IDA
INFORMATION
DATA & ANALYTICS

Databricks IDA Technical
Neighborhood & Databricks CoE
Welcome You To

Databricks Performance Optimization Workshop

10th & 12th of May 2022, 13.30-16.00 CET

Florent Moiny

Resident Solutions Architect
Databricks CoE

Copyright of Shell International

databricks

SAFETY & HEALTH AT HOME

Make sure your smoke alarms work.

Maintain clear walkways and fire exits.

Have an emergency and evacuation plan in place.

Clean surfaces frequently.

Make sure your workspace is ergonomically sound.

Ensure there's adequate lighting in the room when you work.

SAFETY & HEALTH AT WORK

Know what the fire alarm sounds like

Make sure that you can hear the fire alarm

Know locations for fire alarms, extinguishers, emergency exits & muster points

Know your local location's emergency number

Know the location of the nearest AED portable defibrillator

ON THE MOVE?

Do not take this call, or any other call, while driving – ever

In the event of any kind of emergency, please leave the call – promptly and safely

A few remarks on our engagement

1. Compact content – please avoid other distractions ☺
2. Please be polite and respectful when engaging with each other
3. You may type your questions in the chat during the session or attend the optional Q&A at the end; Please keep the mic muted otherwise.
4. Participants who complete both sessions (Tuesday & Thursday) can receive a certificate of completion visible in the **Databricks Academy**, with two conditions:
   a) They must be registered on the Databricks Academy with their @shell.com account (instructions in the chat)
   b) They must fill in the **ATTENDANCE** MS Form at the end of the two days
5. Quiz with prizes on both days ☺
6. Your feedback is extremely important – please fill in the feedback forms at the end

Copyright of Shell International B.V.

RESTRICTED

Agenda

Tuesday

- 15m - Intro
- 15m - Spark UI + Ganglia
- 30m - Spill
- 5m - Break
- 30m - Shuffle & Joins
- 25m - AQE: Join & Skew Optimizations
- 30m - Forms - Quiz - Q&A

Thursday

- 5m - Recap
- 20m - Serialization
- 30m - Key Ingestion Concepts
- 5m - Break
- 30m - Storage
- 30m - Cluster Design & Configuration
- 30m - Forms - Quiz - Q&A

The Five Most Common Performance Problems

Five Basic Problems

The most egregious problems fall into five categories:

- **Spill**: The writing of temp files to disk due to a lack of memory
- **Skew**: An imbalance in the size of partitions
- **Shuffle**: The act of moving data between executors
- **Storage**: A set of problems directly related to how data is stored on disk
- **Serialization**: The distribution of code segments across the cluster

Five Basic Problems - Why it's hard

- Root sourcing problems is hard when one problem can cause another
- **Skew** can induce **Spill**
- **Storage** issues can induce excess **Shuffle**
- Incorrectly addressing **Shuffle** can exacerbate **Skew**
- Many of these problems can be present at the same time

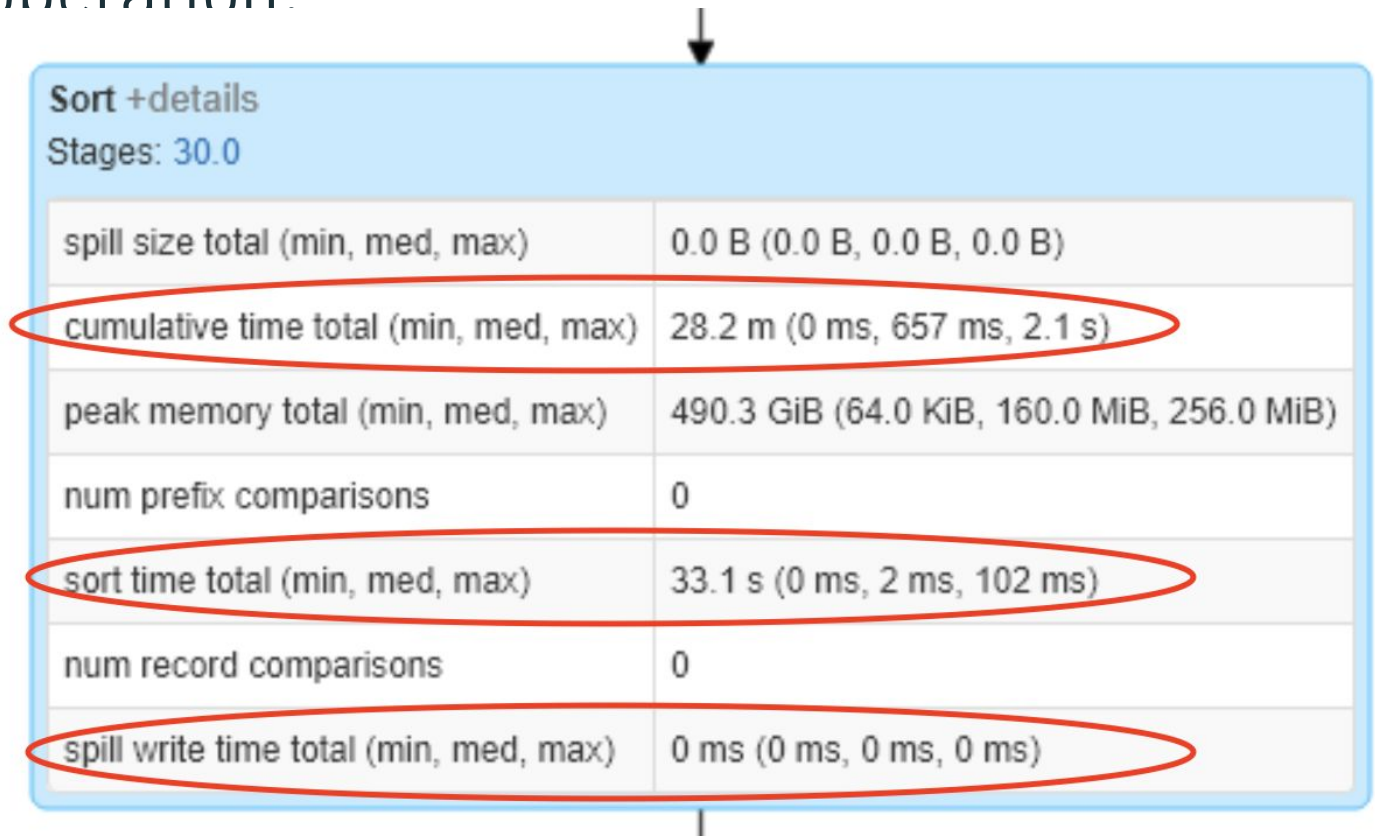
Five Basic Problems - What we need

- An interface to retrieve application performance metrics: the Spark UI
- An interface to retrieve cluster consumption metrics (more about tuning clusters): Ganglia
- A process to benchmark our experiments based on those performance metrics: the *noop* write

Monitoring
Spark UI & Ganglia Metrics

What is the Spark UI?

A suite of web UIs to monitor the status and resource consumption of your Spark cluster at the application level. We can see for instance the execution time for an operation:



What are Ganglia metrics?

Consumption metrics at the cluster level, e.g.:

- CPU usage
- Memory usage
- I/O usage

Example

See the Spark UI Simulator

Benchmarking

Benchmarking

There are generally three common approaches to benchmarking:

- The **`count()`** action
- The **`foreach()`** action with a do-nothing lambda
- A **`noop`** (or no operation) write

Benchmarking

See Experiment #5980

Benchmarking - In Action, Part 1

See Experiment #5980

- Compare **Step B-1** and **Step B-2**
  - Note the total duration
  - Why did **Step B-1** take 2x longer than **Step B-2**?
- See **Step C**, the **`count()`** operation
  - Note the duration
  - Note that the Python and Scala samples are nearly identical
  - Note the number of jobs
  - Why is there one less job as compared to **Step B-2**?

Benchmarking - In Action, Part 2

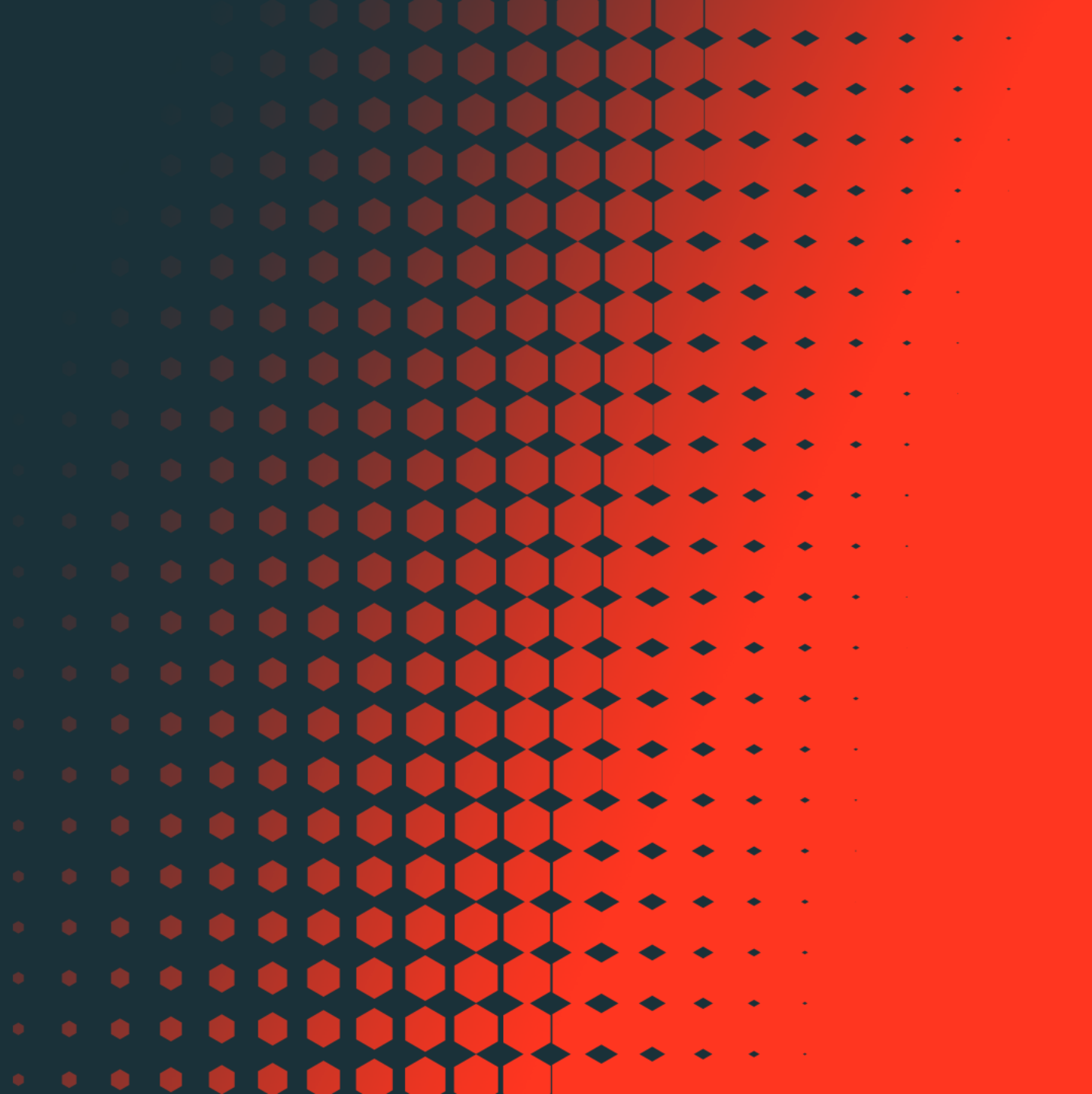
See Experiment #5980

- See **Step D**, the **`foreach()`** action with a do-nothing lambda
  - Note the total duration (esp compared to the **`count()`** action)
  - Compare the Scala a Python versions
  - Why is the Python version significantly slower than the Scala version?
- See **Step E**, the **`noop`** write.
  - Note the total duration of both the Python and Scala

Benchmarking - Review

- About **Step B-1** and **Step B-2**
  - Loading the schema in **Step B-1** and not **Step B-2** provided a side effect
- About the **`count()`** action
  - Count is optimized - doesn't process all the data
  - Metadata & columnar reads affect execution
- About the **`foreach()`** action
  - Simulates processing of every record
  - The serialization side effect is quite significant in Python
- About the **`noop`** with a schema - it just works as expected!

Spill



Spill

- Spill is the term used to refer to the act of moving an RDD from RAM to disk, and later back into RAM again
- This occurs when a given partition is simply too large to fit into RAM
- In this case, Spark is forced into [potentially] expensive disk reads and writes to free up local RAM
- All of this just to avoid the dreaded OOM Error

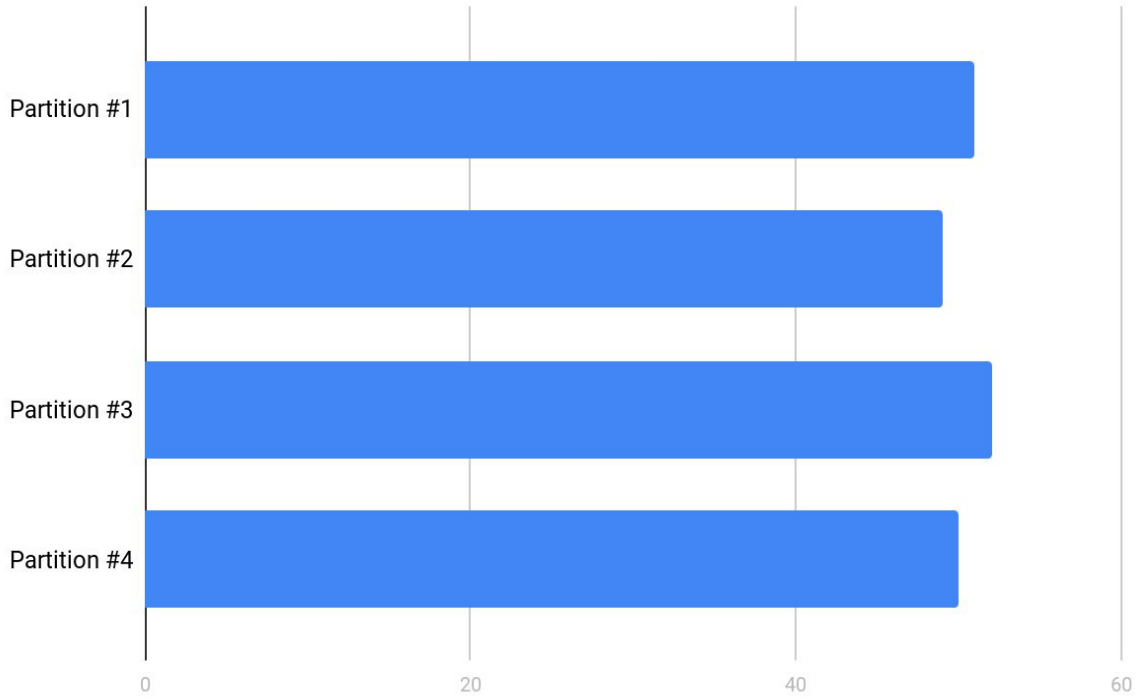
Spill - Examples

There are a number of ways to induce this problem:

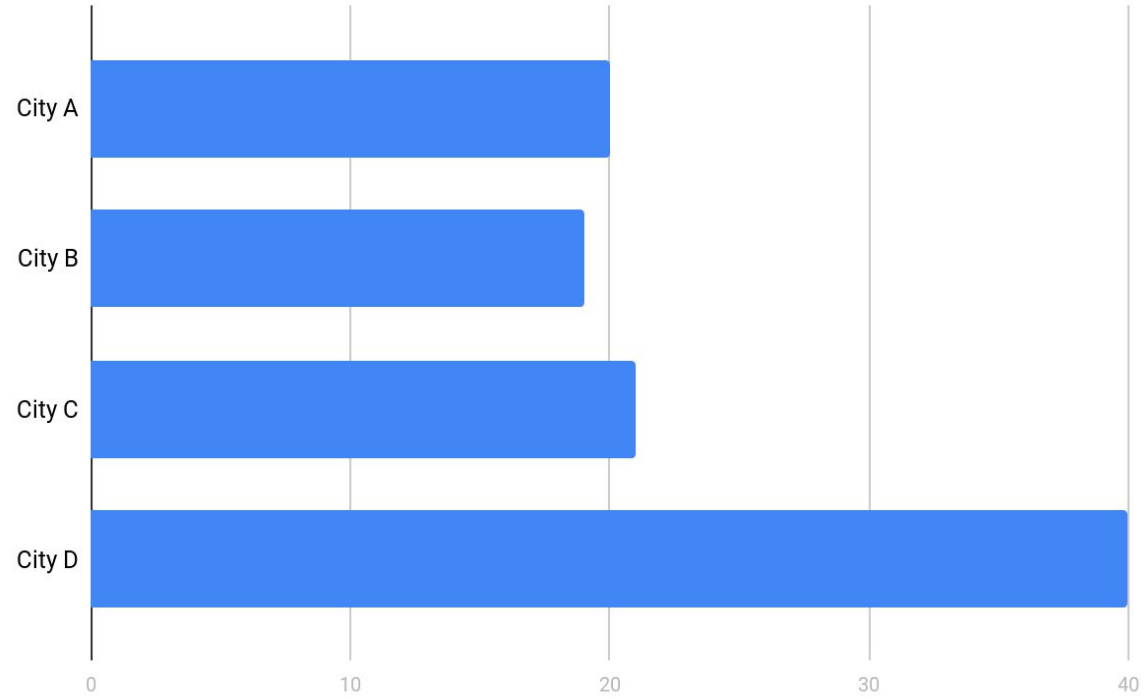
- Set **spark.sql.files.maxPartitionBytes** too high (default is 128 MB)
  ...more on maxPartitionBytes later
- The **explode()** of even a small array
- The **join()** or **crossJoin()** of two tables which generates lots of new rows
- The **join()** or **crossJoin()** of two tables by a skewed key
- Aggregating results by a skewed feature

Skew - Example

Before aggregation



After aggregation by city



Spill - Memory & Disk

In the Spark UI, spill is represented by two values:

- **Spill (Memory):** For the partition that was spilled, this is the size of the data as it existed in memory

- **Spill (Disk):** Likewise, for the partition that was spilled, this is the size of the data as it existed on disk

The two values are always presented together

The size on disk will always be smaller due to the natural compression gained in the act of serializing that data before writing it to disk

Spill - In the Spark UI

A couple of notes:

- Spill is only represented in the details page for a single stage…
  - **Summary Metrics**
  - **Aggregated Metrics by Executor**
  - The **Tasks** table
- Or in the corresponding query details
- This makes it hard to recognize because one has to hunt for it
- When no spill is present, the corresponding columns don’t even appear in the Spark UI - that means if the column is there, there is spill somewhere

Spill - Examples

See Experiment #6518

- Note the four examples:
  - Step B: Spill induced by ingesting large partitions
  - Step C: Spill induced by unioning tables
  - Step D: Spill induced with explode operations
  - Step E: Spill induced by a skewed join
- For each example…
  - Find and note the total **Spill (Memory)** and **Spill (Disk)**
  - Find and note the min, median and max **Spill (Memory)** and **Spill (Disk)**
- Which of the four examples is uniquely different in how it manifests spill?

Spill - Examples, Review

Step	Min	25th	Median	75th	Max	Total
B - shuffle	~2 GB / ~550 MB	~2 GB / ~560 MB	~2 GB / ~565 MB	~2 GB / ~570 MB	~2 GB / ~580 MB	~33 GB
C - union	~2 GB / ~110 MB	~2 GB / ~120 MB	~2 GB / ~125 MB	~2 GB / ~130 MB	~2 GB / ~150 MB	~60 GB
D - explode	0 / ~1.5 GB	0 / ~1.5 GB	0 / ~1.5 GB	0 / ~1.5 GB	0 / ~1.5 GB	~750 GB
E - join*	0 / 0	0 / 0	0 / 0	0 / 0	6 GB / 3 GB	~50 GB

- In **Step B**, the config value **spark.sql.shuffle.partitions** is not managed
- **Steps C & D** simply grow too large as a result of their transformations
- In **Step E** the spill is a manifestation of the underlying skew

What can we do to mitigate spill?

Spill - Mitigation

- The quick answer: allocate a cluster with more memory per worker
  ...more on cluster configurations later

- Decrease the size of each partition by increasing the number of partitions
  - By managing **spark.sql.shuffle.partitions**
  - By explicitly **repartitioning**
  - By managing **spark.sql.files.maxPartitionBytes**
    ...more on maxPartitionBytes later

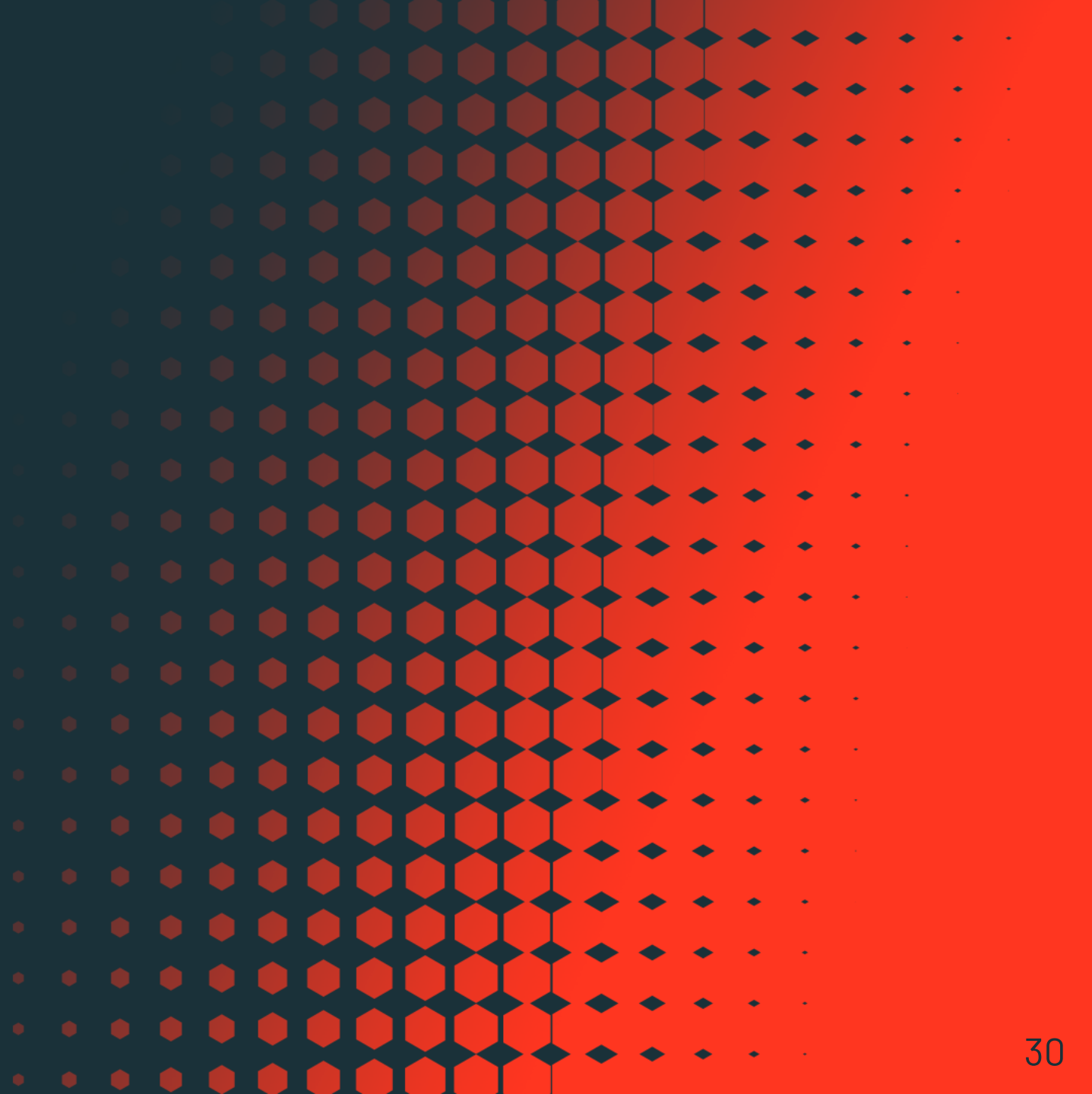
- Make sure AQE is enabled: In the case of skew, this will fix the problem (more on that later)

Spill - Mitigation

- Ignore it - consider the example in **Step E**.
  - Out of ~800 tasks only ~50 tasks spilled
  - Is that 6% worth your time?

