AWS Glue Optimization

Reduce cost and improve performance of your AWS Glue workloads

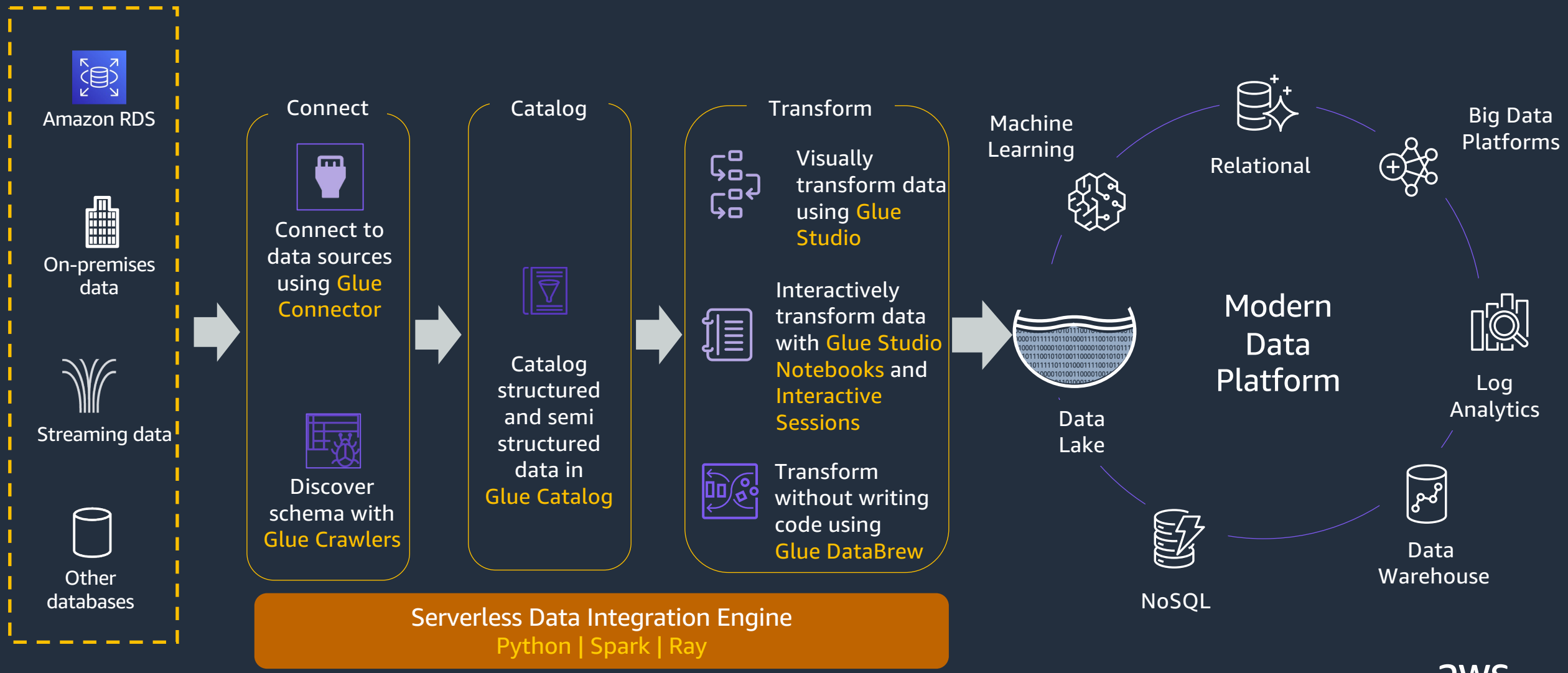
Harsh Vardhan

Sr. Analytics Solutions Architect

Agenda

1. AWS Glue Overview
2. Best Practices: Performance & Cost Optimization
    a. Glue Job Execution Parameters
    b. Glue Job Performance Tuning

AWS Glue Overview



Data integration engines options

AWS Glue for
Apache Spark

~5 sec startup

Performance-optimized Spark

Spark 3.3.0

Native data lake frameworks, Hudi, Iceberg, Delta Lake

AWS Glue for
Ray

Serverless Ray.io

Scale existing Python code on large datasets

Use familiar Python libraries

AWS Glue for
Python Shell

2x faster start times

Dozens of preloaded libraries

Python 3.10

More demanding workloads

Batch

Data warehousing

Business intelligence

Reporting

Nightly or hourly ETL
(extract, transform, and load)
Latency-agnostic long-running jobs

