Summarization of Solutions to Data Skew in Apache Spark

Jiashu Chen

CSE544 Principles of Database System, University of Washington

***Abstract*—*Apache Spark is a big data computing framework for parallel data processing on computer clusters. Spark is developed based on the concept of Resilient Distributed Dataset (RDD), which allows in-memory computation on large clusters in a fault-tolerant manner [4]. Spark's in-memory computing capability makes it ten to a hundred times faster than the disk-based Hadoop MapReduce system, especially for iterative algorithms and interactive data mining tasks. Even though Spark is known for its fast in-memory computation, its performance could degrade due to data skew. This project first provides a summarization of solutions to address data skew in distributed computing systems based on previous studies. In the experiment part, 9 join queries with skew issues are tested in Spark. Following the analysis of experiment results, insights on skew mitigation are discussed from the viewpoint of a Spark user.*

Keywords*—*Data Skew, Apache Spark

1. Review on Apache Spark

1.1. Cluster Computing & Parallel Data Processing. Cluster computing frameworks such as Spark and Hadoop manage task execution on data across a cluster of computers [6]. The Spark cluster follows the master-worker architecture. Driver is the master node that manages a cluster of workers. The driver node is responsible for translating the application program into tasks and sending instructions to worker nodes. Worker nodes are responsible for executing tasks assigned by the master node.

The cluster computing architecture enables parallel data processing by dividing data into partitions and assign them to different worker nodes. Nevertheless, advantages of parallel computing would diminish if tasks are unevenly distributed across workers.

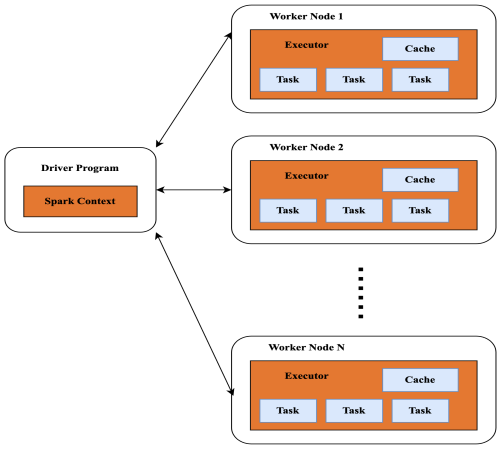

Figure 1. *Spark Cluster*

1.2. RDD Partitioning. Resilient Distributed Dataset (RDD) is the basic data structure in Spark. RDD uses coarse-grained transformation that applies each operation to multiple data items. This is different from systems that use fine-grained transformation where each operation is applied to some specific data items. For instance, update one specific cell in a database. Coarse-grained transformation enables Spark to evaluate transformations lazily as it could keep a record of dependencies between RDD transformations and implement actual transformations when an action is triggered. Actions are operations that return values to the driver node or write data to storage. Transformation operations are lazy while action operations are eager. Datasets are materialized when a action is called [4].

RDDs are partitioned using the key of the dataset and partitions are stored in different worker nodes of a Spark cluster. The number of partitions and the key used to partition RDDs are important factors influencing the load balancing of workers. When one key contains a considerably larger number of tuples compared to others, it results in one partition being significantly larger than others.

1.3. Wide Transformations & Skew. Transformations in Spark could be divided into narrow transformation and wide transformation. In narrow transformations, one partition in the child RDD depends on one partition in the parent RDD. It includes operators such as map, filter. In wide transformations, one partition in the child RDD depends on multiple partitions in the parent RDD. It include operators such as groupBy, join, etc.

Wide transformation is one of the major causes of data skew. For instance, in a groupby transformation, if the groupby field has a skewed distribution, most of the records will be shuffled to a small number of partitions.

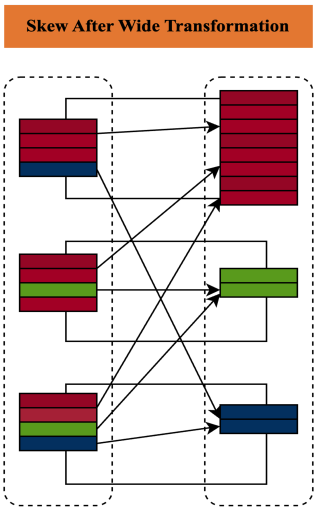

Figure 2. *Data Skew After Wide Transformation*

1.4. Cost of Skew. The direct cost of data skew is time. If a worker node is burdened with a heavy workload, it would take much longer time to complete the task, thus prolonging the

overall job execution time.

