Spark Performance

Techniques for Improving Spark Performance

- Tuning Spark configuration to improve scaling for large workloads, enhance parallelism, and minimize memory starvation among Spark executors.
- Using caching and persistence with appropriate levels to expedite access to frequently used data sets.
- Using the Spark web UI to gain visual perspective on performance.
- Using the right data format.
- Reduce the number of passes by using chunkier lambdas.

Viewing and Setting Spark Configuration

- There are three ways to get and set Spark configuration properties.
  - By specifying them in the configuration files found in `$SPARK_HOME/conf` folder.
  - By specifying directly on the `spark-submit` command as a command-line argument.

    ```
    spark-submit --conf "spark.executor.memory=2g"
        BetterInvertedIndex.py --master local[*]
    ```

  - By specifying them inside our application.

    ```
    spark.conf.get("spark.sql.shuffle.partitions")
    '200'
    spark.conf.set("spark.sql.shuffle.partitions", 5)
    spark.conf.get("spark.sql.shuffle.partitions")
    '5'
    ```

- **Order of precedence**: Form least to highest. (1) Configuration files, (2) command line arguments to spark-submit (3) Set on `SparkSession` configuration in the Spark application.

Scaling Spark for Large Workloads

- **Static versus dynamic resource allocation**: In static allocation (the default), we set a fixed number of executors and Spark queues up jobs if all executors are busy. Dynamic allocation allows for varying number of executors. It is useful for streaming or in multi-tenant environments.

```
spark.dynamicAllocation.enabled true
spark.dynamicAllocation.minExecutors 2
spark.dynamicAllocation.schedulerBacllogTimeout 1m
spark.dynamicAllocation.maxExecutors 20
spark.dynamicAllocation.executorIdleTimeout 2m
```

Scaling Spark for Large Workloads (2)

- **Configuring executors' memory and the shuffle service**. The amount of memory available is controlled by `spark.exceutor.memory`. It is divided into execution memory (60%), storage memory (40%), and reserved memory (300MB).
- Execution memory is used for shuffles, joins, sorts, and aggregations. Storage memory is used for user data structures and partitions
- During map and shuffle, there is heavy I/O activity. This can result in a bottleneck because the default configurations are suboptimal for large scale Spark jobs.

```
spark.driver.memory  1g
spark.shuffle.file.buffer 32k (recommended 1MB)
spark.file.transferTo true  (set it to false to force use of
    buffer)
spark.shuffle.unsafe.file.output.buffer 32k (use 1m for
   larger, used for merges in shuffle)
spark.io.compression.lz4.blockSize  32k (increase to 512k)
spark.shuffle.service.index.cache.size 100m
```

- **Maximize Parallelism**. Use repartition() method to rebalance the workload across cores and threads.

Caching and Persistence of Data

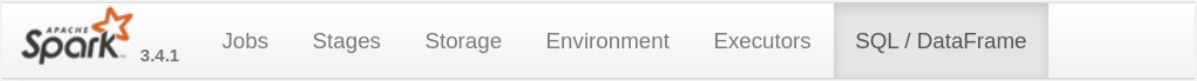
- Use the `cache()` call on a DataFrame or RDD to cache it. The caching only happens after we take an action to realize the data frame or RDD.
- Use the `persist()` method for more control how the data is cached.

`MEMORY_ONLY`	Objects in memory
`MEMORY_ONLY_SER`	Serialized in memory
`MEMORY_AND_DISK`	Objects in memory, serialized on disk
`DISK_ONLY`	serialized on disk
`OFF_HEAP`	Stored off-heap.
`MEMORY_AND_DISK_SER`	serialized in memory and on disk

- **When to cache?** A DataFrame used repeatedly in iterative machine learning training, during frequent transformations during ETL or building data pipelines.
- **When not to cache?** DataFrame is too big to fit in memory. Inexpensive transformation on a DataFrame not requiring frequent use (regardless of size).

SparkUI

- Spark offers an elaborate web UI that allows to examine various components of our applications.



- Debugging through the UI is more like being a detective, following trails of bread crumbs.