Real time

Responsive dashboards

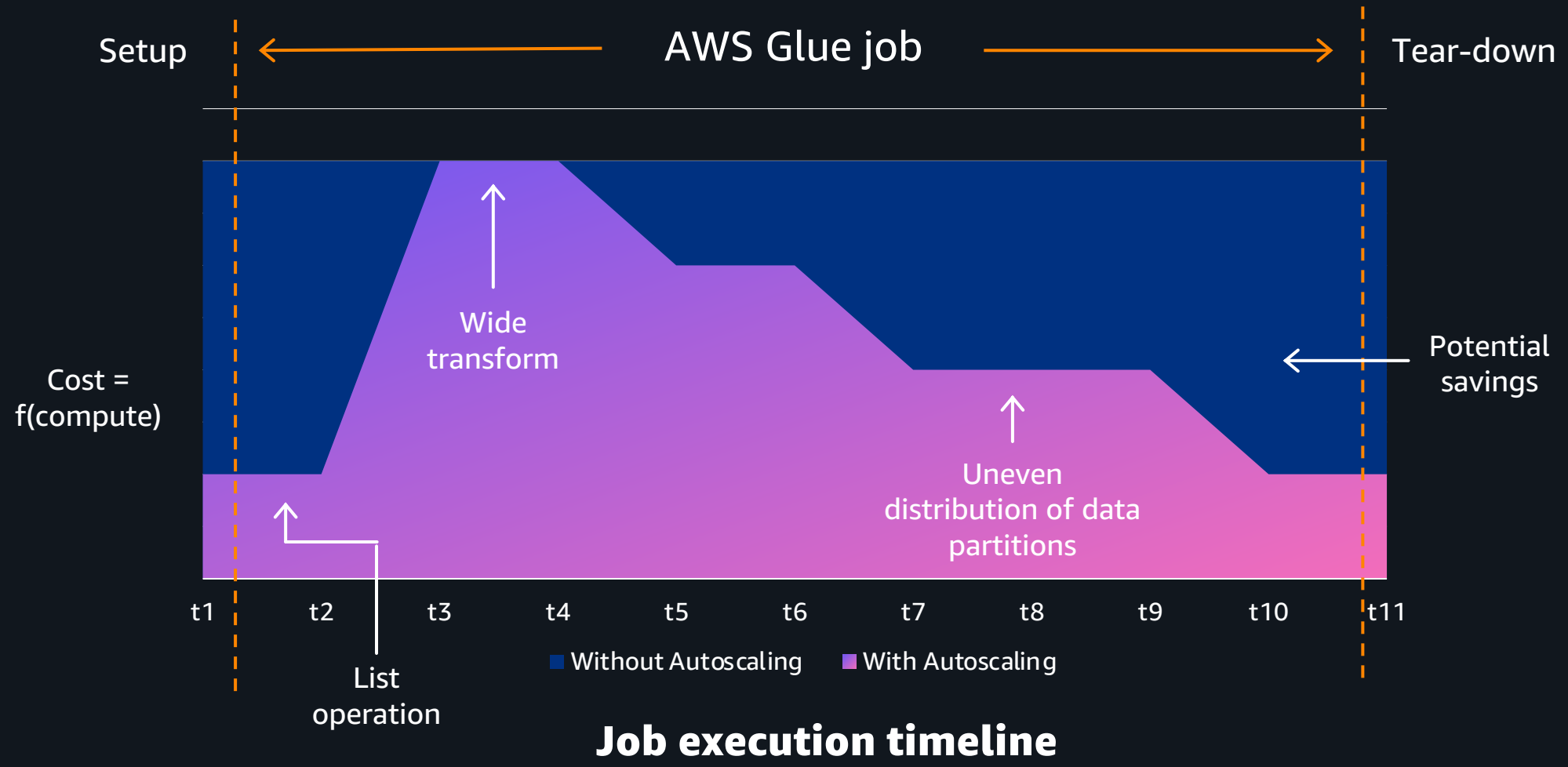
Monitor and alert

Streaming apps

Continual
operation
Latency-sensitive micro-batch jobs

Optimization: Glue Job Execution Parameters

AWS Glue Autoscaling



Glue Flex

New execution option for AWS Glue that allows customers to reduce the costs by up to 35%

Standard
execution-class

10x faster job start times
Predictable job latencies

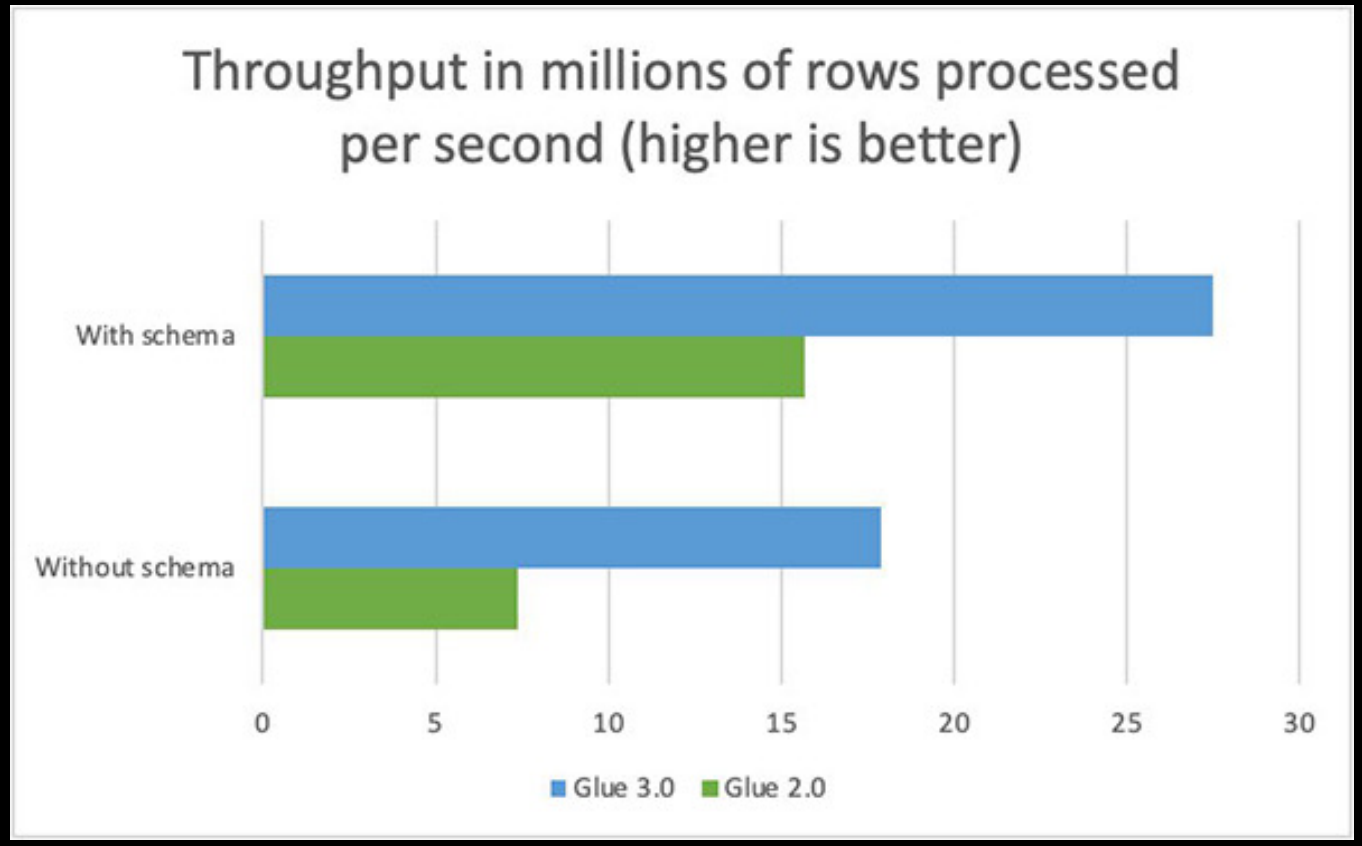
Enables micro-batching
Latency-sensitive workloads

Flex
execution-class

Up to
35% cost savings

Cost effective for non-time sensitive workloads

Use new AWS Glue version: 2.0 vs 3.0



Benchmarking Blog

- Performance-optimized Spark runtime based on open-source Apache Spark 3.1.1
- Faster read and write access - reading csv and writing Apache Parquet
- Faster and efficient partition pruning on highly partitioned tables managed AWS Glue Data Catalog
- Spark 3.1.1 enables an improved Spark UI experience

