Common Performance Problems

Performance issues in Spark are mostly related to the following topics:

- **Skew**: what occurs in case of imbalance in the size of data partitions.
- **Spill**: the writing of temp files to disk due to lack of memory.
- **Shuffle**: moving data between executors due to a wide transformation.
- **Storage**: the way data is stored on disk actually matters.
- **Serialization**: the distribution of code across the cluster (UDFs are evil).

Although finding the root of a problem can be quite hard since one problem can actually cause another one: Skew can induce spill, storage issues can induce excess of shuffle, a wrong way of addressing shuffle can increase skew… And sometimes, many of this causes can be present at the same time!

Skew

In Spark, data is typically read in partitions that are evenly distributed across the cluster executors. As long as we apply transformations to the data, it's possible that some partitions will end up with much more records than others. This imbalance in the size of data partitions is what we call Skew.

Small amount of skew is not a problem and can be ignored. Although, large skews in the data can result in partitions that big that can't fit in the RAM of the workers. This would result in spill, and sometimes, even OOM errors hard to diagnose.

Detecting skew

We have to deep dive into the Spark UI and look at the join job and pay attention to the following:

- **Check Event Timeline:** we consider it unhealthy when we see very unbalance tasks, what means that some tasks are computing way more or less data than others, in other words, partitions are unbalaced. They should all have more or less the same duration.
- **Check Summary Metrics:** pay attention to the Shuffle Read values (min/25th p./median/75th p./max), if the 75th p (meaning most of the tasks) have a high number and the others very low means that most of the tasks are shuffling a lot of data, because the data required is in other partitions. If these Shuffle Read values are are similar to each other, it would indicate that all the tasks are shuffling about the same amount of data, and therefor the partitions are decently balanced.
- **Check Aggregated Metrics by Executor:** If Spill value (memory/disk) is very high, this is caused because of shuffle caused by the skew. Too big partitions can require to store data in temp files in disk because of lack of memory.
- **Inspecting the data:** Once you know you might be facing skew issues, you can get out of doubts by inspecting the data. By performing a count of records per category of "group by" or "join key" and checking if there are way many more records in some

categories than in others. If thre is a big unbalance, the biggest category (partition) will take much longer, causing lots of shuffling and spill.

What can we do to mitigate skew?

In case of OOM issues, you can feel tempted of increasing the RAM of your cluster's workers. This might fix the issue and make your job run, but that won't fix the root of the problem and it might push the problem to later time.. If skew is detected, the first thing to solve is the uneven distribution of records across all partitions. For this purpose we can:

- If in Spark 3, Enable AQE (Adaptative Query Execution).
- If in Databricks, specific skew hint (Skew Join Optimization).
- Otherwise, apply Key Salting. *Salt the skewed column with a random number creating better distribution across each partition at the cost of extra processing*.

Keep in mind that when applying these solutions the job duration can even increase, but remember that even more important than duration, mitigating skew removes potential OOM errors.

How to implement Key Salting for skewed joins?

The general idea of the Key Salting consists on reducing the number of partitions, by increasing the number of join keys. For that we can create a new column wich values are based on the join key, plus a random number within a range. This needs to be applied to all the involved dataframes that are joined by the affected key. After this,

we can perfom the join operation using the new key. As a result, we will end up with more and smaller partitions, and tasks.

The random number range must be chosen after experimentation. In case we choose a large number, we can end up with too many small partitions, and if it's too less, we can keep having the skew problem.

Implementing Key Salting can require lot of effort, consider invest evergy to salt only skewed keys.

Spill

In Spark, this is defined as the act of moving a data from memory to disk and vice-versa during a job. This is a defensive action of Spark in order to free up worker's memory and avoid OOM errors when a partition is too large to fit into memory. In this way, the Spark job can come to an end by paying the high penalty of the compute time caused by the read-write overhead of spilling data.

There are several ways we can get into this problem:

- Having set a `spark.sql.filesMaxPartitionBytes` too high (the default is 128MB) thus ingesting large partitions that might not fit in memory.
- The `explode()` operation of even a small array. Each partition will end up with as much rows as items in the array, therefor the resulting partition size might not fit in memory.
- The `join()` or `crossJoin()` of two tables.

