Distributed and Parallel Databases (2019) 37:411–439
https://doi.org/10.1007/s10619-018-7244-2

Abstract cost models for distributed data-intensive computations

Rundong Li[1] · Ningfang Mi[2] · Mirek Riedewald[1] · Yizhou Sun[3] · Yi Yao[2]

Published online: 24 August 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract
We consider data analytics workloads on distributed architectures, in particular clusters of commodity machines. To find a job partitioning that minimizes running time, a cost model, which we more accurately refer to as makespan model, is needed. In attempting to find the simplest possible, but sufficiently accurate, such model, we explore piecewise linear functions of input, output, and computational complexity. They are abstract in the sense that they capture fundamental algorithm properties, but do not require explicit modeling of system and implementation details such as the number of disk accesses. We show how the simplified functional structure can be exploited to reduce optimization cost. In the general case, we identify a lower bound that can be used for search-space pruning. For applications with homogeneous tasks, we further demonstrate how to directly integrate the model into the makespan optimization process, reducing search-space dimensionality and thus complexity by orders of magnitude. Experimental results provide evidence of good prediction quality and successful makespan optimization across a variety of operators and cluster architectures.

Keywords Distributed analytics · Makespan minimization · Cost model · Data partitioning

This work was supported by a Northeastern University (NU) Tier 1 award, by the National Institutes of Health (NIH) under Award Number R01 NS091421, by the Air Force Office of Scientific Research (AFOSR) under Grant Number FA9550-14-1-0160, and by the NSF Career Award under Award Number 1741634. The content is solely the responsibility of the authors and does not necessarily represent the official views of NU, NIH, AFOSR or NSF.

✉ Rundong Li
rundong@ccs.neu.edu

Extended author information available on the last page of the article

1 Introduction

With the ubiquitous availability of clusters of commodity machines and the ease of configuring them in the Cloud, there is growing interest in executing data analytics workloads in distributed environments such as Hadoop MapReduce and Spark. For effective use of resources, a job needs to be partitioned into tasks running in parallel on different workers. We will use the term *worker* to refer to a single processing unit, i.e., a single physical or virtual core. Hence a c-core machine would support up to c concurrent workers.

Given an analytics operator in a data-intensive computation, our goal is to minimize its total execution time by determining (1) *a partitioning of its work,* (2) *the number of tasks these partitions are mapped to, and* (3) *the degree of parallelism for task execution.* We equivalently refer to this execution time as the **makespan** of the corresponding set of tasks.

In contrast to previous work, our goal is to include *operator-specific* partitioning parameters into the optimization process. This is important, because user-defined data processing operators are common in Hadoop and Spark dataflows, and it is often difficult to determine which partitioning-parameter values will result in the fastest job execution time. For illustration, consider a user who wrote a MapReduce program for dense matrix multiplication based on the well-known block partitioning. It partitions the left matrix into B_0-by-B_1 blocks, and the right one into B_1-by-B_2 blocks. (See Sect. 4.4 for details.) In addition to number of tasks and degree of parallelism during execution, the user now also has to choose the best values for B_0, B_1, and B_2. To do so with state of the art approaches, she essentially had two options.

First, she could train a blackbox machine learning model to predict makespan from a variety of features [2], including the block sizes, input size, output size, number of tasks, degree of parallelism, task size variance, and so on. This approach is convenient for the user, because it does not require deep understanding of distributed system interactions. The model can be trained automatically on labeled data, obtained from an appropriate benchmark that measures makespan for a variety of configurations. Unfortunately, finding the minimum-makespan configuration in a blackbox model requires exhaustive trial-and-error probing of the model. While a single prediction might only take a microsecond, exploring all combinations of just 10 different values for 10 parameters would take 10^{10} microseconds, i.e., almost 3 h.

Second, she could explore DBMS cost models, which estimate the cost of an operator as the sum of the number of operations performed, weighted by per-operation cost. These models require a fairly complete understanding of system-level details, e.g., the number of random and sequential I/O performed. Those depend on implementation details of the underlying system and are difficult to specify for makespan estimation in distributed systems. Furthermore, DBMS cost models do not take resource bottlenecks into account.

To address the shortcomings of existing techniques, we propose to generally follow the machine-learning approach, but to do so with the *simplest possible* model type. The model's structure should enable fast makespan optimization, while at the same time being flexible enough to capture a distributed execution "sufficiently" accurately.

Arguably the simplest approach with any hope for being practically useful is to estimate task execution time as a *linear combination* of its input size (I), output size (O), and computation complexity (C) as $c_0 + c_1 I + c_2 O + c_3 C$. The parameters intuitively represent fixcosts (c_0), data transfer rates (c_1, c_2), and processing speed (c_3). This model is *abstract* in the sense that it reflects algorithm properties, not implementation or system aspects. Since the parameters are estimated based on training data obtained from actual benchmark executions on the same cluster, they represent averages over a large number of low-level processing steps and thus automatically account for underlying processing complexities [5].

To use an abstract model like $c_0 + c_1 I + c_2 O + c_3 C$ for makespan optimization, the user has to express I, O, and C as functions of the partitioning parameters of interest. This requires human expertise, but is strictly easier than for traditional DBMS cost models. Note that the resulting function might not be linear in the partitioning parameters. Consider the first map phase of matrix multiplication, for which in Sect. 4.4 we derive map task duration as $c_{1_0} + c_{1_1}(N_0 N_1 + N_1 N_2)/n_1 + c_{1_2}(N_0 N_1 B_2 + N_1 N_2 B_0)/n_1$. All that was needed to obtain this formula were (1) input size per task ($(N_0 N_1 + N_1 N_2)/n_1$) and (2) output size and computation complexity per task ($(N_0 N_1 B_2 + N_1 N_2 B_0)/n_1$). We believe that this represents a relatively small burden, because the program designer has to understand the algorithmic impact of partitioning choices anyway, in order to design an effective distributed program.

This relatively small additional effort for the programmer to reveal high-level algorithm properties to the optimizer pays big dividends in optimization time, compared to simply providing the operator as a blackbox. For example, matrix multiplication has 10 partitioning parameters (Sect. 4.4), requiring exploration of a 10-dimensional space of combinations. Our approach reduces complexity to three dimensions, because for the other seven our model can derive optimal settings analytically. Assuming 10 values explored in each of those 7 dimensions, this reduces optimization cost by a factor of 10^7!

