Optimizing Data Partitioning for Data-Parallel Computing

Qifa Ke, Vijayan Prabhakaran, Yinglian Xie, Yuan Yu
Microsoft Research Silicon Valley

Jingyue Wu, Junfeng Yang
Columbia University

Abstract

Performance of data-parallel computing (e.g., MapReduce, DryadLINQ) heavily depends on its data partitions. Solutions implemented by the current state of the art systems are far from optimal. Techniques proposed by the database community to find optimal data partitions are not directly applicable when complex user-defined functions and data models are involved. We outline our solution, which draws expertise from various fields such as programming languages and optimization, and present our preliminary results.

1. Introduction

Recent advances in distributed execution engines (MapReduce [8], Hadoop [1], and Dryad [12]) and high-level language support (Pig [15], HIVE [2], and DryadLINQ [18]) have greatly simplified the development of large-scale, distributed data-intensive applications. In these systems, data partitioning is used to control the parallelism, and is central for these systems to achieve scalability to large compute clusters. However, the partitioning techniques employed by the systems are very primitive, leading to serious performance problems.

Consider a real example from our previous work [19] that computes some login statistics for each user using a large service log of 270GB (Section 2.1 shows more examples). It is a MapReduce job written as a simple DryadLINQ program:

```
input.GroupBy(x => x.UserId)
     .Select(g => ComputeStatistics(g))
```

The very first task is to partition the data so that computation can be performed on multiple computers in parallel. Many questions naturally arise here. For example, what partition function shall we choose and how many partitions to generate? Despite the simplicity of the program, a naïve hash partitioning of the input into 1000 partitions using `UserId` resulted in skewed workloads and bad performance. When running on a 240-node cluster, the majority of the partitions finished within 1-2 minutes, while the largest partition had 9.08GB data and ran for 1 hour and 13 minutes before it failed with an out-of-memory exception. A user will have to wait for the slowest node to finish (or fail!).

In current data-parallel computing systems, simple hash and range partitioning are the most widely used methods to partition datasets. However, as the systems are being increasingly used for more complex applications such as building large-scale graphs to detect botnets [19] and analyzing large-scale scientific data [13], these naïve partitioning schemes become a major performance problem for the following reasons:

- Partitioning of data using either a hash function or a set of equally spaced range keys often yields unbalanced partitions in terms of data or computation, resulting in bad performance or failures.
- Balanced workload is not the only factor to achieve optimal performance. Another important factor is the number of partitions. There often exists a tradeoff between the amount of computation per partition and the amount of cross-node network traffic (e.g. Example 3 in Figure 1), making it challenging to identify a sweet point.
- In multiple stage computation (e.g., Example 2 in Figure 1), the data or computation skew may occur in later stages. It is often difficult to predict such skews before running the program.
- Even for a same program, the input datasets may change frequently and have different characteristics (e.g., generating statistics from daily service logs), requiring partitioning schemes that adapt to the changing data to achieve optimal performance.

Thus, the research problem we address in this paper is as follows. Given a data-parallel program (e.g., a DryadLINQ or MapReduce program) and a large input dataset, how can we automatically generate a data partitioning plan that optimizes the performance without running the program on the actual dataset? By performance, we broadly refer to a wide range of cost metrics including the number of processes required, CPU time, job latency, memory utilization, disk and network I/O. And our goal is to minimize these costs.

While database community has studied extensively the important problem of reducing data-skews for SQL queries (e.g. [9]), their solutions are not directly applicable because they, in general, do not support complicated data models and arbitrary user-defined functions. Further, they typically assume highly structured data that are indexed and carefully placed so that one can strategically send a small program (SQL query) to the data nodes based on pre-computed data statistics [16].

(1) Computation skew.

```
images.Select(img => ProcessImage(img));
```

(2) Multiple-stage computation.

```
query.Select(x => x.IP).Distinct();
```

(3) Computation per partition vs. cross-node traffic.

```
var records = input1.Apply(x => SelectRecords(x))
                    .HashPartition(x=>x.label, nump);
var output  = input1.Apply(records,
                    (x,y) => ConstructGraph(x,y));
```

Figure 1. Examples of data-parallel programs.