Despite that Spark uses in-memory computing, if a large partition is assigned to one worker node, the RAM might not have sufficient memory to cache the partition, resulting in data spill to disks. This would compromise Spark's distinctive advantage of fast in-memory computation.

Moreover, if the partition is too large to fit in local disks, it would cause out-of-memory error and result in job failures.

2. Review on Data Skew

2.1. Skew Mitigation for Parallel Databases.

2.1.1. Range Partitioning. The parallel hash join algorithm is sensitive to partition skew. Range partitioning split data based on the range of the key value. Each processor is assigned with a subrange of key value. If there are k processors, the entire key value range could be split into k subranges. The (k-1) splitting boundaries could be determined based on distribution of key values to equalize the number of tuples assigned to each processor. The (k-1) splitting values are called partitioning vector, which could be estimated using sampling [1].

2.1.2. Virtual Processor Range Partitioning. The previous range partitioning algorithm might not be effective in some cases. For instance, suppose there are two relation with 1000 tuples. For each relation, 90% of tuples have join attribute value "1". All other join attribute values appear only in one tuple. Based on the range partition algorithm, 90% of tuples from both relations would be assigned to one processor, which means that the partition is still skewed.

DeWitt et al. (1992) propose the virtual processor partitioning algorithm, which is to make the number of partitions a multiple of the number of processors [1]. Following the previous example, if there are 10 processors, 900 tuples with join attribute value '1' could be split into 100 buckets. Afterwards, each bucket would be mapped to one processor. The Skewed-Join in Pig system is adapted from this algorithm [3].

2.2. Skew Mitigation for Spark. Imperfect cardinality estimate could lead to sub-optimal query plans. Adaptive Query Execution (AQE) is an optimization technique introduced in Spark 3.0, which is to re-optimize query plan based on runtime statistics [7]. AQE include three main features: Dynamically coalescing shuffle partitions, dynamically switching join strategies, and dynamically optimizing skew joins. All three features could mediate data skew in Spark to some extent.

2.2.1. Coalescing Post Shuffle Partitions. Dynamically coalescing shuffle partitions is to coalesce small partitions after shuffling. Spark suggests users to set a relative large number of shuffle partitions at the begining and AQE could adjust the post shuffle partition number by coalescing small ones [7]. The threshold to determine whether a partition needs to be coalesced [2] is controlled by the configuration "minPartitionSize". The default size is 1MB.

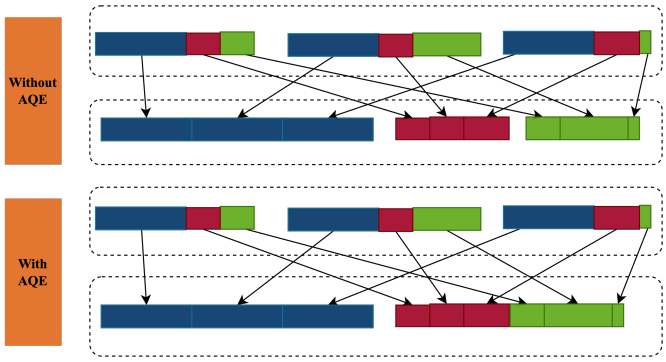

Figure 3. *Coalescing Post Shuffle Partitions*

2.2.2. Dynamically Switching Join Strategies. Broadcast hash join is usually the most efficient join strategy. Spark will choose broadcast hash join if one of the relation is smaller than the "autoBroadcastJoinThreshold" parameter. However, if the join relation is the result after a set of complicated operators, initial cardinality estimation could be wrong. AQE could adjust the join strategy based on runtime statistics. If the materialized join relation is overestimated and the actual size is smaller than the broadcast threshold, AQE would switch to broadcast join, which could reduce data shuffling and mitigate skew.

