databricks

Code Optimization

LECTURE

Spill

This lecture covers what spill is in Spark, its types and examples, and ways to mitigate it.

Spill

- Spill is the term used to refer to the act of moving data 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 is a term used to describe what happens when there is so much data that it can't all fit in RAM, so some of it needs to be pushed out to disk. If more data is needed later, it will have to be pulled back from disk into RAM. This situation occurs when the volume of data simply exceeds the available RAM that has been allocated to the cluster. As a result, the system must rely on disk reads and writes, which are much slower, to free up RAM and allow computations to continue. If this process doesn't happen, and the system runs out of memory, it can lead to an out of memory error and cause the entire job to fail

Spill - Examples

- Set **spark.sql.files.maxPartitionBytes** too high (default is 128 MB)
- 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
  The **groupBy()** where the column has low cardinality
- The **countDistinct()** and **size(collect_set())**
- Setting **spark.sql.shuffle.partitions** too low or wrong use of **repartition()**

Some common situations that can cause a spill include setting the max partition bytes too high, leading to oversized partitions that won't fit in RAM and must be written to disk. The default value is 128 MB, but increasing this can easily lead to large partitions for each worker. Another cause is exploding even a small array, which can quickly generate more data than the available RAM can handle. Performing joins—especially cross joins—between tables can also result in a huge number of new rows, potentially exceeding the memory limit. Issues can also arise from joins on skewed keys, causing certain partitions to become much larger than others. Group by operations on columns with low cardinality, using count distinct, collect set, or having shuffle partitions set too low can all trigger spills as well. Even the wrong use of repartition can lead to large memory demands. In summary, anything that increases the amount of data beyond what your cluster's RAM can support may result in a spill, leading to slower disk-based computation.

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 that 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

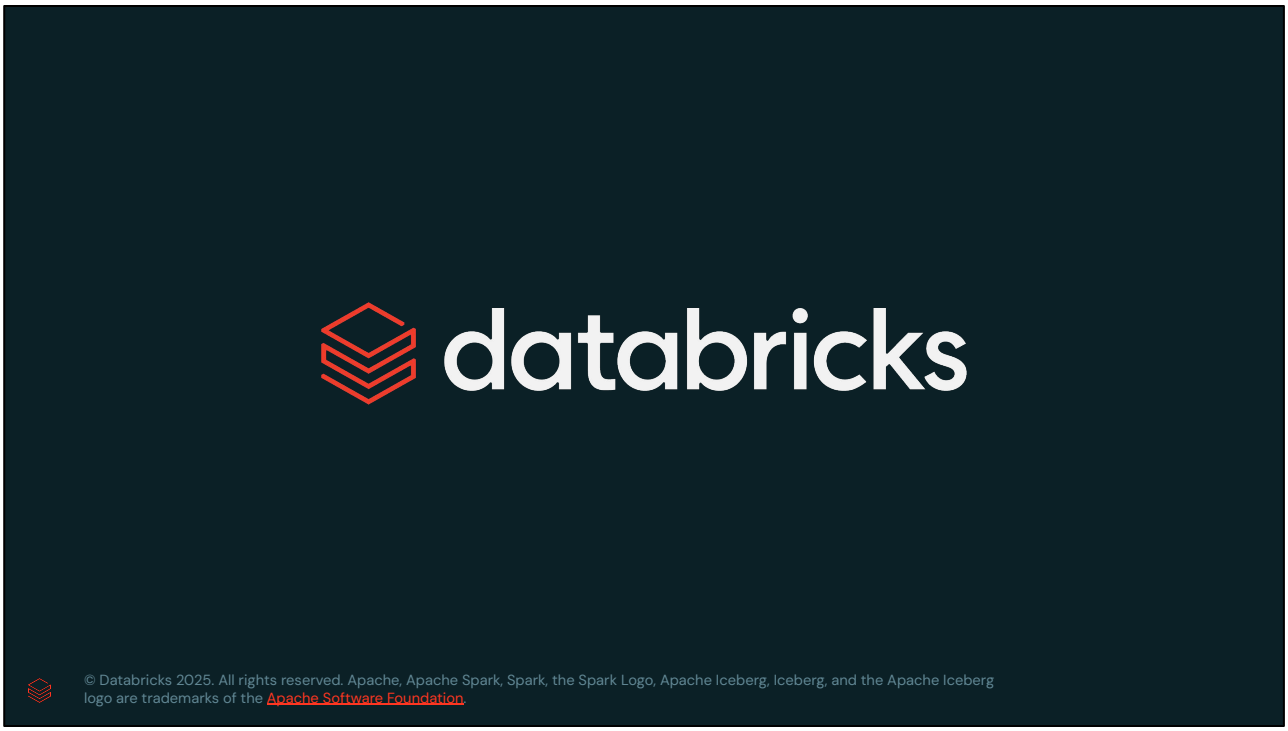
Spill - In the Spark UI

- 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 - Mitigations

- Allocate cluster with more RAM per Core
- Address data skew
- Manage size of Spark partitions
- Avoid expensive operations like explode()
- Reduce amount data preemptively whenever possible

To mitigate spills, one option is to allocate a cluster with more RAM per core, providing a larger memory space to handle the data being processed. Addressing data skew is also important; if one partition is much larger than others, reducing its size can help it fit into memory more easily. Managing the size of Spark partitions is another key approach, which will be demonstrated in the demo. Avoiding expensive operations like explode, when possible, can reduce the risk of spills. Additionally, reducing the amount of data upfront—by excluding unnecessary columns or filtering out rows that aren't needed—can help keep the workload within the available memory and prevent spills during processing.



Thank you for completing this lesson and continuing your journey to develop your skills with us.