We believe that this research problem remains largely unsolved. So, in this paper we propose a framework that takes a holistic view to explicitly measure and infer various properties of both data and computation. It demands techniques from different domains—including database, programming language, optimization, and systems—to perform cost *estimation* and *optimization*. Estimation infers the cost given a candidate data partitioning plan. Optimization generates an optimal partitioning plan based on the estimated costs of computation and I/O.

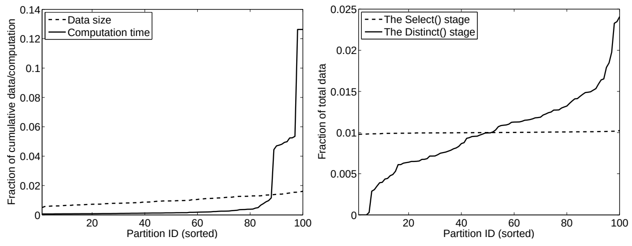
Obviously, performance of a data-parallel program also depends on many other important factors including infrastructure configurations and job scheduling (see Section 4 for discussion). We singled out the data partitioning problem because this is a critical factor that a user can leverage to avoid bad performance. More importantly, we have seen so many jobs failing due to data partitioning problems. We believe any advances in data partitioning would significantly improve the usability of these systems.

2. Background, Examples, and Challenges

A data-parallel program expressed by MapReduce or higher-level languages such as Pig or DryadLINQ is compiled into an execution plan graph (EPG), which is a directed acyclic graph with multiple stages [18]. For each stage, at least one vertex is created to process each input partition; thus multiple vertices can run in parallel to process multiple input partitions.

Data partitioning therefore affects many aspects about how a job is run in the cluster, including parallelism, workload for each vertex, and network traffic among vertices. Below, we present examples from real-world DryadLINQ programs and discuss how data partitioning affects their runtime. All the examples were run on a 240-machine cluster.

Example 1: Computation skew. This example processes image files with a user-defined function `ProcessImage(img)`. The 20GB input data is evenly partitioned into 100 partitions (Figure 2 (a)). However, since some images are more expensive to process than others, the computation is extremely unbalanced. On average, it took 4 minutes to process one partition, but 3 partitions failed after running for 6 hours as they exceeded the 6-hour maximum lease time for the cluster.



(a) Example 1 (b) Example 2

Figure 2. Example 1: data and computation time distribution. Example 2: partition size distribution in multiple stages.

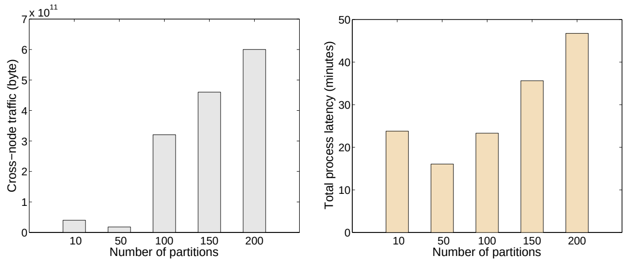


Figure 3. Example 3: Cross-node traffic vs. process time.

Example 2: Multiple stages. This example counts the unique IP addresses of a user-query log. The input data consists of 100 partitions. The first stage selects the IP address of a record using the `Select` operator, and the second stage uses the `Distinct` operator to count distinct IP addresses. Figure 2 (b) shows that even though the input to the `Select` stage is evenly distributed, the input to the `Distinct` stage has significant skews that are difficult to predict beforehand.

Example 3: Computation per partition vs. cross-node traffic. This example takes `(user, IP address)` pairs and constructs a user-user graph, where two user nodes in the graph are connected by an edge if they share an IP address [19]. The program contains two stages. For each partition p_i in the input, we first select `records` of users that are likely to share IP addresses with those in p_i. We then join the original input and the selected `records` to construct the graph. Figure 3 shows that, by increasing the number of partitions, the amount of computation per partition is decreased, but the total network traffic increases. There exists an optimal number of data partitions for minimum job runtime.

2.1. Challenges

Finding an optimal data partitioning for a data-parallel program is challenging. Solutions from parallel database (DB) communities are not directly applicable due to the differences in data and programming models. We list some of the challenges below.