- However, it takes only one long task to delay an entire stage

Shuffle

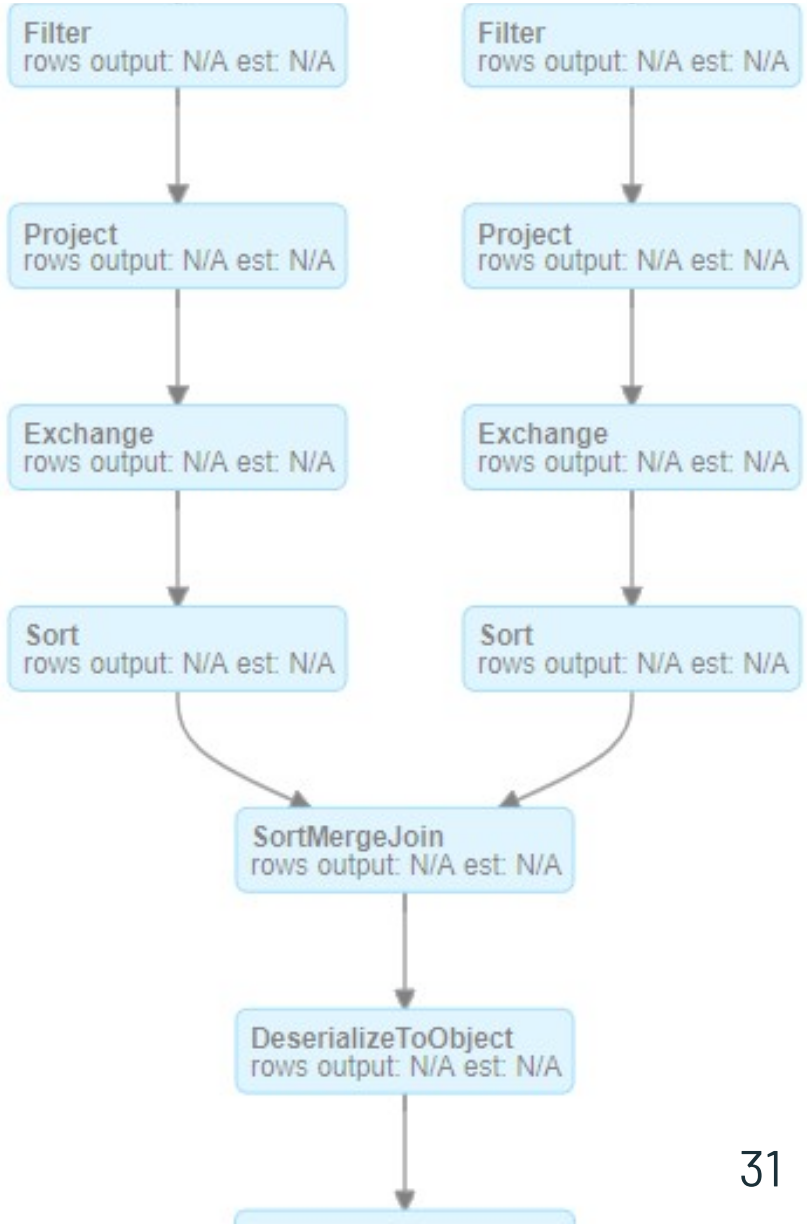


Shuffle

Shuffling is a side effect of wide transformation:

- **`join()`**
- **`distinct()`**
- **`groupBy()`**
- **`orderBy()`**

And technically some actions, e.g. **`count()`**



Shuffle - Not all the same

- The **`distinct`** operation aggregates many records based on one or more keys (the distinguisher) and reduces all duplicates to one record
- The **`groupBy`** / **`count`** combination aggregates many records based on a key and then returns one record which is the count of that key
- The **`join`** operation takes two datasets, aggregates each of those by a common key and produces one record for each matching combination
  (**total record count = max of a.count and b.count**)
- The **`crossJoin`** operation takes two datasets, aggregates each of those by a common key, and produces one record for every possible

Shuffle - Similarities

- They read data from some source
- They aggregate records across all partitions together by some key
- The aggregated records are written to disk (shuffle files)
- Each executors read their aggregated records from the other executors
- This requires expensive disk and network IO

Shuffle - Being Pragmatic

There are some cases in which a shuffle can be avoided or mitigated

TIP: Don't get hung up on trying to remove every shuffle

- Shuffles are often a necessary evil
- Focus on the [more] expensive operations instead
- Many shuffle operations are actually quite fast
- Targeting skew, spill, tiny files, etc. often yield better payoffs

What can we do to mitigate the impact of shuffles?

Shuffle - Mitigation

The biggest pain with shuffle operations is the amount of data that is being shuffled across the cluster.

- Reduce network IO by using fewer and larger workers
  ... more on optimizing cluster designs later
- Reduce the amount of data being shuffled
  - Narrow your columns
  - Preemptively filter out unnecessary records
  - *... more on optimizing data ingestion later*
- Denormalize the datasets - especially when the shuffle is rooted in a join

Shuffle - Mitigation Cont'

- Broadcast the smaller table
  - **`spark.sql.autoBroadcastJoinThreshold`**
  - **`broadcast(tableName)(spark 2.x only)`**
  - Best suited for tables ~10 MB, but can be pushed higher
- For joins, pre-shuffle the data with a bucketed dataset
- Employ the Cost-Based Optimizer
  - Triggers other features like auto-broadcasting based on accurate metadata
  - Possibly negated by Spark 3 & AQE's new features *...more on this later*
  - See our presentation: The Apache Spark™ Cost-Based Optimizer

Apache Spark Join Types

Physical Join Types - A review in performance

Classification by Efficiency

BroadcastHashJoin

BroadcastNestedLoopJoin

ShuffledHashJoin: One side is smaller (3x or more) and a partition of it can fit in memory

(off by default, enabled by `spark.sql.join.preferSortMergeJoin = false`)

SortMergeJoin: Scalable - just works

Cartesian: Try not to do this

Physical Join Types - A review in performance

Classification by Efficiency

- Broadcast Hash Join / Nested Loop
  SELECT **/*+ BROADCAST(a)*/** id FROM a
  JOIN b ON a.key = b.key

  Requires one side to be small.
  Minimal amount of shuffle, no sort,
  very fast.

- Shuffle Hash Join
  SELECT **/*+ SHUFFLE_HASH(a, b)*/** id FROM a
  JOIN b ON a.key = b.key

  Needs to shuffle data but no sort. Can handle large tables, but will OOM too if data is skewed. One side is smaller (3x or more) and a partition of it can fit in memory
  (enable by `spark.sql.join.preferSortMergeJoin = false`)

- Sort-Merge Join
  SELECT **/*+ MERGE(a, b)*/** id FROM a
  JOIN b ON a.key = b.key

  Robust. Can handle any data size. Needs to shuffle and sort data, slower in most cases when the table size is small.

- Shuffle Nested Loop Join (Cartesian)
  SELECT **/*+ SHUFFLE_REPLICATE_NL(a, b)*/** id FROM a JOIN b

  Does not require join keys as it is a cartesian product of the tables. Avoid doing this if you can

Physical Join Types - A review in performance

When is it ok to use Shuffle Hash Join?

- In order to make Spark 2.x more stable in case of skew or partitions too big, Shuffle Hash Join was disabled by default.
- Spark 3.x introduced AQE which will automatically adjust shuffle partitions and use skew joins if needed.
- It is now safer to enable SHJ, but spark.sql.shuffle.partitions is still an upper bound limit to the number of partitions of your dataset, meaning it can still fail if you don't set it to a number which is high enough.
- If you make sure your shuffle partitions is set correctly, it is safe enough to enable SHJ using spark.sql.join.preferSortMergeJoin=false

Help Catalyst choose the right Joins

At the cluster level, set these configs (or in your cluster policy):

spark.sql.autoBroadcastJoinThreshold 100*1024*1024
spark.sql.join.preferSortMergeJoin false

Why do I need the slide before?

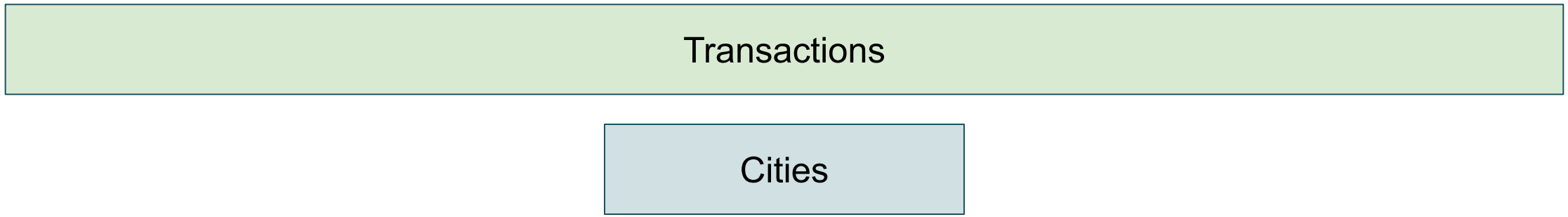
You know your data best. For example, you're joining two tables on a partitioned column and a couple others, a sort merge join would make more sense than a full shuffle

BroadcastHashJoins

- **BroadcastHashJoins** are not a magic bullet
- The use cases are limited to small tables (under 10 MB by default)
- They can put undue pressure on the Driver resulting in OOMs
- In some cases, the alternative **SortMergeJoin** might be faster
- In general, Spark's automatic behavior might be your best bet

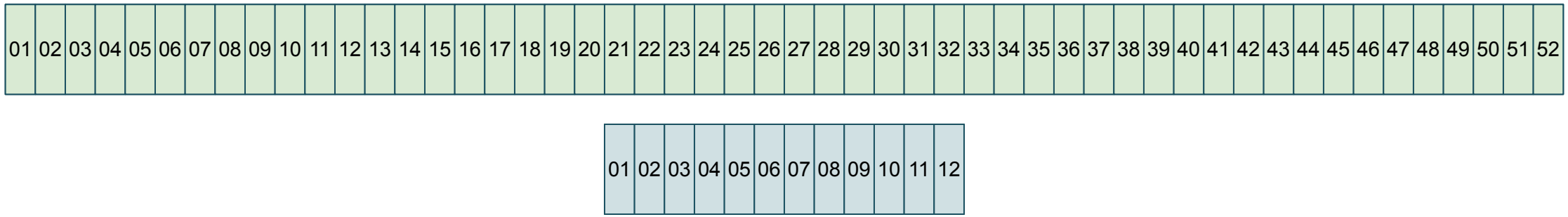
Let’s review how the BroadcastHashJoin works...

Presume we have two tables that we want to join based upon some common column

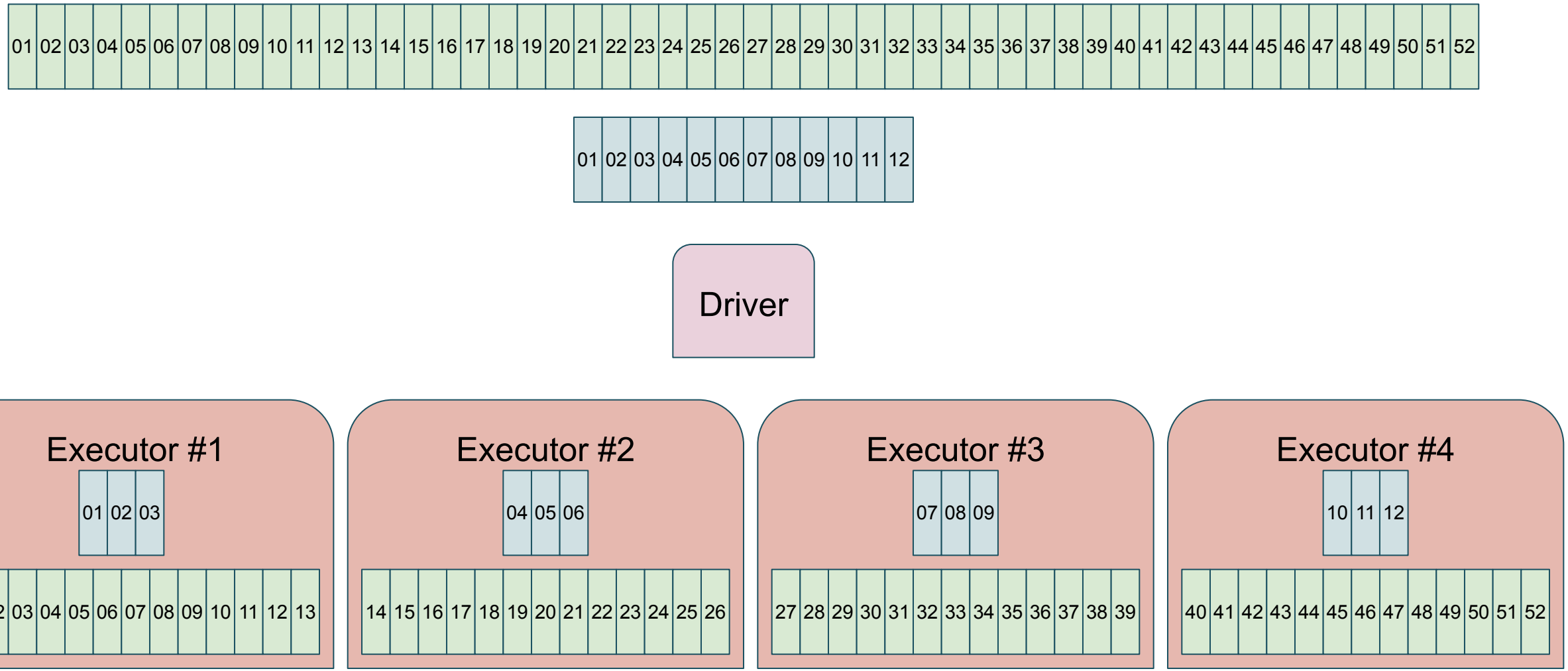


NB: we will broadcast the small Cities DataFrame

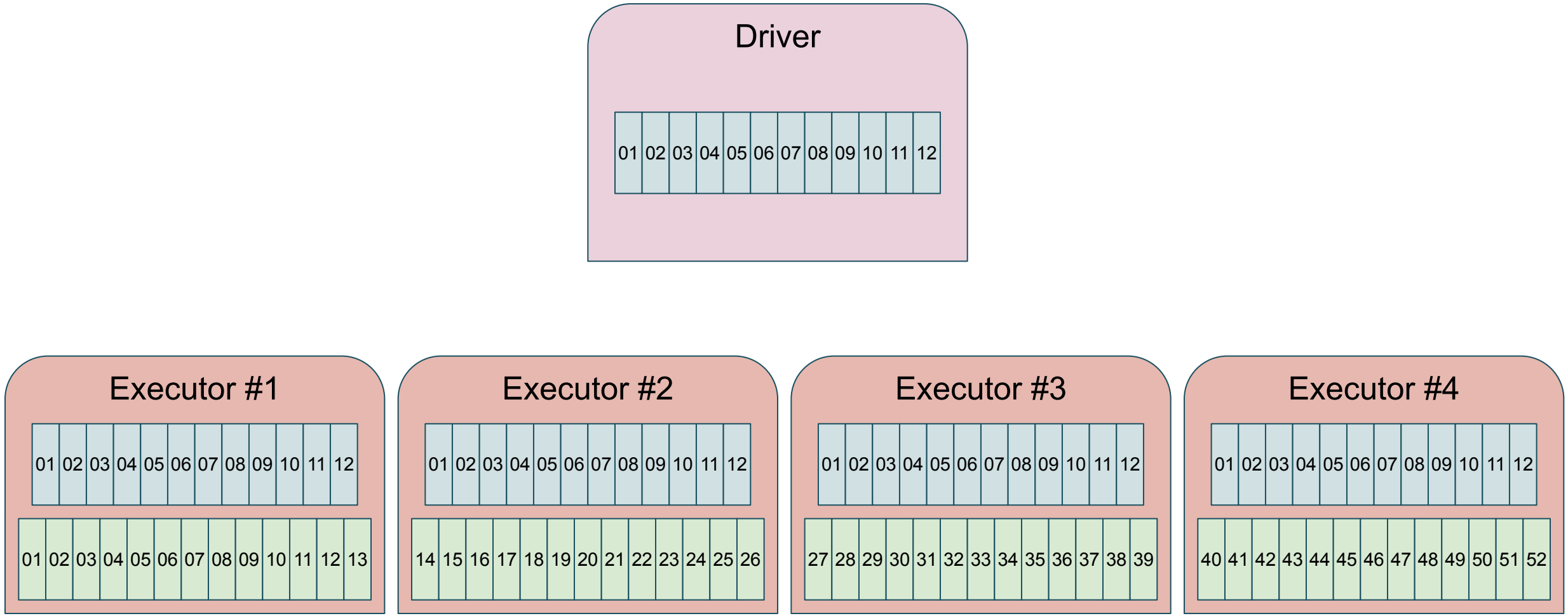
During planning the driver will partition our two datasets



First, collect all Cities partitions from the executors to the driver



Next, a copy of the entire Cities table is sent back by the Driver to each executor



Controlling when Broadcasting is preferred

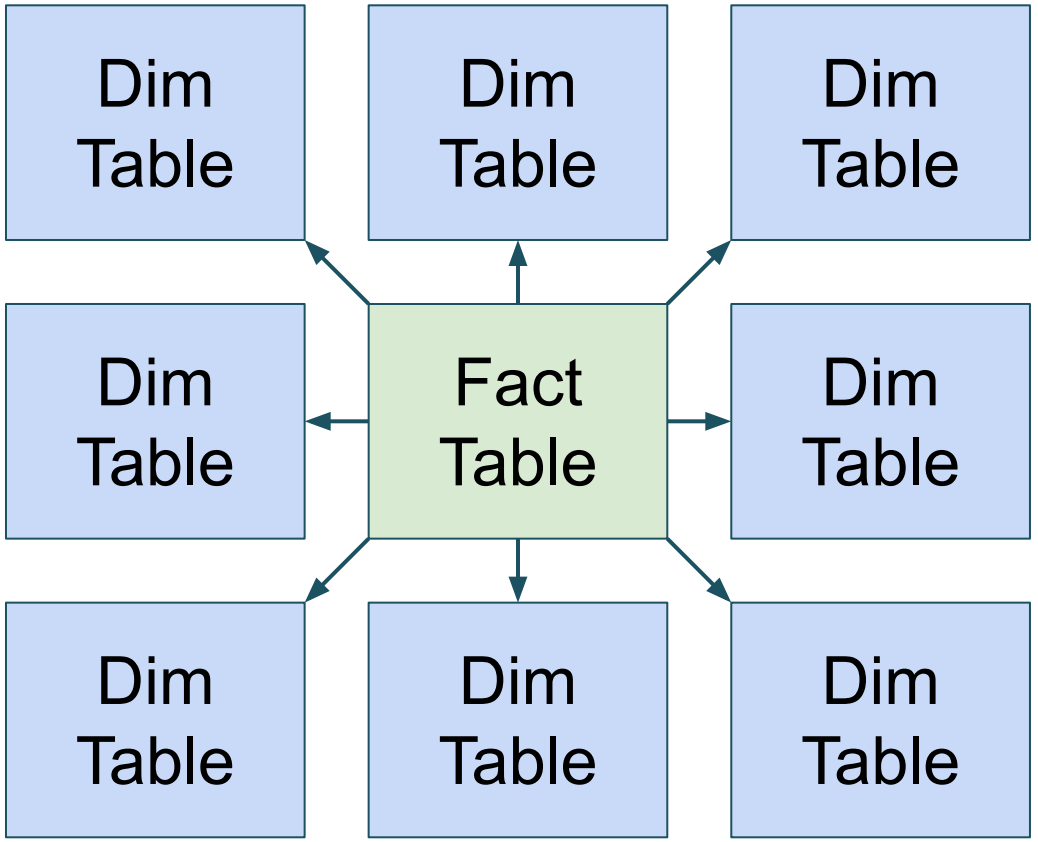
- You can control when a dataframe will be broadcasted setting the configuration spark.sql.autoBroadcastJoinThreshold
- 100MB threshold = 100*1024*1024
- You can push it up to 300MB or higher based on the size of your Driver, but always test

BroadcastHashJoins - Dangers

- Note the high level of IO between the Driver and Executors
- With small tables (e.g. around 10 MB), the cost is lower than the exchange
- When pushed to higher limits (say 100 MB), the balances start to shift
- Similarly, many empty partitions can adversely affect the BHJ
- The Driver & Executors both require enough RAM to receive the fully broadcasted table
- Performance depends on the relative scale of the left and right table

BroadcastHashJoins - w/Many Dim Tables

Even if you don't push the 10 MB limit, joining to many small tables can produce excessive load on the Driver & Executors resulting in GC delays and OOM Errors



BroadcastHashJoins vs SortMergeJoin

BroadcastHashJoin	SortMergeJoin
Avoids shuffling the bigger side	Shuffles both sides
Naturally handles data skew	Can suffer from data skew
Cheap for selective joins	Can produce unnecessary intermediate results
Broadcasted data needs to fit in memory	Data can be spilled and read from disk
Cannot be used for certain outer joins	Can be used for all joins
Overhead of E→D→E is high with few/large executors	Outperforms BHJ with few/large executors

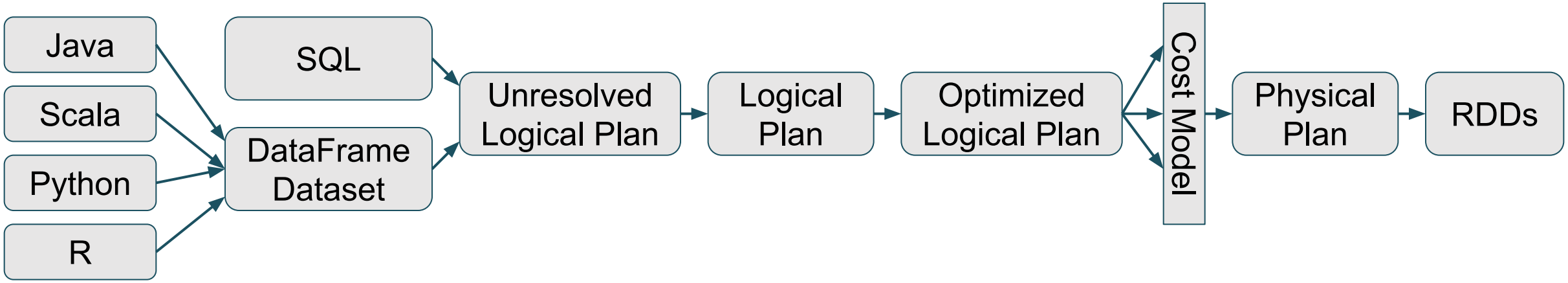
Optimizing Joins and Skew with AQE

Adaptive Query Execution

- Also referred to as
  - Adaptive Query Optimisation
  - Adaptive Optimisation
- Adaptive Query Execution is an extensible framework
- It's akin to writing rules for the Catalyst Optimizer
- And in Spark 3.0, it must be enabled by setting **spark.sql.adaptive.enabled** to **true**
- Other AQE features may default to enabled, but are still gated by this master configuration flag

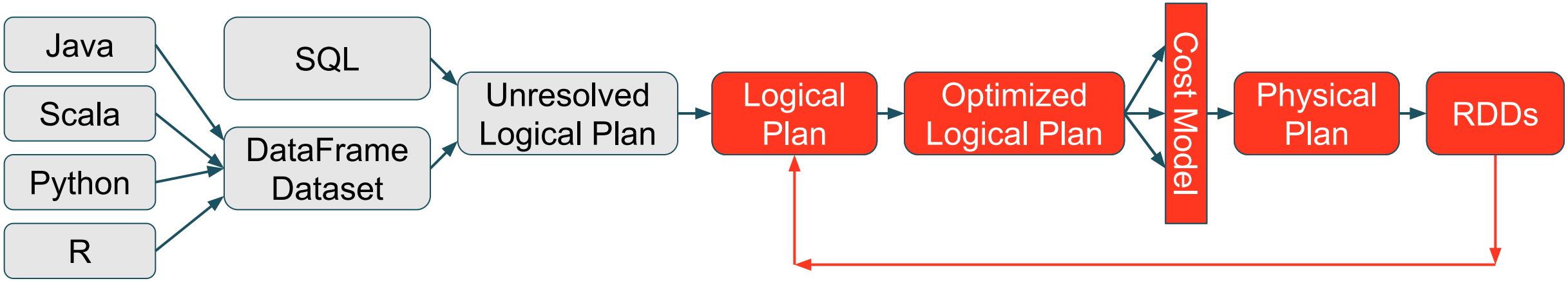
Adaptive Query Execution vs Catalyst Optimizer

- The **Catalyst Optimizer** is a rules based engine
- It takes the **Logical Plan** and rewrites it as an optimized **Physical Plan**.
- The **Physical Plan** is developed **BEFORE** a query is executed
- For Example...



