Automatically Configuring Parallelism for Hybrid Layouts

Rana Faisal Munir[1,2], Alberto Abelló[1], Oscar Romero[1], Maik Thiele[2], and Wolfgang Lehner[2]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain
{fmunir,aabello,oromero}@essi.upc.edu
[2] Technische Universität Dresden, Germany
{maik.thiele,wolfgang.lehner}@tu-dresden.de

Abstract. Distributed processing frameworks process data in parallel by dividing it into multiple partitions and each partition is processed in a separate task. The number of tasks is always created based on the total file size. However, this can lead to launch more tasks than needed in the case of hybrid layouts, because they help to read less data for certain operations (i.e., projection, selection). The over-provisioning of tasks may increase the job execution time and induce significant waste of computing resources. The latter due to the fact that each task introduces extra overhead (e.g., initialization, garbage collection, etc.).
To allow a more efficient use of resources and reduce the job execution time, we propose a cost-based approach that decides the number of tasks based on the data being read. The proposed cost-model can be utilized in a multi-objective approach to decide both the number of tasks and number of machines for execution.

Keywords: Big data, Hybrid storage layouts, Parallelism, Parquet, Spark

1 Introduction

The competition in businesses demands quick insights from data, which is exponentially growing from petabytes to zettabytes [15]. Researchers have proposed distributed processing frameworks (e.g., Hadoop ecosystem[3] and Spark[4]) for quickly processing such large volumes of data to meet the business demands. These frameworks provide distributed storage (e.g., HDFS[5]) and distributed processing [6]. In addition, for more efficient analysis, very wide tables [3, 10] are being used to store non-normalized data in hybrid layouts [2, 11]. Through their built-in operations (e.g., projection, selection), these layouts read data more efficiently from the disk. Hybrid layouts allow to read less data from the disk. This is not thoroughly exploited by distributed frameworks when deciding the number of tasks for processing the data. They always decide the number of tasks based

[3] https://hadoop.apache.org
[4] https://spark.apache.org
[5] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

on the total table size and not on the portion of the table being read. This leads to the over-provisioning of tasks, where many tasks remain idle — without any data to process, still present extra overhead (e.g., initialization time, garbage collection). Furthermore, the idle tasks also waste the computational resources which are assigned to them. The latter is not considered even in the area of cloud computing [9, 16, 18], where computational resources are decided based on the total data size. This leads to wastage of resources and money.

As argued above, we need to decide the number of tasks based on the actual data read from the disk. To do that, we first need to estimate the read size, which can be done by utilizing our cost model presented in [12], which estimates the scan, projection, and selection sizes for hybrid layouts.

In this paper, we propose to extend it further to estimate the makespan of the job implementing a query based on the estimated reading size. Thus, we design a framework which takes a user query and data statistics as inputs to estimate the reading size, and then through a multi-objective optimization method decides the number of *tasks* and *executors*. After configuring the number of tasks and executors, the query would be automatically submitted to a distributed processing framework.

The main contribution of this work is to discuss the main variables to be considered in a multi-objective optimization method to configure the number of tasks and executors of a given query.

The remainder of this paper is organized as follows: In Section 2, we discuss the related work. In Section 3, we present our approach for configuring the number of tasks and executors for a given query. Finally, in Section 4, we conclude the paper.

2 Related Work

Estimating Number of Tasks. There are research works [13, 17] for Hadoop, which estimate the number of mappers and reducers tasks. Moreover, these approaches do not consider the amount of data read, while estimating the number of tasks. These works only estimate the tasks based on the available number of machines and some objectives (such as deadline). As previously argued, the amount of data read is an important factor in deciding the number of tasks.

Resource Provisioning in Cloud. There have been extensive research works [9, 16, 18] by cloud community on resource provisioning. These works focus more on deciding the number of machines to process an application. They aim at saving energy and computational resources, which indirectly leads to cost savings. However, they make these decisions without considering the reading size. Our approach could help them to decide resource provisioning in more granular level and overall, it can help these works to achieve their goals more efficiently.

Tuning Configurable Parameters. There are research works [8, 14] to tune the configurable parameters of distributed processing frameworks. In [5], the shuffle performance in Spark is improved by controlling the total number of shuffle files. These works do not explicitly consider the degree of parallelism. Their main aim is to fine tune a distributed processing framework. Our approach can be complementary to these works.

[1] presents a cost model for Spark SQL to evaluate different query plans. However, it does not configure the number of tasks and executors. We can use this work as complementary to ours, as well.