- Aggregating results by a skewed key. As we saw before, having unbalanced datasets can lead to bigger partitions than others, and in some cases this bigger partitions might not fit in memory.

Detecting Spill

In the Spark UI this is represented by:

- **Spill (Memory)**: size in memory of the spilled partitions
- **Spill (Disk)**: size in disk of the spilled partitions (allways smaller than memory because of compression).

This values are only represented in the details page for a single stage (summary metrics, aggregated metrics by executor, task table) or in SQL query details. This makes it hard to recognize because you have to manually search for it. Be aware that in case there is no spill, you won't find these values.

An alternative to manual search for spill, is implementing a SpillListener to track automagically when a stage spills. Unfortunately, this is only available in Scala.

What can we do to mitigate spill?

Once we find our stages are spilling we have a few options to mitigate it:

- Check if spill is caused by data skew, in that case, mitigate that problem first.

- If possible, increase the memory of the cluster's workers. In this way, larger partitions will fit in memory and Spark won't need to write that much into disk.
- Decrease the size of each partition by increasing the number partition. We can do this by tuning the `spark.sql.shuffle.partitions` and `spark.sql.maxPartitionBytes`, or explicitly repartitioning.

Mitigaing spill is not always worth it, so check first if it makes sense or not.

Shuffle

Shuffle is a natural operation of Spark. It's just a side effect of wide transformations like joining, grouping, or sorting. In these cases, the data needs to be shuffled in order to group records with the same keys together under the same partitions for later on being able to execute the aggregations by those keys. When a wide transformation happens, partitions are written to disk, so the executors of the next stage can read the data and continue the job. Because the data needs to be moved across workers, this behaviour can result in lots of network IO.

As I have mentioned before, shuffling it's inevitable. Just put the focus only on the most expensive operations. At same time, be aware that targeting other problems like skew, spill, or tiny files problems is often more effective.

What can we do to mitigate the impact of shuffles?

The biggest pain point of shuffles is the amount of data that needs to be moved across the cluster. In order to reduce this problem, we can apply the following strategies:

- **Use fewer and larger workers to reduce network traffic**. Having same number of executors (CPUs) in less workers, would reduce the amount of data that needs to be transferred throught the network to other workers.
- **Reduce the amount of data being shuffled**. Sometimes these wide transformations are performed over data that is not needed for the final result, increasing the cost unnecesarily. Therefore, we should filter out those columns and rows that are not required before the execution of a wide transformation.
- **Denormalize datasets**. In case a query that causes expensive shuffling is executed very often by data users, the output of this query can be persisted in a data lake and can be queried directly.
- **Broadcast the smaller table**. When one of the tables involved in a join is way smaller than the others, we can broadcast it to all the executors, so it will be fully present there and Spark will be able to join it to the other table partitions without shuffling it. This is called BroadcastHashHoin and it can be applied by using `.broadcast(df)`. However, the default table size threshold is 10MB. We can increase this number by tuning the `spark.sql.autoBroadcastJoinThreshold`, but not too much since it puts the driver under pressure and it can result in OOM errors. Moreover, this approach increases the IO between driver and executors, it doesn't work well when many empty partitions, and requires enough memory in driver and executors.

- **Bucketed datasets**. For joins, data can be pre-shuffled and store it by buckets, and optionally sorted per bucket. This is worth it when working with terabyte size, tables are joined quite often, and no filters are applied. This requires all tables involved to be bucketed by key with same number of buckets (normally one per core). However, the cost to produce and maintain this approach is very high, and must be justifiable.

Storage

The way initial data is ingested in the data lake can lead to problems which are normally related to: tiny files, directory scanning, or schemas.

Tiny Files

We consider tiny files those that are considerably smaller than the default block size of the underlaying file system (128MB).

Having a dataset partitioned in too many tiny files implies longer total time for openening and closing files, and it leads to very bad perfromance. It often relates to a high overhead with ingesting data, or as a result of a Spark job.

We can use the Spark UI to see how many files are read under the SQL tab and checking the read operation.

This problem can be mitigated by:

- **Compacting the existing small files to larger files** equivalent to block size or the efficient partition size used.

- **Make ingesting tools to write bigger files**.

