2.3 KB	0.0 B	stdout stderr	Thread Dump
driver	172.16.52.249:37694	0	0.0 B / 534.5 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

Spark UI (Skewed Data - Stragglers)

1.5.0-cdh5.5.0

Jobs

Stages

Storage

Environment

Executors

SQL

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 6 s
Input Size / Records: 49.4 KB / 46
Shuffle Write: 2.3 KB / 264

[▶ DAG Visualization](#)
[▶ Show Additional Metrics](#)
[▶ Event Timeline](#)

Summary Metrics for 3 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	5 s	5 s
Scheduler Delay	95 ms	95 ms	0.1 s	2 s	2 s
GC Time	0 ms	0 ms	0 ms	1 s	1 s
Input Size / Records	744.0 B / 1	744.0 B / 1	16.0 KB / 21	32.7 KB / 24	32.7 KB / 24

Tuning

- Resource allocation
 - executors
 - executor-memory
 - executor-cores
 - Role of Serialisation
- Choosing the right transformations (when to use what)
- Impact of Partitioning

Tuning by Resource allocation

`--num-executors`

`--executor-cores`

`--executor-memory`

Data locality related config properties

Memory

Tips:

- Prefer arrays of objects, and primitive types as opposed to standard Java or Scala collection classes
- Avoid nested structures with a lot of small objects
- Use numeric IDs or enumeration objects instead of strings for keys.
- Use caching mechanisms
 - MEMORY_ONLY_SER*
 - This will help reduce the GC overhead.
 - In case the data size is large enough to be folded in the RAM

Data Serialisation

Java Serialisation (default)

Use Externalisation if you think that would help.

Kryo Serialisation (supported by Spark)

More compact than Java serialisation.

Register the classes for best performance.

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

Partitioning scheme and repartitioning

The **coalesce transformation** creates a dataset with fewer partitions than its parent dataset.

The **union transformation** creates a dataset with the sum of its parents' number of partitions.

The **cartesian transformation** creates a dataset with the product of its parents' number of partitions.

If the stage is receiving input from another stage,
the transformation that triggered the stage boundary accepts a **numPartitions argument**:

```
val rdd2 = rdd1.reduceByKey(_ + _, numPartitions = X)
```

Determining the optimal value for X requires experimentation.

Choose the right Transformations

Minimise the number of shuffles and the amount of data shuffled

repartition
join
cogroup
and any of the *By or *ByKey

flatMap-join-groupBy. When two datasets are already grouped by key and you want to join them and keep them grouped, use **flatMap-join-groupBy** or **cogroup**. This avoids the overhead associated with unpacking and repacking the groups.

Other tuning options

reduceByKey when the input and output value types are different. For example, consider writing a transformation that finds all the unique strings corresponding to each key. You could use map to transform each element into a Set and then combine the Sets with reduceByKey:

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2)).reduceByKey(_ ++ _)
```

This results in unnecessary object creation because a new set must be allocated for each record.

Instead, use aggregateByKey, which performs the map-side aggregation more efficiently:

```
val zero = new collection.mutable.Set[String]()  
rdd.aggregateByKey(zero)((set, v) => set += v, (set1, set2) => set1 ++= set2)
```

Ensure decent parallelism (2-3 tasks per core)