Adaptive Query Execution vs Catalyst Optimizer

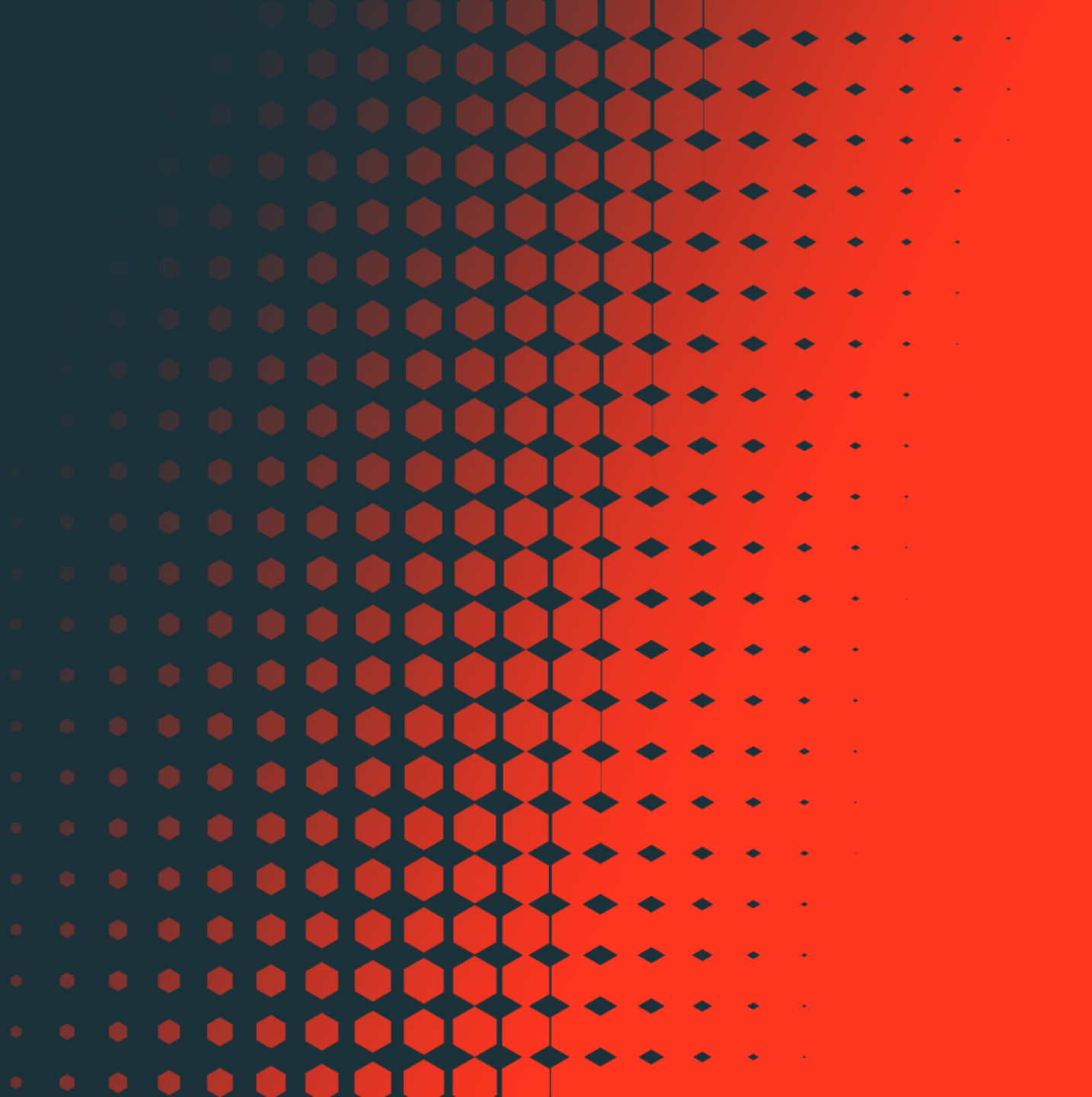
- **Adaptive Query Execution**, on the other hand, modifies the **Physical Plan** based on runtime information, for example…



Adaptive Query Execution

- Let's take a look a three key examples introduced with Spark 3.0
  - Join Optimization
  - Skew Join Optimization
  - Tuning Shuffle Partitions

Join Optimization



databricks

AQE - Join Optimization

Joins can be optimized at runtime, for example:

- Table sizes are estimated at planning:
  - A large table is estimated to be 100 GB
  - A small table is estimated to be 11 MB and thus not a candidate for auto-broadcasting
- Both tables are read in as distinct stages, but **in parallel**
- At runtime, the small table comes in under the 10 MB threshold
- At runtime, AQE adjusts the physical plan to broadcast the small table or potentially employ other strategies like subqueries

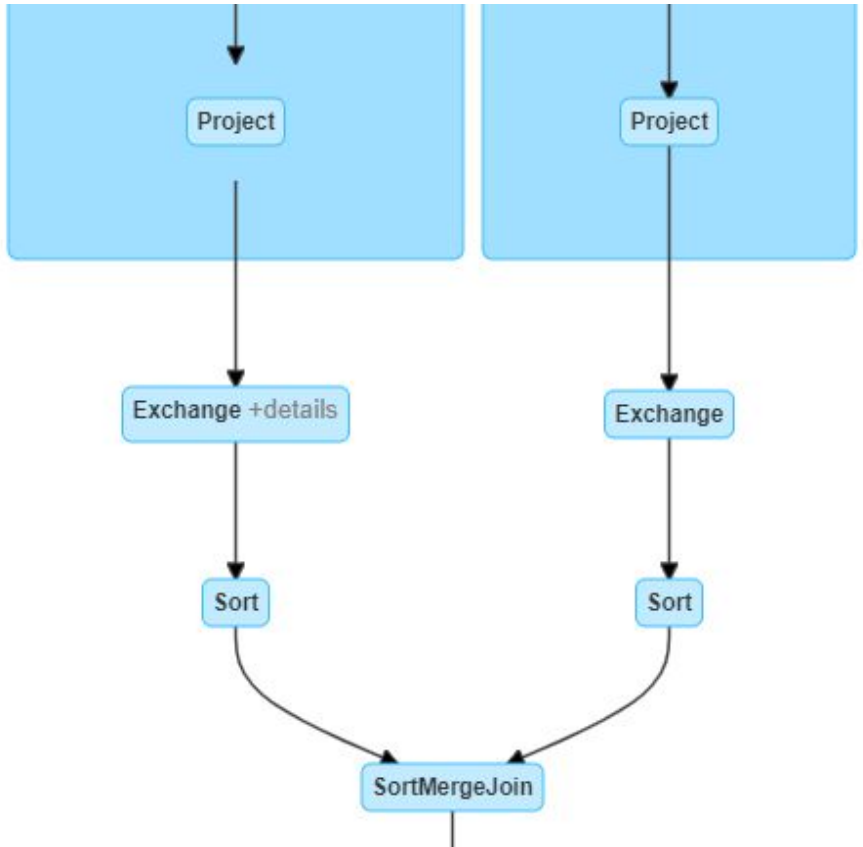
AQE - Join Optimization, In Action

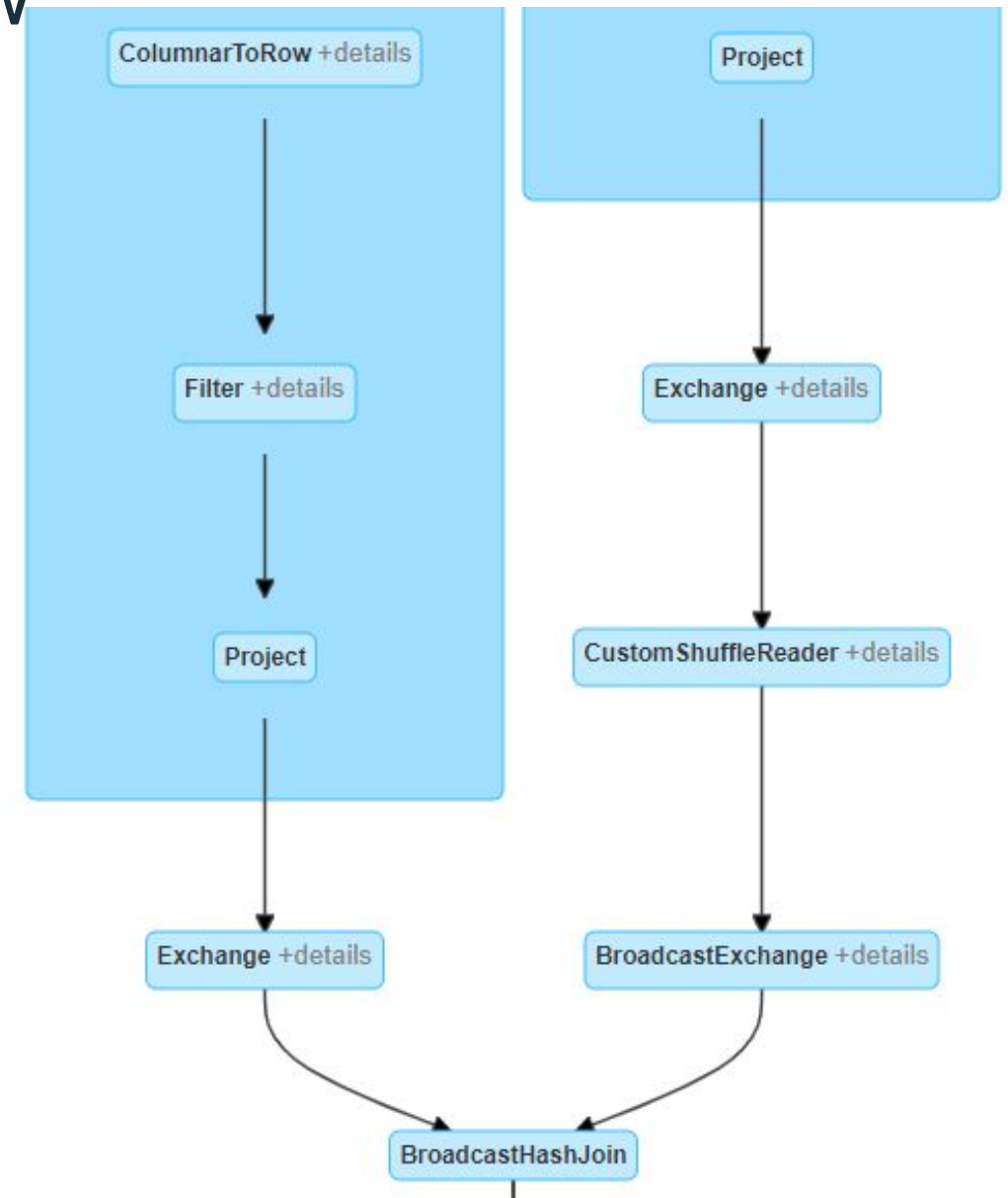
See Experiment #3799A

- Contrast the **Query Plans** for the **last** job for **Step C** (standard) and **Step D** (w/AQE)
- What major difference is there between the two **Query Plans**?

AQE - Join Optimization, Review

Kind of Cool: Run the query, and it will initially show a **SortMergeJoin**. Once the ingest stages (left & right of join) completes, the query and physical plan will update to use a **BroadcastHashJoin**





AQE - Join Optimization, Options

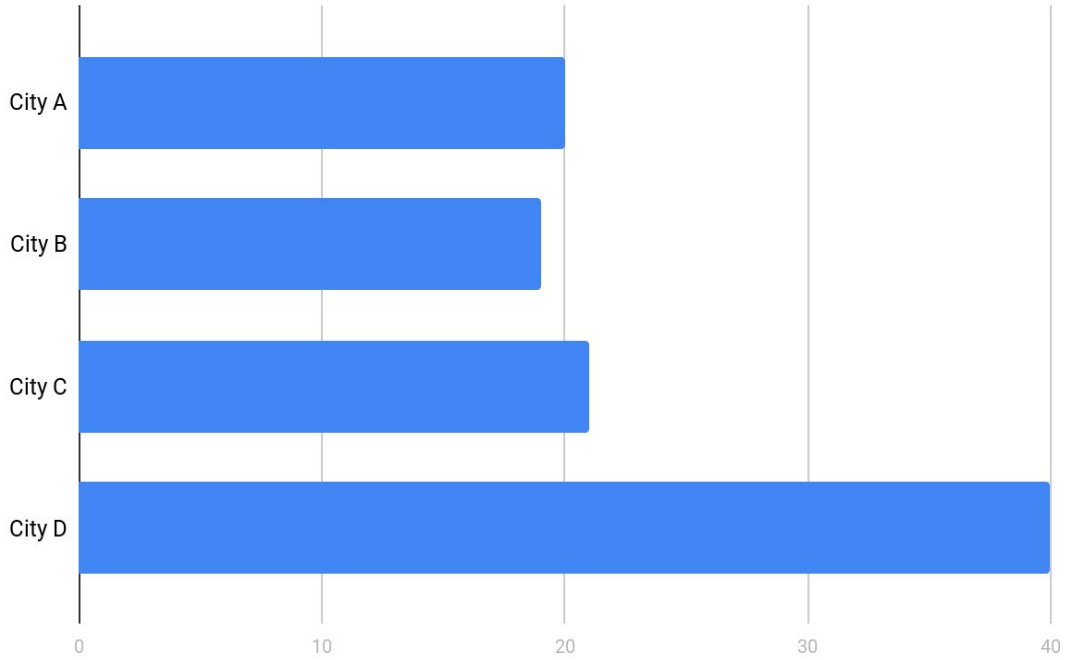
See Converting sort-merge join to broadcast join for more information

Property Name	Default	Meaning
spark.sql.adaptive.localShuffleReader.enabled	true	AQE converts sort-merge join to broadcast hash join when the runtime statistics of any join side is smaller than the broadcast hash join threshold
spark.sql.adaptive.nonEmptyPartitionRatioForBroadcastJoin	0.2	The relation with a non-empty partition ratio lower than this config will not be considered as the build side of a broadcast-hash join in adaptive execution regardless of its size.

Skew Join Optimization

AQE - Skew Join Optimization

- Skew is a hard problem to solve for
- Just to review, we can:
  - Salt the skewed keys
  - Employ skew hints
  - Tweak all kinds of settings:
    - Level of Compute
    - Available RAM for Execution
    - The Broadcast-Join Threshold
    - spark.sql.shuffle.partitions
  - Or we can just use Spark 3.x...



AQE - Skew Join Optimization

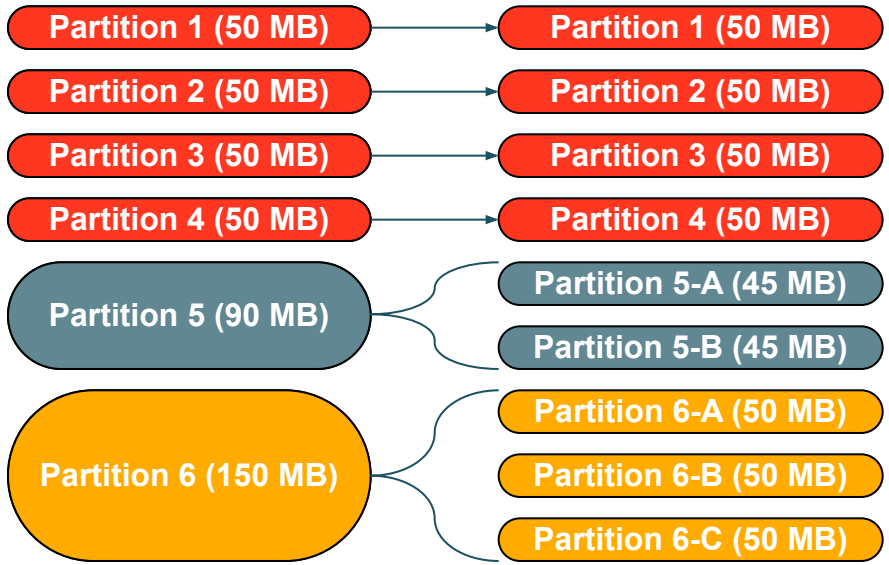
With Spark 3.x, solving for skew is easy and automatic:

- Enable by setting **spark.sql.adaptive.skewedJoin.enabled** to **true**

- A partition is skewed if its data size or row count is N times larger than the median & also larger than a predefined threshold

- When skewed, AQE subdivides the one partition into many partitions and employs additional tasks to process

- For example...



AQE - Skew Join Optimization, In Action

See Experiment #1596

- Contrast the **last** job for **Step C** (standard) and **Step E** (w/AQE)
  - ...the total execution time (entire command)
  - ...the number of tasks for the final stage of each job
  - ...the stage details for the final stage of each job
    - Event Timeline
    - Presence or absence of Spill
    - Shuffle Read Size / Records
  - ..the query plans for the two jobs
- What major difference is there in the **Query Plan** for **Step E** vs **Step C**?
- How many skewed partitions were reported in the **Query Plan**?

AQE - Skew Join Optimization, Review

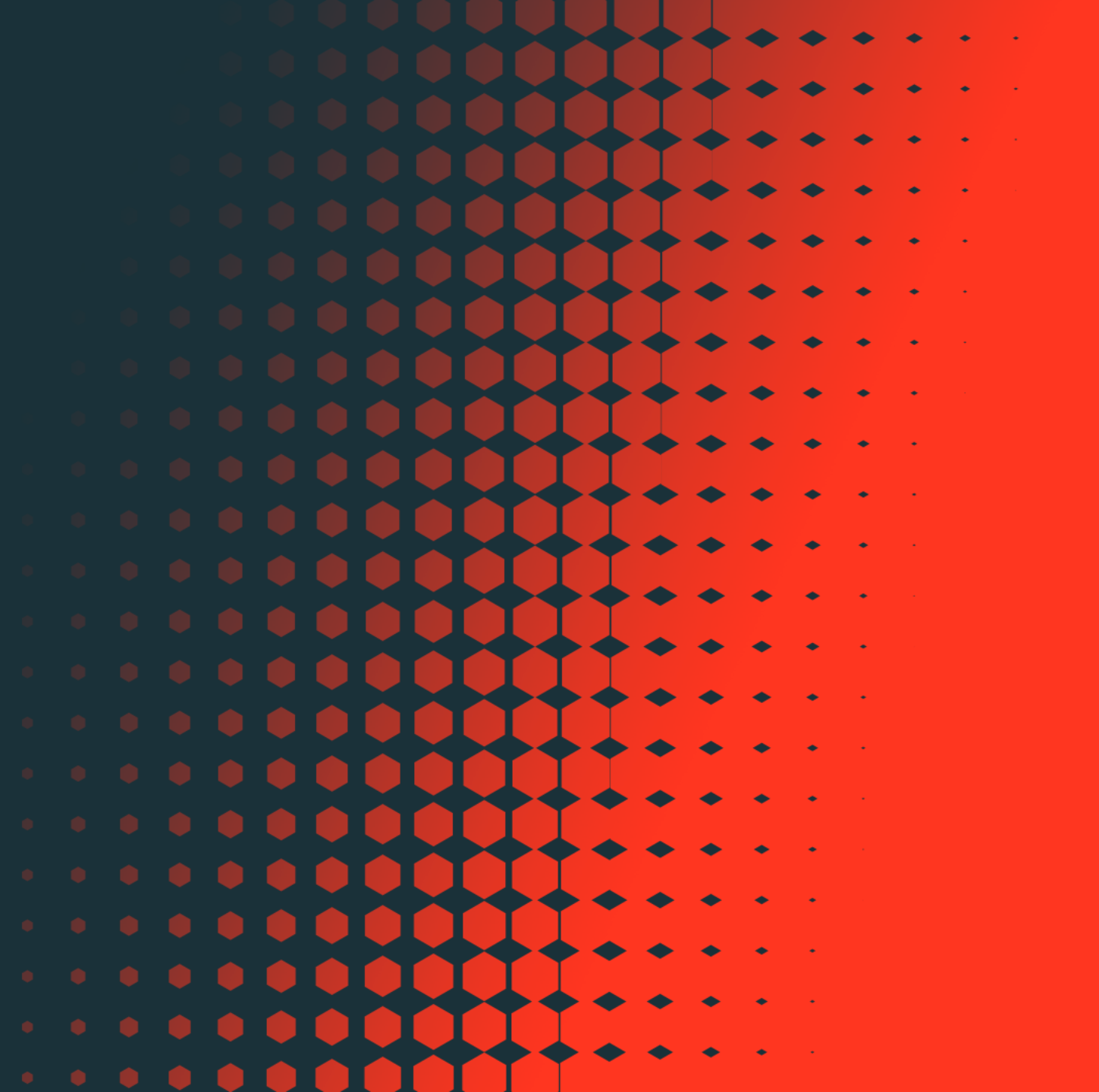
Step	Cmd Duration	Tasks	Event Timeline	Spill	Shuffle Read Size / Records	Skew Count	Query Plan Optimization
C	~29 min	832	Bad	Yes	**0/0/<100K/<500M/2.6G**	n/a	-none-
E	**~25 min**	**1489**	**Good**	**No**	**0/115M/115M/125M/130M**	**1,327**	**CustomShuffleReader SortMergeJoin(skew)**

AQE - Skew Join Optimization, Options

See Optimizing Skew Join for more information

Property Name	Default	Meaning
spark.sql.adaptive.skewJoin.enabled	true	When true and **spark.sql.adaptive.enabled** is true, Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions.
spark.sql.adaptive.skewJoin.skewedPartitionFactor	10	A partition is considered as skewed if its size is larger than this factor multiplying the median partition size and also larger than **spark.sql.adaptive.skewedPartitionThresholdInBytes**
spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes	256 MB	A partition is considered as skewed if its size in bytes is larger than this threshold and also larger than **spark.sql.adaptive.skewJoin.skewedPartitionFactor** multiplying the median partition size. Ideally this config should be set larger than **spark.sql.adaptive.advisoryPartitionSizeInBytes**.

