**Analysis of a Slow PySpark Query and Optimization**

We examine a representative slow PySpark query and apply best practices to optimize it. The original query (from a user forum) joins a large sales table with a customers table and then filters and aggregates. This approach causes excessive data shuffling and redundant computation. For example:

***salesDF = spark.read.parquet("s3://data/sales")***

***customersDF = spark.read.parquet("s3://data/customers")***

***result = (***

***salesDF***

***.join(customersDF, on="customerId", how="inner")***

***.filter(customersDF.country == "US")***

***.groupBy("customerId", "customerName")***

***.agg(sum("amount").alias("total\_sales"))***

***)***

In this code, the entire customersDF is joined before filtering by country, and the join is a full shuffle on "customerId". Also, no caching or broadcast is used. The **key bottlenecks** here are:

* **Large shuffle**: The join sends all rows across the cluster, then the filter eliminates rows. Spark *shuffles* data on the join key, which is expensive.
* **Filter pushdown missed**: Filtering after the join means unnecessary data movement; instead, filters (especially on small tables) should be applied **before** the join to reduce data size.
* **No broadcast**: If customersDF is relatively small, broadcasting it can avoid the shuffle altogether.
* **No caching**: If intermediate results (e.g. filtered DataFrames) are reused, caching can prevent recomputation.
* **Python loops / UDFs**: Although not explicit here, many PySpark queries suffer from using Python loops or UDFs, which bypass Catalyst optimizations. It’s recommended to use Spark’s built-in functions, which are highly optimized (up to 10× faster than Python UDFs).

These inefficiencies lead to long task runtimes and uneven load (often one or a few tasks dominate due to skew or extra work).

**Optimized Query**

We can rewrite the query to address these issues. Key improvements are **filtering and broadcasting the small table before the join**, and **reducing shuffles**:

from pyspark.sql.functions import broadcast, col, sum

***salesDF = spark.read.parquet("s3://data/sales")***

***customersDF = spark.read.parquet("s3://data/customers")***

***usCustomers = customersDF.filter(col("country") == "US")***

***result = (***

***salesDF***

***.join(broadcast(usCustomers), on="customerId", how="inner")***

***.groupBy("customerId", "customerName")***

***.agg(sum("amount").alias("total\_sales"))***

***)***

This version first filters customersDF to the US subset, significantly reducing its size before joining. We then use broadcast(usCustomers) to hint Spark to perform a **broadcast hash join**. Because usCustomers is small after filtering, broadcasting it sends its data to every executor and avoids the expensive shuffle on "customerId". The groupBy still incurs a shuffle, but now only on the already-joined (smaller) dataset.

**Explanations of Optimizations**

* **Filter Before Join**: By filtering customersDF on country == "US" first, we drastically reduce the data to join. This early *predicate pushdown* reduces I/O and shuffle volume. Partition pruning or column filters are crucial: “A good query uses filters wherever possible” to limit data movement.
* **Broadcast Join**: We wrap the small table with broadcast(), forcing a broadcast hash join. Spark’s optimizer will also do this automatically if the table size is under the spark.sql.autoBroadcastJoinThreshold (default 10MB). Explicitly broadcasting ensures the smaller side is sent to all executors, avoiding a shuffle. As the Spark docs note, using the broadcast hint directs Spark to use the broadcast join on the marked relation.
* **Reduce Shuffles & Parallelism**: We should ensure the cluster and Spark configs are tuned. For instance, enable adaptive execution (spark.sql.adaptive.enabled=true) so Spark can automatically convert joins to broadcast or optimize skew at runtime. Also enable skew join handling (spark.sql.adaptive.skewJoin.enabled=true) if data skew is detected. Adjusting spark.sql.shuffle.partitions to match the cluster’s cores can ensure tasks finish faster.
* **Caching Intermediate Results**: If parts of the pipeline (e.g. filtered DataFrames) are reused in multiple actions, caching them in memory avoids recomputation. Cache only when reused frequently: “Only cache datasets that you'll reuse multiple times”, and call unpersist() when done.
* **Avoid Python UDFs / Loops**: This example didn’t use UDFs or driver-side loops, but it’s worth noting: pure Python loops (using .collect() inside a loop) or Python UDFs prevent Spark from optimizing and parallelizing the work. As one expert notes, using Python loops creates multiple Spark jobs and shuffles, leading to “very un-sparky” plans . Whenever possible, use Spark SQL/DataFrame functions (which are compiled by Catalyst) instead of Python UDFs. In fact, built-in Spark functions have been measured to be an order of magnitude faster than equivalent Python UDFs.

**Performance Gains:** By applying these optimizations, the join no longer requires a full shuffle of the large salesDF, and the intermediate dataset is much smaller. Broadcasting can reduce join time from minutes to seconds for large data skew. Caching and proper partitioning prevent redundant work. In practice, these changes typically yield **orders-of-magnitude speedup** on the join and overall job. For example, filtering early can cut data size significantly, and avoiding repeated shuffles (by removing loops and using built-ins) removes the largest overhead.

**Sources:** These optimizations are recommended best practices in the Spark community and documentation [pepperdata.com](https://www.pepperdata.com/blog/why-is-spark-so-slow#:~:text=select%20all%20the%20columns%20of,results%20in%20a%20higher%20overhead) [spark.apache.org](https://spark.apache.org/docs/latest/sql-performance-tuning.html#:~:text=The%20join%20strategy%20hints%2C%20namely,either%20broadcast) [stackoverflow.com](https://stackoverflow.com/questions/73747793/optimize-filter-update-join-loops-in-pyspark-dataframes#:~:text=The%20code%20you%20shared%20is,hence%20the%20poor%20performance) [chaosgenius.io](https://www.chaosgenius.io/blog/spark-performance-tuning/#:~:text=,to%20free%20up%20resources) [stackoverflow.com](https://stackoverflow.com/questions/38296609/spark-functions-vs-udf-performance#:~:text=Spark%20now%20offers%20predefined%20functions,an%20identical%20spark%20function%20exists), including use of broadcast joins, predicate pushdown, caching, and avoiding Python loops. By rewriting the query with these practices, we ensure Spark can fully optimize execution and use the cluster efficiently.