3 Our Approach

In this section, we discuss our proposed approach. It is based on a cost model which can be utilized in a multi-objective optimization method for configuring the number of tasks and executors.

For cost model, we propose to extend our previous work [12], that estimates the reading size for hybrid layouts. The reading size can be further used in estimating the number of tasks and executors. The number of tasks always depends on the size of partition (also known as *input split*), which we need to consider in the extended cost model.

Moreover, we focus on read-only analytical jobs, to estimate the amount of data read for their first operation and based on that, we try to find the best partition size to control the number of tasks. Given the simplicity of a file system (far from that of a DBMS), only three operations need to be considered: scan, projection, and selection. These three operations can be generalized to *selection sorted* and *selection unsorted*, because scan and projection operations are just the extreme cases of selection unsorted with selectivity factor of 1 (i.e., they read all Row Groups - *RGs*), and when you can choose the attributes in the output.

3.1 Estimating Number of Tasks

Modern distributed processing frameworks decide the number of tasks based on the total file size (which is the actual size of data without metadata) and the partition size. Moreover, all tasks cannot be executed at once, if the number of executors is less than the total number of tasks. Thus, we need multiple rounds/waves to finish the job.

3.2 Types of Partitions

As discussed earlier, data is processed by dividing into multiple partitions and each partition is processed in a separate task. These tasks process different amount of data presented in each partition, based on the number of referred attributes and the selection predicate. For instance, *selection unsorted* always reads all RGs, thus every task processes a full partition except the last one, whose partition might not be completely full, as shown in Figure 1a.

On the other hand, *selection sorted* has high probability of skipping some RGs, thus, it can have empty partitions, which only read metadata. Additionally,

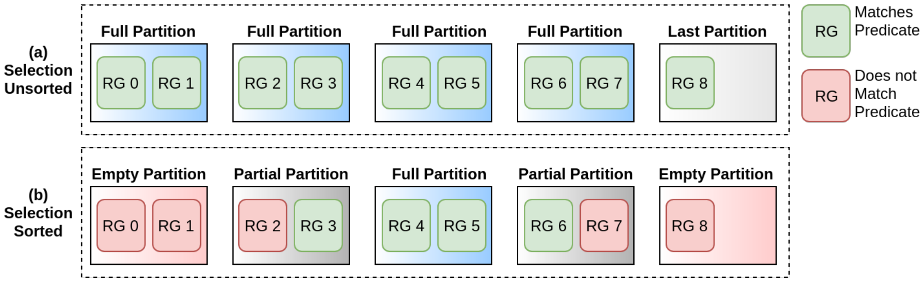

Fig. 1. Type of partitions in selection sorted and unsorted

it has full partitions that contain all matching RGs and two partial partitions, the first (from where selection starts) and last one (where selection ends), because requested data will not start just at the beginning and finish just at the end of a partition, as exemplified in Figure 1b.

3.3 Task's Cost Estimation

The total cost of a task depends on four factors: *initialization cost*, *I/O cost*, *CPU cost*, and *networking cost*. The *initialization cost* is constant and can be determined according to the execution environment. The *I/O cost* depends on the amount of data read within a task and the disk bandwidth. We do not consider *CPU cost* due to its negligible impact compared to *I/O cost* (existing works [2, 11] already proved that this is enough to capture the execution trend). Finally, we focus on the first operation loading data, thus *networking cost* for shuffling is also considered to be zero [2]. However, there is still a networking cost for metadata, because current solutions require to sequentially transfer metadata to all other executors before start processing the data. Typically, it is read and transferred by the master or driver executor.

Each partition has an initialization cost, which is a constant, and I/O cost (which depends on the amount of metadata and data read inside the partition). As shown in Figure 1, *full partitions* read all RGs inside and *partial partitions* only read the matching RGs. Whereas, *last partitions* read the remaining data and *empty partitions* only read metadata. These costs help to estimate the total cost of each task, which can help to estimate the average cost.

3.4 Estimating Makespan

As discussed earlier, each task processes different amounts of data and thus, some tasks can finish earlier compared to others. Likewise, each executor can finish its assigned tasks in different times. Thus, we should estimate makespan based on the executor that is processing largest stack of tasks, which can be estimated using the number of executors active in the last wave. This would help to estimate standard deviation among tasks and used it further for estimating overall makespan of an operation.