Use new AWS Glue version: 3.0 vs 4.0

Upgrades AWS Glue engines

Apache Spark 3.3.0

Python 3.10

Scala 2.1

Native support for Hudi, Delta and Iceberg

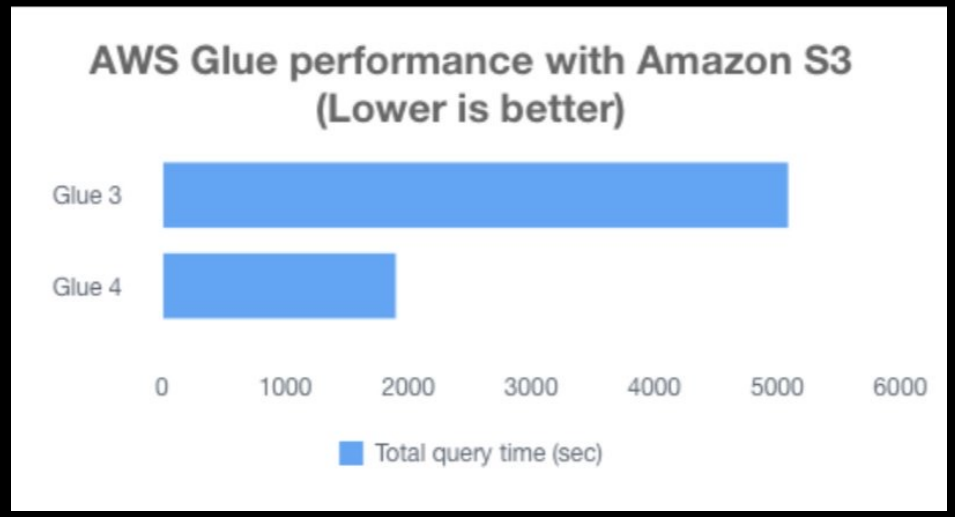
Apache Hudi

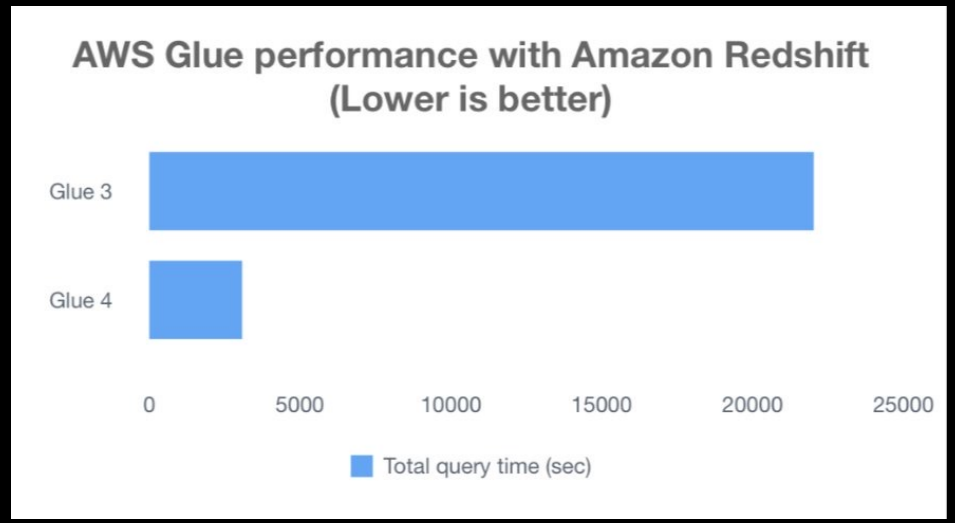
Delta Lake

Apache Iceberg

Adds more options for scaling, storing, and running your jobs

- Performance-optimized Spark 3.3
- Distributed Pandas API on Spark–improved data processing
- Amazon Redshift integration Apache Spark–10x faster in TPC-DS at 3TB scale





[Benchmarking Blog]

Choosing right Glue Worker Type

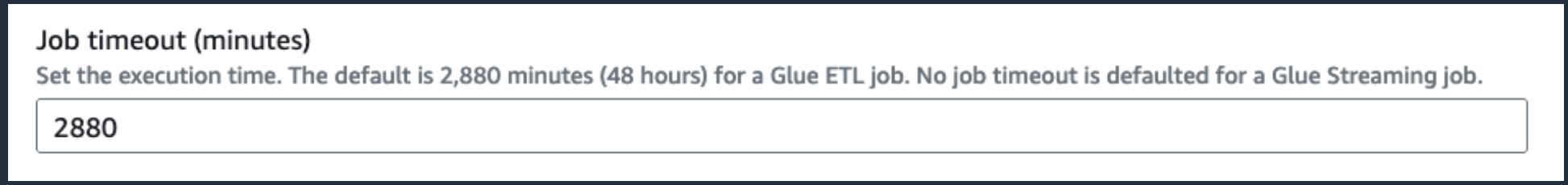
- The processing power allocated at the time of job execution is called DPU (Data Processing Unit).
- 1DPU = 4vCPU, 16GB memory
- Each Worker Type has different resource capacity and configuration.

AWS Glue Worker Type	DPU per Node	vCPU	Memory (GB)	Disk (GB)	Number of Spark Executors per Node	Number of Cores per Spark Executor
G. 025X*	0.25	2	4	16	1	2
G 1X	1	4	16	64	1	4
G 2X	2	8	32	128	1	8
G 4X (new)	4	16	64	256	1	16
G 8X (new)	8	32	128	512	1	32