Tuning Shuffle Partitions



databricks

spark.sql.shuffle.partitions

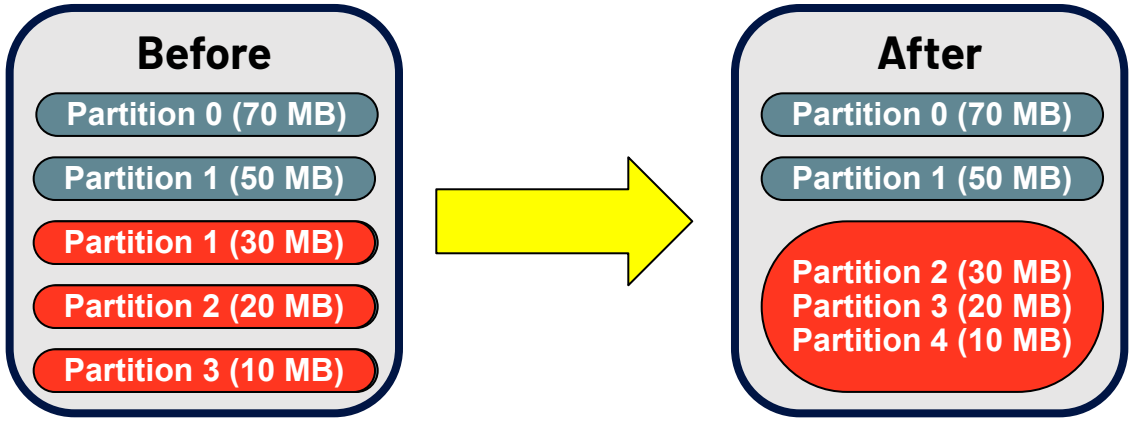
- **spark.sql.shuffle.partitions** - everyone should know of this by now!
  - After every wide transformation, Spark needs to repartition the data
  - Indicates how many partitions Spark will create for the next stage
  - This setting MUST be managed by every user for every job

- The problems with this:
  - Too many shuffle partitions, and one has empty/small spark-partitions putting undue pressure on the scheduler/driver
  - Too few shuffle partitions, and one has larger partitions resulting in spill during the exchange and sort if not OOM Errors
  - It can only be set once per job
  - As the number of stages increases, so do the odds of this value being inappropriate for all of the stages

AQE - Tuning Shuffle Partitions

- To enable the coalescing of shuffle partitions set **spark.sql.adaptive.coalescePartitions.enabled** to **true**

- The net effect is fewer partitions for subsequent stages, for example...



- Over simplifying, but we now only need to manage **spark.sql.shuffle.partitions** for the expected maximum

AQE - Tuning Shuffle Partitions, In Action

See **Experiment #2653**

- Contrast the **last** job for **Step B** (default), **Step C** (832) and **Step D** (w/AQE)
  - ...the total execution time (entire command)
  - ...the number of tasks in the final stage of each job
  - ...the stage details for the final stage of each job
  - ...the query plans for the three jobs
- What major element is different in the **Query Plan** for **Step D** versus the other two?

AQE - Tuning Shuffle Partitions, Review

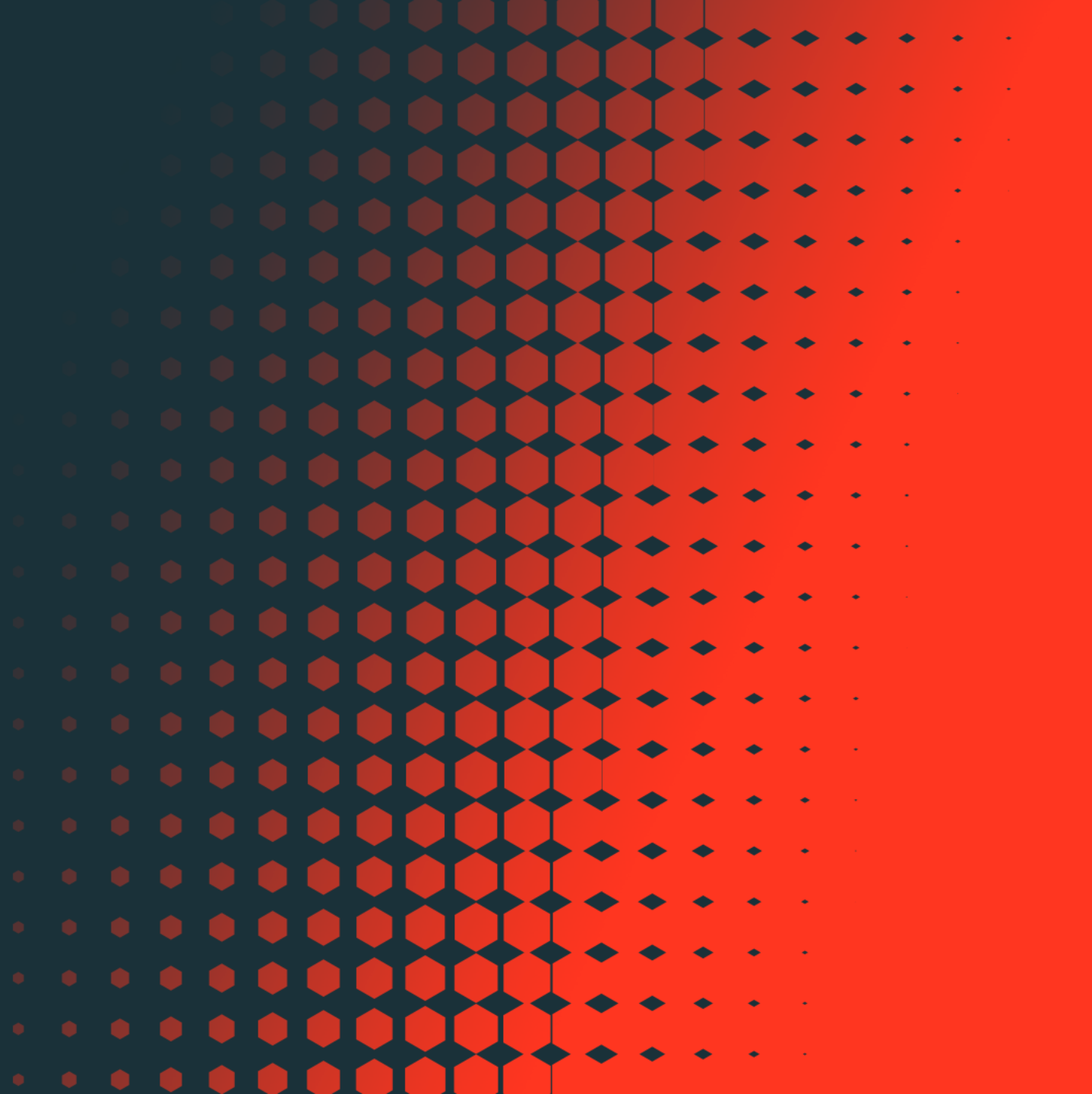
Step	Total Duration	Number of Partitions	Stage Details Conclusions	Query Plan Optimization
Step B	~1.5 minutes	825 / 200	Bad distribution / Overhead @200 partitions are 4x Larger Potential Spill	-none-
Step C	~1 minute	825 / 832	Horrible distribution / Overhead	-none-
Step D	**~¾ of a minute**	825 / **17**	Good Distribution / Minor Overhead	**CustomShuffleReader**

AQE - Tuning Shuffle Partitions, Options

See Coalescing Post Shuffle Partitions for more information

Property Name	Default	Meaning
spark.sql.adaptive.coalescePartitions.enabled	true	When true and **spark.sql.adaptive.enabled** is true, Spark will coalesce contiguous shuffle partitions according to the target size (specified by **spark.sql.adaptive.advisoryPartitionSizeInBytes**), to avoid too many small tasks.
spark.sql.adaptive.coalescePartitions.minPartitionNum	Default Parallelism	The minimum number of shuffle partitions after coalescing. If not set, the default value is the default parallelism of the Spark cluster. This configuration only has an effect when **spark.sql.adaptive.enabled** and **spark.sql.adaptive.coalescePartitions.enabled** are both enabled.
spark.sql.adaptive.coalescePartitions.initialPartitionNum	200	The initial number of shuffle partitions before coalescing. By default it equals to **spark.sql.shuffle.partitions**. This configuration only has an effect when **spark.sql.adaptive.enabled** and **spark.sql.adaptive.coalescePartitions.enabled** are both enabled.
spark.sql.adaptive.advisoryPartitionSizeInBytes	64 MB	The advisory size in bytes of the shuffle partition during adaptive optimization (when **spark.sql.adaptive.enabled** is true). It takes effect when Spark coalesces small shuffle partitions or splits skewed shuffle partition.

Serialization



Serialization

- Serialization is the last of our “most common problems”
- Spark 1.x provided significant performance gains over other solutions
- Spark 2.x, namely Spark SQL & the DataFrames API, brought even more performance gains by abstracting out the execution plans
- We no longer write “map” operations with custom code but instead express our transformations with the SQL and DataFrames APIs
- That means that with Spark 2.x, when we regress back to code, we are going to see more performance hits

Serialization - Why it's bad

- Spark SQL and DataFrame instructions are compact, and optimized for distributing those instructions from the Driver to each Executor
- When we use code, that code has to be serialized, sent to the Executors and then deserialized before it can be expected
- Python takes an even harder hit because Python code has to be pickled **AND** Spark must instantiate an instance of the Python interpreter in each and every Executor
- Compared to the Python version of the DataFrames API which uses Python to express the operations executed in the JVM

Serialization - Catalyst Optimizer

- Besides the cost of serialization, there is another problem...
- These features create an analysis barrier for the Catalyst Optimizer
- The Catalyst Optimizer cannot connect code before and after UDF
- The UDF is a black box which means it limits optimizations to the code before and after, excluding the UDF and how all the code works together

Serialization

- See Experiment #5980 (Benchmarking)
- See Experiment #4538 for Scala (UDF)
- See Experiment #4538 for Python (UDF)

Serialization - How Bad?

Remember our benchmarks?

- See Experiment #5980, **Step D-S** and **Step D-P**
- These use a do-nothing Lambda and a strait read
- The Scala version takes < 15 minutes
- The Python version takes ~2.5 hours!
- All because we executed this python code: `lambda x: None`

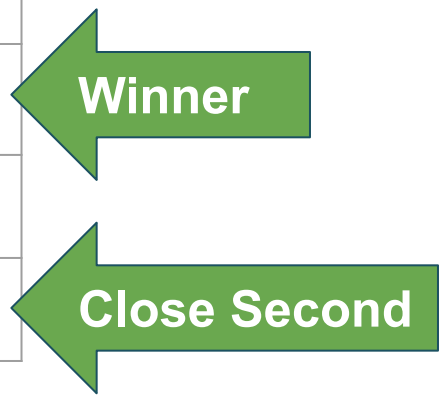
Serialization - Scala's Overhead

Let's see how serialization affects Scala in Experiment #4538 for Scala

- See **Step D** which uses higher-order functions
  - Uses functions from **`org.apache.spark.sql.functions`**
  - Note the execution time
- See **Step E** which uses two UDFs
  - See **`parseId(..)`** and **`parseType(..)`**
  - Note the execution time
- See **Step F** which uses Typed Transformations
  - See the **`map(..)`** operation
  - Note the execution time

Serialization - Scala's Overhead, Review

Step	Type	Duration
C	Baseline	~3 min
D	Higher-order Functions	~25 min
E	UDFs	~35 min
F	Typed Transformations	25+ min



Serialization - Python's Overhead

Let's see how serialization affects Python in Experiment #4538 for Python

- See **Step D** which uses higher-order functions
  - Uses functions from **`pyspark.sql.functions`**
  - Note the execution time
- See **Step E** which uses two UDFs
  - See **`parseId(..)`** and **`parseType(..)`**
  - Note the execution time
- See **Step F** which uses Pandas (or Vectorized) UDFs
  - See **`@pandas_udf parseId(..)`** and **`@pandas_udf parseType(..)`**
  - Note the execution time

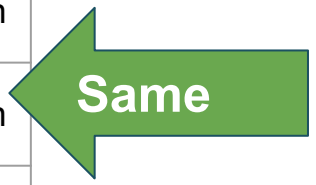
Serialization - Python's Overhead, Review

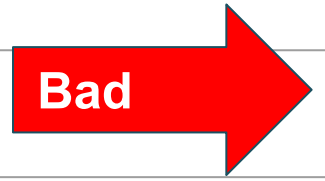
Step	Type	Duration
C	Baseline	~3 min
D	Higher-order Functions	~25 min
E	UDFs	~105 min
F	Panda/Vectorized UDFs	~70 min

How do the two stack up against each other?

Serialization – Python vs Scala

Step	Type	Scala/Java Duration	Python Duration
C	Baseline	~3 min	~3 min
D	Higher-order Functions	~25 min	~25 min
E	UDFs	~35 min	~105 min
F - Scala	Typed Transformations	~25 min	n/a
F - Python	Panda/Vectorized UDFs	n/a	> 70 min





Serialization - Why?

Why do we still see such a proliferation of these [poorly performing] features?

- Integration with 3rd-party libraries
  - Common in data science
  - In some cases there is no other choice
- Attempting to integrate with the company's existing frameworks
  - e.g. custom business objects
  - or proprietary libraries
- Migrations from legacy systems like Hadoop
  - Copy and pasting code instead of rewriting them as higher-order functions

What can we do to mitigate the serialization issues?

Serialization - Mitigation

- Short answer, don't use UDFs, Vectorized UDFs or Typed Transformations
- The need for these features is REALLY rare
- The SQL higher-order functions are very robust
- But if you have to...
  - Use Vectorized UDFs over "standard" Python UDFs
  - Use Typed Transformations over "standard" Scala UDFs

Should we rewrite our Spark code to use Scala instead of Python?

No!

Serialization

Is there something I can do if I have lots of python UDFs, I cannot avoid them, and this is impacting performances very badly?

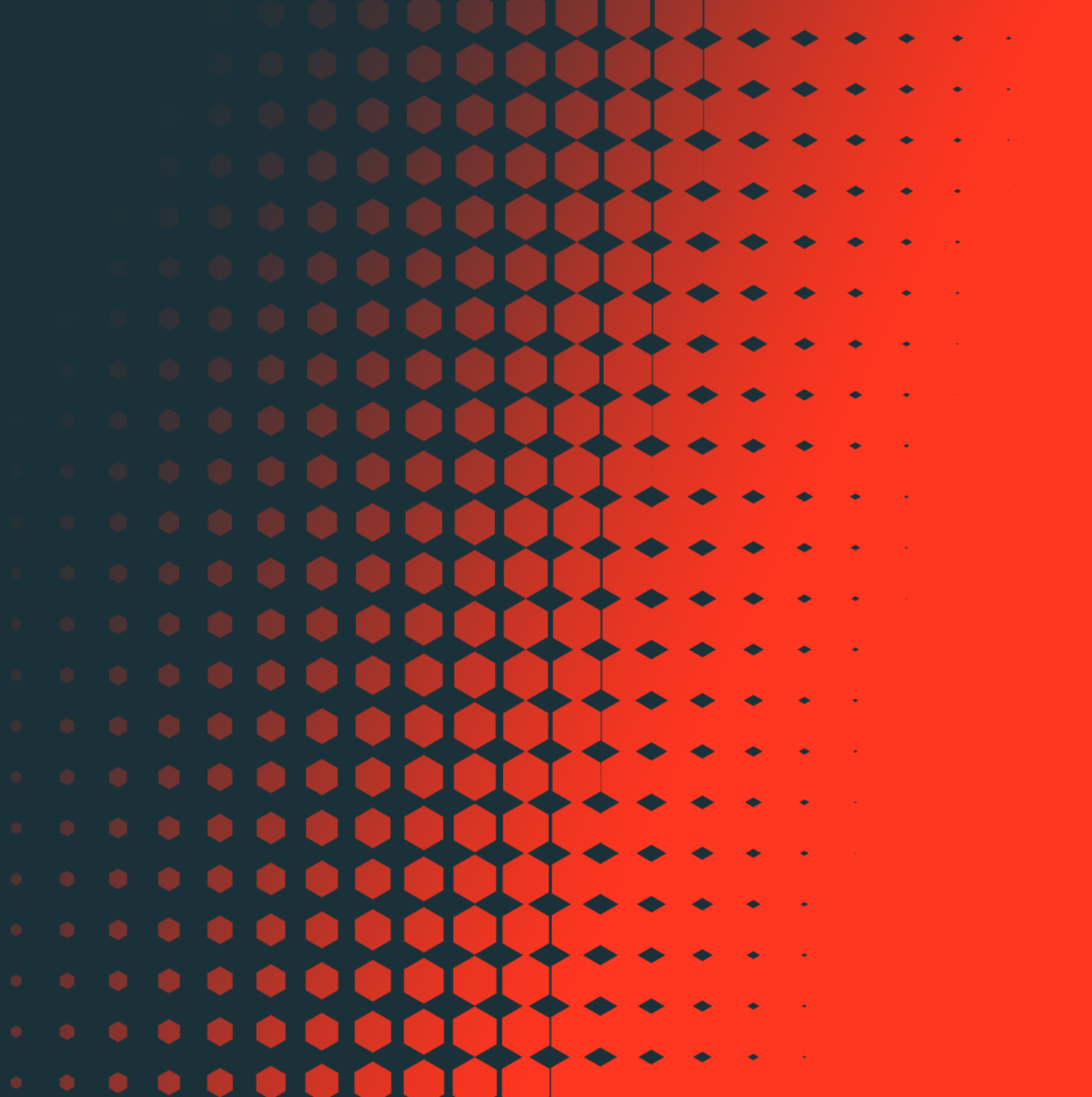
- You could define and register your UDFs in scala/java, then use it in python using SQL queries
- Potentially, you could build a scala/java jar library to install as a dependency in your cluster
- From Python, you can use spark._jvm to execute a setup method in the JVM which would register your UDFs
- The rest of the code can stay in python, but you won't pay performance penalties anymore.
- This works only if the UDF is not used to integrate with an external python framework which is not available in java/scala

Key Ingestion Concepts

A few key concepts when ingesting data

- Predicate pushdown
- Disk partitioning
- Dynamic partition pruning

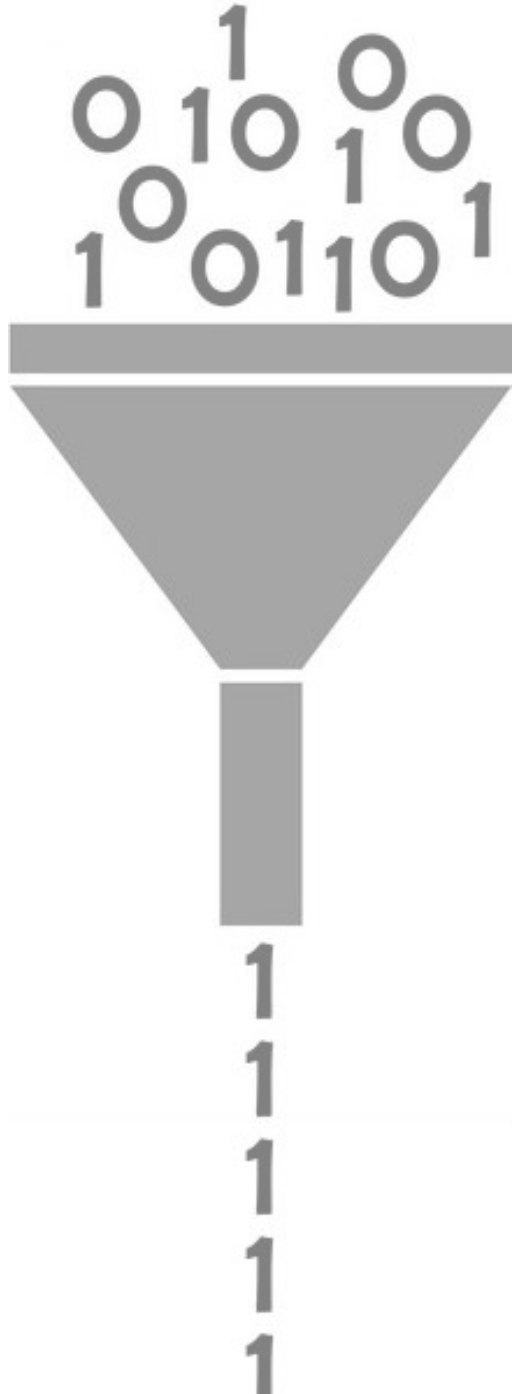
Predicate Pushdown



databricks

Predicate Pushdown

- Reducing Data Ingestion is one of the best strategies for mitigating performance problems in Apache Spark
- The **Predicate Pushdown** is the basis for most of the reduction strategies
- A Predicate is simply a condition applied to records read in by a Spark job
- More specifically, it's employing schemas, **filter(..)**, **where(..)**, **select()**, and **drop()** transformation (to name a few) that are then pushed down to the underlying data store



How Is The Predicate Applied?

- How is 100% dependent on the underlying data source
- For a relational database like PostgreSQL this involves a **SELECT** statement and a **WHERE** condition
- In this scenario, only qualifying records and columns are returned to Apache Spark by the PostgreSQL server

When Is The Predicate Applied?

- The predicate is applied as early as possible
- If applied early enough (e.g. as the first transformation), then records can be excluded and never pulled into the executor
- If applied late, all records might be pulled into the executor, after which non-matching records are discarded
- One of the most common “bugs” results from introducing transformations between the initial read and filtering transformation as we will see next

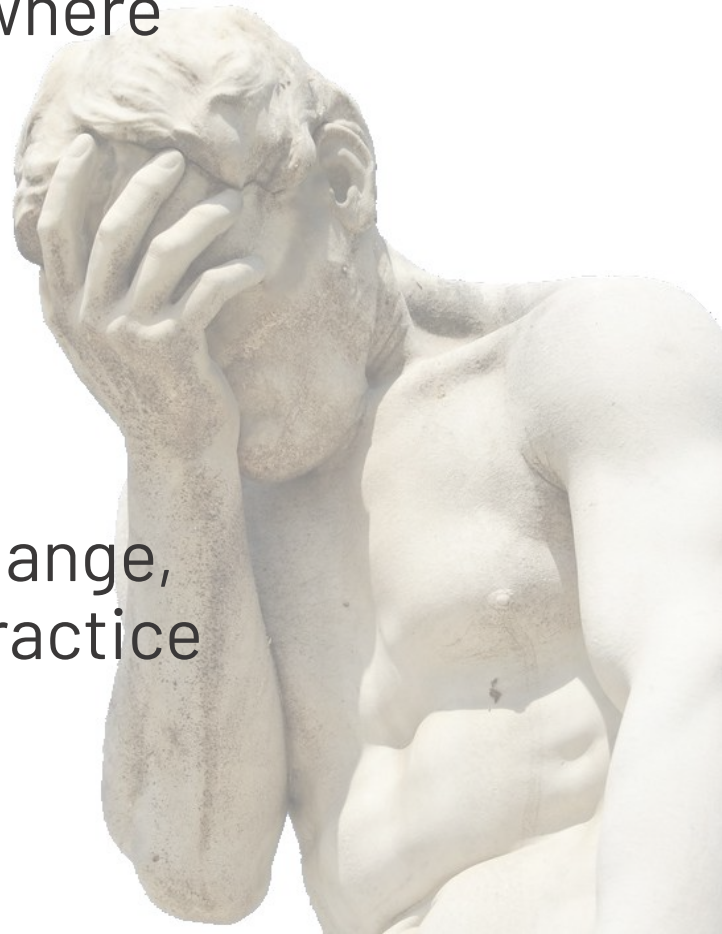
Predicate Pushdown w/PostgreSQL

See Experiment #8342

- Contrast **Step B** and **Step C** which query a table with 1M records
  - Note the one line code difference between each command
  - Note the execution duration of both commands
  - In the **Spark UI**, **Query Details**
    - Note how many records were returned by the **Scan JDBCRelation**
    - In the **Physical Plan**, note the values passed to the **PushedFilters**

Crippling the Predicate

- One common performance problem is the crippling of the predicate
- This often happens when refactoring existing code where the full context of the Spark query might get lost
- In nearly all cases, it is the result of introducing operations between the **read** and the **filter** with unintended side effects
- They can be recognized before and after the code change, but a review of the query should become standard practice



Crippling the Predicate

See Experiment #8342

- Contrast **Step D** with **Steps B** & **C**
- Performance wise, which step (**B** vs **C**) does **Step D** most closely resemble?
- What has changed in the code?
- What has changed in the **Query Details**?
- What has changed in the **Physical Plan**?