But can an abstract makespan model capture the complexities of a distributed system, in particular **task interactions** and resource **bottlenecks**? Fortunately, any function can be approximated with multiple linear pieces. Our experiments show that for a *piecewise linear* model (Fig. 1), it only takes a small number of pieces to be sufficiently accurate. The reason for this lies in the way resources are consumed. Consider a network link that can transmit data at a certain rate. Ideally, transmitting twice the amount of data should take twice as long. However, in practice greater competition for resources typically increases overhead cost and hence the effective transmission rate may drop. Figure 2 shows a typical observation for a MapReduce program, where the time for shuffling data across the network increases more rapidly after about 600MB. The model can capture this behavior by using a different slope for larger data.

Piecewise linear models also offer two additional benefits. First, the program designer does not need to specify the dependency of I, O, and C on the partitioning parameters overly accurately, *as long as the formula captures the dominating terms*. For instance, for a program whose computation cost is $C = n \log n + n + \sqrt{n}$, it suffices to specify $C = n \log n$—something the programmer is familiar with from traditional O-notation complexity analysis. The only downside is that the model may potentially need more linear pieces to be sufficiently accurate. As a second benefit, the

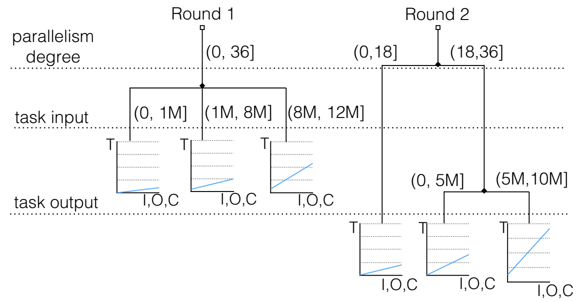


Fig. 1 Schematic illustration of piecewise linear models for a 2-round computation with homogeneous tasks. The model for round 1 is partitioned on task input size only. The model for round 2 is partitioned on both parallelism degree and task output size

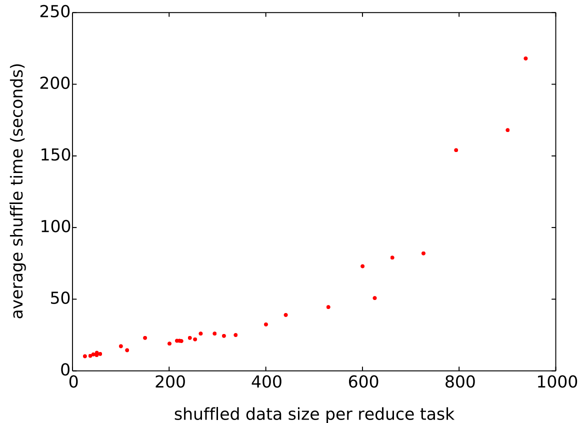


Fig. 2 Shuffle time versus data size (MB) for round 2 of the matrix product algorithm

model pieces provide insights about bottlenecks. For example, for the reduce phase of sorting (Sect. 4.3), model training for a cluster of quad-core machines determined that three pieces were needed when all four cores were used. Input coefficient c_1 had value 5.5, 9.9, and 12 for "small", "medium", and "large" input size, respectively. For executions using only two cores per machine, the model created only two such pieces with c_1 equal to 4.4 for "small", and 4.9 for "large" inputs. Hence it *automatically* captured the I/O-dominated nature of sorting. With four cores competing for data access, larger input size stresses I/O and memory bus more than when only two cores are used.

This work makes the following main contributions:

1. We propose a linear makespan model for the rounds of a data-intensive computation (Sect. 2) and show how it can account for bottlenecks through domain partitioning into a piecewise linear model. For makespan optimization, we show how model structure can be exploited to prune the optimization-parameter search space (Sect. 3).
2. In Sect. 4, we introduce an instantiation of the general model for problems with homogeneous tasks. It enables us to prove even stronger results, significantly reducing the dimensionality of the optimization-parameter search space and thus decreasing optimization cost by orders of magnitude.
3. We present a framework for model training in Sect. 5.

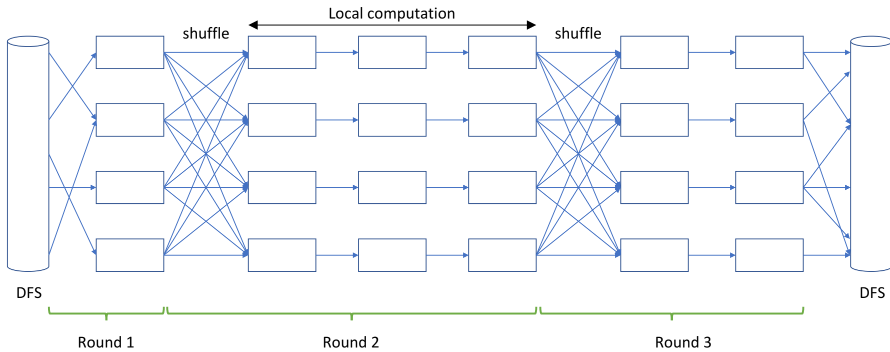


Fig. 3 Distributed data-intensive computation as sequence of rounds, consisting of shuffle followed by local computation. Each box in a column symbolizes a worker

4. We show through extensive experiments (Sect. 6) that the proposed models are sufficiently accurate, i.e., capture the relative makespan behavior for different optimization-parameter settings. This is explored for essential data analytics operators (join, sort, matrix product).

Related work is discussed in Sect. 7, and we conclude in Sect. 8.

2 General model

Despite the diversity of analytics operators, at the system level every distributed data-intensive computation relies on the same basic building block: *local* data processing on multiple worker machines in parallel, preceded by *global* data exchange to get the appropriate input to each worker. In line with nomenclature of modern big-data processing platforms Hadoop MapReduce and Spark, we will refer to the latter as *shuffle phase*; and we will use the term *round* (of computation) for the building block (see Fig. 3). We are interested in the simplest possible, but "sufficiently accurate" cost model for the running time of a round, which we refer to as *makespan model*.

Our proposed function for modeling running time T of a round is defined as

$$T = \beta_0 + \beta_1 \mathcal{I} + \beta_2 I_m + \beta_3 O_m + \beta_4 C_m + \beta_5 \tau_m. \quad (1)$$

β_0 accounts for the fixcosts of starting up the round, which can be significant in a distributed setting. Term $\beta_1 \mathcal{I}$ captures the impact of the *total* amount of input $\mathcal{I}$ to be shuffled and transferred to the workers. For the remaining four terms, notice that computation on the different workers happens in parallel. Hence makespan, in contrast to traditional DBMS cost optimization, is determined by the *most loaded* worker, a.k.a. "straggler". Consequently, the model does not depend on the total input, output, and computation on all workers, but only on the input (I_m), output (O_m), and computation (C_m) on that one straggler. Term $\beta_5 \tau_m$ accounts for the fixcost for starting up and shutting down the τ_m tasks assigned to this worker. This and other important notation is shown in Table 1.