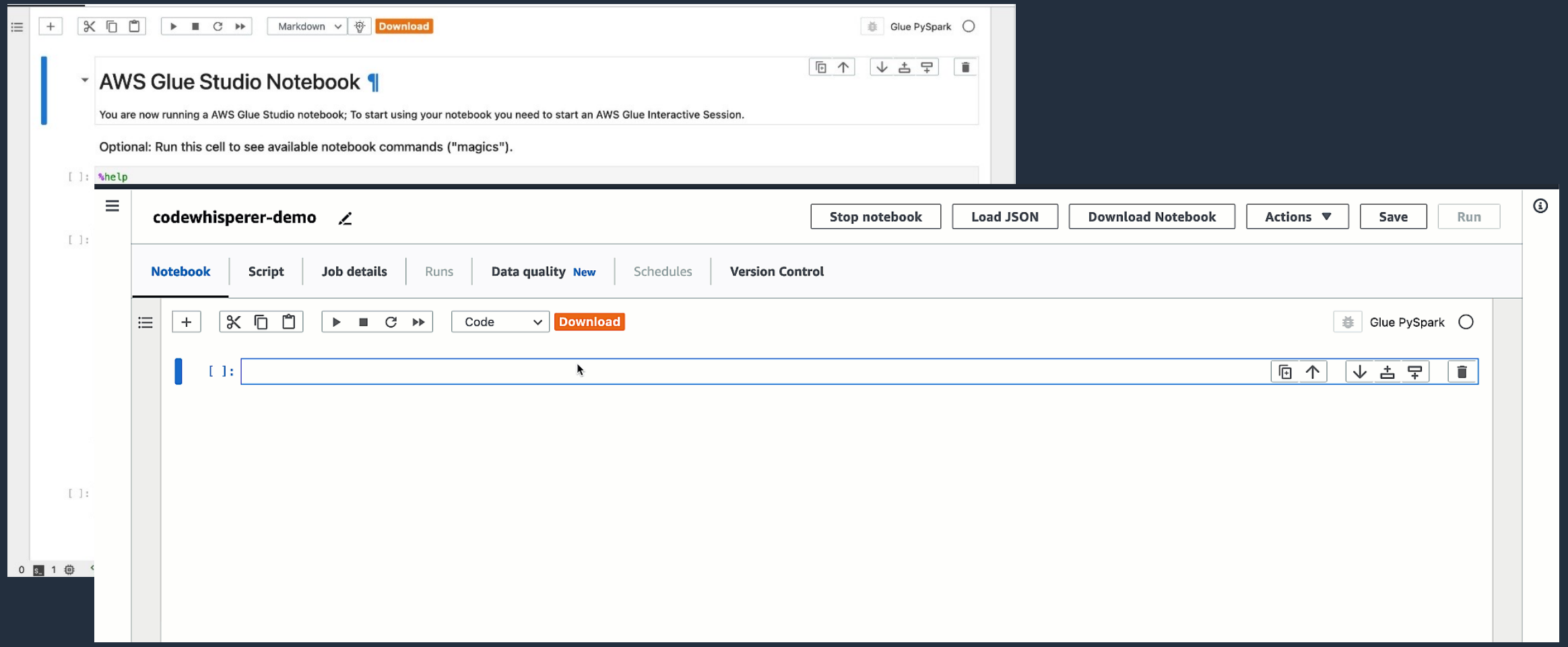
* Glue Streaming Job

Job Timeout

- By default, 2880 minutes (48 hours).
- Due to configuration issues, script coding errors, or data anomalies could result in unexpected charges
- Identify average execution time of your job, adjust accordingly.



AWS Glue Interactive Sessions



Optimization: Job Performance Tuning

Basic strategy for tuning AWS Glue ETL jobs

- Scale cluster capacity
- Reduce the amount of data scan
- Parallelize tasks
- Optimize shuffles
- Overcome data skew

Choosing right Glue Worker Type

- The processing power allocated at the time of job execution is called DPU (Data Processing Unit).
- 1DPU = 4vCPU, 16GB memory
- Each Worker Type has different resource capacity and configuration.

AWS Glue Worker Type	DPU per Node	vCPU	Memory (GB)	Disk (GB)	Number of Spark Executors per Node	Number of Cores per Spark Executor
G. 025X*	0.25	2	4	16	1	2
G 1X	1	4	16	64	1	4
G 2X	2	8	32	128	1	8
G 4X (new)	4	16	64	256	1	16
G 8X (new)	8	32	128	512	1	32

* Glue Streaming Job

How to minimize the data I/O load

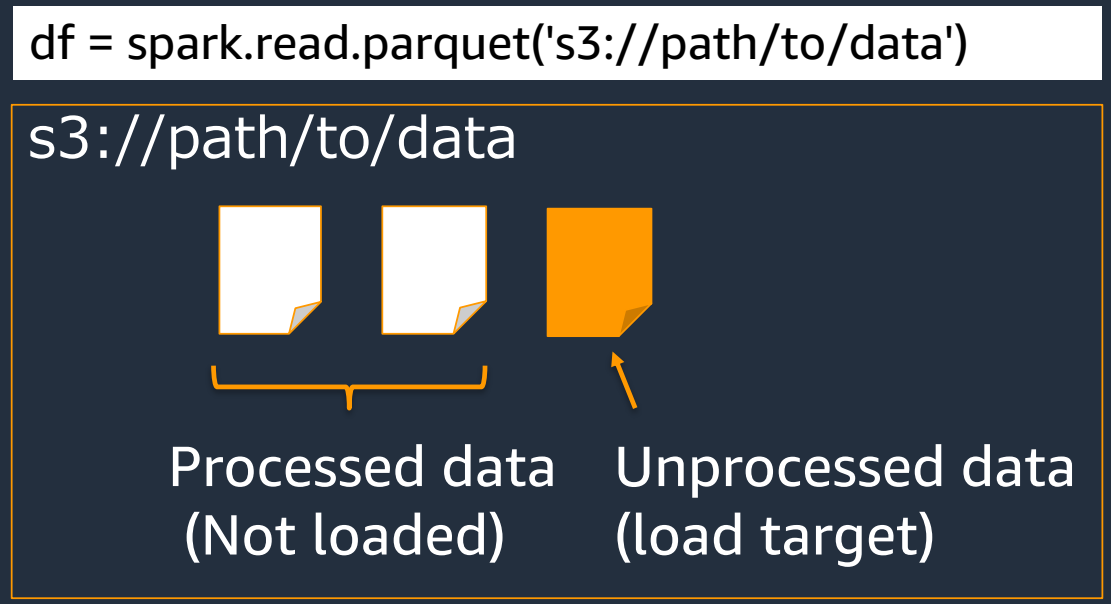
- **Read only the data you need.**
  - ✓ Reduce unnecessary data I/O by selectively reading required data.
  - ✓ Utilize projections or filters to limit the amount of data accessed during queries.

- **Control the amount of data read in one task.**
  - ✓ Manage data read in each task to avoid overwhelming system resources.
  - ✓ Set appropriate batch sizes or row limits for data retrieval operations.

- **Choose the right file size, file format and compression algorithm.**
  - ✓ Reduce data size and I/O load by employing efficient compression techniques.
  - ✓ Select compression formats (e.g., gzip, Snappy) based on data characteristics and workload requirements.

Job Bookmarks

- Glue keeps track of data that has already been processed by a previous run of an ETL job. This persisted state information is called a bookmark.
- Function to process only the delta data when performing steady-state ETL
- Use file timestamps to process only the data that has not been processed in the previous job to prevent duplicate processing and duplicate data.