Columnar Data Formats

- Filters are not the only “predicate” that can be pushed down
- Column selection can also be pushed down
  - With a database like PostgreSQL, this is done with a **SELECT** statement
  - For files, we require a **Columnar File Format**
- What constitutes a “Columnar File Format?”
  - Textbook Definition: **Data is stored by column, not by row**
  - Examples include Delta, Parquet, and ORC
- Compared to Row-Based File formats that store data by row
  - Examples include CSV, TSV, JSON, and AVRO

An Example: Columnar vs Row-Based

Row-Based

	name	color	city	age
Row 1	Tom	red	Chicago	32
Row 2	Sally	blue	Paris	87
Row 3	Mike	green	London	20
Row 4	Mary	yellow	Fresno	55

Reads Row #1

Columnar

	Row 1	Row 2	Row 3	Row 4
name	Tom	Sally	Mike	Mary
color	red	blue	green	yellow
city	Chicago	Paris	London	Fresno
age	32	87	20	55

Reads the “name” column

Columnar Reads

- By reading in only specific columns, Spark can reduce the overall IO
- Consider the schema from the previous example:
  `name:STRING, color:STRING, city:STRING, age:INTEGER`
- And the following SQL Query:
  `SELECT name, age FROM whatever`
- Only the qualifying columns (name & age) are returned
- Compare this to JSON and CSV in which the entire row must be read into the executor before unused columns can be discarded

Columnar Reads w/Parquet

See Experiment #4112

- Contrast **Steps B, C & D**
  - Note the execution duration of each command
  - Note the code differences of each command
  - In the **Spark UI**, **Query Details**, note
    - The **Scan parquet** block, **scan time total**
    - The **Scan parquet** block, **filesystem read data size**
    - The **WholeStageCodegen(1)** block, **rows output**

What is different in each of these queries' **Physical Plan**?

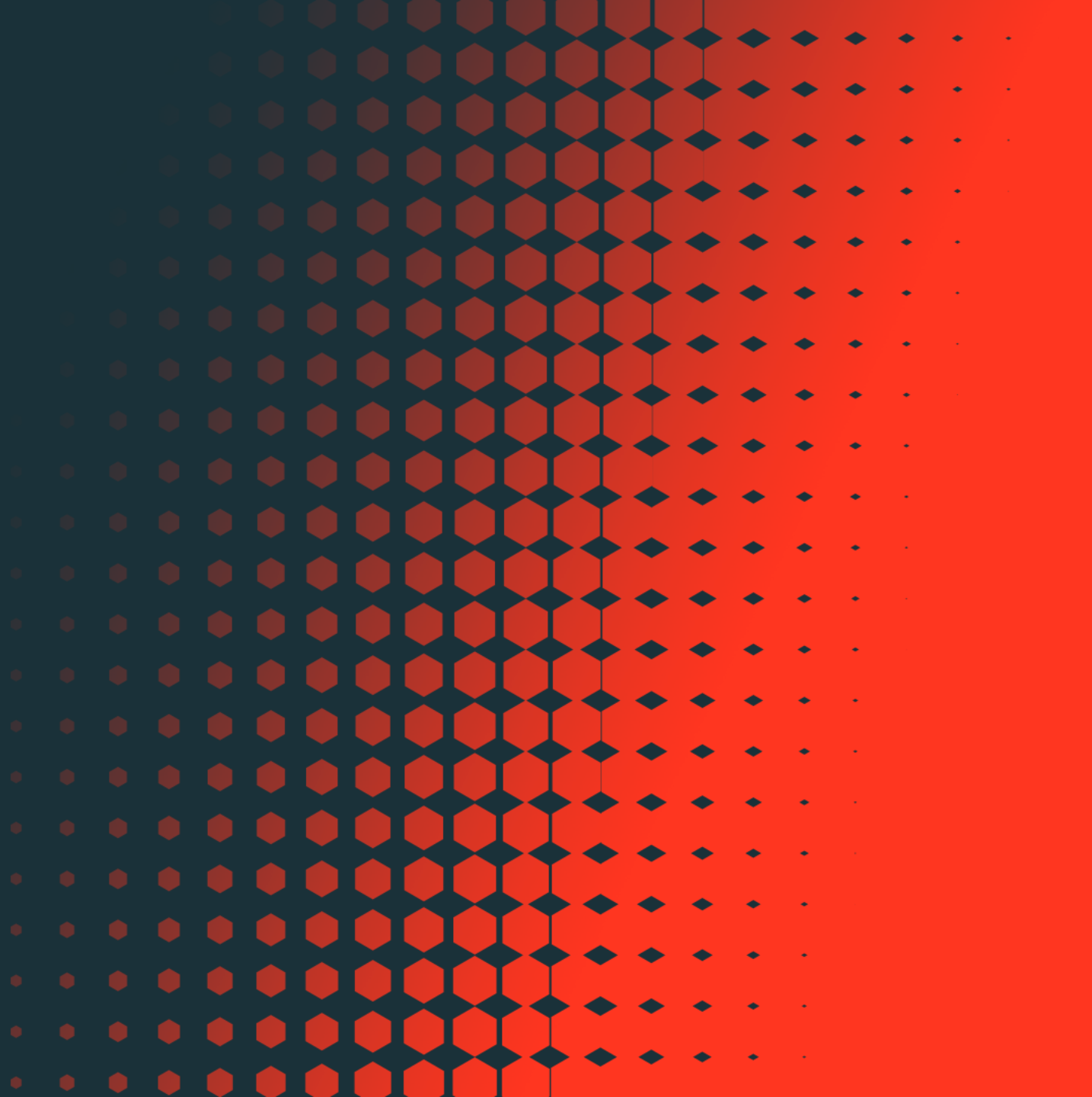
Columnar Reads w/Parquet, Review

We can see in the Spark UI how different (or similar) these strategies are:

Step	Filesystem read data size	scan time total	Physical Plan FileScan | ReadSchema
B	~100 GiB	43.2 minutes	all 6 columns
C	~30 GiB	16.9 minutes	trx_id, retailer_id & city_id
D	~30 GiB	16.2 minutes	trx_id, retailer_id & city_id

Note: The **Physical Plans** for **Step C** and **Step D** are **100% identical**!

Disk Partitioning



databricks

Disk Partitioning - Why?

- Spark can push a predicate down to the file scanner
- Only those directories that match the predicate are read in
- Spark infers column types and names from the directory structure further reducing the number of bytes read from disk
- Works for Delta, Parquet, ORC, CSV, JSON and many more
- See Partition Discovery for more information and the configuration setting **spark.sql.sources.partitionColumnTypeInference.enabled**

Predicate Pushdown w/Disk-Partitioning

See Experiment #2934

- Contrast **Step B** and **Step C** where the data is filtered by **city_id**
  - Note the total execution time of each command
  - In the **Spark UI**, **Query Details**
    - Note the **number of files read**
    - Note the **filesystem read data size total**
    - Note the **filesystem read time**

- What is different in each of these queries' **Physical Plans**?
- What is unusual/unexpected in **Step B**'s **Physical Plan**?

Predicate Pushdown w/Disk-Partitioning, Review

Step	Execution Time	number of files read	filesystem read data size	filesystem read time	Physical Plan FileScan	…?
B	~7 minutes	100	102 GB	~7 minutes	Includes **PushedFilters** (why?)
C	~10 seconds	1	285 MB	10 seconds	Includes **PartitionFilters**

Disk-Partitioning Warnings

- Over partitioning can lead to Tiny Files.
  - Consider a 1024 MB part-file now partitioned by hour
  - This will produce 24 part-files, each one ~42 MB (1024 / 24)
- Over partitioning can result in excessive directory scanning by creating a highly nested structure (recall Experiment #8973)
- Remember: Advertise partitioned columns so that they get used
- One of the most common mistakes consumers make is to create their own indexes as opposed to using the provided, optimized columns

Bad

```
👎 someDF.filter(year($"trx_date") === 1975)
```

Good

```
👍 someDF.filter($"p_trx_year") === 1975)
```

Dynamic Partition Pruning

New Strategies for Spark 3.x

Dynamic Partition Pruning

- Consider a 100GB table of **transactions** and a 30MB table of **cities**
- Without any filtering, this is a massive shuffle operation
- A 1-minute query on **transactions** can easily become an hour long join with the subsequent shuffle
- Filtering the **cities** table by country (e.g. USA only) means we are now joining a 100GB table to a 15MB table
- This is still a massive shuffle operation!

New Strategies for Spark 3.x

DPP - What Can We Do?

Both of these solutions can work but only to a very limited degree

We can broadcast the cities table:

- At 10+ MB, it's still too big for auto-broadcasting
- We would have to force a broadcast with a hint
- This wouldn't be an option if our "big" table was **1 TB** and our "small" table was **100 GB**

But it may not be reasonable to expect the consumers of your data to do this

We can create our own subquery

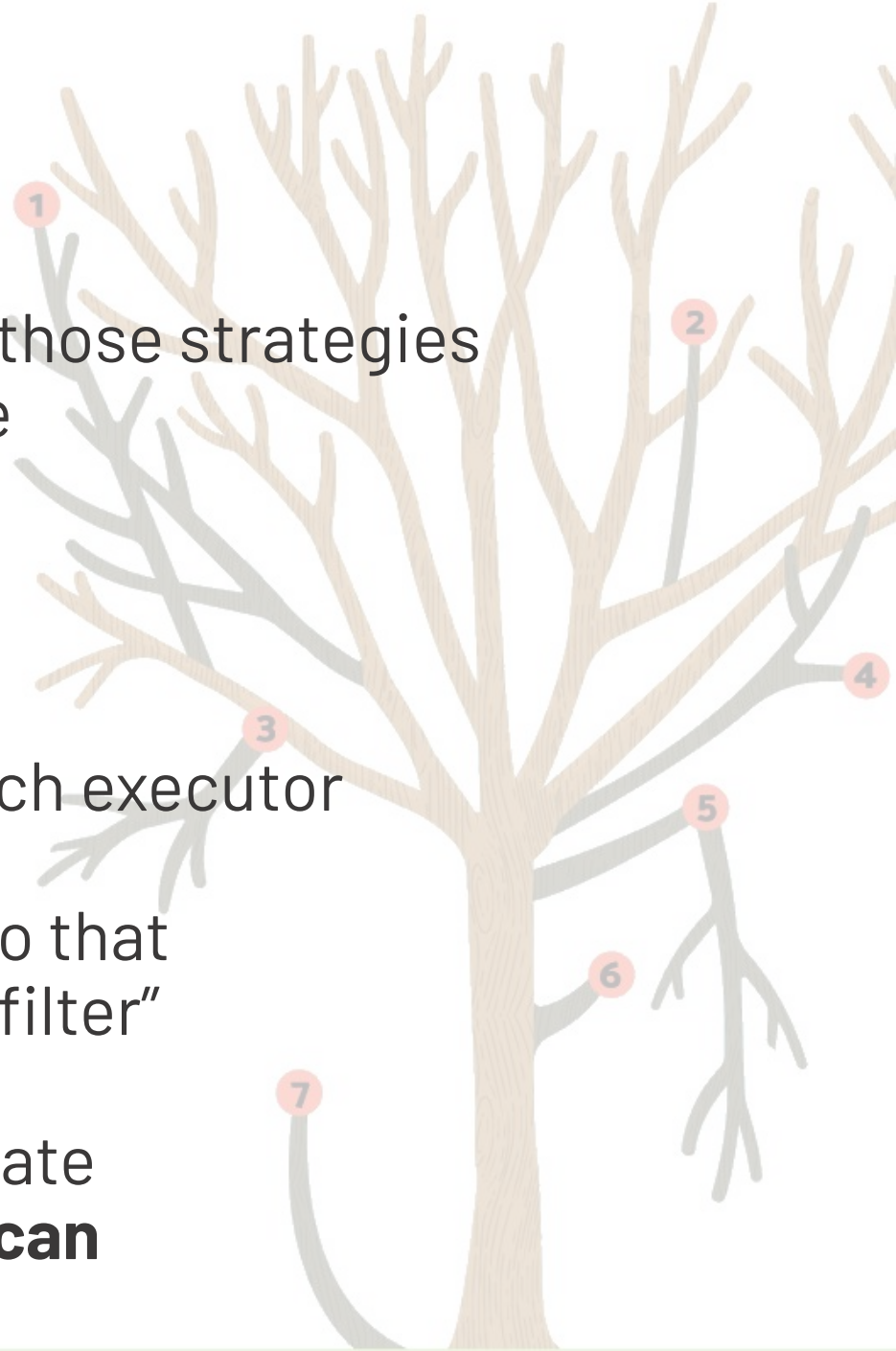
- Explicitly select all the US city ids and **collect()** them as the array **city_ids**
- Filter both the **transactions** and **cities** table with **$"city_id".isin(city_ids)**
- Join the ~70GB **transactions** table to the ~15MB **cities** table

New Strategies for Spark 3.x

What Does DPP Actually Do?

Dynamic Partition Pruning uses a combination of those strategies

- Spark will produce a query on the “small” table
- The result of which is used to produce a “dynamic filter” similar to our list of city_ids
- The “dynamic filter” is then broadcasted to each executor
- At runtime, Spark’s physical plan is adjusted so that our “large” table is reduced with the “dynamic filter”
- And if possible, that filter will employ a predicate pushdown so as to avoid an **InMemoryTableScan**



Storage

Storage

- Storage is composed of many smaller subproblems. When you have a choice, Delta is always the most performant option
- In some cases, Delta is not an option, e.g. ingesting data coming from an external system or data is being read by a system that doesn't support Delta
- For sake of time, we will only focus on Delta and the two most important points: the tiny files problem and correctly ordering data inside files
- NB: you can see later in the deck how you can optimize the storage for other file formats (CSV, JSON, etc.)

Storage - Spoiler Summary

Storage - Choose the right file format

- The main issues of reading data from conventional file formats are:
  - Row-Based storage makes compression algorithms very inefficient
  - Inability to filter columns at the source
  - Schema enforcement is not provided
  - Schema inference is very slow (provide schema with JSON and CSV to optimize)
  - Schema evolution is difficult to implement

- The Solution: Parquet
  - Columnar storage format
  - Very efficient and fast compression
  - Schema defined into header of each file (no more schema inference)
  - Enable schema evolution

Storage - Choose the right file format

- Issues with Parquet storage format:
  - Still no schema enforcement
  - Schema evolution is still not easy to maintain
  - No statistics are maintained about file content, therefore no data skipping
  - No ability to delete or update records, no ACID support
  - Directory Listing is still affecting performances
  - Spark ends up creating tiny files which affects performances
- The partial solution: Register Parquet tables on Hive metastore
  - Statistics are kept into the metastore and used for data skipping
  - Data skipping is still not effective because data is not ordered
  - Directory Listing problem is gone
  - Schema is defined in the metastore
  - All other issues remains: no ACID, no compaction
  - Other issues are added: Hive metastore needs to be maintained (repair and analyze table)

Storage - Choose Delta

- Based on Parquet
- Statistics are maintained into the metadata layer
- No need to use Hive metastore and repair/analyze table is gone
- ACID support for Update, Delete, Merge
- Compaction provided with OPTIMIZE and Auto-Optimize
- Z-ORDER makes data skipping very effective
- Schema enforcement and Schema evolution improved
- Bloom filters improve performances of joins
- Support for Time Travel
- Support for concurrent read/write
- Support for Dynamic File Pruning

Storage - Delta Optimizations

Storage

- Delta performance optimizations can be found here: https://docs.databricks.com/delta/optimizations/index.html
- Remember to schedule Vacuum and Optimize on all your Delta lakes

Delta & The Tiny Files Problem

The Tiny Files Problem

- Reading a file from cloud storage has an overhead
- We want to minimize that overhead
- We want to have less files per partition (possibly, a multiple of the worker nodes)
- We also have to consider impact on Delete/Update operations
- Size of partitions is important: compaction is done on per partition bases

Storage - Tiny Files In Action

See Experiment #8923, contrast **Step B**, **StepC** and **Step D**

- Note the total execution time of each job
- In the **Spark UI**, see the **Stage Details** for the last stage of each step and note the **Input Size / Records**
- In the **Spark UI**, see the **Query Details** for the last job of each step and note the…
  - **number of files read**
  - **scan time total**
  - **filesystem read time total**
  - **size of files read**

Storage - Tiny Files, Review

Step	Record Count	Execution Time	number of files read	scan time total	filesystem read time total	size of files read
B - Benchmark #1	~41 M	~3 minutes	6,273	20 minutes	~10 minutes	1,209 MB
C - Benchmark #2	~2.7 B	~10 minutes	100	1 hour	~1 hour	102 GB
D - Tiny Files	~34 M	~1.5 hours	345,612	12 hours	> 6 hours	2.1 GB

What can we do to mitigate the impact of tiny files?

Delta Lake Optimize and Auto Optimize

Tiny Files - The Basics

- Make sure you constantly Optimize and Vacuum your delta tables
  - Optimize will compact and Z-order your files
  - Vacuum will delete old versions and cleanup the metadata
  - Vacuum is necessary also if you are not updating/deleting records

- Enable Auto Optimize on all your tables unless there is a good reason for not doing so (e.g. streaming low latency requirements)

Storage - Why Auto Optimize?

- Manually compacting files, or writing them out correctly the first time, is the most efficient process
- Delta's Optimize and Auto Optimize can afford us the ability to focus on [potentially] bigger problems
- Auto Optimize adds a buffering layer to automatically compact data before writing them to cloud storage

Choose the Right File Size in Delta

- The spark configuration spark.databricks.delta.optimize.maxFileSize controls the file size for the OPTIMIZE command (default value 1GB)
- 1GB is good for very large delta lakes, but optimal size vary from 128MB to 1GB
- If you are running frequent updates, 128MB or less is probably the optimal size
- Check the size of your storage partitions:

Partitions Size	Recommended File Size
100GB or more	1GB is ok
10GB up to 100GB	500MB to 1GB
1GB to 10GB	128MB to 500MB
Less than 1GB	The table is over-partitioned

Delta Lake Z-Ordering

Delta Lake Z-Ordering

```
SELECT input_file_name() as "file_name",
       min(col) AS "col_min",
       max(col) AS "col_max"
FROM table
GROUP BY input_file_name()
```

file_name	col_min	col_max
data_file_1	6	8
data_file_2	3	10
data_file_3	1	4

Z-Order Clustering

- Drastically reduces the scan time for highly selective queries
- Supports binary comparison operators such as **StartsWith**, **Like**, **In <list>** as well as **AND**, **OR** and **NOT** operations and others
- Disk-partitioning targets large data segments (e.g. partitioned by year)
- Z-Ordering is ideal for the needle-in-the-haystack type of query
- Z-Ordering a table is as simple as adding **ZORDER BY** to **OPTIMIZE**

```
OPTIMIZE tableA ZORDER BY (column1, column2)
```

- Note: This features requires Delta on Databricks

Predicate Pushdown w/Z-Ordered Data

Revisit Experiment #2934, see **Step D**

- Note the total execution time of the command
- In the **Spark UI**, **Query Details**
  - Note the **number of files read**
  - Note the **filesystem read data size total**
  - Note the **filesystem read time**
- Compared to **Step B** and **Step C**, how is the **Physical Plan** different?

Predicate Pushdown w/Z-ordered Data, Review

Step	Execution Time	number of files read	filesystem read data size	filesystem read time	Physical Plan FileScan | …?
B	~7 minutes	100	102 GB	~7 minutes	~~Includes **PushedFilters**~~
C	~10 seconds	1	285 MB	~10 seconds	Includes **PartitionFilters**
D	< 1 minute	33	10.3 GB	~4 minutes	Includes **PushedFilters** & **DataFilters**

Z-order Use Cases

- In the last example, we used a Z-ordered index to filter by **city_id** (with low-cardinality)
- But Z-order is most effective with indexes that feature high-cardinality
- This would be equivalent to searching a relational database by primary key or other unique id
- For example, what if we wanted to find one record out of a Terabyte dataset?

Z-ordered, Haystack Query, In Action

See Experiment #1337 which queries a 1-Terabyte dataset for a single ID

- Contrast **Step B** (unoptimized) to **Step C** (indexed)
  - Note the total execution time of each command
  - In the **Spark UI**, **Query Details**,
    - Note the **number of files read**
    - Note the **filesystem read data size total**
    - Note the **filesystem read time**

Z-ordered, Haystack Query, Review

Step	Execution Time	number of files read	filesystem read data size	filesystem read time
B	17 minutes	1,024	1016 GB	~7.5 hours
C	2.5 minutes	360	117 GB	~1 hour

One more note:

- **2.5 minutes** is with only **128 cores** processing **993 tasks** in **8 iterations**
- With a properly sized cluster (say **1024 cores**), this same query can complete in as little as **20 seconds**

Z-Order Clustering

For more information, see:

- Databricks: Z-ordering (multi-dimensional clustering)
- Databricks: Z-order by columns
- Wikipedia: Z-order curve
- Processing Petabytes of Data in Seconds with Databricks Delta

Delta Lake Z-Ordering

- Z-Ordering will order the storage inside a Parquet file to improve data skipping
  - https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html

- Now available a new ordering algorithm: Hilbert Curve (DBR 7.2+)
  - Spark conf: spark.databricks.io.skipping.mdc.curve=hilbert (default:zorder)

Additional Tuning for BI Queries

		delta.checkpoint.writeStatsAsStruct	
		false	true
delta.checkpoint.writeStatsAsJson	false	! No data skipping	• Faster queries on Databricks Runtime 7.3 LTS and above • Slightly slower checkpoints • No data skipping in readers on Databricks Runtime 7.2 and below
	true	• Databricks Runtime 7.2 and below • Slower queries	• Faster queries on Databricks Runtime 7.3 LTS and above • Maintains compatibility with readers on Databricks Runtime 7.2 and below • Checkpoints have the highest latency (order of seconds)

https://docs.databricks.com/delta/optimizations/file-mgmt.html#improve-interactive-query-performance

What do I do when I have too many columns to Zorder?

CREATE BLOOMFILTER INDEX

databricks

History of Bloom Filters

A **Bloom filter** is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set."

CREATE BLOOMFILTER INDEX

- To accelerate “Needle in the haystack” type filter queries
- Use for attributes not in the ZORDER index
- Use it for attributes not in a partition key
- Use it for attributes that don’t fit easily within the min/max data skipping filters

Test Results

Databricks ran a series of performance tests to determine the value of Bloom Filters in three sample data sets:

1. Very High Cardinality: 50B rows with unique hashes (10 TB data on disk). Each column had a unique SHA256 hash (the ID number combined with a column-specific salt): hash_basic, hash_bloom, hash_zorder.

Dataset One: Very High Cardinality

Description

50 billion rows, three columns each with a unique SHA256 hash: hash_basic, hash_bloom, hash_zorder. 10 TB data on disk, zordered by hash_zorder (bloom filter took ~ 36-136 GB). 41 x i3.xl workers.

Bloom Settings	Column	# Hashes Searched	Fastest	Median	Average	Slowest	% vs Slowest (Avg)
N/A - Z-ordered Column	zorder	8	8.74	10.93	11.07	15.78	4121%
fpp=0.01, numItems=10000000	bloom	8	66.59	71.58	88.32	205.97	517%
fpp=0.1, numItems=10000000	bloom	8	75.91	89.14	100.29	190.73	455%
fpp=0.001, numItems=10000000	bloom	8	100.86	103.75	131.12	327.59	348%
fpp=0.0001, numItems=10000000	bloom	8	129.21	137.53	179.14	454.12	255%
fpp=0.00001, numItems=10000000	bloom	8	157.22	160.62	196.4	450.43	232%
N/A - Unaccelerated Column	basic	8	268.47	273.8	445.85	1654.23	102%
No Options Specified	bloom	8	240.36	248.46	456.2	1831.91	100%

Test Results ...

Low Cardinality: Same as dataset one (50B rows, three cols each), but only 10k hashes repeated to get to 50B events (sha256 of ID col % 10000 with col-specific salt), consuming 420 GB on disk.

Dataset Two: Low Cardinality

Description

Same as dataset one (50B rows, three cols each), but only 10k hashes repeated to get to 50B events (sha256 of ID col % 10000 with col-specific salt). 420 GB on disk, zordered by hash_zorder (bloom filter took 35-145MB). 41 x i3.xl workers.