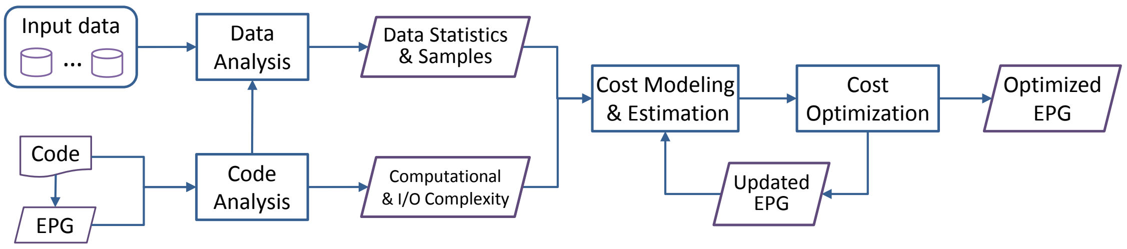


Figure 4. System architecture.

Programming Model. Traditionally, DB systems use a set of predefined operators such as `Select` or `Join`; but they provide little support for user-defined functions (UDF), and some systems do not support UDF at all (e.g., Vertica [3]). In contrast, data-parallel computing often involves arbitrary user-defined functions, which makes it harder to understand how the data is accessed, processed, and transformed to the next computation stage.

Data Model. Compared with parallel DB systems, our data model is also different in terms of how the data is represented, accessed, and stored.

• Unstructured data. DB systems operates on highly structured schema with built-in indices, whereas data-parallel programs compute on unstructured data. Quickly computing data statistics (e.g., key distribution) without data indices is difficult.

• Dynamic datasets. DB systems are better suited at querying static datasets because of the overhead of storing data and building indices. In contrast, data-parallel computing often processes different and new datasets (e.g., daily service logs). Frequent dataset changes require adapting data partitioning schemes accordingly.

• Large intermediate data. To minimize writing intermediate data, parallel DB sends an optimized query plan to all nodes at the beginning of the query [16]. In contrast, data-parallel computing uses disks as communication channels for fault tolerance. How to efficiently analyze the “materialized” intermediate data is less well studied in DB.

3. System Architecture

To optimize data partitioning, we advocate an approach that leverages techniques from many domains. We present the architecture of the system we are building, our early results, and new research opportunities.

3.1. System overview

Figure 4 shows the system architecture. The system first compiles a given data-parallel program into a job execution plan graph (EPG) with initial data partitions (e.g., supplied by the user). The Code Analysis module takes this EPG and the code for each vertex in EPG as input to derive (1) the computational complexity of each vertex program and (2) important data features. This step is important as it not only provides information about the relationship between input data size vs.computational and I/O cost, but also guides the data analysis process, e.g., providing hints to strategically sample data and to estimate data statistics. For example, in Example 1 (Figure 1), it would be desirable to understand what image features (e.g., texture richness, resolution) determine the computational cost. Such information can then be used to identify image records that are expensive to process and distribute them more evenly.

The Data Analysis module linearly scans the data to generate compact data representations. We consider the following data representations:

- a representative sample set for input data;
- data summarizations including the number of input records, data size, etc;
- an approximate histogram of frequent data records;
- the approximate number of distinct keys.

The first two items provide general statistics that are useful for all operators. The histogram of frequent items [7] is important for estimating skews. The number of distinct keys [4] is useful for estimating the output size for a MapReduce job.

The Cost Modeling/Estimation module uses the code and data analysis results to estimate the runtime cost of each vertex including CPU time, output data size, and network traffic. We consider two approaches. The first is a white-box approach that analytically estimates the costs using code analysis results. The second is a black-box approach that empirically estimates the costs by running the job on the sample data and then performing regression analysis on the measured job performance for each vertex. We can further combine these two approaches to improve the estimation accuracy. Once we estimate the cost of each vertex in an EPG, we can identify the critical path (using techniques such as dynamic programming) for estimating the cost of the entire job.

Finally, given the estimated cost of the input EPG, the Cost Optimization module searches for an improved data partitioning plan and generates a new EPG accordingly. The updated EPG can be looped back into the Cost Estimation module for another around of optimization. Using small sample sets allows us to efficiently iterative this process until it converges. The output is the final optimized EPG for execution.