For makespan, there are two scenarios based on the number of executors active in the last wave. In the first scenario, there is only one executor in the largest stack. In this case, the last task is processing the remaining data and then, we do not need to take any standard deviation, because there is one single largest stack. Thus, we just add the average duration of all task in that stack. In the second scenario, the makespan depends on metadata transfer, the average cost of a task, the number of executors running in the last wave, and their standard deviation. Thus, we need to estimate expected maximum [4] of those, which accounts for the standard deviation of the addition of tasks, as well as the maximum among executors in the last wave.

3.5 Multi-Objective Optimization

As presented earlier, we would like to optimize two objectives (i.e., makespan and resource usage), which are mutually contradicting, i.e., if we want to reduce makespan, we require more computational resources and vice versa. Thus, we need to find a trade-off between them that satisfies user requirements and constraints. Additionally, to avoid unfavorable or even impossible configurations, we also need to consider three constraints. Firstly, the partition size must always be greater than or equal to the RG size. Secondly, we must have enough partitions to utilize all assigned executors. Finally, it must enforce the maximum limit on the number of executors.

We propose to use an existing multi-objective optimization approach, namely NSGA-II [7], implementing genetic algorithms. It takes objective functions along with constrains as input, and produces the Pareto front as an output. Typically, there is no single optimum in a multi-objective optimization problem, but a Pareto front which contains many potentially optimal solutions depending on user prioritization of one objective or another. Our framework[6] facilitates the user choice by reducing the many possible configurations to very few (belonging or close to the Pareto front), so helping her to select one according to her preferences.

4 Conclusions

Big Data systems process data on a cluster by creating multiple tasks. Typically, they create tasks based on the total size of the table, rather than based on the reading size of the query. Thus, we propose a multi-objective approach based on our extended cost model to configure the number of tasks and executors for a given query based on the reading size. The proposed approach will be implemented as a framework, that automatically configures the number of tasks and executors for a given query.

[6] http://www.essi.upc.edu/dtim/tools/adbis2019

Acknowledgement

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC).

References

1. L. Baldacci and M. Golfarelli. A cost model for Spark SQL. *TKDE*, 31(5):819–832, 2019.
2. H. Bian, Y. Tao, G. Jin, Y. Chen, X. Qin, and X. Du. Rainbow: Adaptive layout optimization for wide tables. In *ICDE*, pages 1657–1660, 2018.
3. H. Bian, Y. Yan, W. Tao, L. J. Chen, Y. Chen, X. Du, and T. Moscibroda. Wide table layout optimization based on column ordering and duplication. In *SIGMOD*, 2017.
4. G. Dasarathy. A simple probability trick for bounding the expected maximum of n random variables. Technical report, Arizona State University, 2011.
5. A. Davidson and A. Or. Optimizing shuffle performance in Spark. Technical report, UC Berkeley, 2013.
6. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
7. K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.
8. A. Gounaris and J. Torres. A methodology for Spark parameter tuning. *Big Data Research*, 11:22–32, 2018.
9. M. T. Islam, S. Karunasekera, and R. Buyya. dSpark: Deadline-based resource allocation for big data applications in Apache Spark. In *e-Science*, pages 89–98, 2017.
10. Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10), 2014.
11. R. F. Munir, A. Abelló, O. Romero, M. Thiele, and W. Lehner. ATUN-HL: Auto tuning of hybrid layouts using workload and data characteristics. In *ADBIS*, pages 200–215, 2018.
12. R. F. Munir, A. Abelló, O. Romero, M. Thiele, and W. Lehner. A cost-based storage format selector for materialization in big data frameworks. *Distributed and Parallel Databases*, 2019.
13. P. P. Nghiem and S. M. Figueira. Towards efficient resource provisioning in MapReduce. *JPDC*, 95:29–41, 2016.
14. P. Petridis, A. Gounaris, and J. Torres. Spark parameter tuning via trial-and-error. In *INNS*, pages 226–237, 2016.
15. K. V. Shvachko. HDFS scalability: the limits to growth. *Login*, 35(2):6–16, 2010.
16. S. Sidhanta, W. M. Golab, and S. Mukhopadhyay. Optex: A deadline-aware cost optimization model for Spark. In *CCGrid*, pages 193–202, 2016.
17. A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *USENIX*, pages 165–186, 2011.
18. W. Wu, W. Lin, C. Hsu, and L. He. Energy-efficient Hadoop for big data analytics and computing: A systematic review and research insights. *Future Generation Comp. Syst.*, 86:1351–1367, 2018.