2.2.3. Splitting Skewed Shuffle Partitions. AQE skew join optimization could detect skewed partition based on shuffle file statistics. If some partition is too big, it would be split into smaller partitions. There are two configurations determining whether a partition is skewed [2]. One is "skewedPartitionFactor". A partition is considered to be skewed if the partition is larger than the median partition size multiply this parameter. The default is set at 5. The second configuration is "skewedPartitionThresholdInBytes". A partition is considered to be skewed is its size is larger than this parameter. The default is 256MB.

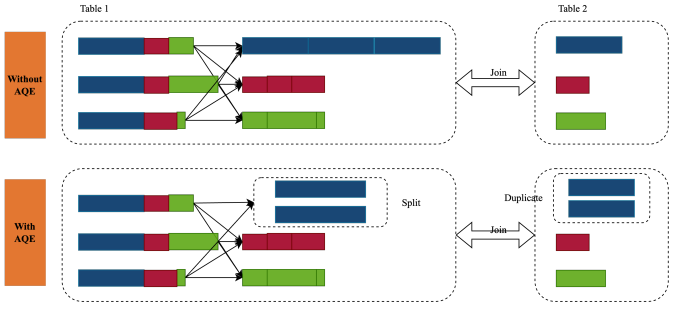

Figure 4. *Splitting Skewed Shuffle Partitions*

2.3. Salting Technique. Salting technique is an approach to handle data skew adopted from Cryptography. Key salting is to add a random number to the skewed key which helps to distribute data more evenly across worker nodes. Figure 6 and Figure 7 shows an exmaple of skewed join with and without key salting. Using key salting to mitigate skewed join could be divided into following steps:

- Step 1: Choose a salting number, which determines the number of small partitions to divide the large partition into. In the example, the salting number is 3.