When produced as a result of a **Spark job**, Spark is partitioning the data way more than required for its size, and it's reflected when writing. We can mitigate this by:

- **Changing default partition number**.
  Tuning `spark.sql.shuffle.partitions` (200 by default).
- **Explicitly repartitioning the data before writing**.
  Applying `repartition()` or `coalesce()` functions to decrease the number of partitions, or in case of Spark 3.0+ with AQE enabled set the `spark.sql.adaptive.coalescePartitions.enabled` to true.

Directory Scanning

Having too many directories for some dataset (because of data partioitning) lead to performance issues at scanning time. Too partitioned datasets with no much data also ends up into tiny files problem

We can detect this by paying attention to the scanning time under the SQL tab read operation.

We can mitigate this problem by:

- **Partitioning stored data in a smarter way**.
- **Registering datasets as a tables**. When doing so, metadata like where to find the files belonging to that dataset are stored in the Hive Metastore, thus it's not needed to scan the directory

anymore. However, first time we register the table, it will need some time to retrieve the metadata by scanning direcotries first.

Schemas

Inferring schemas require a full read of a file to determine the data type of each column. This involves time for opening up and scan the files. Reading parquet files requires a one-time read of the schema, because schema is included on the files itself. On the other hand, supporting schema evolution is potentially expensive if you have hundred or thousands of part-files, each schema has to be read in and then merged collectively, and that might be really expesive. Schema merging can be enabled via `spark.sql.parquet.mergeSchema`.

We can mitigate the schema problem by:

- **Providing schema every time**.
- **Registering datasets as tables**. In this way the schema will also be stored in the Hive Metastore.
- **Using Delta format**. Merges schemas automatically for supporting schema evolution.

Serialization

It happens when we need to apply a non-native API transformation, known as UDFs. This implies to serialize the data into a JVM object to be modified outside Spark. The impact of this is way worse in Python, since JVM objects are not native for Python, so they need to be transformed first, execute the code on top of it, and seriallize the

result back to JVM object, causing a big overhead. On the other hand, this part is not needed in Scala since it's JVM native language.

In any case, UDFs are a barrier for the Catalyst Optimizer, since it can't connect the code before and after applying the UDF, since it's impossible for it to know what the UDFs are doing, and how to optimize the overal job execution.

We can mitigate serialization issues by:

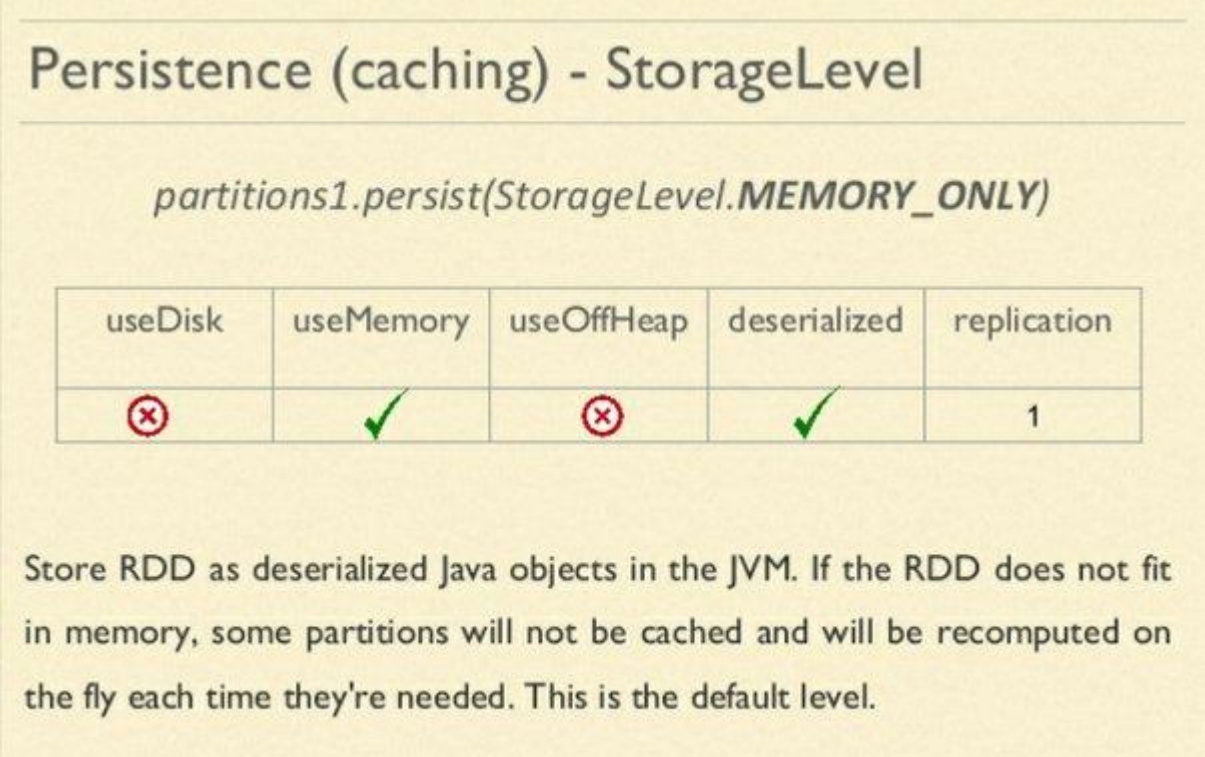
- **Avoid using UDFs, Pandas UDFs or Typed Transformations** whenever possible and use native Spark high order functions instead.
- **If there is no other option**:
  – **Python**: use **Pandas UDF** over "standard" Python code. Pandas UDF uses PyArrow to serialise batches of records (treated as a Pandas Series or Data Frame) to later apply the UDF to every record in Python. On the other hand, regular Python UDFs serialises every single record individually, and executes the UDF on it.
  – **Scala**: use **Typed Transformations** over "standard" Scala code.