Table 1 Important notation

Variable	Meaning
w	Number of worker nodes in the cluster
p	Degree of parallelism during distributed execution
n	Number of tasks
Z	Set of operator-specific parameters, controlling its partitioning
$\mathcal{I}, \mathcal{O}, \mathcal{C}$	Total input size, output size, and computation cost of a round of computation
I_m, O_m, C_m, τ_m	Input size, output size, computation cost, and number of tasks assigned to the most loaded worker
R, S	Join input relations
A	Join attribute and set of its possible values
R_a, S_a	$R_a = \{r \in R : r.A = a\}$, $S_a = \{s \in S : s.A = a\}$
r_a, s_a	Number of partitions for input R_a and S_a, respectively; $a \in A$

2.1 General practical aspects

Model training happens offline, i.e., before the model can be used for optimization of a given job. It follows the standard approach of supervised learning in general, and linear regression in particular. First, a suite of benchmark jobs is executed, covering a variety of values for model variables $\mathcal{I}$, I_m, O_m, C_m, and τ_m. For each job, round execution time T is recorded, resulting in a 6-tuple $(\mathcal{I}, I_m, O_m, C_m, \tau_m, T)$. Given a set of such tuples, standard least-squares estimation produces the best-fit values of the β-coefficients. Due to the small number of variables, overfitting is not a concern and hence we do not apply regularization. (Our experiments confirm similar prediction accuracy on both training and withheld test data.)

For more details about the model training process, refer to Sect. 5. Training of piecewise linear models is analogous, with the additional step of data-driven determination of a domain partitioning when needed (see Sect. 2.2).

Model use In order to use the proposed model, the programmer has to express variables $\mathcal{I}$, I_m, O_m, C_m, and τ_m as functions of the partitioning parameters she would like to tune. We demonstrate this for three diverse operations below, showing that often the model can also be simplified. It will become clear that our abstract model is much easier to determine than a corresponding DBMS-style cost model based on low-level operations.

In addition to the automatically learned β-coefficients and the user-provided functions for variables $\mathcal{I}$, I_m, O_m, C_m, and τ_m, the optimizer only needs traditional selectivity estimation, identical to the same functionality in a DBMS, in order to

estimate intermediate result size for data processing pipelines consisting of multiple rounds. Then it can estimate the values of $\mathcal{I}$, I_m, O_m, C_m, and τ_m for each round, and simply plug them into Eq. 1 to estimate round time.

Model realism Will this abstract model be sufficiently accurate to be useful? Indeed, it is more powerful than it may at first seem. To see this, consider the following possible concerns.

The shuffle phase does not transfer all data in bulk before the local computation phase—both are usually interleaved. This means that after completing a task belonging to a round, the worker might later process another task of the same round. In that case, the worker requests input data for the new task *after* the completion of the previous one. Equation 1 still applies, because, mathematically, it simply captures the fact that *total* input $\mathcal{I}$ to a round is essential for capturing shuffle cost, no matter the actual interleaving of data transfer and local computation. If data transfer is spread over multiple waves of tasks, then the model might automatically determine a lower value of β_1, i.e., lesser impact of larger total input on makespan.

When input is not evenly balanced across workers, then total input $\mathcal{I}$ might not suffice to explain variations in shuffle time. In that case, term $\beta_2 I_m$ can pick up some of the effects. An analogous argument applies to other low-level system operations. For instance, a map task might have to spill buffer content to disk. The corresponding reading and writing time will be accounted for by the I_m and O_m terms. When larger output causes more frequent buffer spilling, the model can capture this automatically by learning that a larger β_3 value is needed for a model piece covering larger O_m values. (See discussion of piecewise linear models below.) In general, since the β-parameters are estimated from actual benchmark executions, they represent averages over a large number of low-level processing steps. This agrees with recent results by Duggan et al. [5], who showed that a single variable can account for underlying processing complexities in their performance prediction approach.

A MapReduce *combiner* is treated like any other local computation functionality in a round. It affects the user-specified functions for O_m and C_m, as well as the value of $\mathcal{I}$ for the following reduce phase.

We next discuss the two major challenges in making the linear model practically applicable: accounting for interaction effects and determining I_m, O_m, C_m, and τ_m.

2.2 Accounting for task interactions and bottlenecks

Interaction effects occur when tasks executed in parallel on a multicore processor compete for resources, e.g., memory bus and local disk(s). They also occur when multiple machines compete for access to shared network links or switches, slowing down data transfer and local computation. This can be captured by partitioning our model into $k \geq 1$ ranges $(p_0, p_1]$, $(p_1, p_2]$,..., $(p_{k-1}, p_k]$ of degrees of parallelism. *Bottlenecks* appear not only when multiple tasks compete for resources. The local computation of a task might also get delayed by I/O wait time caused by its own I/O operations, requiring different model coefficient values for different ranges of input and output size.

The result of partitioning the design space is a family of piecewise linear models, each with its own combination of values for $(\beta_0, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5)$. We say that a model *covers* the corresponding partition defined by a range of parallelism degrees (p), total input size $(\mathcal{I})$, and input size (I_m) and output size (O_m) on the most loaded worker. The partitioning can be determined in a fully data-driven manner from the training data, e.g., by minimizing the residual sum of squares [30] or by using a model tree [23]. For parallelism degree, we ensure that the number of cores per CPU is considered as follows: For a cluster consisting of w/c c-core machines, all interval endpoints that are multiples of the number of workers, i.e., all values in $\{i \cdot w/c : i = 1, 2, \ldots, c\}$, are explicitly considered as possible split points for a parallelism-degree range. Intuitively, these values correspond to a degree of parallelism of 1 to c per physical machine. Figure 1 illustrates the overall structure of the proposed models. It shows a stylized example for the homogeneous task case, discussed in Sect. 4. For each round of the computation, there is a separate piecewise linear model.

2.3 Estimating max load

Estimating I_m, O_m, C_m, and τ_m, i.e., the input size, output size, computation, and number of tasks on the most loaded worker can be challenging. If the number of tasks in a round is less than or equal to the number of workers, w, then $\tau_m = 1$ and it suffices to identify the "heaviest" individual task. When the number of tasks exceeds w, then some workers will receive multiple tasks and I_m, O_m, C_m, and τ_m will depend on the actual scheduling policy used for assigning tasks to workers.

Notice that schedulers in distributed data-processing systems like MapReduce and Spark actively attempt to balance load at runtime by assigning tasks incrementally. In particular, initially just w tasks will be scheduled—one task per worker. Only after a worker reports completion of a task, will it receive the next. Hence when the number of tasks is "sufficiently" large and load between tasks does not vary "too much", then each worker will receive a similar share of the total load. This implies that one could estimate I_m as total input divided by w, O_m as total output divided by w, and C_m as total computation time divided by w.