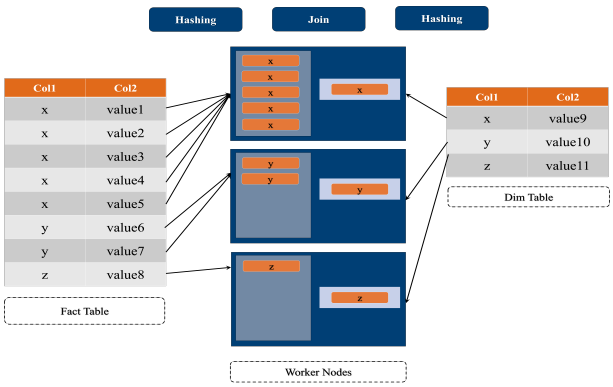

Figure 5. *Skewed Join without Salting Example*

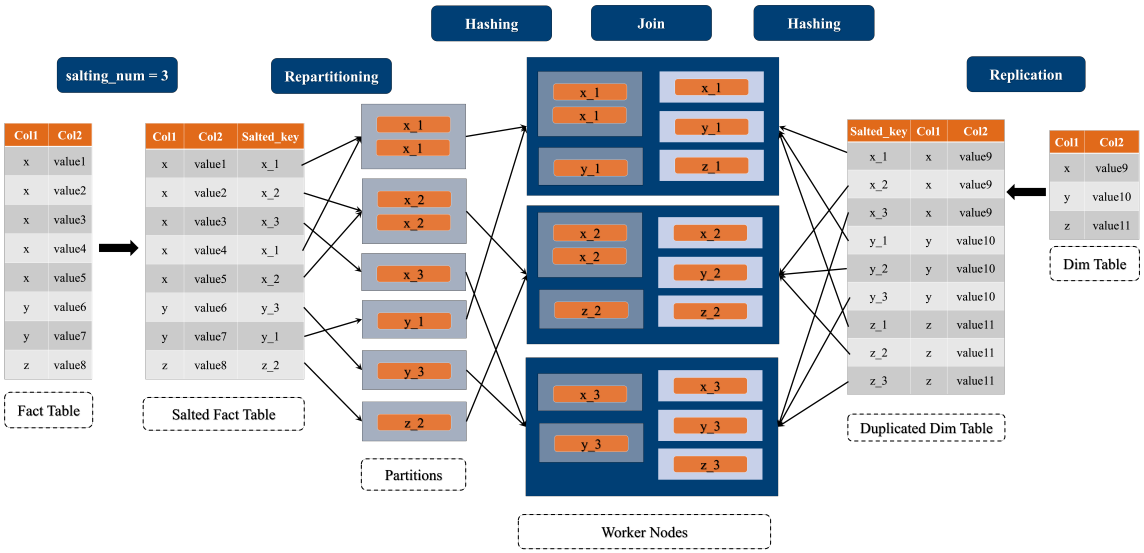

Figure 6. *Skewed Join with Salting Example*

- Step 2: Add a random number between [1, salting_num] to each join key in the large relation. In the example, originally there are 6 tuples with key "x". After salting, 6 tuples with key "x" are transformed into 2 tuples with key "x_1", 2 tuples with key "x_2", and 1 tuple with key "x_3".
- Step 3: Redistribute data using salted key and divide the data into a larger number of smaller partitions. Afterwards, hash each partition to one worker node. With key salting, tuples in the large relation are more evenly distributed across worker nodes.
- Step 4: Since the join key becomes the salted key. The smaller table in the join relation needs to be modified. Each tuple in the smaller table is duplicated by the salting number. In the example, the tuple with key "x" is transformed into 3 tuples with key "x_1", "x_2", and "x_3". The smaller table originally has 3 tuples. After salting, it has 9 tuples.
- Step 5: Tuples in the duplicated smaller table are hashed into different worker nodes using the same hashing function used in step3.

3. Experiment

3.1. Spark Environment. Spark is installed locally and all the computations are performed locally using 10 cores. Query plans and computing timelines are obtained from Spark UI.

3.2. Two Datasets.

3.2.1. New York Taxi Trips Data. One table is taxi trip records, which is a big table with 19,817,583 records. Another table is the taxi zone look up table, which is a small table with only 265 records [9].

3.2.2. IMDB Data. One table is name.basics, which is a big table with 13,302,452 records. Another table is title.principals, which is also a big table with 60,717,362 records [8].

3.3. Skewness Manipulation. Both datasets are manipulated to increase the skewness level. For instance, In the taxi_trip table, a significant number of records in the PULocation field are altered to a specific locationID, resulting in one partition containing significantly more records than others during shuffling

3.4. Four Configurations & Four Join Commands. Four configurations are used in the experiment to test the effectiveness of different techniques in addressing data skew.

Configuration	Broadcast Join	AQE	AQE Parameters
Conf1	Disabled	Disabled	Disabled
Conf2	Enabled	Disabled	Disabled
Conf3	Disabled	Enabled	Default: minPartitionSize(1MB); skewedPartitionFactor(5); skewedPartitionThresholdInBytes(256MB)
Conf4	Disabled	Enabled	Adjusted: minPartitionSize(128K); skewedPartitionFactor(3); skewedPartitionThresholdInBytes(256K)

Figure 7. *Four Configurations*

Four join commands are used to test the effectiveness of key salting.

Join Command	Data	Key Salting
Join Command 1	Taxi Trips Data	No
Join Command 2	Taxi Trips Data	Yes
Join Command 3	IMDB Data	No
Join Command 4	IMDB Data	Yes

Figure 8. *Four Join Commands*

4. Results

4.1. Join Results of Taxi Trip Data. 6 join queries are tested on the taxi trips data. These join queries represent join between a large table and a small table. Join results are summarized in Figure 15.

4.1.1. Query1. Query1 uses configuration 1 and join command 1. Since broadcast join is disabled, the query uses sort merge join. There is a very obvious straggling task in the join stage. The straggling task took 9 seconds to complete, while the 75th percentile of task duration is 1s.

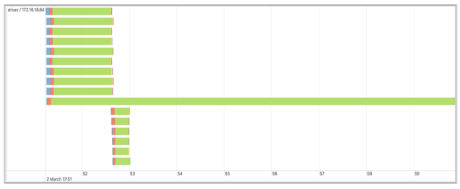

Figure 9. *Query1 Timeline*

4.1.2. Query2. Query2 uses configuration 1 and join command 2. The only difference between Query1 and Query2 is that the salting technique is applied here. Compared with Query1, the join stage time reduced from 9s to 4s. The median, 75th percentile, and the max of task duration is 2s, 2s, 4s.