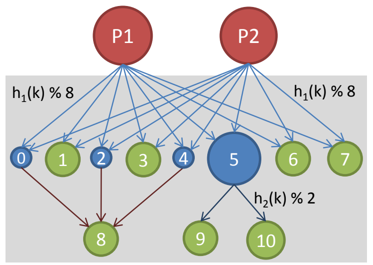


Figure 5. Example of generating an optimized data partitioning scheme represented by the partition graph including two hash functions and a 3-entry ID to ID mapping table.

The entire optimization process can be applied to all stages *offline* before computation starts. We may also couple the optimization process with the computation to dynamically partition data *online*. One solution is to introduce an optimization phase to existing programming models. The data analysis process can thus be piggybacked when the system writes immediate data to disks so that it adds little overhead to the overall computation.

Benchmark	Computation
Clicklog1	`Distinct`
Clicklog2	`GroupBy` + user-defined Reducer
Clicklog3	`GroupBy`
WordTopDocs	`GroupBy` + `Count` Reducer
SkyServer	Scientific computation
UDMap	Preprocessing followed by `OrderBy`
PageRank	Pagerank computation

Table 1. Benchmark program description.

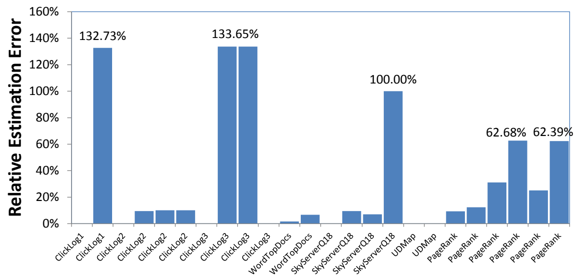


Figure 6. Relative errors of estimated output sizes. Multiple stages in one program are labeled by the same program name.

3.2. Optimizing Execution Plan Graph

We propose a flexible and expressive partitioner for the iterative cost estimation and optimization process. This partitioner derives an optimal partitioning scheme stage by stage for the EPG. It uses a hierarchical partitioning graph, where large partitions are recursively split and small partitions are merged, so that the final partitions are balanced in cost. By balancing the cost at each stage, we essentially minimize the total cost along the critical path and thus the overall cost of the job.

Figure 5 shows an example of generating a partitioning graph. The two root nodes represent two partitions of the sampled input data. The Cost Optimization module inserts an additional partition stage into the current EPG to greedily search for an optimized partitioning scheme. First, the two inputs are split into 8 initial partitions by any existing partitioner (e.g., a hash partitioner $h_1(k)$ mod 8), and the EPG is updated accordingly. (One can try a different number of initial partitions.) The Cost Estimation module then identifies the critical path up to the current stage in the updated EPG, which includes the vertex associated with Partition 5. To reduce cost, it splits Partition 5 into two partitions by another partitioner (e.g., hash partitioner $h_2(k)$ mod 2). Meanwhile, Partition 0, 2, and 4 all have small costs and are merged in order to reduce I/O, the overhead of launching vertices, and thus the potential overall cost.

This process of cost estimation and optimization by recursive data merging and splitting is iterated until it converges. Each iteration is a greedy step towards minimizing the overall cost. The EPG is then updated with the final partitioning scheme (represented by the partitioning graph, two hash functions, and one 3-entry mapping table), and the optimization process continues for the next stage in the EPG. Note that once the partitioning scheme is derived, a data record from the input can be directly assigned to the appropriate data partition, without the intermediate data splitting or merging.

3.3. Preliminary Cost-Estimation Results

To sanity check our framework, we implemented a simple Data Analysis module with uniform sampling, and a Cost Modeling/Estimation module using the black-box approach (see Section 3.1) to estimate output data size and CPU runtime. These two modules together estimate the runtime performance of a given job, which would allow us to apply the optimization algorithm described in Section 3.2.