If your data transformations require the application of many UDFs, consider Scala as a programming language.

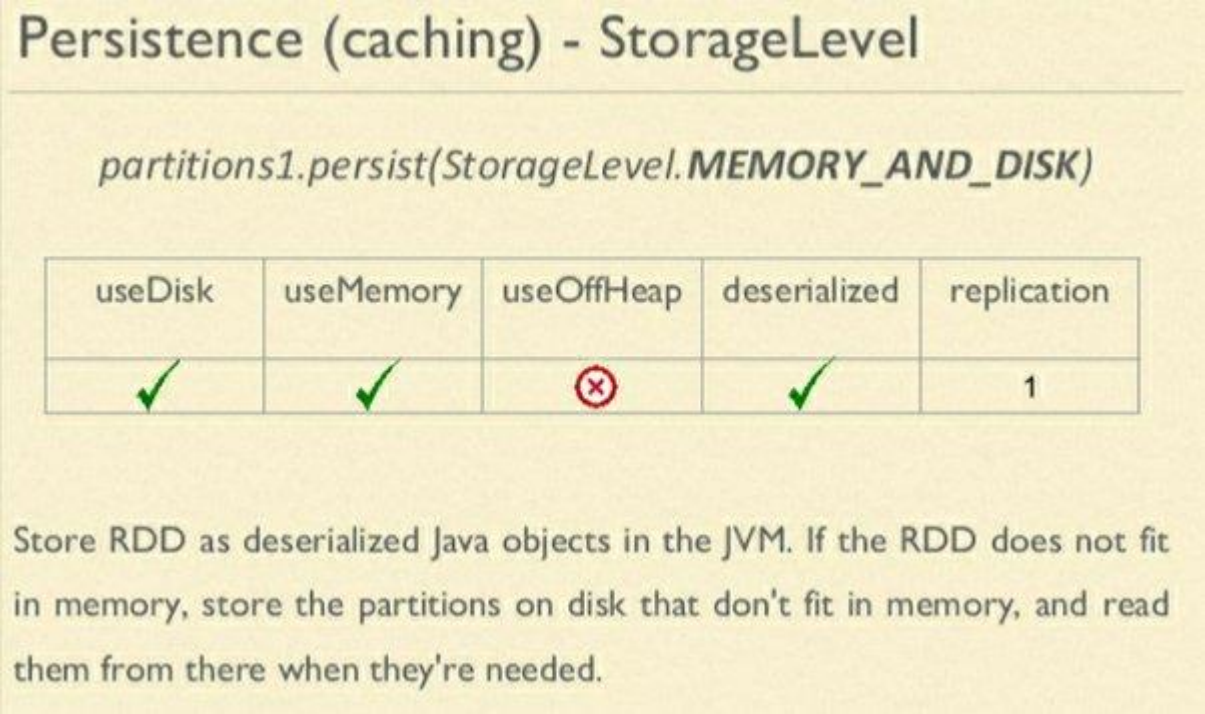
In Apache Spark if the data does not fits into the memory then Spark simply persists that data to disk.