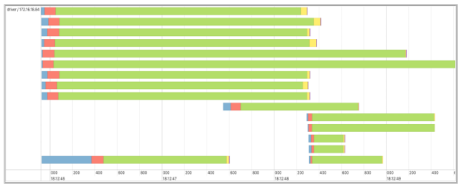

Figure 10. *Query2 Timeline*

4.1.3. Query3. Query3 is the same as Query2, except that the salting number is increased from 20 to 40, which means that each partition is split into more small partitions. The median, 75th percentile, and the max of task duration is 1s, 2s, 2s. The work distribution is even more balanced compared to Query2.

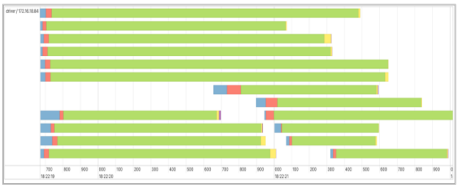

Figure 11. *Query3 Timeline*

4.1.4. Query4. Query4 uses configuration 2 and join command 2. The broadcast join is enabled. Since the smaller table is less than the broadcast threshold, broadcast hash join is used. The join stage uses only 0.8s. No straggling task identified. The median, 75th percentile, and the max of task duration 0.5s, 0.7s, 0.7s.

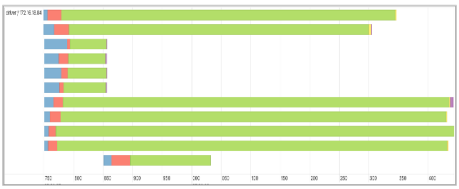

Figure 12. *Query4 Timeline*

4.1.5. Query5. Query5 uses configuration 3 and join command 1. The broadcast join is disabled and AQE is used with all related parameters keeping its default values. The default value for Spark to flag a partition as skewed is 256MB. Since this dataset is relatively small and the largest partition is less than this threshold (122.6MB), this large partition is not split. Only two partitions are used in the join stage. This means that only 2 out of 10 cores are used. One task takes 1s while another task takes 8s. This query reveals that AQE does not guarantee to maximize parallelism and balance workload.

Figure 13. *Query5 Timeline*

4.1.6. Query6. Query6 is the same as Query5, except that the AQE parameters are adjusted. The "skewedPartitionThresholdInBytes" parameter is set to 256K so that the largest partition in Query 5 would be detected as skew partition. The workload is slightly more balanced compared to Query5.

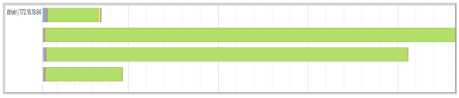

Figure 14. *Query6 Timeline*

Query	Configuration	Join Command	Key Salting	Query Plan	Straggling Task	Join Stage Duration	Median/75th Percentile/Max Join Task Duration
Query 1	Conf1	Join Command 1	No	Sort Merge Join	Yes	9s	1s; 1s; 9s
Query 2	Conf1	Join Command 2	Yes (20)	Sort Merge Join	No	4s	2s; 2s; 4s
Query 3	Conf1	Join Command 2	Yes (40)	Sort Merge Join	No	2s	1s; 2s; 2s
Query 4	Conf2	Join Command 1	No	Broadcast Hash Join	No	~0.8s	0.5s; 0.7s; 0.7s
Query 5	Conf3	Join Command 1	No	Sort Merge Join	Yes	8s	1s; 8s; 8s
Query 6	Conf4	Join Command 1	No	Sort Merge Join	No	5s	4s; 5s; 5s

Figure 15. *Join Results of Taxi Trips Data*

Query	Configuration	Join Command	Key Salting	Query Plan	Straggling Task	Join Stage Duration	Median/75th Percentile/Max Join Task Duration
Query 7	Conf2	Join Command 3	No	Sort Merge Join	Yes	13s	6s; 7s; 12s
Query 8	Conf2	Join Command 4	Yes (5)	Sort Merge Join	No	23s	15s; 15s; 16s
Query 9	Conf3	Join Command 3	No	Sort Merge Join	Yes	15s	9s; 9s; 15s

Figure 16. *Join Results of IMDB Data*

4.2. Join Results of IMDB Data. 3 join queries are tested on the IMDB data. These join queries represent join between two large tables. Join results are summarized in Figure 16.