Unfortunately, when task load is highly skewed, e.g., one of the tasks accounts for half of the total load, then those averages would result in significant under-estimation. We propose the following general technique for addressing this problem through lightweight *simulation*. Given a set of tasks, one can simply execute the task assignment algorithm used by the scheduler. Task running time is estimated using the model parameters, i.e., for task i with input I_i, output O_i, and computation complexity C_i, the estimate is $\beta_2 I_i + \beta_3 O_i + \beta_4 C_i + \beta_5$.

3 Makespan optimization using the general model

The structure of a linear model provides valuable insights about the importance of the different terms. In particular, the larger the value of a coefficient, the greater the term's impact on makespan. In addition to insights, simple model structure can be exploited

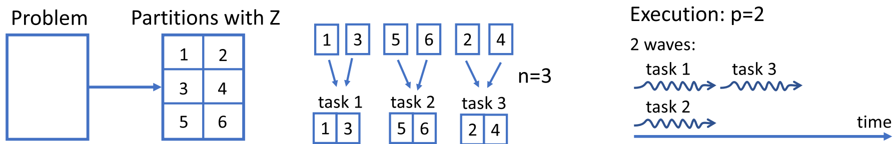


Fig. 4 Relationship between partitioning parameters

to *reduce optimization cost*. We show this for the general model in this section, then discuss even stronger results in Sect. 4.2 for operators with homogeneous tasks.

3.1 Search space exploration

The optimization-parameter search space consists of all combinations of possible values for the tuning parameters of interest. We focus on parameters controlling problem partitioning into tasks, and their parallel execution:

- number of tasks: n,
- degree of parallelism during execution: p ($p \leq w$),
- a set Z of operator-specific parameters controlling problem partitioning.

Figure 4 illustrates the relationship between the parameters. Notice that changing the value of a parameter $z \in Z$ may affect total input, output, and computation cost of the round. For example, a more fine-grained partitioning may require additional input duplicates. On the other hand, different values for n or p do *not* affect total input size ($\mathcal{I}$), total output size ($\mathcal{O}$), or total computational complexity ($\mathcal{C}$)—they only control how the different partitions are "packaged" into tasks and how many tasks are executed concurrently, respectively. We will explain this in more detail for an example in Sect. 3.3.

3.2 Search space pruning

The following lemma will enable us to limit the search space of operator-specific parameters controlling problem partitioning, by establishing a lower bound for makespan of a round. Intuitively, this lower bound corresponds to the (possibly unattainable) ideally balanced load assignment where each worker receives the same number of tasks and the same share of total input, output, and computation. The $\min\{p, n\}$ term accounts for scenarios when the number of tasks (n) is smaller than the degree of parallelism (p): then at most n of the p workers can receive a task.

Lemma 1 *No matter how n tasks of a round with total input $\mathcal{I}$, total output $\mathcal{O}$, and total computation $\mathcal{C}$ are assigned to p concurrent workers, round time T is lower-bounded by* $\bar{T} = \beta_0 + \beta_1\mathcal{I} + \beta_2\frac{\mathcal{I}}{\min\{p,n\}} + \beta_3\frac{\mathcal{O}}{\min\{p,n\}} + \beta_4\frac{\mathcal{C}}{\min\{p,n\}} + \beta_5\frac{n}{\min\{p,n\}}$.

Proof Let I_i, O_i, and C_i denote input, output, and computation for task i, $1 \leq i \leq n$. Then $\mathcal{I} = \sum_{i=1}^{n} I_i$, $\mathcal{O} = \sum_{i=1}^{n} O_i$, and $\mathcal{C} = \sum_{i=1}^{n} C_i$. This implies for the total load induced by all tasks:

$$\sum_{i=1}^{n}(\beta_2 I_i + \beta_3 O_i + \beta_4 C_i + \beta_5) = \beta_2\mathcal{I} + \beta_3\mathcal{O} + \beta_4\mathcal{C} + \beta_5 n.$$

The load on the most loaded worker must be greater than or equal to the average load per worker, i.e., $(\beta_2\mathcal{I} + \beta_3\mathcal{O} + \beta_4\mathcal{C} + \beta_5 n)/p$. Also notice that the "heaviest" task even by itself will induce a load at least as high as the average over all tasks, i.e., $(\beta_2\mathcal{I} + \beta_3\mathcal{O} + \beta_4\mathcal{C} + \beta_5 n)/n$. The load of the worker receiving that task will therefore be lower-bounded by the per-task average as well. This immediately implies the same lower bound for the most loaded worker in the system (whose total assigned load is at least as high as that of the worker receiving the heaviest task). Putting these lower bounds together, we obtain

$$\beta_2 I_m + \beta_3 O_m + \beta_4 C_m + \beta_5 \tau_m \geq \\ \max\left\{\frac{\beta_2\mathcal{I} + \beta_3\mathcal{O} + \beta_4\mathcal{C} + \beta_5 n}{p}; \frac{\beta_2\mathcal{I} + \beta_3\mathcal{O} + \beta_4\mathcal{C} + \beta_5 n}{n}\right\},$$

and hence $T \geq \frac{\beta_2\mathcal{I}+\beta_3\mathcal{O}+\beta_4\mathcal{C}+\beta_5 n}{\min\{p,n\}}$, completing the proof of the lemma. □

Lemma 1 can be exploited for search space pruning as follows. For given task number n and degree of parallelism p, we immediately obtain a lower bound $\bar{T}$ based on total inherent input size, inherent output size, and inherent computation of a round. The *inherent* values are those for the unpartitioned execution of the operator. Partitioning can never decrease them, but will typically increase them, e.g., require additional input copies or additional computation steps for post-processing.

Whenever more fine-grained partitioning increases $\mathcal{I}$, $\mathcal{O}$, or $\mathcal{C}$, the corresponding value of lower bound $\bar{T}$ will increase accordingly. Hence exploration of even more fine-grained partitioning can be terminated as soon as the lower bound exceeds the best makespan found so far. We illustrate this for joins in Sect. 3.3. Even when more fine-grained partitioning does not increase the lower bound, it still provides valuable information. Knowing that the best makespan found so far is within a small factor of the lower bound, the user may decide to stop exploration early.

Note that when applying Lemma 1 to piecewise linear models, each piece could return a different lower bound. The entire model's lower bound is the minimum of the per-piece lower bounds, over all pieces that could still be reached during the optimization-parameter space exploration. (For instance, if application parameter settings are explored in increasing order of total input size $\mathcal{I}$, then model pieces for smaller input size ranges do not need to be considered for the lower bound.)