Figure 6 shows the relative errors in estimating the output data sizes (sampling rate = 0.001) for seven benchmark programs listed in Table 1, which contains 24 stages in total. Each bar in Figure 6 represents the estimated error for one stage. The majority of these 24 stages have relatively small estimation errors ($< 10\%$). We found stages involving the `GroupBy` operator had large estimation errors because uniform sampling could not generate representative samples for `GroupBy`. For these programs, data analysis should be guided by program semantics. An efficient streaming algorithm [4] for counting distinct keys is more suitable in this case.

3.4. Research Opportunities

The design and implementation of each module we discussed also provide many new research opportunities.

Programming-language (PL) analysis: The Code Analysis module can leverage automatic static and dynamic PL analysis techniques (e.g., [10, 11]) to understand both the data model and the semantics of a program in terms of its processing flow, computational complexity, and relevant data features. We can also let users manually specify attributes of UDFs to provide hints for PL analysis. Alternatively, we can predefine a set of callback APIs that users can implement with domain knowledge to explicitly specify important data attributes (e.g., image features) or to define the way to measure computational complexity based on input.

Data analysis: The task of the Data Analysis module is to efficiently derive a compact data representation for cost modeling and estimation. While there exists many efficient (streaming) algorithms for this purpose (e.g., [4, 6, 7]), we also encounter new challenges. First, most existing algorithms are designed to run on a single processor. We need to extend them to a distributed setting. Second, determining what data records are representative depends on the program semantics and we need to strategically sample them accordingly (e.g., leveraging importance sampling techniques [17]). Finally, in multi-stage computation, a representative input sample (or summary) for the first stage may not generate representative outputs to use in later stages (e.g., Example 3 in Figure 1). How to generate data representations for multiple stages is a challenging task.

Optimization: Optimization techniques can used to model cost objectives and help search for improved data partitioning schemes (e.g., [5]). There are two major challenges here. The first is to define a cost model that is expressive and flexible enough to include heterogenous types of costs. For example, a user submitting a job to Amazon EC2 may wish to minimize the total price paid, while programmers who have access to inhouse data centers may want to minimize their job latency. The second challenge is to identify tradeoffs and to provide a spectrum of options for applications with multiple, and possibly conflicting cost objectives (e.g., job running time vs. price budget). While we can optimize for each individual cost dimension, it may be more difficult to identify the relationships across multiple cost dimensions to achieve a user-desired solution.

Systems: The entire framework for optimizing data partitioning is itself a distributed system. We need system components to measure and predict resource consumptions (e.g., CPU utilization, memory usage, disk I/O, and communication traffic). Despite existing techniques (e.g., [14]), accurately deriving costs may also require detailed information about the infrastructure setup. For example, the amount of network traffic highly depends on the switch configuration and processing node locations (e.g., on a same rack or across different racks).

4. Discussion

While finding an ideal partitioning scheme is hard, it is made worse by the network traffic introduced by repartitioning data for more efficient program executions. An interesting tradeoff thus exists between repartitioning a dataset versus running a job on existing partitions.

In addition to a program and its dataset, there are several other factors that affect the run time of a data-parallel program, for example, job scheduling policies and machine configurations. In many cases, optimizing data partitioning can lead to improved job scheduling decisions and resource utilization. For example, if a machine runs slower than the others, we can store a smaller partition to balance the runtime across machines. More importantly, optimizing data partitioning avoids failures and bad performance from a program's perspective and is thus a critical step for preparing data inputs.

References

[1] The Hadoop Project. `http://hadoop.apache.org`.

[2] The HIVE Project. `http://hadoop.apache.org/hive/`.

[3] Vertica. `http://www.vertica.com`.

[4] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD 2007*.

[5] S. Boyd and L. Vandenberghe. Convex optimization. *Cambridge University Press*, 2004.

[6] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD 1998*.

[7] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *VLDB 2008*.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*.

[9] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB 1992*.

[10] S. F. Goldsmith, A. S.Aiken, and D. S.Wilkerson. Measuring empirical computational complexity. In *ESEC/FSE 2007*.

[11] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL 2009*.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Eurosys 2007*.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC 2010*.

[14] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *MobiSys 2003*.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD 2008*.

[16] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD 2009*.

[17] L. Wasserman. All of statistics. *Spring-Verlag, New York*, page 408, 2004.

[18] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI 2008*.

[19] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. BotGraph: Large scale spamming botnet detection. In *NSDI 2009*.