4.2.1. Query7. Query7 uses configuration 2 and join command 3. The broadcast join is not disabled. As both tables exceed the broadcast join threshold, sort merge join is used. There is one task that takes relatively longer time than other tasks. The median, 75th percentile, and the max of task duration 6s, 7s, 12s.

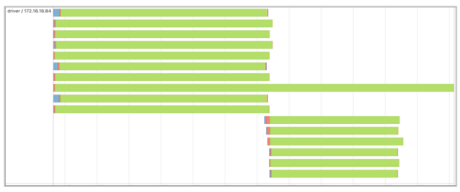

Figure 17. *Query7 Timeline*

4.2.2. Query8. Query8 uses configuration 2 and join command 4. Salting technique is used and AQE is disabled. Salting involves duplicating one of the join relations. In this case, both relations in this query are large. The name table has 13,302,452 rows originally. After salting, it has 66,512,260 (5 * 13,302,452) rows. The join stage takes more time compared to Query9. Moreover, the query failed several times due to out-of-memory error.

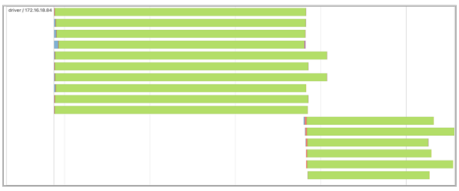

Figure 18. *Query8 Timeline*

4.2.3. Query9. Query9 uses configuration 3 and join command 2. AQE is used. From the query plan, no partition is split or coalesced after shuffling. The workload distribution is almost the same as Query7 as AQE doesn't flag the largest partition as a skew task. The join stage takes even longer time due to additional overhead introduced by AQE. The median, 75th percentile, and the max of task duration 9s, 9s, 15s.

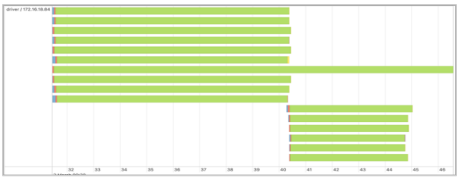

Figure 19. *Query9 Timeline*

5. Analysis

5.1. Broadcast Hash Join. From experiment on the taxi trips data, broadcast hash join is the most efficient join strategy. If one of the join relations is less than the broadcast join threshold controlled by the "autoBroadcastJoinThreshold" parameter, Spark will choose broadcast hash join. However, there are some issues related to broadcasting.

In Spark, about 60% of memory in executor memory is evenly allocated for storage and execution. The storage part

is for caching intermediate results and the execution part is for computing [5]. During broadcast hash join, the broadcast table is replicated and cached in each worker node. If the cached table exceeds available storage memory, it would be spilled to disks. In scenarios where local disk space is insufficient, this can result in out-of-memory errors and job failures. Therefore, the selection of broadcast threshold parameter is critical. Setting the parameter too high will lose the advantages of the broadcast hash join. Setting this parameter too low might cause insufficient storage memory and memory spill.

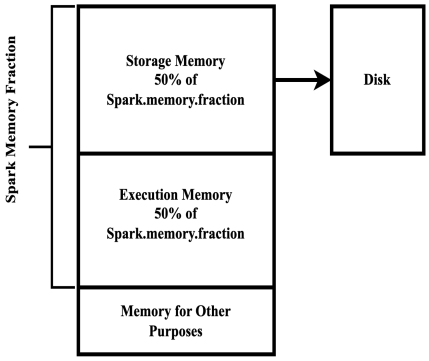

Figure 20. *Simplified Spark Memory Management Model*

5.2. Key Salting. While the key salting technique could help distribute data evenly across data nodes, its usage should also be carefully considered as it also involves data replication. Each tuple of one relation will be duplicated by the salting number. Therefore, the salting technique is not suitable for join between two large relations. In the experiment, Query8 applies the salting technique to join between two large relations and the query failed several times due to out-of-memory error.