Bloom Settings	Column	# Hashes Searched	Fastest	Median	Average	Slowest	% vs Slowest (Avg)
fp=0.1, numItems=10000	bloom	8	19.69	21.88	22.15	28.35	265%
fp=0.001, numItems=10000	bloom	8	19.55	22.6	22.92	28.35	256%
fp=0.00001, numItems=10000	bloom	8	20.61	22.71	23.54	31.39	249%
fp=0.0001, numItems=10000	bloom	8	19.22	23.4	24.23	35.3	242%
N/A - Z-ordered Column	zorder	8	18.02	20.96	26.4	60.96	222%
N/A - Unaccelerated Column	basic	8	49.42	52.16	58.59	101.52	100%

Storage - Non-Delta Formats

When you don’t have a choice?

- Always specify the schema, especially with JSON and CSV
- Use Autoloader with queue to avoid listing costs
- Use Streaming and Streaming Trigger Once for ETL
- For large files: the format might require spark to read on a single core
  - Gzip compression
  - Multiline CSV or JSON
- Should I prefer Kafka/Kinesis over dumping files on a cloud storage?
  - Consider the limits (max 1 MB of size per message)
  - Consider the infrastructure and maintenance costs

The most common issues with non-Delta formats

- Directory Scanning Costs
- Schema Inference Costs
- Non splittable file formats

Storage - Directory Scanning

The next version of the “Tiny Files Problem” is Directory Scanning

- Here is the idea:
  - One can list the files in a single directory
  - A single list with thousands of files is still OK
  - The scan is still not as bad as the overhead of reading tiny files
- Highly partitioned datasets (data on disk) present a different problem:
  - For every disk-partition there is another directory to scan
  - Multiplied that number of directories by N secondary & M tertiary partitions
  - These have to be scanned by the driver one directory at a time

Storage - Scanning Example

Consider this common scenario:

- Consider 1 year's worth of data partitioned by year, month, day & hour
- 1 year * 12 months * 30 days * 24 hours = 8,640 distinct directories
- 10 years of data becomes 86,400 directories.

Storage - Scanning In Action

See Experiment #8973

- Contrast **Step B**, **Step C** and **Step D**
  - Note the we are not executing any actions - only declaring the DataFrames
  - Note the total execution time for each command
  - Note the results for **countFiles(..)**
    - The **Records** total
    - The **Files** total
    - The **Directories** total
- See **Step E, F & G** for more variants and how they affect scanning
- For **Step J** open the **Spark UI** and look at the **Query Details** for the last job
  - Identify the proof that scanning is the root cause of this performance problem

Storage - Scanning, Review

Step	Description	Duration	Records	Files	Directories
B	~100 records per part-file (tiny files)	~1 minute	34,562,416	345,612	1
C	Partitioned by year & month	~ 5 seconds	36,152,970	6,273	12
D	Partitioned by year, month, hour & day	~15 minutes	37,413,338	6,273	8,760

Storage - Scanning, Prove It

What proof is there in the **Query Details** for Experiment #8973, **Step J**, that scanning is the root cause of these performance problems?

Scan parquet +details
Stages: 75.0

number of files read	8,760
filesystem read data size total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
scan time total (min, med, max)	1.71 h (1.9 s, 6.3 s, 7.6 s)
estimated repeated reads high size total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
filesystem read data size (sampled) total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
filesystem read time (sampled) total (min, med, max)	16.6 m (137 ms, 1.2 s, 1.9 s)
metadata time	36 ms
size of files read	1108.6 MiB
estimated repeated reads low size total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
number of partitions read	8,760
rows output	37,413,338

What can we do to mitigate the impact of scanning?

Storage - Can we...?

What happens if we were to specify the schema?

- See Step H and determine if this solution works

Nope

What happens if we were to register it as a table?

- See Step I and determine if this solution works

Yes!

Storage - Avoiding Scanning costs

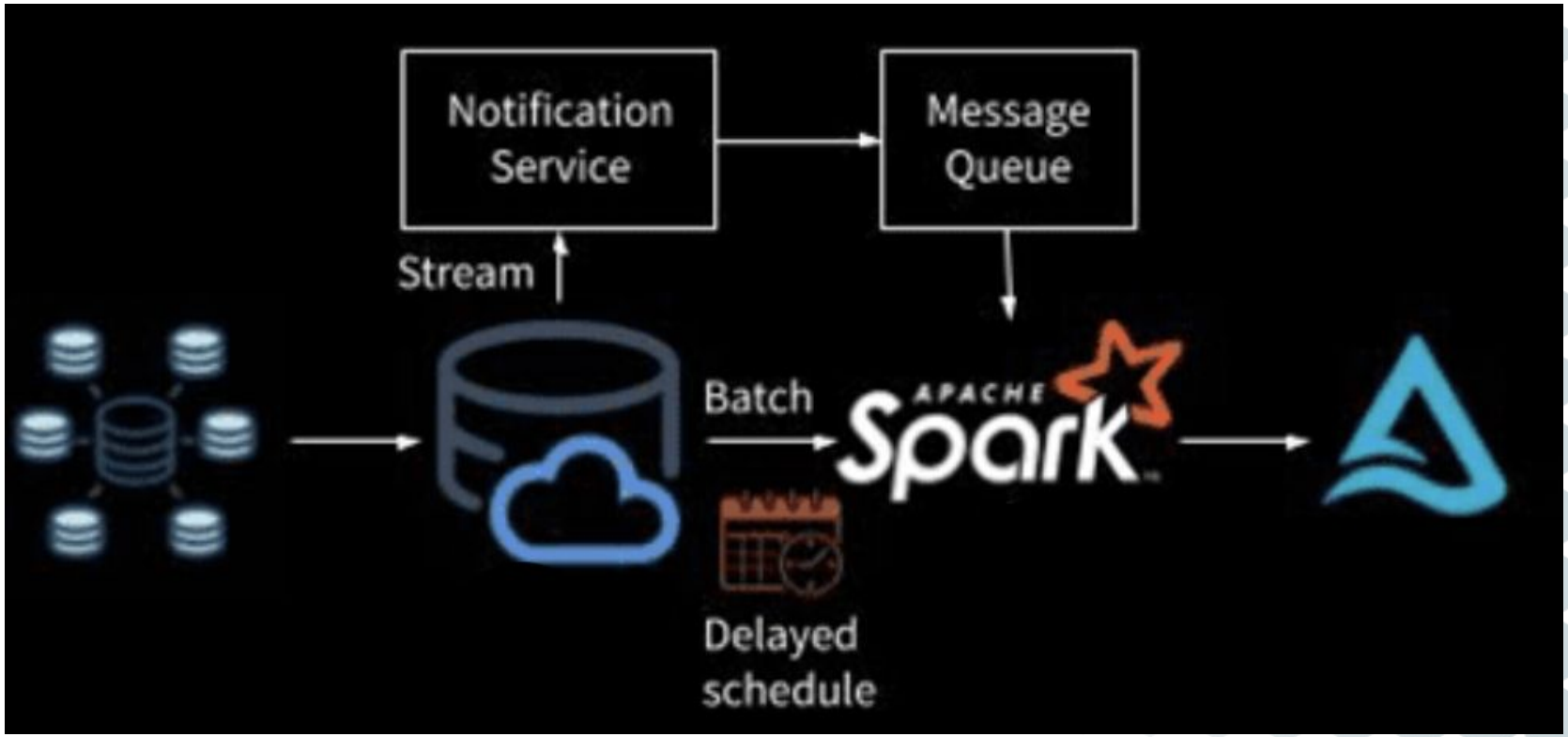
- Use Autoloader with cloudFiles.useNotifications = True https://docs.databricks.com/spark/latest/structured-streaming/auto-loader.html

- Register the table to the Hive metastore (however, you will then need to maintain the Hive metastore, repairing the table when you add new partitions)

- Recommended: Use Delta when it is possible
  - No Hive metastore maintenance required
  - Metadata layer will be used instead of scanning

Storage - Autoloader

- The notification system of Autoloader uses cloud storage events generated by the cloud storage to obtain a list of new files to be processed.
- No listing/scanning required!



Storage - Schema Inference and Schema Evolution

- Inferring schemas (for JSON and CSV) require a full read of the file to determine data types, even if you only want a subset of the data
- Reading Parquet files requires a one-time read of the schema
- However, supporting schema evolution with Parquet is [potentially] expensive
  - If you have hundreds to thousands of part-files, each schema has to be read in and then merged which collectively can be fairly expensive
  - Schema merging was turned off by default starting with Spark 1.5
  - Enabled via the **spark.sql.parquet.mergeSchema** configuration option or the mergeSchema option

What can we do to mitigate the schema issues?

Storage - Schema Mitigation

There are several ways to mitigate some of these issues:

- Provide the schema every time
  - Especially for JSON and CSV
  - Applicable to Parquet and other formats
- Use tables - the backing meta store will track the table's schema

Splittable vs Unsplittable Compression Formats

- You have to ingest 100GB of CSV data.
- Every line represent a record (no newlines within quotes, multiline=false)
- Possible alternatives:
  - Uncompressed CSV (1 big file)
  - GZIP compressed (1 big file)
  - BZIP2 compressed (1 big file)
- Do they all perform exactly the same way?

Splittable vs Unsplittable Compression Formats

Files	Compression	Multiline	Spark Partitions	Spark Tasks	Worker cores used
1	uncompressed	False	many	many	many
1	uncompressed	True	1	1	1
1	GZIP	*	1	1	1
1	BZIP2	False	many	many	many
1	BZIP2	True	1	1	1
10	GZIP	*	10	10	10

Storage - Summary

Storage - Choose the right file format

- The main issues of reading data from conventional file formats are:
  - Row-Based storage makes compression algorithms very inefficient
  - Inability to filter columns at the source
  - Schema enforcement is not provided
  - Schema inference is very slow (provide schema with JSON and CSV to optimize)
  - Schema evolution is difficult to implement

- The Solution: Parquet
  - Columnar storage format
  - Enable schema evolution
  - Schema defined into header of each file (no more schema inference)
  - Very efficient and fast compression

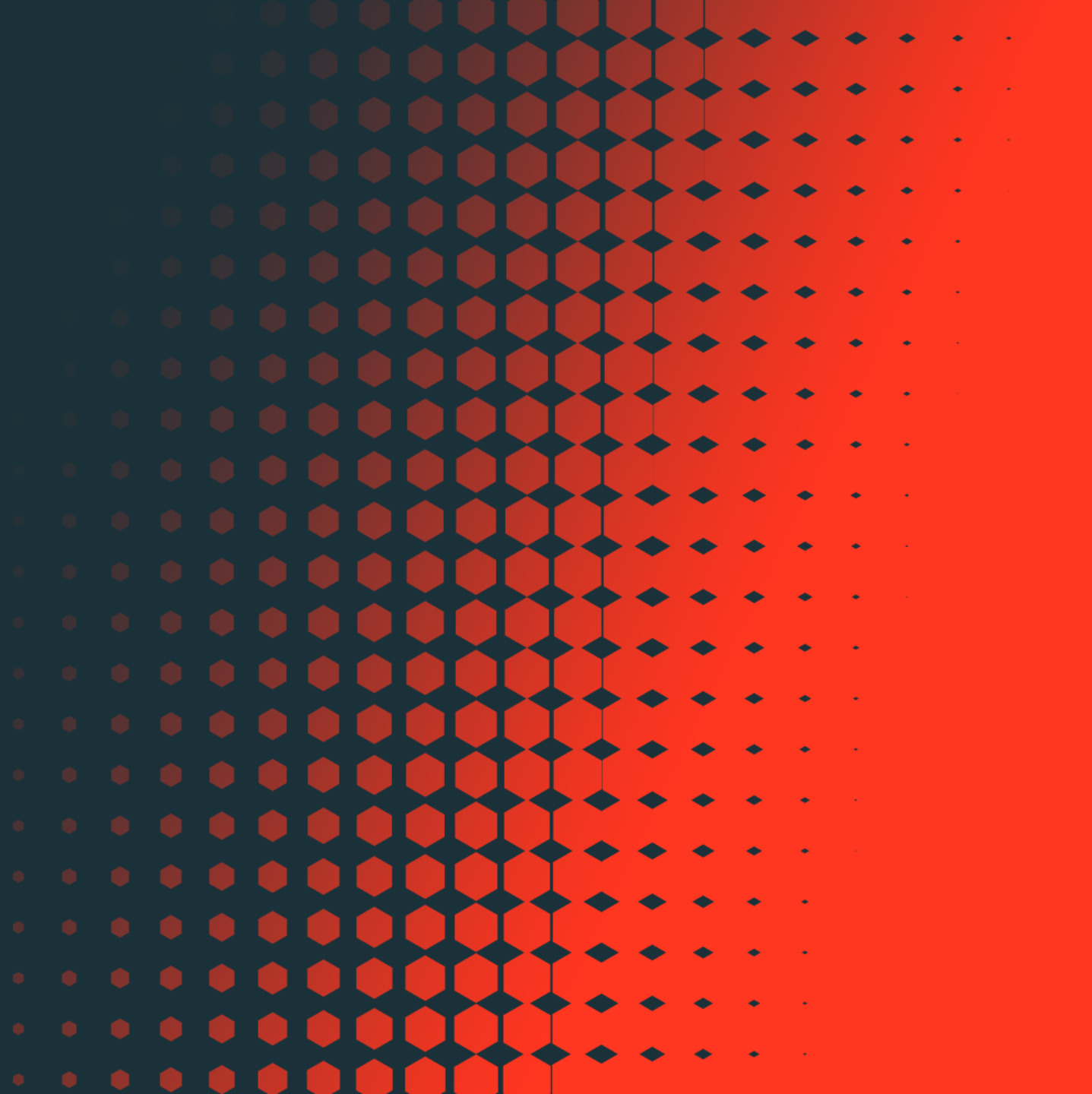
Storage - Choose the right file format

- Issues with Parquet storage format:
  - Still no schema enforcement
  - Schema evolution is still not easy to maintain
  - No statistics are maintained about file content, therefore no data skipping
  - No ability to delete or update records, no ACID support
  - Directory Listing is still affecting performances
  - Spark ends up creating tiny files which affects performances
- The partial solution: Register Parquet tables on Hive metastore
  - Statistics are kept into the metastore and used for data skipping
  - Data skipping is still not effective because data is not ordered
  - Directory Listing problem is gone
  - Schema is defined in the metastore
  - All other issues remains: no ACID, no compaction
  - Other issues are added: Hive metastore needs to be maintained (repair and analyze table)

Storage - Choose Delta

- Based on Parquet
- Statistics are maintained into the metadata layer
- No need to use Hive metastore and repair/analyze table is gone
- ACID support for Update, Delete, Merge
- Compaction provided with OPTIMIZE and Auto Optimize
- Z-ORDER makes data skipping very effective
- Schema enforcement and Schema evolution improved
- Bloom filters improve performances of joins
- Support for Time Travel
- Support for concurrent read/write
- Support for Dynamic File Pruning

Designing Clusters for High Performance

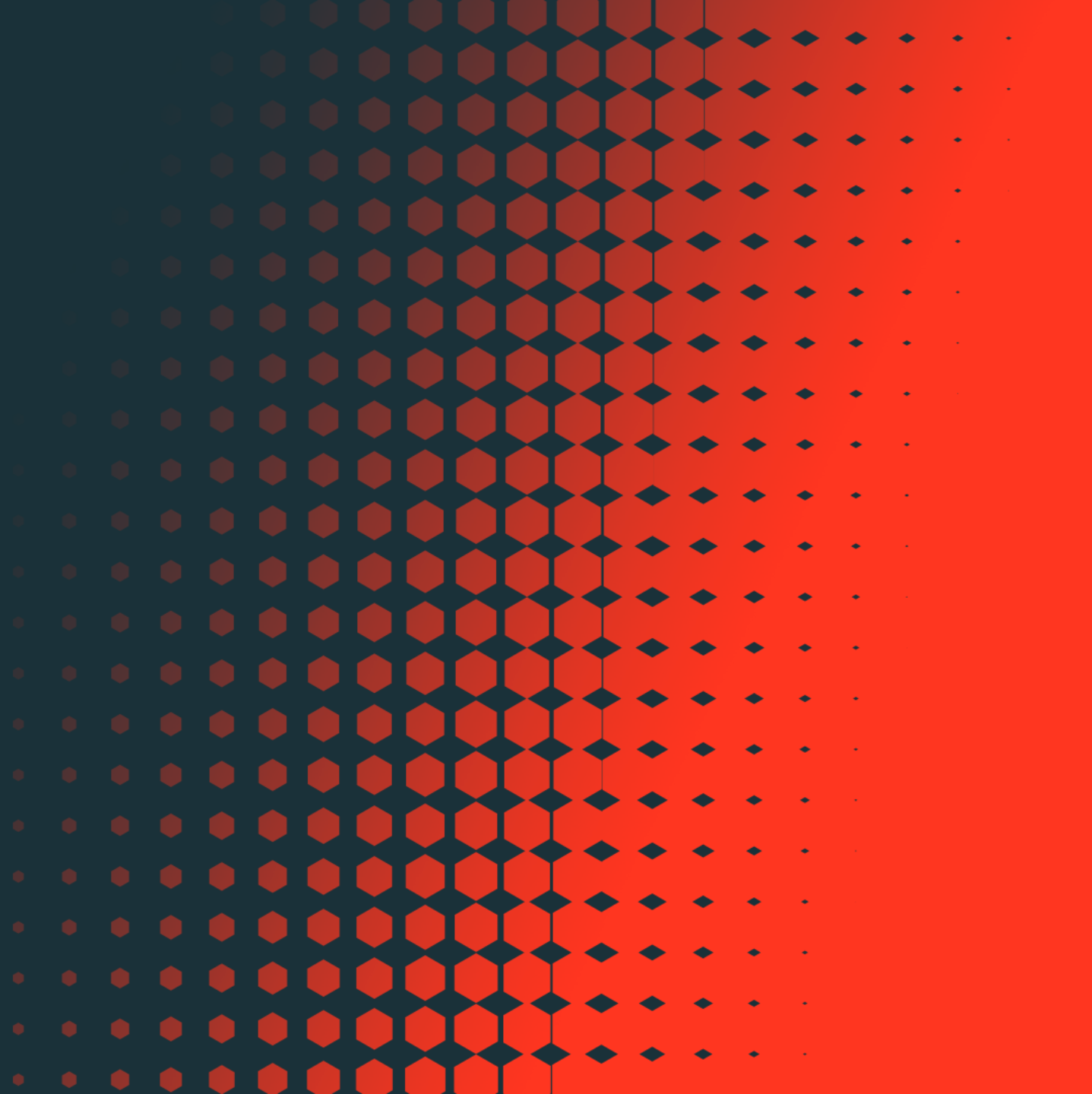


WWWWHW

Before designing the cluster, we need to answer 6 questions:

- Who will be using the cluster?
- What will the cluster be used for?
- Where will the cluster [and data] reside?
- When are the results needed?
- How do I control/predict the costs?
- Why do I care about all these details?

Who will be using the cluster?



Who will be using the cluster?

At one level, we can split on personas...

- Data Analyst
- SQL Analyst
- Data Scientist
- Data Engineers
- ...and everyone else (intentionally oversimplified)

Who - SQL & Data Analyst

- Pull data, aggregate it, report
- Should be working on clean (e.g. silver or gold) data
- Highly repetitive queries
- Large gaps between queries
- Quick to join two datasets

- Regularly employs caching
- SQL Analyst generally use little to no code
- Data Analyst generally use a mix of SQL and code (often Python)

Who - Data Scientist

- Behavioral overlap with Data Analyst (DS often wear both hats)
- Approach the data from a statistical perspective (e.g. more compute)
- Regularly engage in data preparation
- Train ML models with highly iterative jobs
- Need to cache full datasets for training

Who - Data Engineers

- Produce end-to-end applications
- Driven by SLAs more so than Data Analyst and Data Scientist
- Build complex data pipelines
  (raw→bronze→silver→gold)
- Productionizing jobs, ML models and reports
- Non repetitive queries, scheduled jobs

Who - Groups

Besides personas, we also have to consider groups:

- Different data restrictions for Group A vs Group B
- Groups with heavy cluster demands (e.g. engineers)
- Groups with light cluster demands (e.g. SQL Analyst)
- Groups that will share a cluster

What will the cluster be used for?

What will the cluster be used for?

- Ad Hoc Data Analysis
- Reporting Generation
- Training ML & Deep Learning Models

- Structured Streaming Jobs
- Batch ETL
- Data Pipelines

How a cluster is used often follows the persona of the person using it

What - Ad Hoc Data Analysis

- **Level**: The amount of memory and compute is difficult to estimate
- **Shuffles**: Often employs arbitrary wide operations (e.g. joins). Tuning needs to be done in the general case, not query specific.
- **Caching**: Repetitive queries can benefit from Delta caching. Spark caching should only be used to temporarily store aggregates.
- **Opportunity**: Little opportunity for specific tuning
- **SLAs**: Generally not applicable

The dynamic nature of this activity makes planning and tuning hard

What - Reporting Generation

- **Level**: The amount of memory and compute can be fine tuned
- **Shuffles**: Shuffles can be mitigated with use-case specific datasets
- **Caching**: Generally does not require repetitive reads
- **Opportunity**: More opportunity for job-specific tuning
- **SLAs**: May be imposed in that the report is a job in and of itself

Presumably “lighter” than full batch ETL, this activity
is still nothing more than periodically running a job

What - Training ML & Deep Learning Models

- **Level - first run**: Memory and compute is difficult to estimate
- **Level - secondary runs**: Memory and compute can be fine tuned
- **SLAs - first run**: Difficult to impose requirements on first runs
- **SLAs - secondary runs**: May be imposed when retraining a model

- **Shuffles**: Shuffles can be mitigated with use-case specific datasets
- **Caching**: Entire datasets are often cached to achieve maximum performance via Spark and Delta caches
- In some cases GPUs may be warranted to maximize memory and/or compute

What - Structured Streaming Jobs

- **Level**: The amount of memory and compute are often over-provisioned to enable a stopped stream to catch up
- **Shuffles**: Shuffles MUST be mitigated by tuning the cluster, and when joining other datasets, with use case specific datasets
- **Caching**: Generally not employed in a streaming job
- **Opportunity**: High opportunity for specific tuning
- **SLAs**: Dictated by the throughput of the stream - possibly schedule & terminate short-running streams (e.g. 2 hr, every 1 wk)

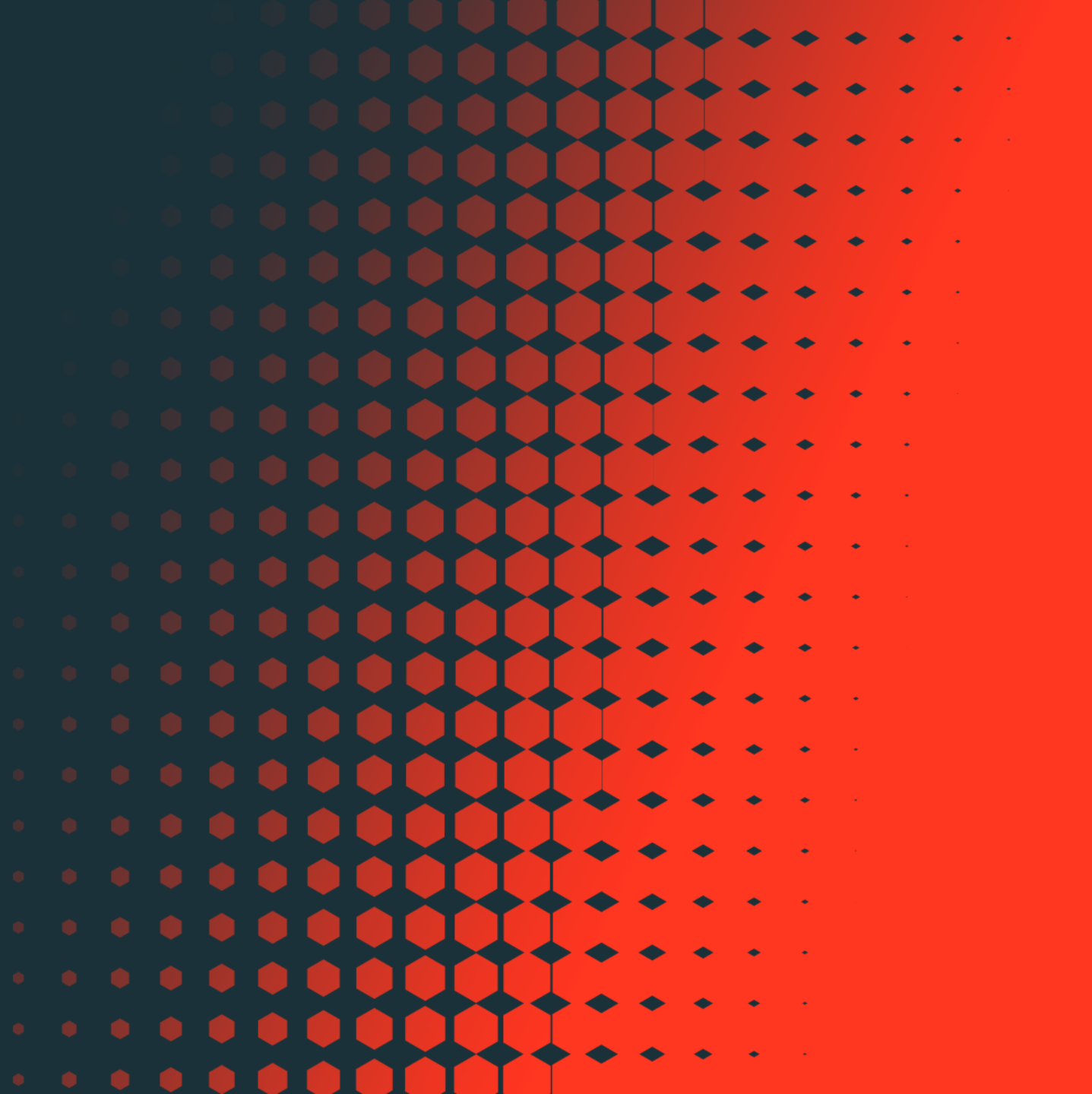
What - Batch ETL

- **Level**: The amount of memory and compute can be fine tuned
- **Shuffles**: Shuffles "cannot" be mitigated with use-case specific datasets - should be mitigated by fine tuning
- **Caching**: Generally does not require repetitive reads (opt for Delta caching over Spark caching if needed)
- **Opportunity**: High tuning opportunity as it pertains to a specific job
- **SLAs**: Post tuning, different SLAs levels are met by adjusting compute levels (e.g. size of and number of executors and thus cores)

What - Data Pipelines

- Batch ETL can be thought of as one instance of data-in and data-out
- Data Pipelines can be thought of as many batches orchestrated to work together
- Do you use one general-purpose cluster for the entire pipeline? Or does each "job" get a highly-tuned configuration?
- SLAs will most likely decide between a shared-cluster design vs a cluster-per-batch design
- For example, low-latency pipelines might require a cluster-per-batch design where higher latency pipelines might use a shared cluster.