The persist method in Apache Spark provides 6 persist storage levels to persist the data. That are as follows:

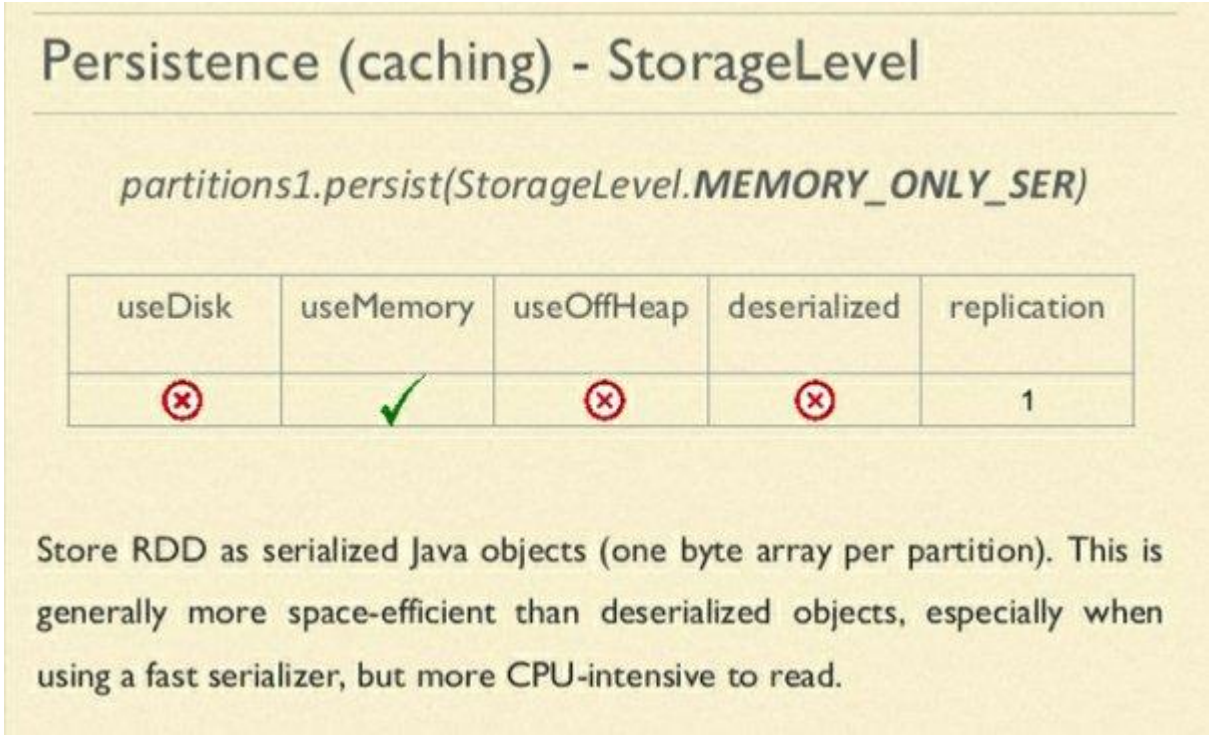
1. **MEMORY_ONLY**



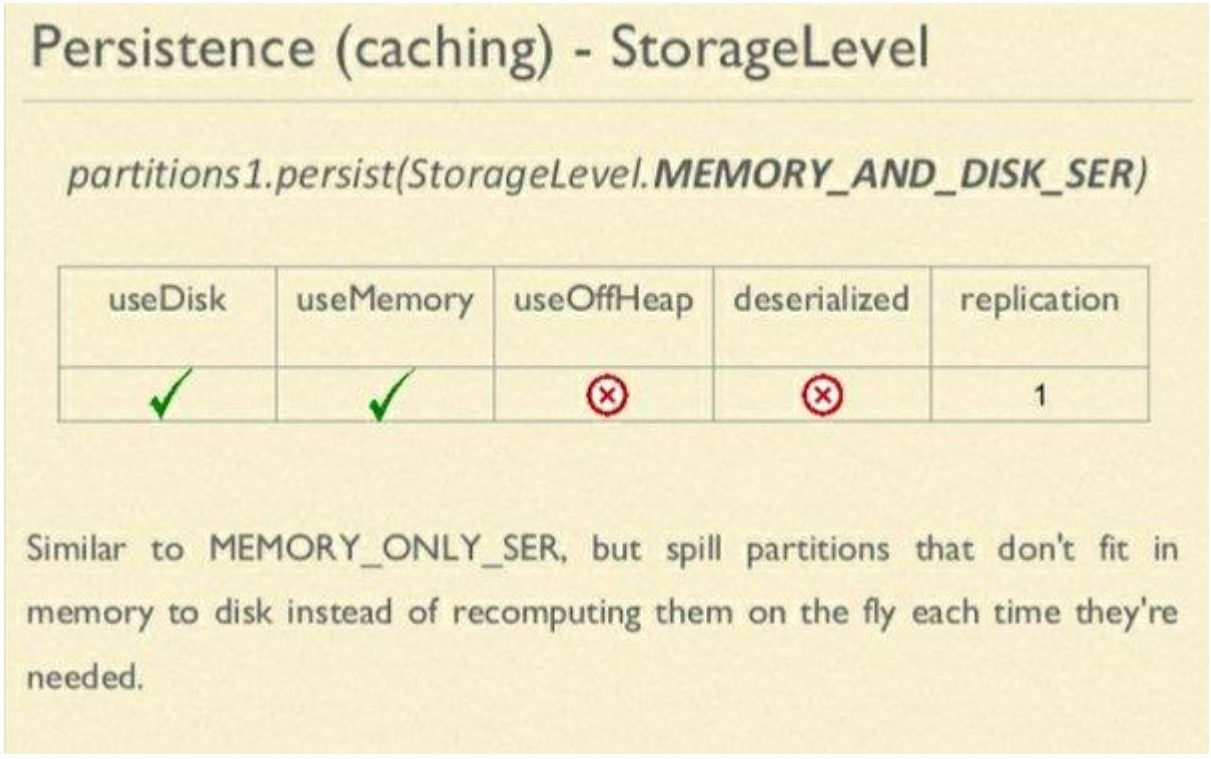