5.3. Adaptive Query Execution (AQE). AQE uses a set of configuration parameters to flag skewed partition and small partition, and to switch join strategies. For instance, in Query5, since the largest partition is less than the parameter "skewedPartitionThresholdInBytes", Spark will ignore the skewness. It would assume the skewness does not worth the cost of data splitting and redistribution.

AQE could help mitigate data skew. However, as these parameters are hard-coded and data-independent, its effectiveness depends on whether the configuration thresholds align with the data.

6. Conclusion

Based on experiment results, some insights are summarized from the viewpoint of a Spark user.

- If one relation in the join query is small, broadcast hash join would be the most efficient method.
- Key salting technique could be used for join between a small relation and a large relation. It is not suitable for join between two large relations.
- Adaptive Query Execution (AQE) is effective in most cases, but it does not guarantee skew mitigation and it may not maximize parallelism. Users could adjust configuration parameters based on data distribution.

In conclusion, while there are various ways to address data skew in Spark, there lacks a fully transparent and automatic solution for skew mitigation. Spark uses a large set of hard-coded configuration parameters to determine join strategy, repartition, splitting and coalescing etc. Since these parameters does not consider data distribution, it does not guarantee skew mitigation. In certain cases, users might need to check Spark Web UI for query plan, adjust configuration parameters, or apply some techniques to modify key distribution.

7. Limitations

- The datasets utilized in the experiments remain relatively small in size. Spark is usually deployed for computing data on the scale of terabytes or petabytes. Queries tested in this experiment are completed within seconds, while a Spark job might take hours in a industry setting. Consequently, certain observations in this experiment may not be applicable to large-scale data processing.
- The experiment is conducted on a local machine, utilizing a single node with 10 cores. This setup may not accurately represent the complexities of cluster computing environments.
- The experiment primarily focuses on the user's perspective. Further experiments could provide valuable insights from a developer's viewpoint, such as exploring the integration of range partitioning in Spark.

References

[1] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins", in *Proceedings of the 18th International Conference on Very Large Data Bases*, ser. VLDB '92, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 27–40, ISBN: 1558601511.

[2] Apache Software Foundation, *Spark*, version 3.5.0, Feb. 19, 2010. [Online]. Available: https://spark.apache.org/docs/latest/sql-performance-tuning.html#join-strategy-hints-for-sql-queries.

[3] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications", in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 25–36, ISBN: 9781450312479. DOI: 10.1145/2213836.2213840. [Online]. Available: https://doi.org/10.1145/2213836.2213840.

[4] M. Zaharia, M. Chowdhury, T. Das, *et al.*, "Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing", in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28, ISBN: 978-931971-92-8. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[5] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "Memtune: Dynamic memory management for in-memory data analytic platforms", in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 383–392. DOI: 10.1109/IPDPS.2016.105.

[6] B. Chambers and M. Zaharia, *Spark: The Definitive Guide Big Data Processing Made Simple*, 1st. O'Reilly Media, Inc., 2018, ISBN: 1491912219.

[7] W. Fan, H. van Hövell, and M. Xue, *Adaptive query execution: Speeding up spark sql at runtime*, https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html, Accessed: 2024-02-26, 2020.

[8] https://developer.imdb.com/non-commercial-datasets/, [Accessed 09-03-2024].

[9] *TLC Trip Record Data - TLC — nyc.gov*, https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page?source=post_page-----e2934b00b021----------------------------------, [Accessed 09-03-2024].