Partition Index for partitioned tables

Improve query performance for highly partitioned data

For highly partitioned data query performance can degrade

Partition Indexes allow retrieval of relevant partitions quickly

Improves query performance and reduces data transfer

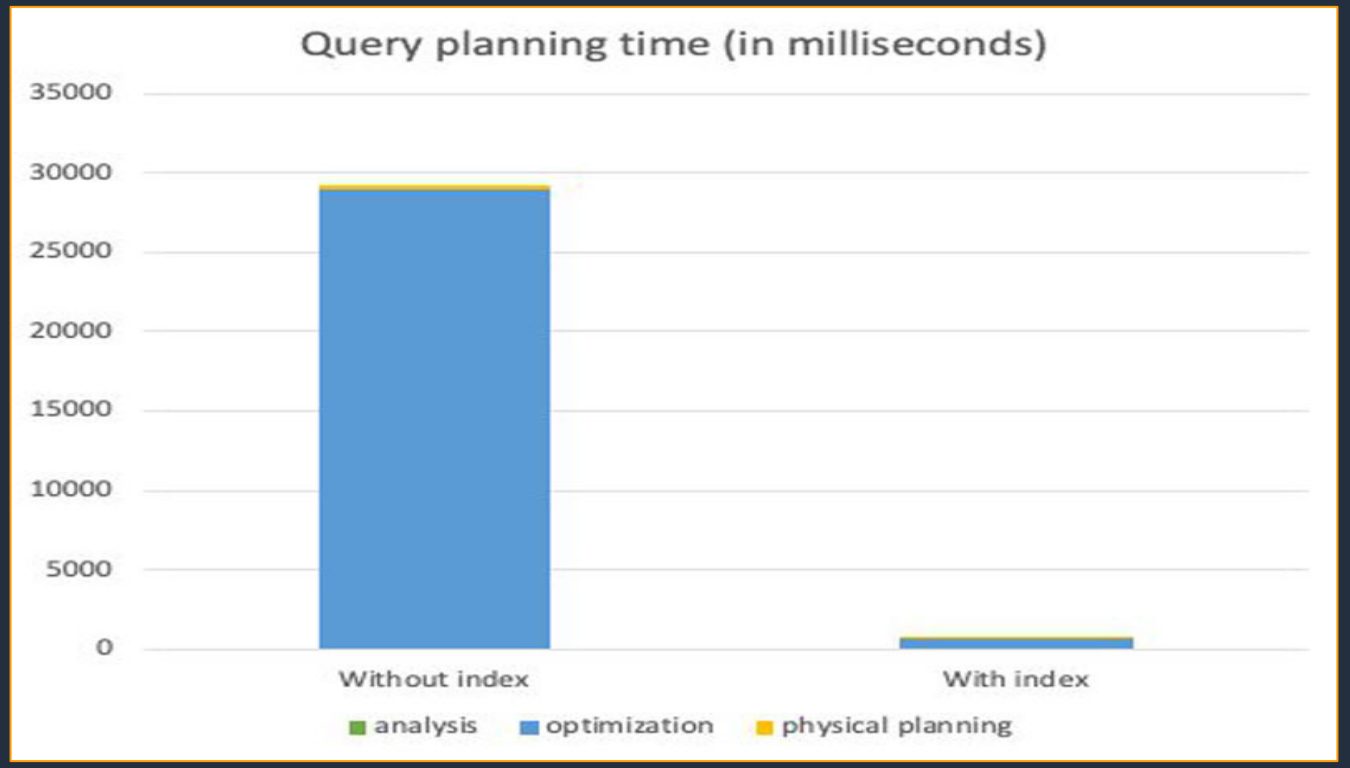
Enables interactive exploration of very large data sets

Powered by Glue Data Catalog

aws

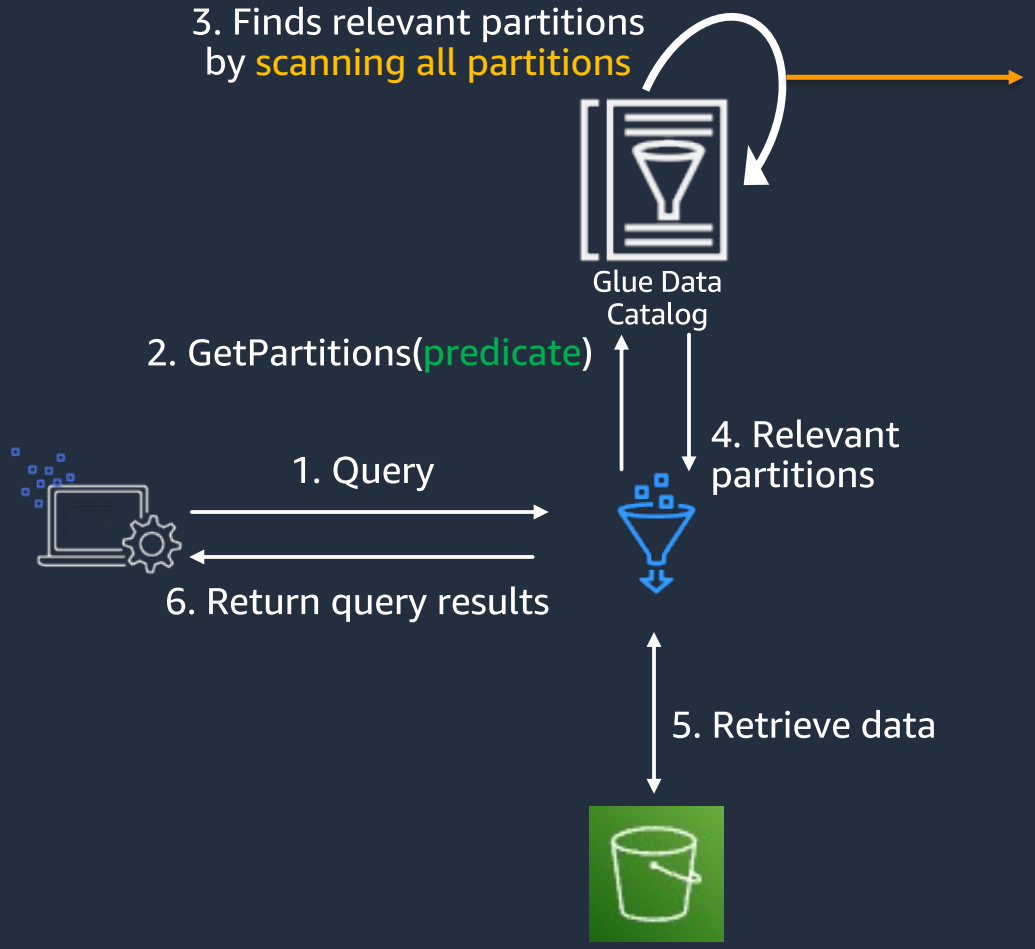
With Partition Index vs Without Partition Index

- The difference of query planning time between using Partition Index and not using Partition Index



New with Glue 3.0: Catalog partition predicates

Server-side partition pruning combined with push-down predicate reduces the time and cost of data integration jobs



Push-down predicate

```
dynamic_frame = glueContext.create_dynamic_frame.from_catalog(
database=dbname, table_name=tablename,
transformation_ctx="datasource0",
push_down_predicate = "year=='2017' and month=='04'"
)
```

Server-side partition pruning

```
dynamic_frame = glueContext.create_dynamic_frame.from_catalog(
database=dbname, table_name=tablename,
transformation_ctx="datasource0",
additional_options={"catalogPartitionPredicate":"year='2017' and
month='04'"}
)
```

JDBC Connections: Parallel data reading in Spark DataFrame

- spark.read.jdbc() only allows one Executor to access the target database by default.
- For parallel reading, partitionColumn, lowBound, upperBound, and numPartitions must be specified.
- The partitionColumn must be one of the following types: numeric, date, or timestamp.

```
df = spark.read.jdbc(
    url=jdbcUrl, table="sample",
    partitionColumn="col1",
    lowerBound=1L,
    upperBound=100000L,
    numPartitions=100,
    fetchsize=1000,
    connectionProperties=connectionProperties)
```

JDBC Connections: Parallel data reading in Glue DynamicFrame

- Specify hashfield/hashexpression If you want to read data from a JDBC connection as a DynamicFrame.
- hashfield - strings and other columns can also be used as partition columns (“month”, “customer_name”)
- hashexpression - SQL expression that returns a whole number. (“customer_id”, “store_id”)
- hashpartitions - number of parallel reads from the JDBC table. Default is 7.