2. **MEMORY_AND_DISK**



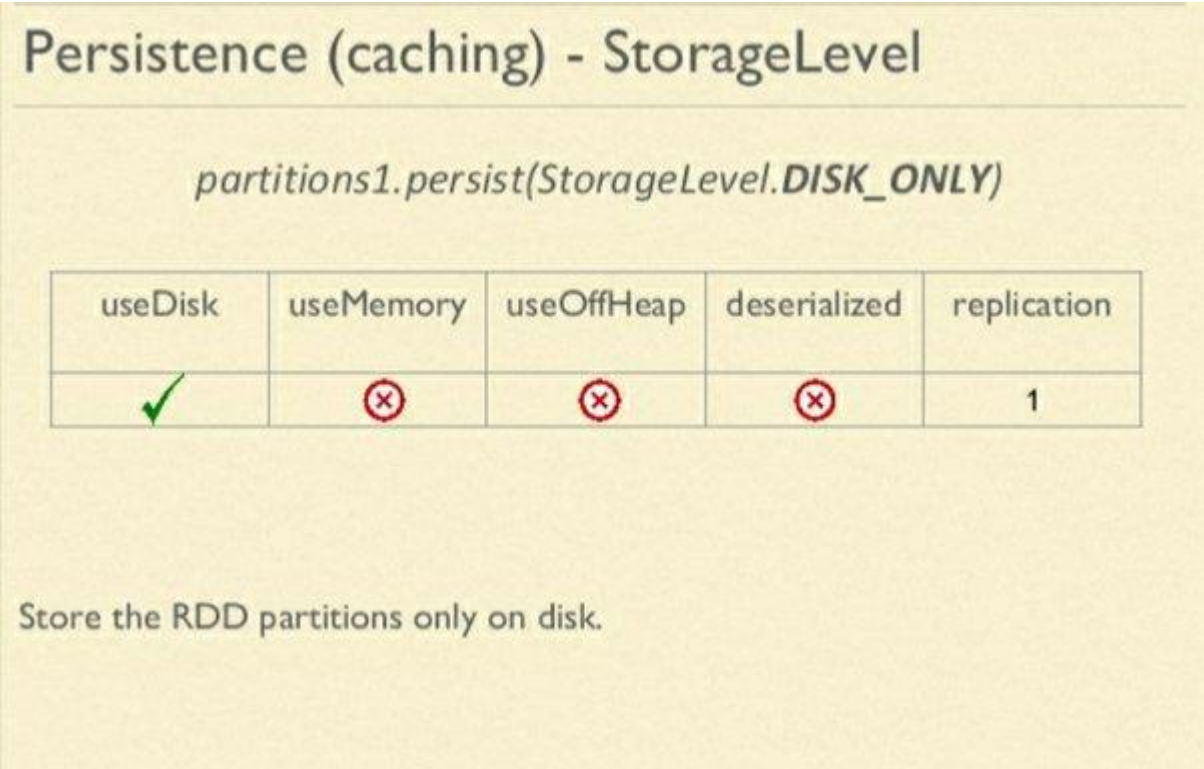
3. **MEMORY_ONLY_SER**



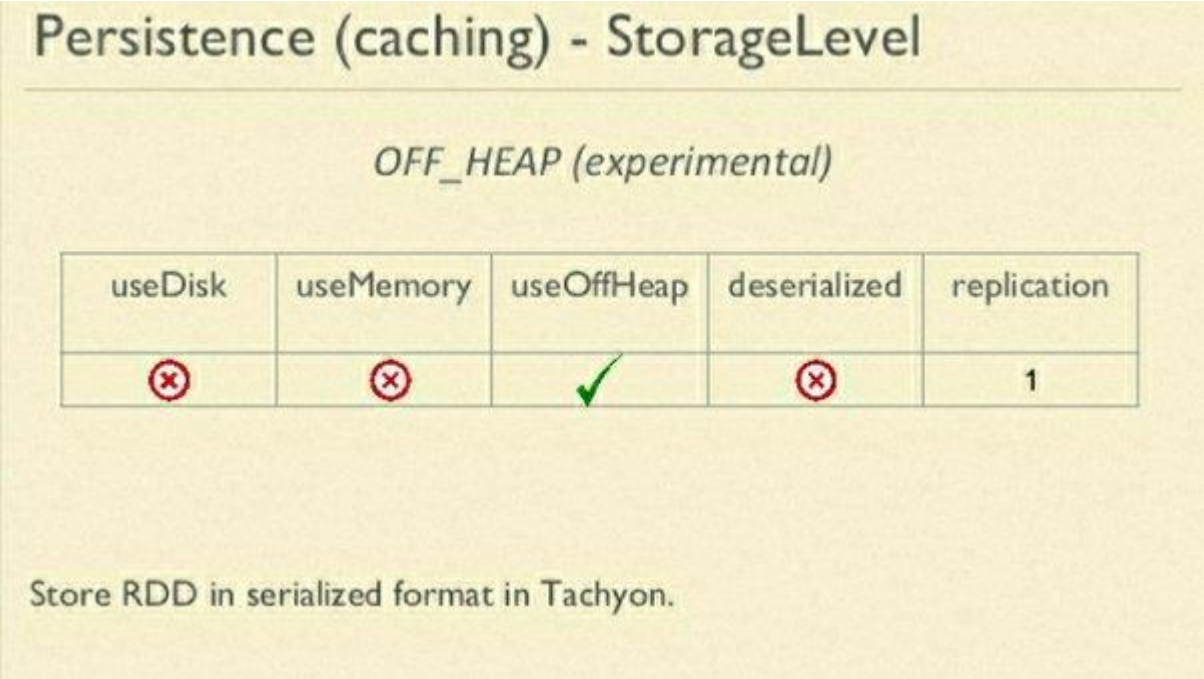
4. **MEMORY_AND_DISK_SER**



5. **DISK_ONLY**



6. **OFF_HEAP**



Which Storage Level to Choose?

- RDD fits comfortably in memory, use MEMORY_ONLY
- If not, try MEMORY_ONLY_SER
- Don’t persist to disk unless, the computation is really expensive.