When are the results needed?



When are the results needed?

A Spark job's SLA generally refers to how long it takes to "deliver" data

- Real-Time
- Near Real-Time
- ...and everything else kind of depends

When - Real-Time

- The data needs to be “processed” as soon as it arrives
- Generally involves Structured Streaming jobs
  (but Structured Streaming jobs are not only for real-time SLAs)
- Requires super-low, if not near-zero, latency
  (e.g. processing the data as soon as it arrives)
- This cluster’s configuration will generally have a
  priority to compute and IO throughput over memory
- Clusters are often design with fault tolerance in mind
  (e.g. recovering from random executor failures)

When - Near Real-Time

- The data needs to be “processed” faster than it arrives
- Generally involves Structured Streaming jobs
- Does not require low-latency, only to process the data as fast as it arrives (not as soon as)
- The cluster will generally have a priority to compute over latency and memory

When – "It Depends" #1

Example #1: Yesterday's Sales

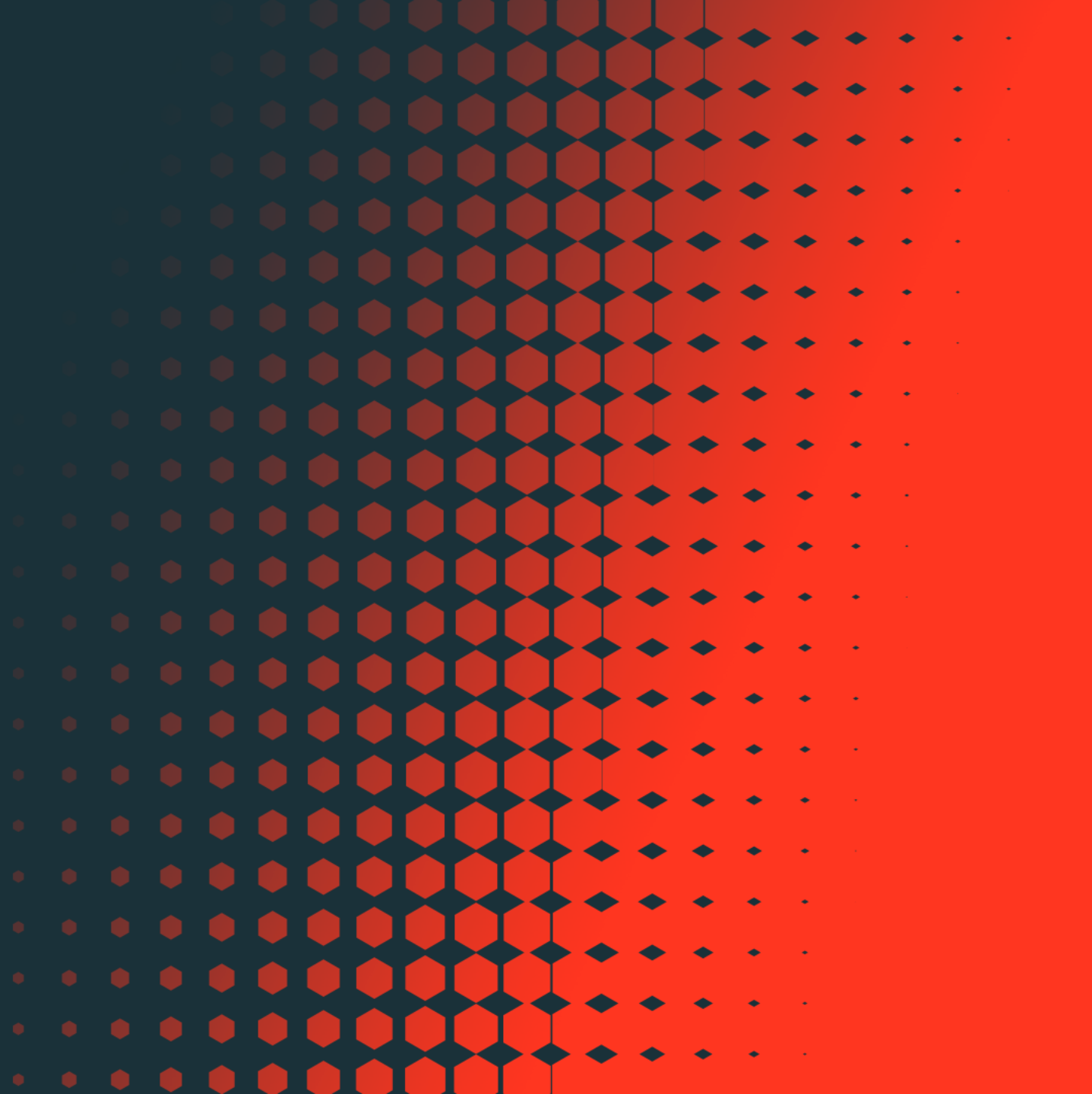
- Data arrives at midnight
- The report must be ready by 9 AM the following morning
- We have up to 9 hours to "deliver" the data
- Given the hours of execution, cluster stability might be a concern
- Multiple executors will help mitigate this, but we may want to limit ourselves to 4 hours of execution in case it has to be reran
- In this case a job-specific cluster sized and tuned to 4 hours of execution would be enough to support retrying the job
- There is no need/harm to tune to 1 hour of execution

When – "It Depends" #2

Example #2: Last Month's Sales

- Data is collected over the course of the month (1st to ~31st)
- The report must be ready by the 7th of the following month
- We have up to 7 days to "deliver" the data
- Our untuned implementation takes 24 hours to complete
- A commodity or even a shared cluster would suffice for this job
- If performance is impacted by low memory (e.g. spill) or other jobs, there is still plenty of time. A job-specific cluster may be unwarranted.
- Prudence would dictate that one not tune this job
- The cost of tuning this job is not justifiable given its SLA and possible labor

How do I control/predict the costs?



How do I control/predict the costs?

When it comes to cost, there are a couple of things to keep in mind:

- **Rule #1**: Use a cloud provider
- **Rule #2**: It's the labor, not the service
- **Rule #3**: Select the right [level] of service
- **Rule #4**: Reduce service costs by shortening compute time

Costs - Compute Levels

The price between a **Level-N** VM and a **Level-N+1** VM is 2x the cost, with 2x the resources

Level	Cores	Size	Cloud-A		Cloud-B	
1	4	Small	30.5 GB	$0.266 / hour	28 GB	$0.299 / hour
2	8	Medium	61.0 GB	$0.532 / hour	56 GB	$0.598 / hour
3	16	Large	122.0 GB	$1.064 / hour	112 GB	$1.196 / hour

Costs - Actual Consumption Cost

Assume you have a job with 256 partitions and

Level	Cores	VMs	Max Compute (cores * VMs)	Iterations (max/part)	Actual Durations (iterations * min)	~Price/Hour (level $ * VMs)	VM Costs (VMs * dur * price / 60)
1	4	1	4	64	128 minutes	$0.283	60¢
1	4	64	256	1	minutes	$0.283	60¢
2	8	1	8		utes	$0.565	60¢
3	16	1	1		s	$1.130	60¢
3	16	8	1		minutes	$1.130	60¢

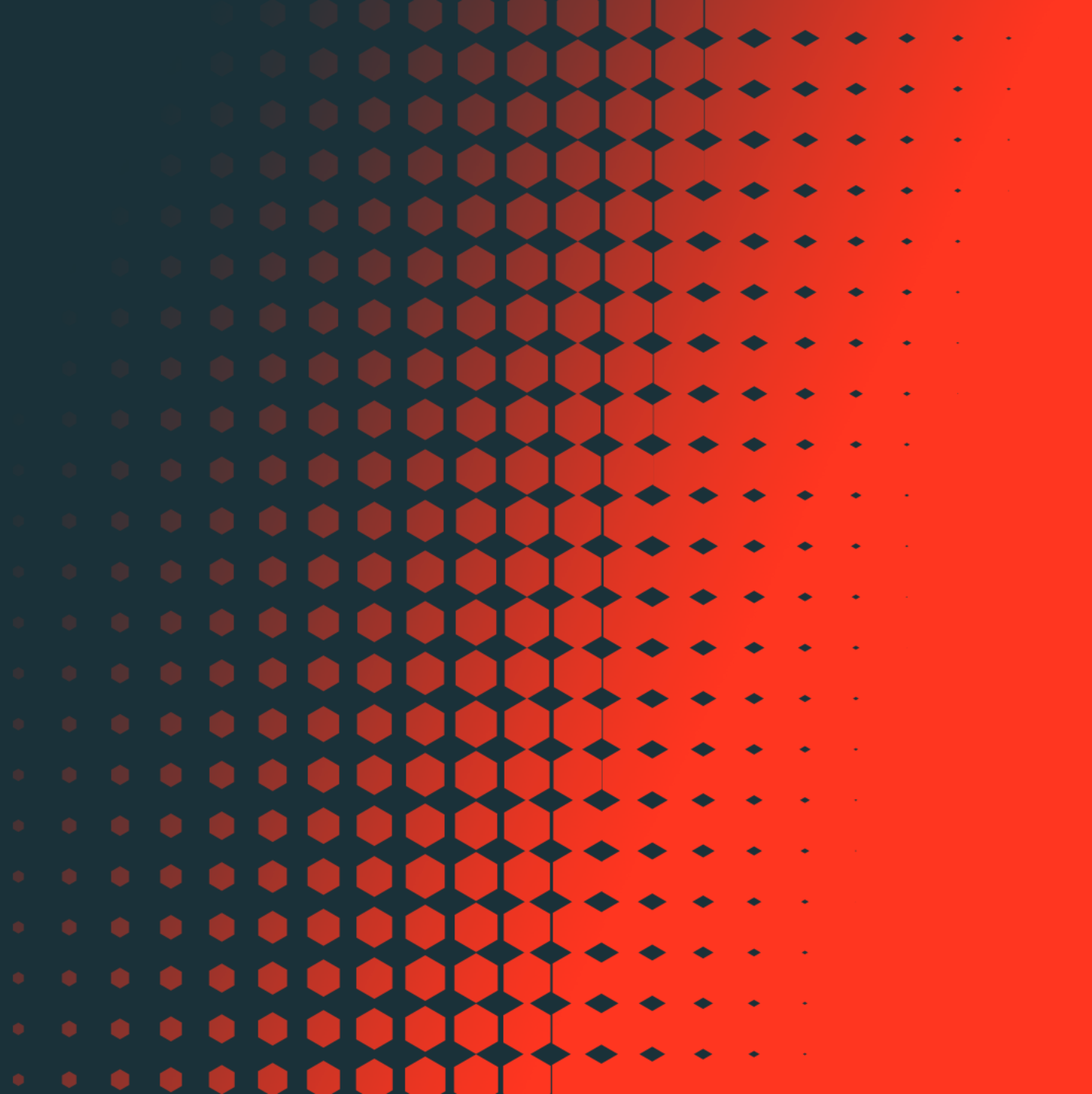
Costs - Developer Costs

Another factor to consider is the cost of the developers:

- What are they doing when the job is running?

Level	Cores	VMs	Max Compute (cores * VMs)	Iterations (max/part)	Actual Durations (iterations * min)	~Price/Hour (level $ * VMs)	Costs	$/Hour ($50 * dur / 60)
1	4	1	4	64	128 min	$0.283	60¢	$106.66
1	4	64	256	1	2 min	$0.283	60¢	$1.66
2	8	1	8	32	64 min	$0.565	60¢	$53.33
3	16	1	16	16	32 min	$1.130	60¢	$26.66
3	16	8	128	2	4 min	$1.130	60¢	$3.33

VM Selection



VM Selection: Effect on Shuffles

Level	Cores	VMs	Max Compute (cores * VMs)	Iterations (max/part)	Actual Durations (iterations * min)	Notes
1	4	1	4	64	128 minutes	No network IO
1	**4**	**64**	**256**	**1**	**2 minutes**	**Heavy network IO between 64 VMs**
2	8	1	8	32	64 minutes	No network IO
3	16	1	16	16	32 minutes	No network IO
3	16	8	128	2	4 minutes	Reasonable(?) network IO
7	**256**	**1**	**256**	**1**	**2 minutes**	**Most optimal shuffle experience**

VM Selection: Categories

Categorization	Amazon	GBs	Cores	MS Azure	GBs	Cores
Memory Optimized	r4.xlarge	30.5	4	DS12_v2	28.0	4
Compute Optimized	c5.xlarge	8.0	4	F4s	8.0	4
Storage Optimized	i3.xlarge**	30.5	4	L4s**	32.0	4
GPU Optimized	p2.xlarge	61.0	1	NC6s_v3	112.0	1
General Purpose	m5.xlarge	16.0	4	DS3_v2	14.0	4

Only a sample of VMs are shown here. Each type is represented by N different levels of memory and cores. Availability varies by cloud.

VM Categories

Memory Optimized

- ML workload with data caching
- If shuffle-spill remains a problem (no other mitigation strategy)
- When spark-caching is a requirement

Compute Optimized

- ETL with full file scans and no data reuse
- Structured Streaming Jobs

General Purpose

- Used in absence of specific requirements

Storage Optimized

- Optimized with Delta IO Caching !!
- ML & DL workloads with data caching
- Data Analysis / Analytics
- If shuffle-spill remains a problem (no other mitigation strategy)
- When spark-caching is a requirement

GPU Optimized

- ML & DL workloads with exceptionally large memory and compute requirements (presumes caching)

Guessing Compute Level

Experimentation is easy...

- Make a guess
- If you are spilling, assume you need more RAM
  (unless you have skew)
- If your shuffles are slow, increase VM Level while decreasing the number of VMs

- How many iterations did it take? Increasing the VM Level or number of VMs for more cores
- Is your cluster underutilized? Reduce the VM level or number of VMs for fewer cores
- Expect this processes to take a fair amount of trial and error (aka time, aka money)

Estimate Compute Level

1. Calculate the data's size on disk
2. **spark.sql.files.maxPartitionBytes?**
3. Compute the number of partitions or cheat and call **df.rdd.getNumPartitions()**
4. Decide which category of VM you want
5. Based on the SLA, quota, and budget, select the type and level of VM
6. Select the number of iterations
7. Compute the number of VMs
8. Adjust, experiment and retest - at least time (and money) is saved by starting with a semi-reasonable configuration

How?

1. Assume we have 100 GB or **102,400 MB**
2. Assume **maxPartitionBytes** is **128 MB**
3. **102,400 MB** / **128 MB** = **800 partitions**
4. **Compute Optimized**
5. **Level 5**, **144 GB**, **72 cores** each
6. **2 iterations**
7. **800 par** / **72 cores** / **2 iterations** = **6 VMs**

Cluster Configurations Scenarios

Getting Started...

Taking into consideration everything we know now...

- Who will be using the cluster?
- What will the cluster be used for?
- Where will the cluster and/or data reside?
- When are the results needed?
- How do I control/predict the costs?

Can we predict, for a given scenario, which cluster configuration and set of features will best meet the needs of each specific scenario?

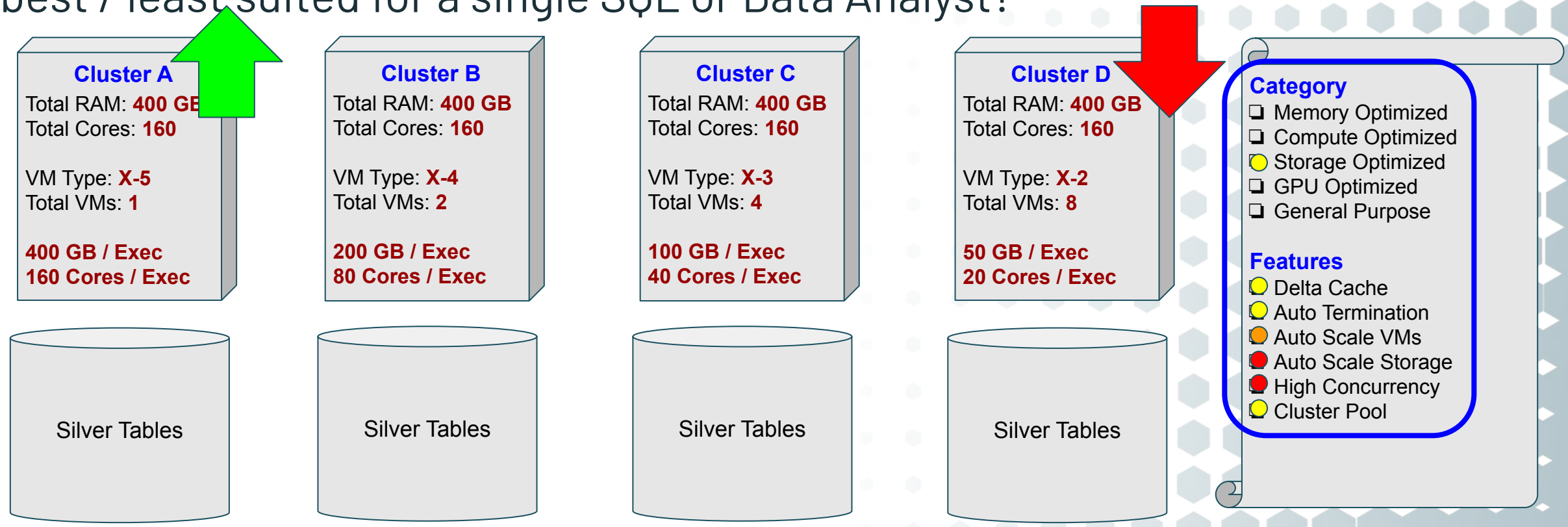
“It Depends”

Really… it does depend…

- There is rarely a black or white, right or wrong, answer
- There are many different factors that could justify various decisions
- The conclusions presented here are generalizations only

Typical Analyst

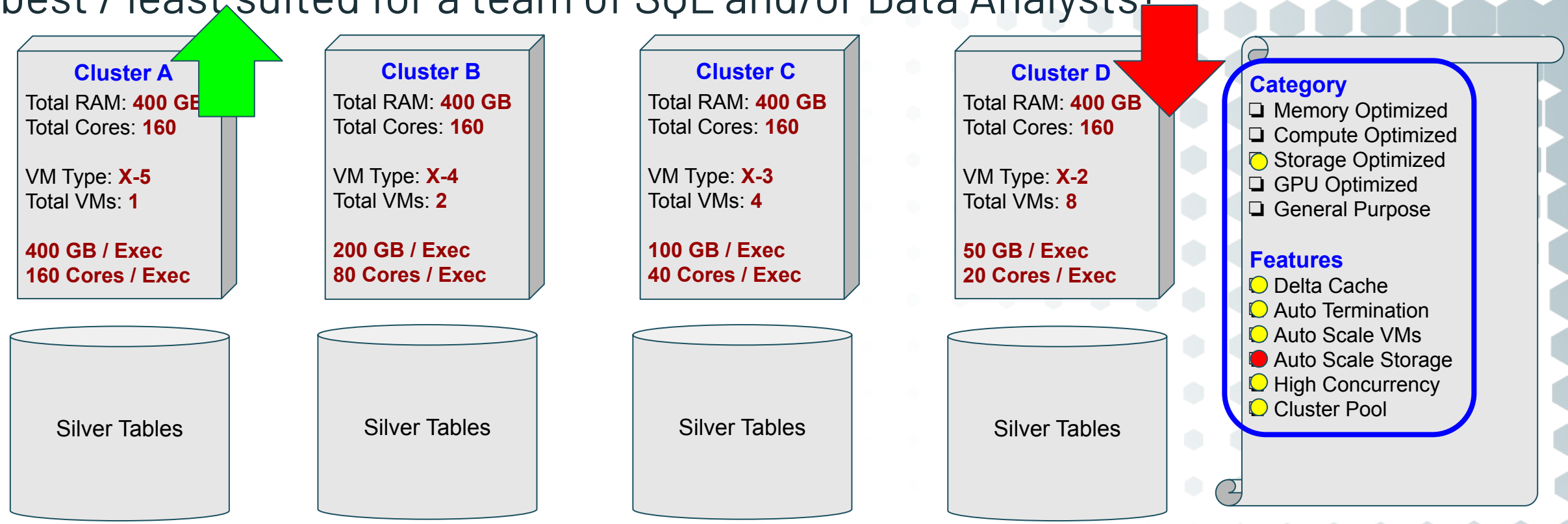
Which of the following cluster configurations is best / least suited for a single SQL or Data Analyst?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Team of Analyst