```
spark.default.parallelism
```

Convert large objects that are being used in the tasks in to Broadcast variables.

tasks larger than about 20 KB should be considered for optimisation.

Spark 2.0 (stepping towards optimisation)

Spark as a Compiler

- Whole-stage code generation (optimise code by eliminating virtual functions)
- Catalyst optimizer (nullability propagation, vectorized Parquet decoder)

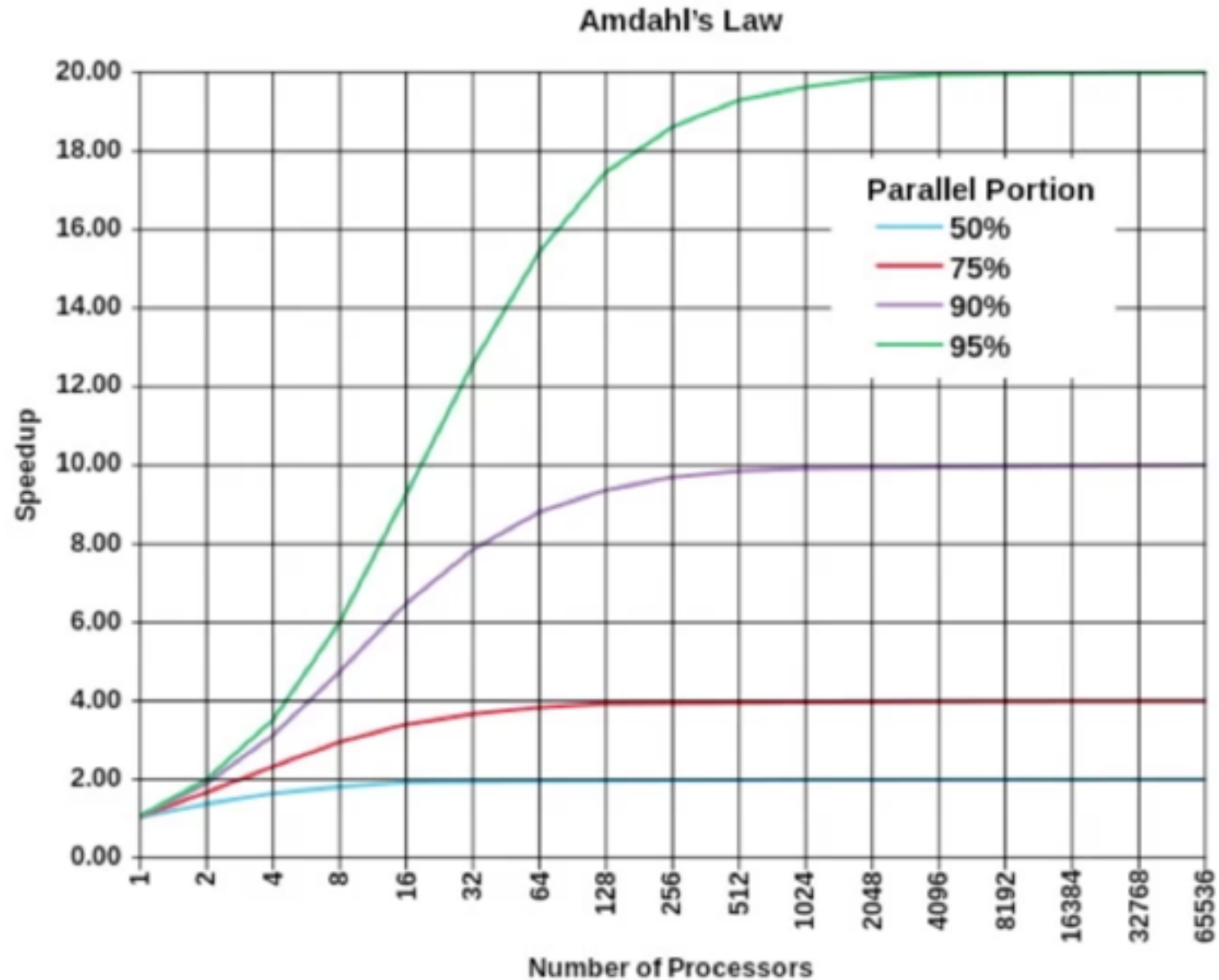
SQL and Streamlined APIs

- *Standard SQL support*
- Unifying DataFrame/Dataset API
- Unifying DataFrames and Datasets in Scala/Java
- SparkSession
- DataFrame based ML APIs (spark.ml, the new lib)
- etc

Structured Streaming

- a (surprisingly small!) extension to the DataFrame/Dataset API

Amdahl's Law



Amdahl's Law

It doesn't how many machines you throw at a problem... at some point it is just not going to run any faster

THANK YOU