3.3 Example: equi-join

Consider equi-join $R \bowtie S = \{(r, s) \in R \times S : r.A = s.A\}$ and let $R_a = \{r \in R : r.A = a\}$ and $S_a = \{s \in S : s.A = a\}$ be the subsets of tuples from R and S, respectively, with join attribute value a. We will refer to $R_a \cup S_a$ as the *group* for join attribute value a. Then the equi-join can be expressed as $R \bowtie S = \bigcup_{a\in A} R_a \times S_a$, i.e., the union of Cartesian products for each group.

Algorithm 1 : GreedyJoinPartition

Input: count statistics $[|R_a|]_{a\in A}$, $[|S_a|]_{a\in A}$
Input: starting partitioning $[(r_a = 1, s_a = 1)]_{a\in A}$
Input: weight function `weight`
1: **while** termination condition not met **do**
2: Increment the r_a or s_a that maximizes the ratio of benefit and cost
3: // Benefit = load variance reduction when incrementing the corresponding r_a or s_a
4: // Cost = weight assigned by `weight` to the corresponding r_a or s_a
5: Determine $\mathcal{I}$, I_m, and O_m for the new partitioning
6: Evaluate makespan model $T = \beta_0 + \beta_1\mathcal{I} + \beta_2 I_m + \beta_3 O_m$ for the new $\mathcal{I}$, I_m, O_m
7: **return** partitioning $[(r_a, s_a)]_{a\in A}$ with lowest predicted makespan

For skewed input, some groups are significantly larger than others, causing load imbalance and hence a delay in job completion. Skew can be addressed by splitting large groups into smaller sub-groups, e.g., using rectangular partitioning. More formally, the set Z of operator-specific partitioning parameters is defined as the set of integer pairs $\{(r_a, s_a) : r_a \geq 1, s_a \geq 1, a \in A\}$, or $[(r_a, s_a)]_{a\in A}$ for short. The best partitioning algorithm to date by Li et al. [19] explores Z by greedily incrementing the r_a or s_a that maximizes a benefit-cost ratio based on load variance reduction versus additional input duplication due to subgroup partitioning (Algorithm 1). Note that the algorithm relies on a simplified version of Eq. 1: There the term for C_m is omitted, because computation cost is linear in input and output; and the $\beta_5\tau_m$ term collapses into β_0, because the number of tasks is set equal to the degree of parallelism and hence $\tau_m = 1$.

[19] determines the values for $\mathcal{I}$, I_m, and O_m in line 5 by executing a load assignment strategy such as random or LLD when packing the (sub) groups into tasks. Since the makespan model in Algorithm 1 is a special case of our general model (Eq. 1), we can leverage Lemma 1 to terminate the loop in a principled way. More precisely, it is easy to see that with increasing r_a and s_a, the lower bound $\bar{T}$ will increase because $\mathcal{I}$ keeps increasing due to the additional input duplicates. Hence the while-loop can be terminated safely (i.e., with the guarantee that no better makespan can be found for more fine-grained partitioning) as soon as the lower bound exceeds the predicted makespan for the best partitioning found so far.

4 Homogeneous tasks

This section presents analytical results that enable a significantly greater reduction in optimization cost for a class of problems where all tasks have "similar" load. We refer to these as *homogeneous* tasks. Task homogeneity occurs frequently in practice, typically by design, because the programmer attempts to distribute load evenly over the workers. For example, for distributed sorting, input is range-partitioned based on (approximate) quantiles, so that each partition receives about the same amount of data. Even for equi-joins, hash partitioning often distributes load fairly evenly as long as groups are not "overly skewed."

4.1 Makespan model for homogeneous tasks

In the homogeneous model, each of the n tasks handles approximately $1/n$ of the total input, output, and computation. Schedulers also can easily balance load across workers, assigning about n/p tasks to each of the p workers. When n is not divisible by p, the most loaded worker will receive $\lceil n/p \rceil$ of the tasks. Together with Eq. 1, we obtain makespan $\beta_0 + \beta_1 \mathcal{I} + \lceil \frac{n}{p} \rceil \left(\beta_2 \frac{\mathcal{I}}{n} + \beta_3 \frac{\mathcal{O}}{n} + \beta_4 \frac{\mathcal{C}}{n} + \beta_5 \right)$.

We propose to further simplify this formula by dropping the first two terms, resulting in the following makespan model H for homogeneous tasks:

$$H = \left\lceil \frac{n}{p} \right\rceil \left(\beta_2 \frac{\mathcal{I}}{n} + \beta_3 \frac{\mathcal{O}}{n} + \beta_4 \frac{\mathcal{C}}{n} + \beta_5 \right) \tag{2}$$

Notice that dropping terms does not make the model "less correct," but simply reduces its flexibility in capturing real-world behavior. For instance, β_0 is a per-job fixcost, while β_5 represents per-task fixcost. Without β_0 in the formula, the model can implicitly account for the effect of β_0 by increasing β_5. The same applies to $\beta_1 \mathcal{I}$ and $\beta_2 \mathcal{I}/n$, which both model a dependency of makespan on input $\mathcal{I}$. Our experiments will show that the resulting models are sufficiently accurate for makespan optimization.

Analogously to the discussion in Sect. 2.2, we propose a piecewise linear model to account for task interactions and bottlenecks. For Eq. 2, partitioning is considered for parallelism degree (exactly as for the general model), per-task input size, and per-task output size. Figure 1 shows a stylized example, with 1-dimensional lines as stand-in for a plane in 3-dimensional space.

4.2 Makespan optimization for homogeneous tasks

The following powerful lemma enables us to derive the optimal task number and parallelism degree for all applications with homogeneous tasks. Recall that in a piecewise linear model, each piece covers some range $(p_l, p_h]$ for parallelism degree, range $(i_l, i_h]$ for input, and range $(o_l, o_h]$ for output. Intuitively, the lemma states that parallelism degree should be set to the largest value possible for the linear piece, i.e., p_h. And the number of tasks, n, should be set to the smallest possible multiple of p_h that is allowed based on the input and output range constraint for the linear piece. Note that changing the number of tasks affects the input and output per task, i.e., for some values of n, $\mathcal{I}/n$ or $\mathcal{O}/n$ might not be inside range $(i_l, i_h]$ and $(o_l, o_h]$, respectively. If n cannot be set to a multiple of p_h, then it should be set to the largest possible value allowed for this linear piece. In the special case of the model being a single linear piece, the lemma implies that both parallelism degree p and number of tasks n should be set to w. This makes perfect sense, because in the absence of non-linear behavior, it is best to use all machines and to achieve this with the smallest number of tasks possible (i.e., one task per worker).