8. Appendix

```python
import findspark
findspark.init()
import time
from pyspark.sql import SparkSession
from pyspark import SparkConf
from pyspark.sql import DataFrame
import pyspark.sql.functions as F


def conf1_setup():
    conf = SparkConf() \
        .set('spark.driver.memory', '20G') \
        .set('spark.sql.autoBroadcastJoinThreshold', '-1') \
        .set('spark.sql.shuffle.partitions', '16') \
        .set('spark.sql.adaptive.enabled', 'false')

    spark_session = SparkSession\
        .builder\
        .master('local[8]')\
        .config(conf = conf)\
        .appName("conf1") \
        .getOrCreate()

    url = spark.sparkContext.uiWebUrl
    print("Spark web UI: ", url)

    return spark_session
```

Code 1. *Example of Configuration Code*

```python
def conf2_query9():
    # set up spark session
    spark = conf2_setup()

    # read data
    principals = spark.read.format("csv")\
        .option("header", "true")\
        .option("delimiter", "\t")\
        .option('inferSchema','true')\
        .load("file:///Users/jiashu/Documents/UW_CSE544/project/experiments_finals/data/data2/title.principals.tsv")

    names = spark.read.format("csv")\
        .option("header", "true")\
        .option("delimiter", "\t")\
        .option('inferSchema','true')\
        .load("file:///Users/jiashu/Documents/UW_CSE544/project/experiments_finals/data/data2/name.basics.tsv")

    # manipulate data to make it more skew
    top4000 = principals\
    .groupBy('nconst')\
    .agg(F.count(F.lit(1)).alias("num_rows")).sort(F.col("num_rows").desc())\
    .select('nconst')\
    .limit(4000)

    name_to_change = top4000.rdd.map(lambda x: x[0]).collect()

    # manipulate data to make it more skew
    principals = principals.withColumn('nconst',
                    F.when(F.col('nconst').isin(name_to_change), F.lit("nm0914844"))
                    .otherwise(F.col('nconst')))

    """
    # check data skewness
    principals\
        .groupBy('nconst')\
        .agg(F.count(F.lit(1)).alias("num_rows")).sort(F.col("num_rows").desc())\
        .show(truncate=False, n = 30)
    """

    # join two tables and group by
    principals.join(names, principals['nconst'] == names['nconst'], 'inner')\
        .groupBy("birthYear") \
        .agg(F.avg(F.length('primaryName')).alias("avg_name_length")) \
        .show(truncate = False, n = 500)
```

Code 2. *Example of Query Code*

```python
def conf2_query10():
    # set up spark session
    spark = conf2_setup()

    # read data
    principals = spark.read.format("csv")\
        .option("header", "true")\
        .option("delimiter", "\t")\
        .option('inferSchema','true')\
        .load("file:///Users/jiashu/Documents/UW_CSE544/project/experiments_finals/data/data2/title.principals.tsv")

    names = spark.read.format("csv")\
        .option("header", "true")\
        .option("delimiter", "\t")\
        .option('inferSchema','true')\
        .load("file:///Users/jiashu/Documents/UW_CSE544/project/experiments_finals/data/data2/name.basics.tsv")

    # manipulate data to make it more skew
    top4000 = principals\
    .groupBy('nconst')\
    .agg(F.count(F.lit(1)).alias("num_rows")).sort(F.col("num_rows").desc())\
    .select('nconst')\
    .limit(4000)
```

```
24
25      name_to_change = top4000.rdd.map(lambda x: x
        [0]).collect()
26
27      # manipulate data to make it more skew
28      principals = principals.withColumn('nconst',
29                    F.when(F.col('nconst').isin(
        name_to_change), F.lit("nm0914844"))
30                    .otherwise(F.col('nconst')))
31      # key salting
32      salting_n = 5
33
34      principals = principals\
35          .withColumn("salt_key", F.floor(F.rand
        (222)*(salting_n-1)))
36
37      names = names \
38          .withColumn('key2', F.array([F.lit(num)
        for num in range(0, salting_n)]))\
39          .withColumn('key2', F.explode(F.col('
        key2')))
40
41      # join two tables and group by
42      principals.join(names, (principals['nconst']
         == names['nconst']) & (principals['salt_key
        '] == names['key2']),
43                             'inner')\
44          .groupBy("birthYear") \
45          .agg(F.avg(F.length('primaryName')).
        alias("avg_name_length")) \
46          .show(truncate = False, n = 500)
```

Code 3. *Example of Key Salting Code*