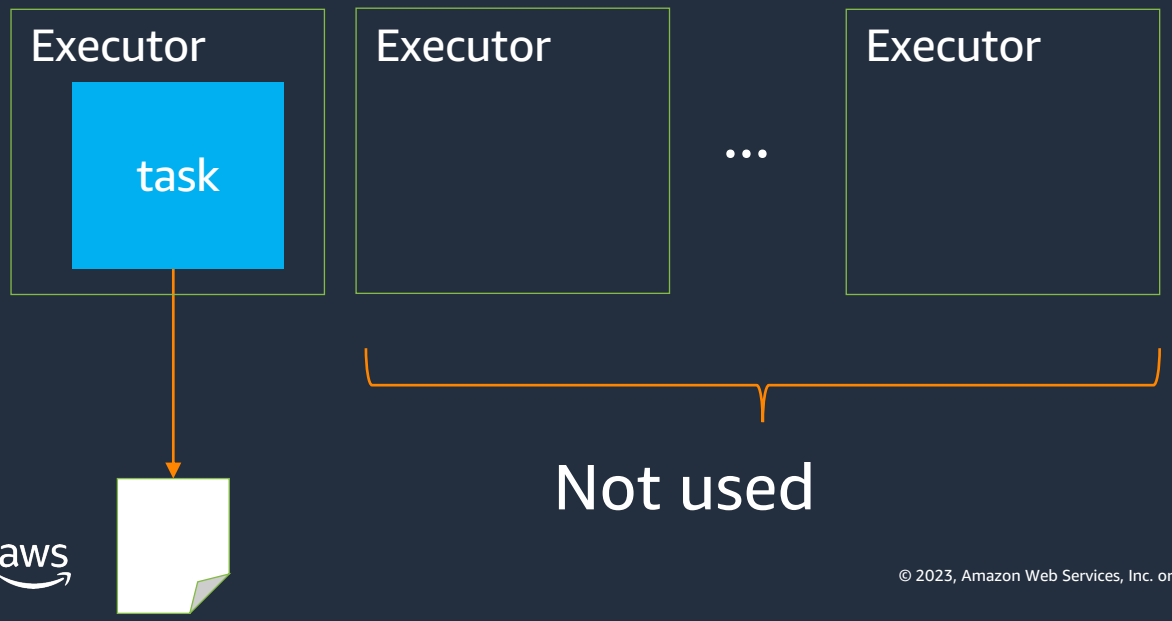
```
glueContext.create_dynamic_frame.from_catalog(
      database="mysql_jdbc_db”,
      table_name=”my_table”,
      transformation_ctx=”my_ transformation_ctx”,
      additional_options = {"hashfield":"month", "hashpartitions":10}
)
```

https://docs.aws.amazon.com/glue/latest/dg/run-jdbc-parallel-read-job.html

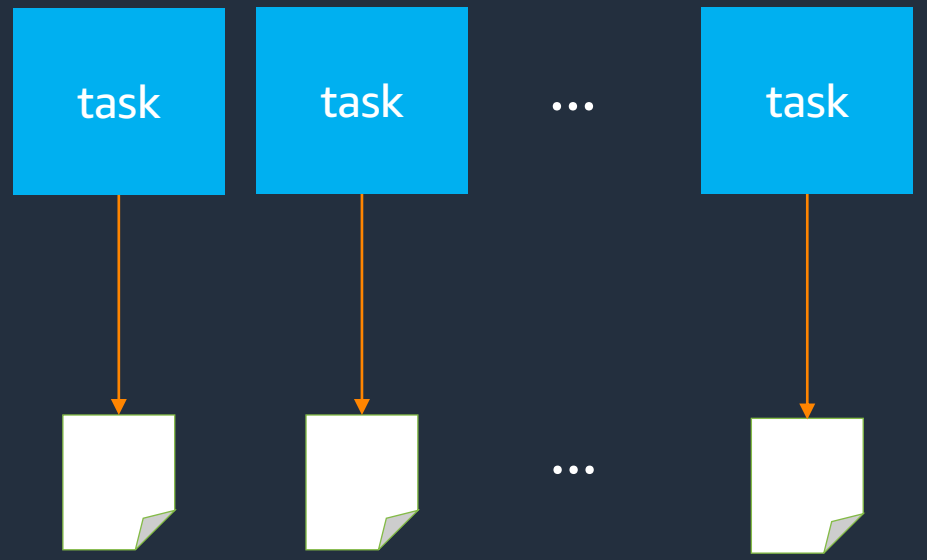
Store data in appropriate file sizes.

- Data read/write tasks are basically tied to a single file. (If the file is splitable, one file can be split into multiple tasks.)
- The recommended file size for AWS Glue is 128MB-512MB.