Lemma 2 *Let* $H = \lceil \frac{n}{p} \rceil \left(\beta_2 \frac{\mathcal{I}}{n} + \beta_3 \frac{\mathcal{O}}{n} + \beta_4 \frac{\mathcal{C}}{n} + \beta_5 \right)$ *be a makespan model covering range* $(p_l, p_h]$ *for parallelism degree, range* $(i_l, i_h]$ *for input, and range* $(o_l, o_h]$ *for*

output. Then H is minimized by setting $p = p_h$ *and* $n = \min\{\lceil n_l/p_h \rceil p_h; n_h\}$*, where* $n_l = \max\{\lceil \mathcal{I}/i_h \rceil; \lceil \mathcal{O}/o_h \rceil\}$ *and* $n_h = \min\{\lfloor \mathcal{I}/(i_l+1) \rfloor; \lfloor \mathcal{O}/(o_l+1) \rfloor\}$.

Proof First consider the constraints on the number of tasks, n, imposed by the range for input and output size. For input, per-task input $\mathcal{I}/n$ has to fall into range $(i_l, i_h]$, which implies $\mathcal{I}/i_h \leq n < \mathcal{I}/i_l$; and analogously $\mathcal{O}/o_h \leq n < \mathcal{O}/o_l$. Together, and taking into account that n has to be integer, this yields $n \in [n_l, n_h]$, where $n_l = \max\{\lceil \mathcal{I}/i_h \rceil; \lceil \mathcal{O}/o_h \rceil\}$, and $n_h = \min\{\lfloor \mathcal{I}/(i_l+1) \rfloor; \lfloor \mathcal{O}/(o_l+1) \rfloor\}$.

To find the value of p that minimizes $H = \lceil n/p \rceil (\beta_2 \mathcal{I}/n + \beta_3 \mathcal{O}/n + \beta_4 \mathcal{C}/n + \beta_5)$, notice that none of the terms other than n/p is affected by the choice of p, and that $\lceil n/p \rceil$ in monotonically non-decreasing in p. Hence the optimal choice for p is the largest value possible, i.e., $p = p_h$. This implies that we are left to determine the value of n that minimizes $H(p = p_h) = \lceil n/p_h \rceil (\beta_2 \mathcal{I}/n + \beta_3 \mathcal{O}/n + \beta_4 \mathcal{C}/n + \beta_5)$. To deal with the ceiling function, we separate the problem into two cases.

Case 1 the range of possible values for n contains a multiple of p_h. We show that the smallest such multiple minimizes H. Formally, the case condition states that there exists an integer $k \geq 1$ such that $n_l \leq kp_h \leq n_h$. For any such k, consider all $n \in [n_l, n_h]$ with $\lceil n/p_h \rceil = k$, i.e., all n that satisfy $(k-1)p_h < n \leq kp_h$. For these values of n, let $H_k = k(\beta_2 \mathcal{I}/n + \beta_3 \mathcal{O}/n + \beta_4 \mathcal{C}/n + \beta_5)$. Note that $\mathcal{I}$, $\mathcal{O}$, and $\mathcal{C}$ are not affected by the choice of n. The problem partitioning is controlled by the operator-specific partitioning parameters in set Z; the choice of n only determines how these partitions are grouped into tasks. As a consequence, H_k is minimized by choosing the largest n in $(k-1)p_h < n \leq kp_h$, i.e., $n = kp_h$.

We now determine the optimal choice for k. For $n = kp_h$, $H = \lceil kp_h/p_h \rceil (\beta_2 \mathcal{I}/(kp_h) + \beta_3 \mathcal{O}/(kp_h) + \beta_4 \mathcal{C}/(kp_h) + \beta_5)$, which simplifies to $H = (\beta_2 \mathcal{I} + \beta_3 \mathcal{O} + \beta_4 \mathcal{C})/p_h + k\beta_5$. Since $(\beta_2 \mathcal{I} + \beta_3 \mathcal{O} + \beta_4 \mathcal{C})$ is not affected by the choice of n, this function is minimized for the smallest possible k, i.e., for $k = \lceil n_l/p_h \rceil$ and hence $n = \lceil n_l/p_h \rceil p_h$.

Case 2 the range of possible values for n *does not* contain a multiple of p_h. Then there exists an integer $k' \geq 1$ such that $(k'-1)p_h < n_l \leq n_h < k'p_h$. This implies $\lceil n/p_h \rceil = k'$ for *all* values of n in $(n_l, n_h]$. Like for case 1, this function is minimized for the largest possible choice of n, i.e., n_h.

To combine the solutions derived for both cases, note that in case 1, $\lceil n_l/p_h \rceil p_h \leq n_h$. For case 2, the case condition implies $\lceil n_l/p_h \rceil = \lceil n_h/p_h \rceil$. Together with $\lceil n_h/p_h \rceil \geq n_h/p_h$ (by definition of the ceiling function), this implies $\lceil n_l/p_h \rceil \geq n_h/p_h$ and hence $\lceil n_l/p_h \rceil p_h \geq n_h$. Hence setting $n = \min\{\lceil n_l/p_h \rceil p_h; n_h\}$ will minimize H, no matter which of the two cases applies. □

4.3 Example: sorting

Sorting plays a central role in data analysis, therefore we first demonstrate how to apply abstract piecewise linear makespan models to the classic sort algorithm in Hadoop MapReduce. Consider a user who is satisfied with the Hadoop defaults for the map phase (one map task per file chunk, assign tasks to all workers), but would like to optimize the reduce phase. She performs the following analysis to leverage our approach.

Algorithm 2 : Find p_r and n_r that minimize H_r of sorting

Input: N; M = set of models H_r, each covering some range $(p_l, p_h]$ of parallelism degrees, range $(i_l, i_h]$ of reduce-task input sizes, and range $(o_l, o_h]$ of reduce-task output sizes

1: **for all** $m \in M$ **do** // Model piece m covers $(p_l, p_h]$, $(i_l, i_h]$, $(o_l, o_h]$
2: $t \leftarrow$ time returned by model m when setting p_r and n_r according to Lemma 2
3: Keep track of smallest t
4: Return minimal time t and its (p_r, n_r) combination

Let N denote the size of the input data. The map phase only shuffles the input, hence reduce phase input size is $\mathcal{I} = N$. Its output is the sorted data set, resulting in $\mathcal{O} = N$. Reduce tasks simply merge pre-sorted runs they receive from the mappers, therefore $\mathcal{C} = N$ as well. Note that each reduce task is responsible for a range of keys. A good implementation creates q ranges based on (approximate) q-quantiles, therefore $Z = \{q\}$ is the set of operator-specific parameters. Note that in order to generate n_r reduce tasks, one simply sets $q = n_r$. With p_r denoting parallelism degree in the reduce phase, this analysis implies for Eq. 2:

$$H_\mathrm{r} = \left\lceil \frac{n_r}{p_r} \right\rceil \left(\beta_2 \frac{N}{n_r} + \beta_3 \frac{N}{n_r} + \beta_4 \frac{N}{n_r} + \beta_5 \right) = \left\lceil \frac{n_r}{p_r} \right\rceil \left(c_{r_0} + c_{r_1} \frac{N}{n_r} \right),$$

where $c_{r_0} = \beta_5$ and $c_{r_1} = \beta_2 + \beta_3 + \beta_4$. Note how terms for variables with the same function *collapse* in the linear model.