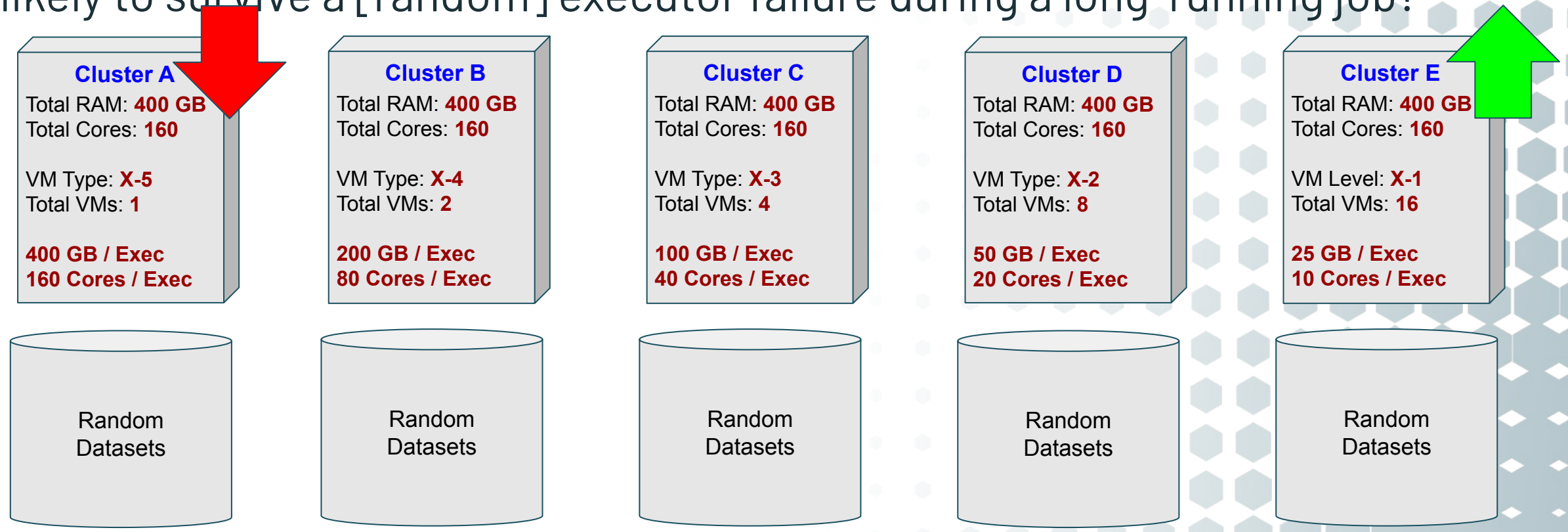
Which of the following cluster configurations is best / least suited for a team of SQL and/or Data Analysts?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Stability

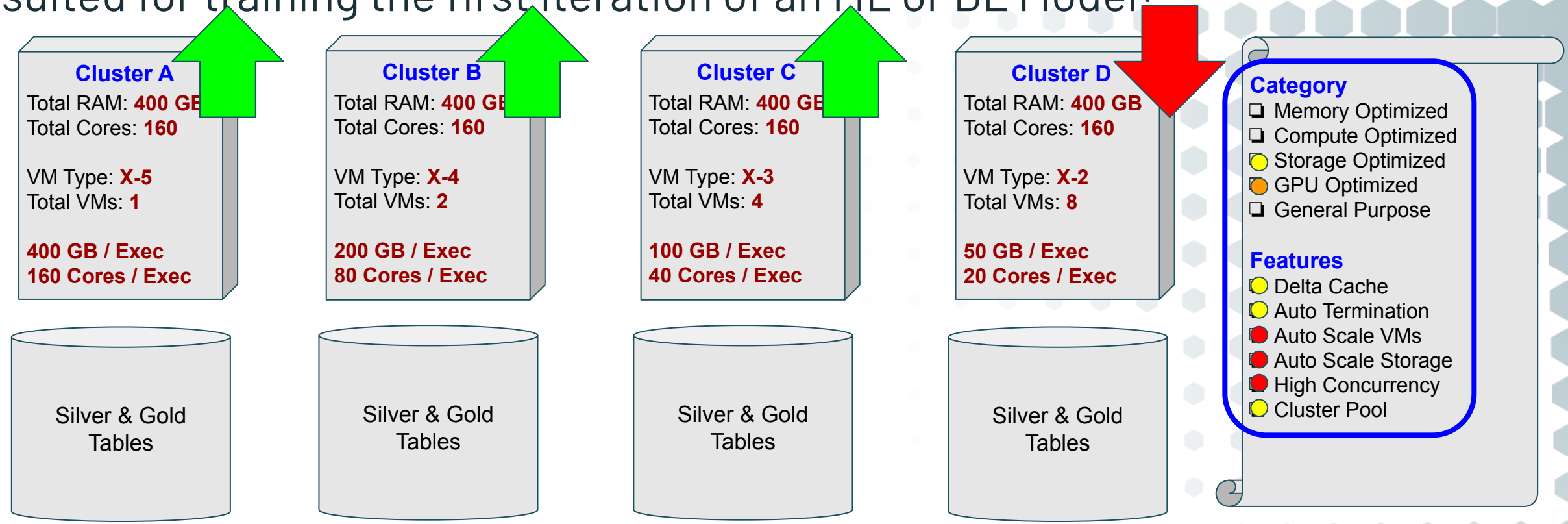
Which of the following cluster configurations is most / least likely to survive a [random] executor failure during a long-running job?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Training ML Models, 1st Iteration

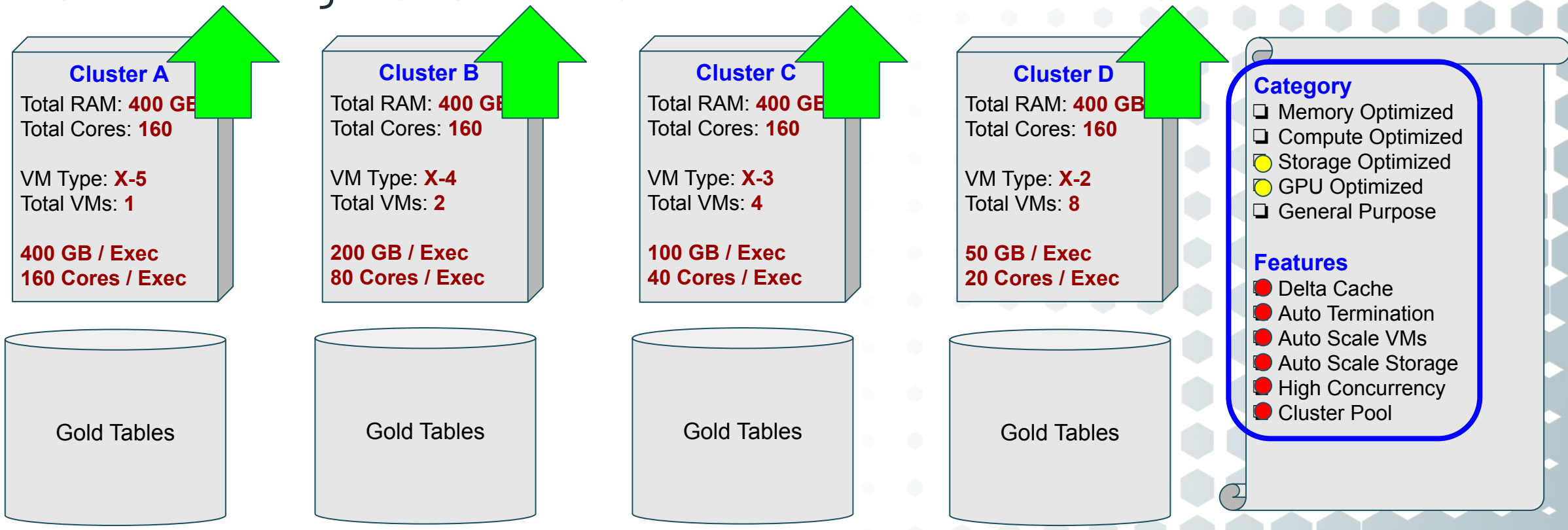
Which of the following cluster configurations is best / least suited for training the first iteration of an ML or DL Model?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Training ML Models, 2nd+ Iteration

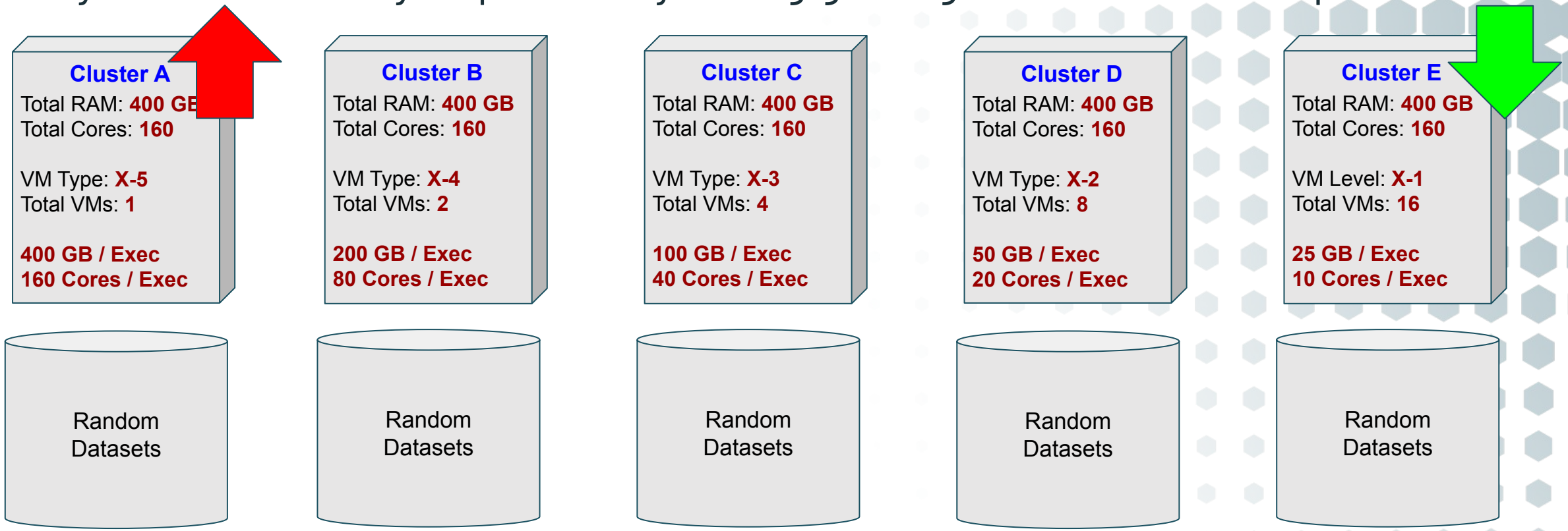
Which of the following cluster configurations is best / least suited for training the second iteration of an ML or DL Model?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Garbage Collection

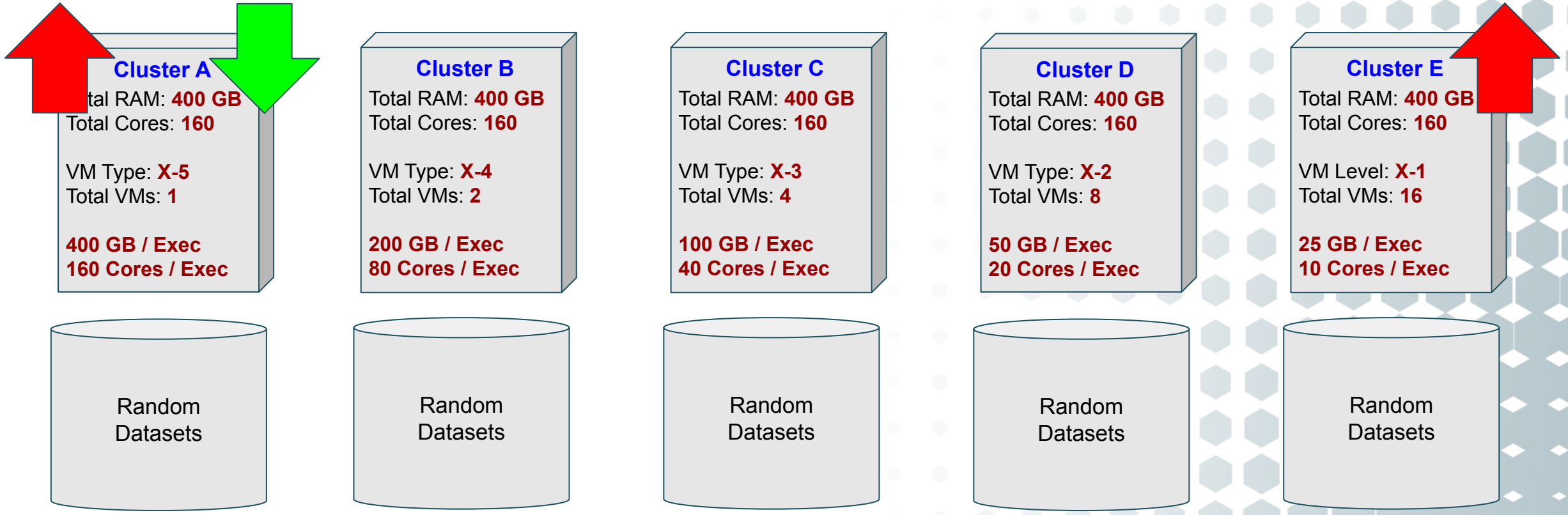
Which of the following cluster configurations is most / least likely to be adversely impacted by a long garbage collection sweep?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

General OOM Error

Which of the following cluster configurations is most / least likely to encounter an OOM Error?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Caching Induced OOM Error

Which of the following usage cases is most / least likely to induce a OOM Error induced by caching?

Heavy Caching

A data scientist that is training the first iteration of a model against a 1,000 GB dataset

Not Caching

An ETL Job that is consuming CSV data, updating data types, removing duplicates and then writing it out to Parquet

Here Too!

Excessive Caching

A team of 5 analyst engaged in heavy, ad hoc analysis against a single shared cluster

A report that joins three tables and writes the result to a Delta table used by BI tools

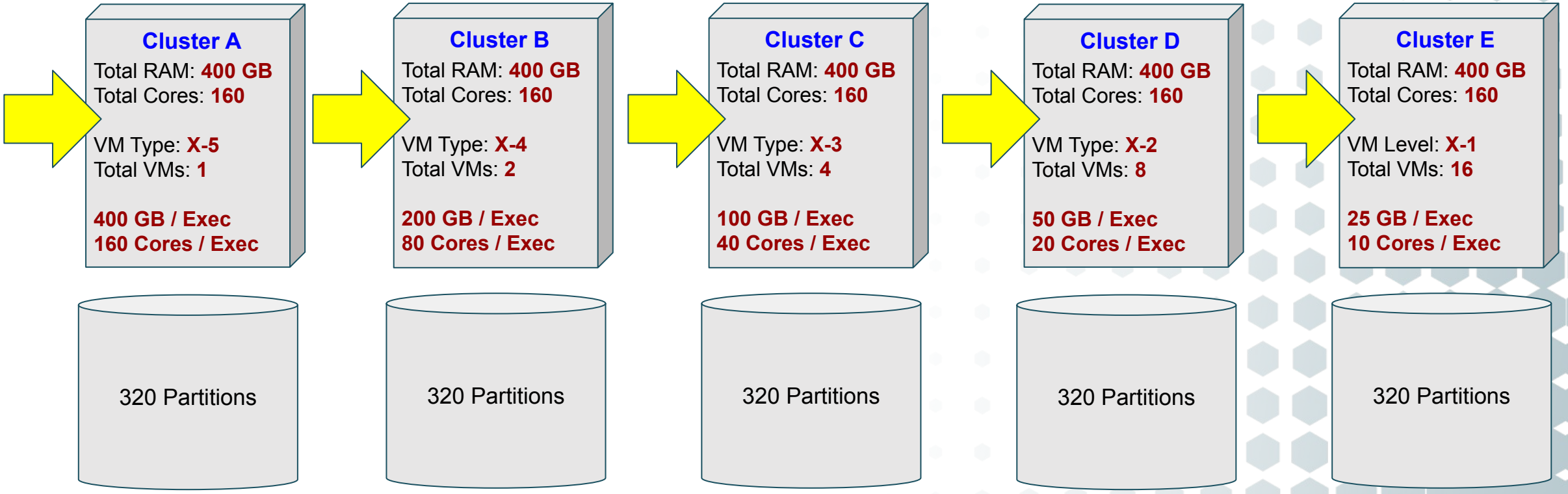
Not Caching

A single analyst attempting to validate sales-tax calculations for the previous year against a well formed 100 GB dataset

Light Caching

More Cores == More Money Version #1

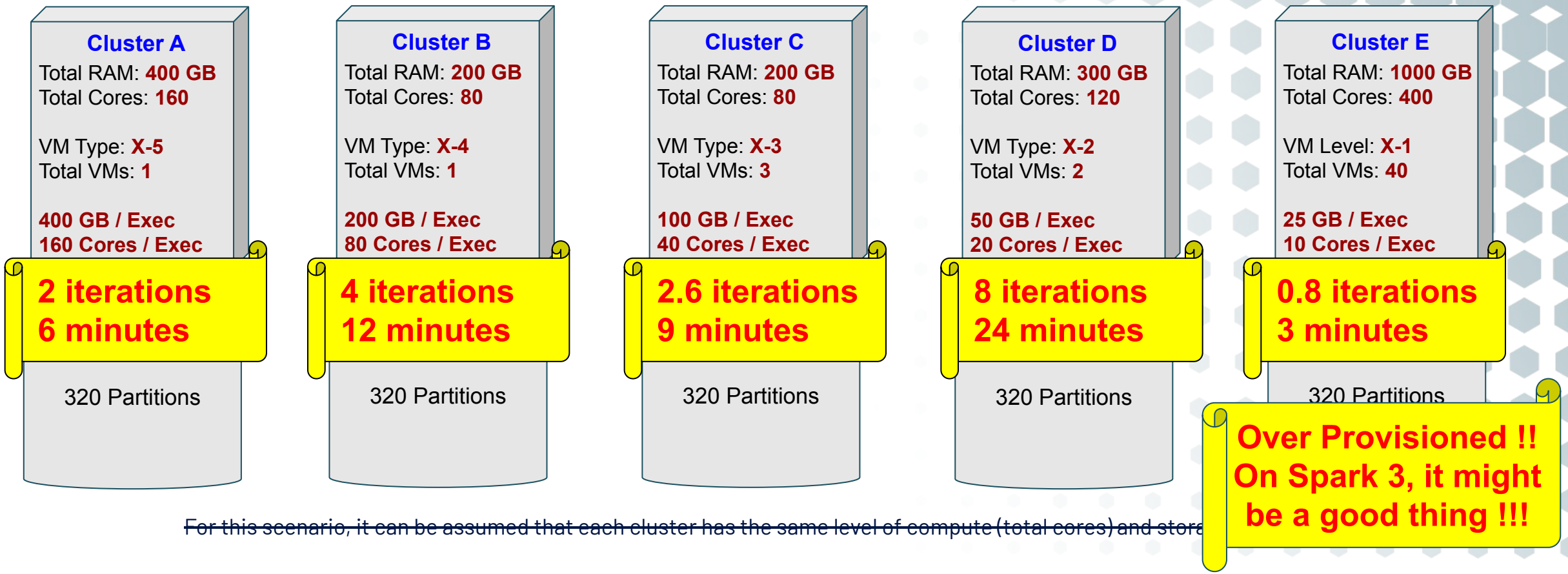
Assuming the data in 320 partitions is equally distributed, which cluster configuration will cost the most / least amount of money for this job?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

More Cores == More Money Version #2

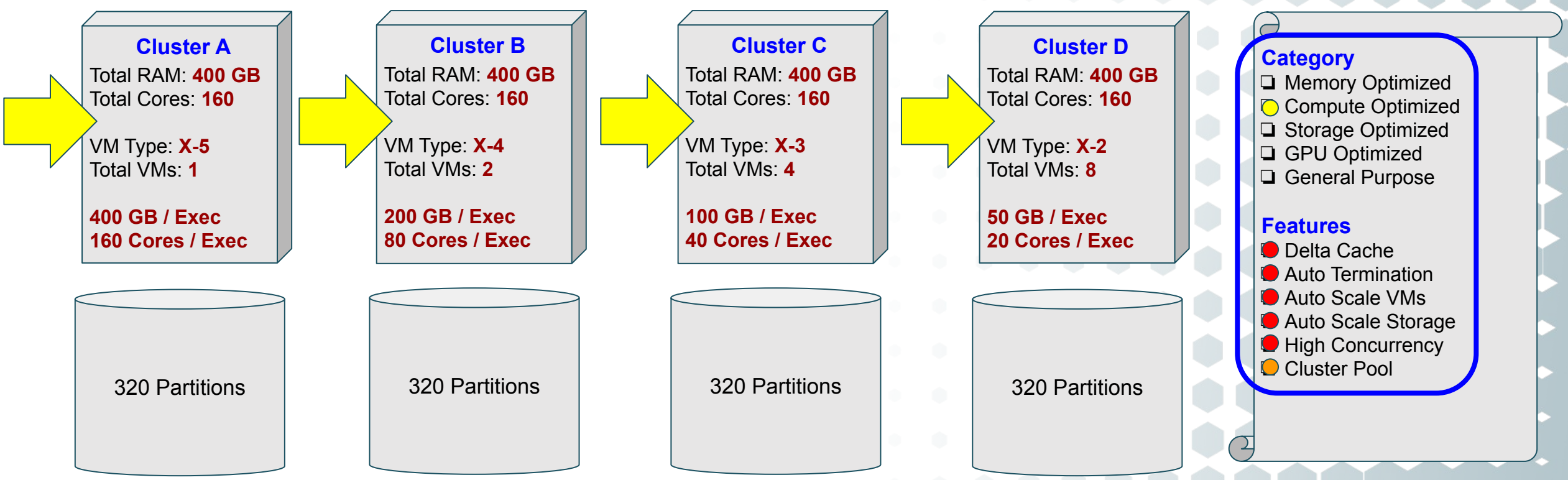
Assuming each partition takes 3 minutes to process... Calculate the **compute-time**, **number of iterations** and **run-time** for each scenario:



~~For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and stora~~

Batch ETL: Raw -> Bronze

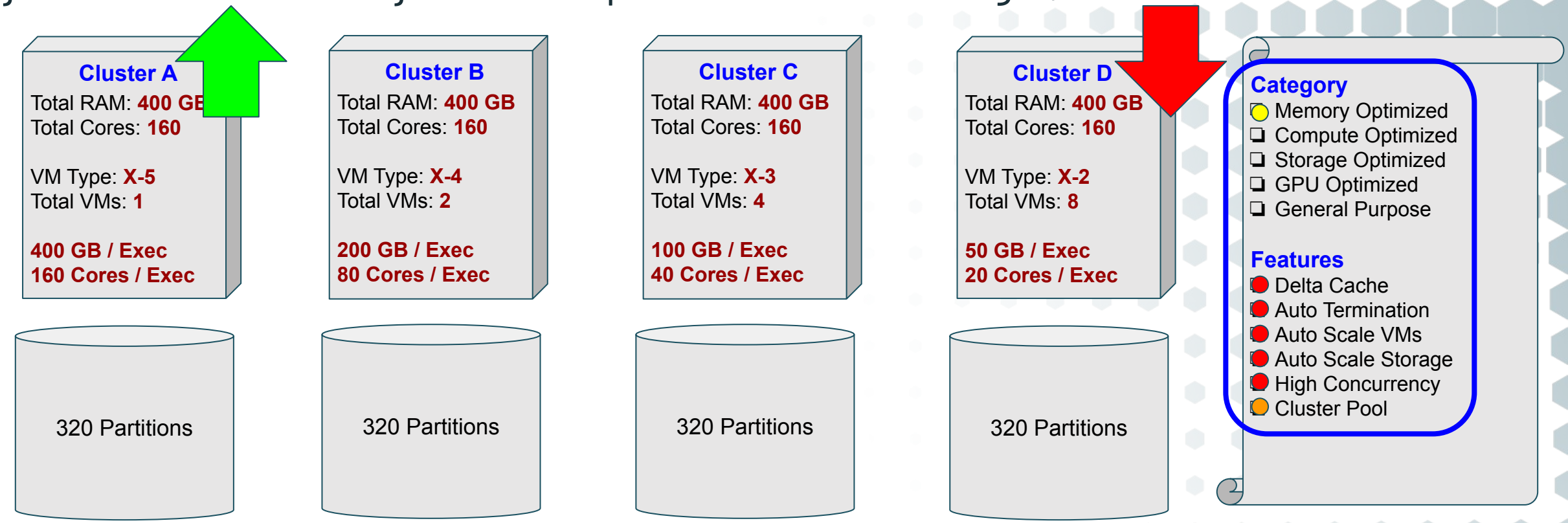
Which of the following cluster configurations is best / least suited for a simple ETL job that does not employ wide transformations (no joins)?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Batch ETL: Silver -> Gold

Which of the following cluster configurations is best / least suited for an ETL job that unions and joins multiple tables into a single, new table?



For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

databricks