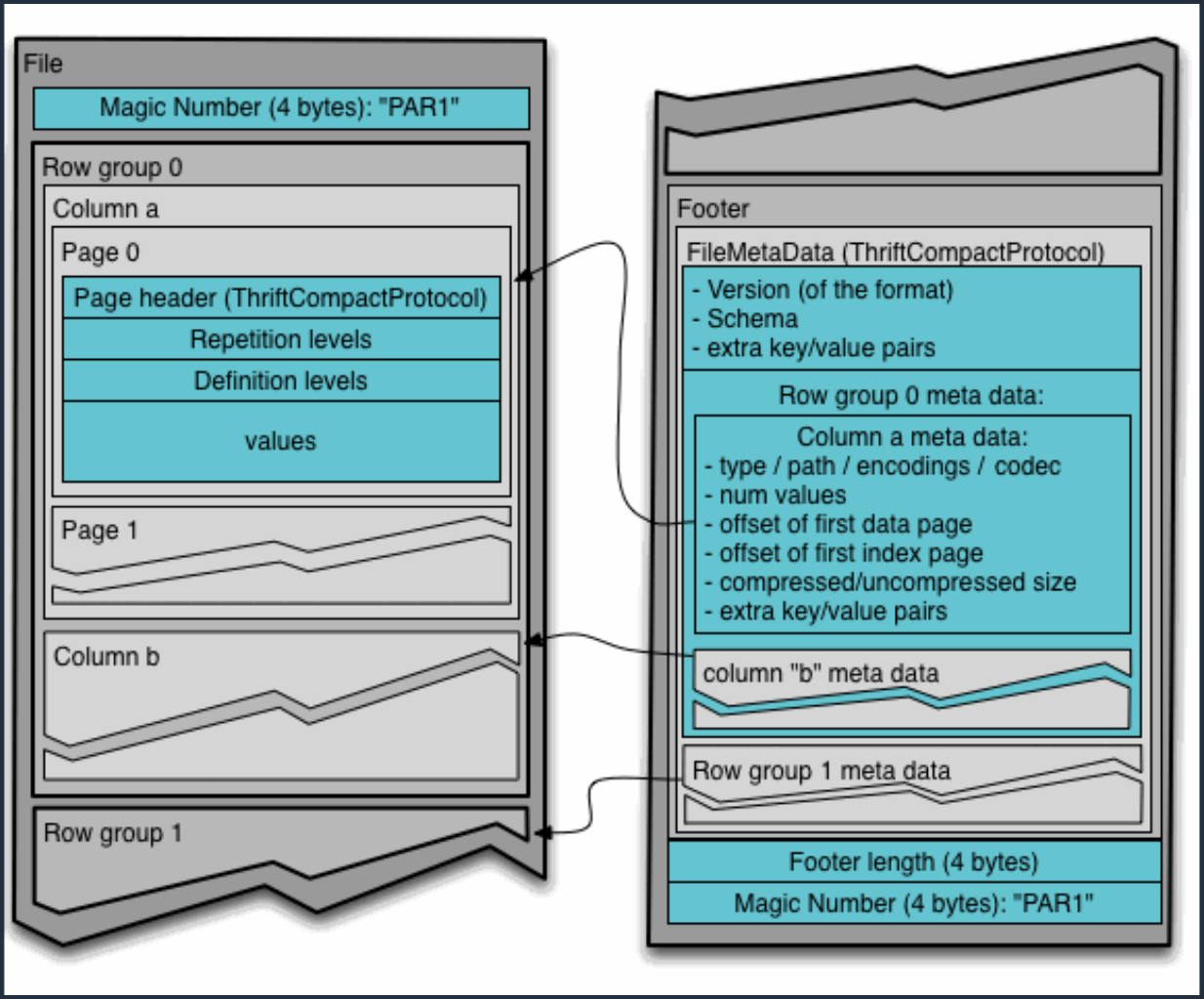
When there is large unspilitable data in one file
- Data is not fully loaded into memory on a single node.
- No distributed processing



When the data is too small
- Overhead due to large number of small tasks



Using Apache Parquet



https://parquet.apache.org/docs/file-format/

- Column-oriented format for data arrangement suitable for analytical applications
- Data type is preserved
- Compressed effectively
- Aggregation by skipping unnecessary data and using metadata
- The Spark engine can efficiently use Apache Parquet
  Integration is in place

Choose a compression codec based on your application

- Compression codec can be selected at data writing.
- Trade-off between compression rate and compression/decompression speed
- BZIP2 and LZO compressed files can be split and processed when read. GZIP files cannot!
- No compression/decompression time for uncompressed files but data transfer cost may be a bottleneck
- If processing speed is important to you, choose snappy or lzo.

Algorithm	Splittable?	Compression ratio	Compress / Decompress speed
Gzip (DEFLATE)	No	High	Medium
Bzip2	Yes	Very high	Slow
LZO	Yes	Low	Fast
Snappy	No	Low	Very fast

Using JOIN hints

Sort Merge Join - */*+ MERGE(table) */*

- Distribute the two tables to be joined by their respective keys, sort them, and then join them.
- Suitable for joining large tables together.

Broadcast Join - */*+ BROADCAST(table) */*

- Transfer one table to all Executors, and distribute the other table to all Executors and join them.
- Suitable for when one table is smaller than the other.

Shuffle Hash Join - */*+ SHUFFLE_HASH(table) */*

- Distribute the two tables to be joined and join them without sorting.
- Suitable for joins between tables that are not so large.

Broadcast Join

- By default, if the table size is less than or equal to the value specified in spark.sql.autoBroadcastJoinThreshold (default 10MB), Broadcast Join will be used.
- The Join strategy in use can be seen in the Spark UI or by using explain().
- Utilize join hints for broadcast join - `/*+ BROADCAST(table) */`

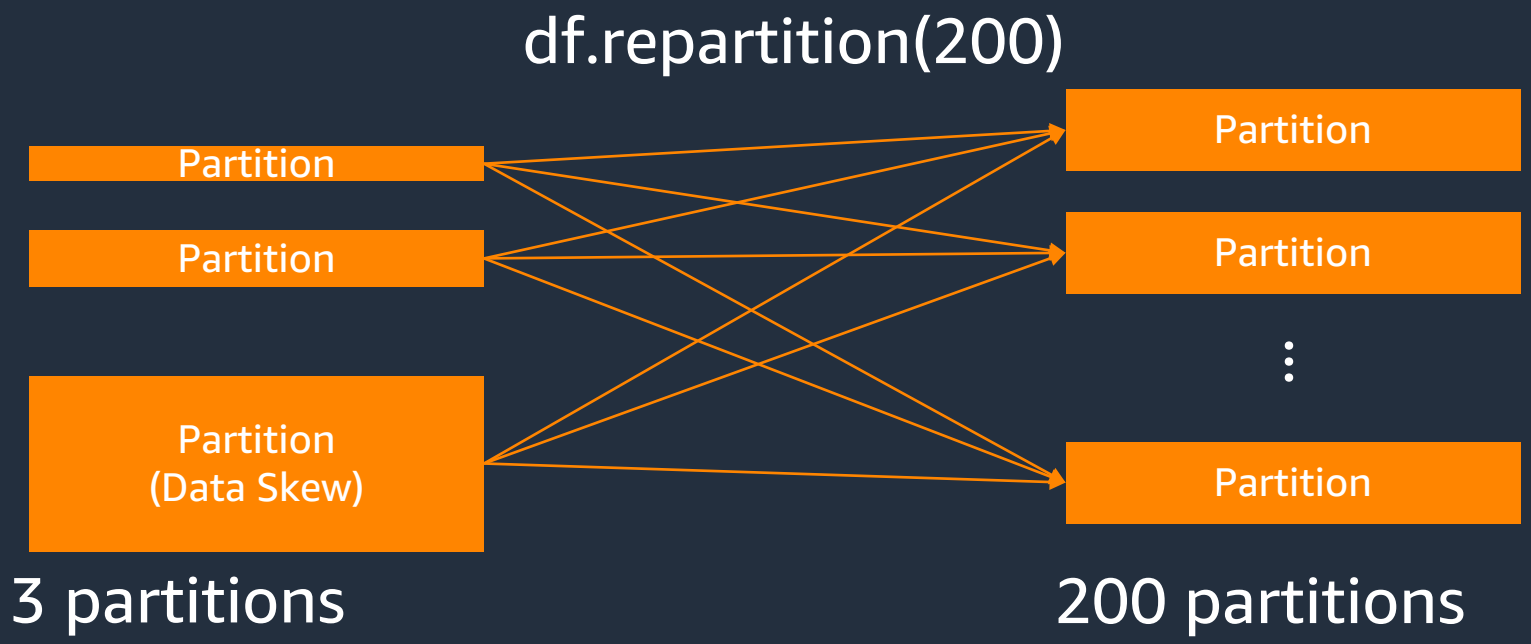
```
df1.join( broadcast(df2), df1("col1") <=> df2("col1") ).expand()
```

```
== Physical Plan == BroadcastHashJoin [coalesce(col1#6, )], [coalesce(col1#21, )], Inner, BuildRight, (col1#6 <=> col1#21)
 :- LocalTableScan [first_name#5, col1#6].
+- BroadcastExchangeHashedRelationBroadcastMode(List(coalesce(input[0, string, true], ))) +- LocalTableScan [col1#21, col2#22, population#23]
```