Overall, the user only had to select the homogeneous-task case and specify $\mathcal{I} = \mathcal{O} = \mathcal{C} = N$. Then our approach automatically solves $\mathrm{argmin}_{n_r, p_r} H_\mathrm{r}$. Using Lemma 2, the optimizer immediately derives the optimal settings of p_r and n_r for each piece of the piecewise linear model, selecting the pair with the lowest predicted makespan as the global winner. Details are shown in Algorithm 2. Instead of exhaustively exploring many (n_r, p_r) combinations, optimization cost is *linear in the number of model pieces*. Using a larger number of linear pieces improves model accuracy, but increases optimization cost—a directly tunable tradeoff.

To appreciate how the optimization process takes task interactions and bottlenecks into account, consider first the special case where the model consists of a single linear piece covering parallelism degrees $(0, w]$, input size $(0, x]$, and output size $(0, x]$, for some sufficiently large $x > N$. The for-loop in Algorithm 2 would be executed once, returning $p_r = w$ and $n_r = \min\{w; N\} = w$. (Note that $N/x < 1$ and we assume $N \geq w$, i.e., the number of workers does not exceed the number of input records.) Stated differently, the algorithm determines that the problem should be partitioned into w tasks—one per worker—and all tasks should be executed in a single wave in parallel.

Now consider a cluster of $w/2$ dual-core machines and assume that when using both cores on a worker, the memory bus on the worker slows down data transfer rate from memory to core, causing the cores to wait for data. During model training, our approach would automatically determine from the training data that two different linear models are needed: one covering parallelism degree $p_r \in (0, w/2]$, and the other $p_r \in (w/2, w]$. The for-loop in Algorithm 2 now compares predicted makespan for two configurations of (p_r, n_r): $(w/2, w/2)$ for the model covering $p_r \in (0, w/2]$

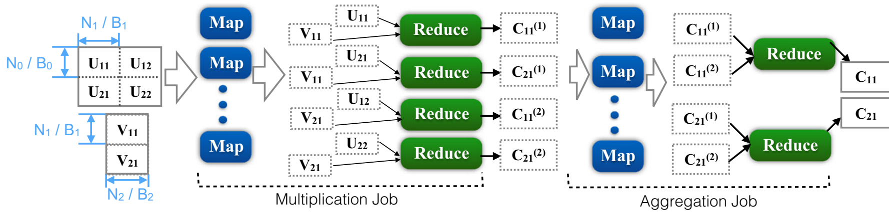


Fig. 5 Block-wise parallel matrix multiplication in 4 rounds. U is partitioned into 2×2 blocks, V into upper and lower half, i.e., $(B_0, B_1, B_2) = (2, 2, 1)$

and (w, w) for the model covering $p_r \in (w/2, w]$. Stated differently, if the memory-bus bottleneck leads to a severe slowdown, the optimal solution may be to use only half of the cores—one per machine—and execute the reduce phase in a single wave of $w/2$ concurrently executed tasks. This perfectly captures the intuition that if the memory bus is the bottleneck (and not the CPU), then it may be better to only use one of the two cores per machine.

4.4 Example: dense matrix product

Dense matrix multiplication represents a more challenging workload with high data transfer costs, but also significant CPU load in some rounds due to the large number of multiplications and additions. Furthermore, matrix partitioning increases total cost due to data replication. Dense matrix multiplication was identified as an important computation problem in a recent UC Berkeley survey on the parallel computing landscape [3].

Consider a programmer who implemented the classic block-partitioning algorithm for dense matrix-matrix multiplication in MapReduce. As illustrated in Fig. 5, input matrix U with dimensions $N_0 \times N_1$ is partitioned into $B_0 \cdot B_1$ blocks, each of size N_0/B_0 by N_1/B_1; V (with dimensions $N_1 \times N_2$) is partitioned into $B_1 \cdot B_2$ blocks, each of size N_1/B_1 by N_2/B_2. Each block from U will be multiplied with the B_2 corresponding blocks from V, for a total of $B_0 \cdot B_1 \cdot B_2$ block-pair multiplication tasks. Note that each U block is duplicated B_2 times, each V block B_0 times. The data duplication (map: round 1) and local multiplication (reduce: round 2) form the **multiplication job (m-job)**. If $B_1 > 1$, then each block-pair product represents only a partial result. In that case an **aggregation job (a-job)** needs to read and re-shuffle these partial results (map: round 3) and sum them up (reduce: round 4).

Based on this understanding of the computation, the programmer now proceeds as discussed for the sort program, expressing input, output, and computation in terms of the partitioning parameters. In addition to p_i and n_i, $i \in \{1, 2, 3, 4\}$—the parallelism degree and number of tasks in each of the four rounds—this includes the operator-specific partitioning parameter set $Z = \{B_0, B_1, B_2\}$. Note that $n_2 = B_0 B_1 B_2$ according to the above analysis.