https://spark.apache.org/docs/3.3.0/sql-ref-syntax-qry-select-hints.html

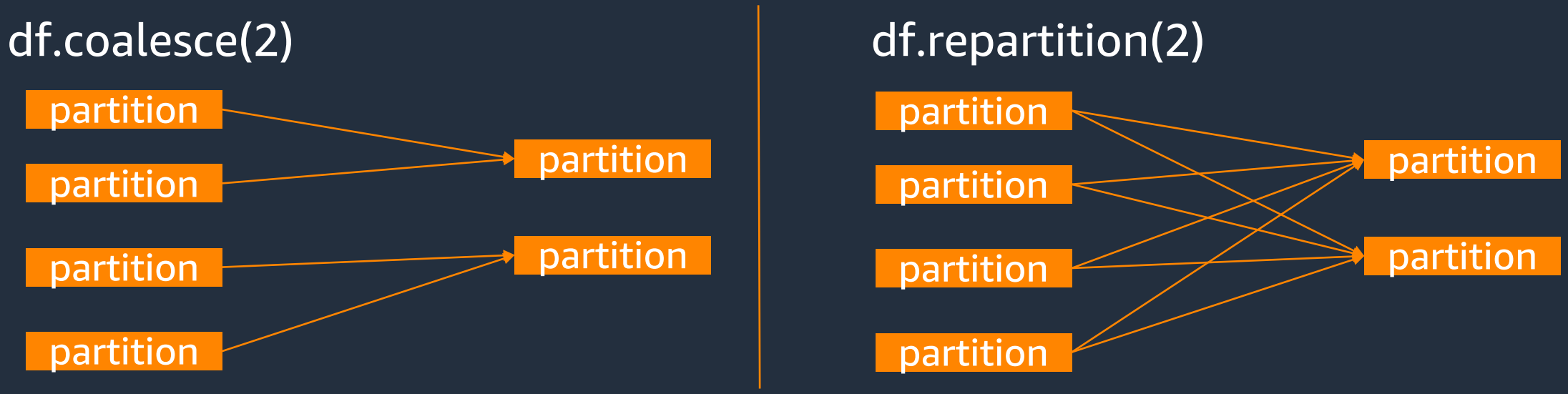
Dealing with Skewness - Repartition()

- If the subsequent process is not a key-by-key process (partitioning and storing data by date, window processing by key, etc.), repartition will resolve the skew.
- repartition() induces data shuffle to re-distribute data
- It can be used to decrease/increase partitions



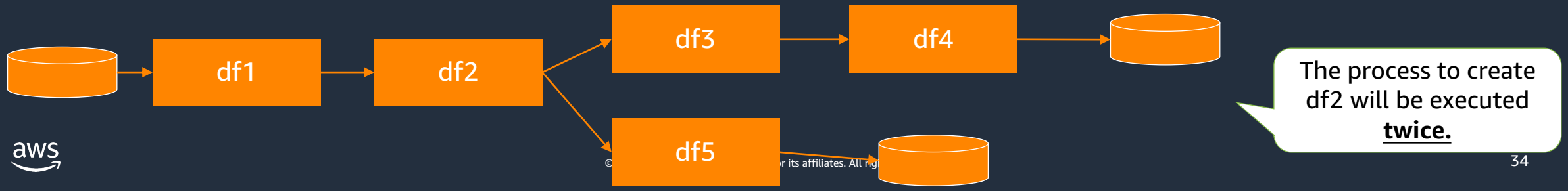
Repartition() and Coalesce()

- Partitions may be split into smaller pieces during processing due to load a large number of small files or when performing groupBy on columns with high cardinality
- In such a case, it is better to merge the partitions before the next process to reduce the overhead of the subsequent process.
- Since repartition involves shuffling, it may be desirable to use coalesce. However, since it is a simple merge, the data after coalesce may be biased.



Make good use of cache

- When branching the processing of a single Dataframe to perform multiple outputs, you can prevent recalculation by inserting cache() just before the branch.
- Note that it may be faster not to use cache, and that too much use of cache will use local disk space.
- A cached Dataframe will continue to occupy memory and local disk space. Save memory and disk space by deleting the Dataframe cache when it is no longer needed - `df.unpersist()`



AWS Glue: Best Practices

- Use Glue execution parameters - latest version, auto-scaling, flex and job timeout
- Use the appropriate type of scaling
    - Horizontal scaling – Data parallelism, splitable datasets, file size and format
    - Vertical Scaling/Right Worker – Memory Intensive
- Use partitioning to query exactly what you need
    - Push down predicate
    - Glue Partition indexes for highly partitioned tables
- Use Job Bookmarks – Incremental processing
- Perform JOIN Optimization – Filter before JOIN, Broadcast, remove data skew
- Enable Job Metrics, Spark UI and real-time monitoring
- Optimize memory management
- Leverage AWS Glue interactive sessions for development work

Important Links

Best Practices

https://docs.aws.amazon.com/prescriptive-guidance/latest/serverless-etl-aws-glue/best-practices.html

AWS Blogs

1. Best practices to scale Apache Spark jobs and partition data with AWS Glue
2. Optimize memory management in AWS Glue
3. Improve query performance using AWS Glue partition indexes
4. Introducing Amazon S3 shuffle in AWS Glue

What's New in AWS Glue

Thank you!