The resulting per-round makespan models are as shown below. For readability, we present the version with collapsed terms for variables with identical functions. (Note that rounds 3 and 4 are executed if and only if $B_1 > 1$.)

$$H_1 = (c_{1_0} + c_{1_1}(N_0N_1 + N_1N_2)/n_1 + c_{1_2}(N_0N_1B_2 + N_1N_2B_0)/n_1) \cdot \lceil n_1/p_1 \rceil,$$
$$H_2 = (c_{2_0} + c_{2_1}(\frac{N_0N_1}{B_0B_1} + \frac{N_1N_2}{B_1B_2}) + c_{2_2}\frac{N_0N_2}{B_0B_2} + c_{2_3}\frac{N_0N_1N_2}{B_0B_1B_2}) \cdot \lceil B_0B_1B_2/p_2 \rceil,$$
$$H_3 = (c_{3_0} + c_{3_1}N_0N_2B_1/n_3) \cdot \lceil n_3/p_3 \rceil,$$
$$H_4 = (c_{4_0} + c_{4_1}N_0N_2B_1/n_4 + c_{4_2}N_0N_2/n_4) \cdot \lceil n_4/p_4 \rceil.$$

The problem partitioning that minimizes estimated makespan is defined as $\text{argmin}_{B_0,B_1,B_2,p_1,p_2,p_3,p_4,n_1,n_3,n_4} H_1 + H_2 + H_3 + H_4$. With traditional cost models, this would require trial-and-error exploration of a *10-dimensional search space*. Using our approach, we can again leverage Lemma 2 to derive optimal settings for all parallelism degrees and task numbers. Hence the optimization problem simplifies to

$$\underset{B_0,B_1,B_2}{\text{argmin}}\ H_1 + H_2 + H_3 + H_4. \tag{3}$$

This reduces optimization cost by *orders of magnitude*, from search in 10 dimensions to 3 dimensions. (Note that optimization cost is linear in the total number of linear pieces, across all rounds.)

5 Model training

Recall the basic approach to model training as introduced in Sect. 2.1: A suite of profiling benchmarks is executed on the cluster, producing a labeled set of training records. Each record is a tuple of values for the model variables (input, output, computation, number of tasks), and its label is the corresponding observed makespan. The values of the β-parameters are determined from the labeled data using the standard least squares approach. While this follows standard machine learning practice, there are a few subtleties in the specific context of our problem.

Representative training data Mainstream supervised learning methods, including linear regression, assume that training data is drawn from the same distribution as the "unseen" data, i.e., inputs for which the model will be used to predict the label. For our makespan model, this means not only should the profiling benchmarks include a variety of input sizes, output sizes, and computation costs, but they also need to be executed for a *variety of parallelism degrees*. Furthermore, we propose *on-the-fly* model fitting based on the following two observations: First, due to the small number of variables and the simple model structure, it does not take more than a few dozen training records to estimate the model coefficients. Second, we are dealing with a relatively simple function surface. Makespan is (approximately) monotonically increasing in input size, output size, and computation cost. Hence the best training records for predicting the makespan of an unseen data point are those "closest" to it. When predicting makespan for a given configuration at runtime, we therefore select

Table 2 Cluster specifications

Name	#Machines	#Cores per machine	#Workers	Memory per machine (GB)	Software
9h36	10	4	36	8	Hadoop 1.2
2h24	3	12 (virtual)	24	47	Hadoop 2.4
20h160	21	8	160	64	Hadoop 2.4
7h14	8	2	14	8	Hadoop 1.2
Emr$X$	$X+1$	1 (virtual)	X	3.75	Hadoop 2
6s12	7	2	12	8	Spark 1.6.1
Emr12s	7	2 (virtual)	12	7.5	Spark 1.6.1

the 30 most similar training records and determine the β-coefficients on-the-fly from these 30 training records. On-the-fly model fitting takes only milliseconds.

A surrogate measure for makespan In a distributed execution, determining start and end time of a round requires careful measurement and solid understanding of system-internals. (Notice that this is a challenge for the system admin, not for users and application developers!) On the other hand, it is fairly straightforward to measure running times of individual tasks. We therefore are interested in exploring if our makespan models are still usable if actual makespan measurements in the training data are replaced by a surrogate measure based on individual task running times. In particular, for the homogeneous-task case we propose as a surrogate measure the product of number of task waves $\lceil n/p \rceil$ and average task running time. Our experiments in Sect. 6.2 show, that it indeed works very well.

Determining the value of C_m Values for input size $\mathcal{I}$ of a round, as well as total input size I_m, output size O_m, and number of tasks τ_m on the most loaded worker are easy to observe during benchmark execution. For computation cost C_m, we have to apply the user-provided function to the observed variables the function depends on. Consider an operator where a partition's computation complexity is $n \log n$ for input of size n. If the most loaded worker received two partitions whose input sizes are n_1 and n_2, respectively, then we record $C_m = n_1 \log n_1 + n_2 \log n_2$.

Simpler model for collapsed terms Recall from the analysis of sorting and matrix product, how model terms collapse when variables are expressed by the same function. To exploit this, one can fit a model in the low-dimensional space. This enables use of smaller training sets.

6 Experiments

We implemented all algorithms in Hadoop MapReduce or Spark, and conducted experiments on eleven different systems with diverse properties. They include in-house clusters (9h36, 2h24, 6s12, 7h14), a research cluster (20h160) provided by CloudLab [32] and EMR clusters with various sizes (Emr$X$, where $X =$ 10, 20, 30, 40, 50, and Emr12s) on Amazon Web Services. For details see Table 2.

For simplicity, in most experiments on Hadoop, the number of map tasks is left at the default value, i.e., total map input size divided by Hadoop Distributed File System (HDFS) block size. Only for small data sets whose size is smaller than the product of desired parallelism degree and HDFS block size, we set the number of map tasks equal to the desired parallelism degree.

The design of the profiling benchmark for model-parameter fitting is an open challenge. For applications where the same queries are periodically re-executed, e.g., social network analysis as the network evolves, it would be beneficial to execute a specialized benchmark containing only those queries, and covering a narrow range of input and output sizes based on current and near-future graph properties. On the other hand, for an infrequently used custom operator, it might not be feasible to include it in the benchmark at all. Optimal profiling-benchmark design is beyond the scope of this paper. For proof-of-concept of our approach, we need to show that for some "reasonable" benchmark, the resulting model is effective for makespan optimization. Note also that the system can collect free training data each time an operator is executed during data analysis. Since join, sorting, and matrix product are common operators, it is reasonable to assume that dozens of training records for them exist. In the experiments, we then fit the collapsed models (e.g., $H_\mathrm{r} = \lceil n_r/p_r \rceil (c_{r_0} + c_{r_1} N/n_r)$ for reduce phase of sorting as discussed in Sect. 4.3) to the training records from the same operator. To avoid overly optimistic results, we ensure that there are no more than about 100 training records and they cover a wide variety of values for model variables. Furthermore, only simple synthetic data is used for profiling. This way predictions made for other synthetic data distributions and for real data are a true test how well the model extrapolates to distributions it has never seen before.

6.1 General model

We first study the general model (Eq. 2), applied to join and sorting.

Queries `JOIN` computes the full equi-join on the join key, emitting all result tuples. `JOIN-AGG` computes an equi-join whose results are aggregated on-the-fly as they are generated. (We compute the sum over a non-join attribute.) Only a single output tuple is emitted for each join group. `Sorting` sorts a set of integers in increasing order.

Data For joins, we use both synthetic and real data sets. `Zipf`-$n - z$ denotes a pair of synthetic data sets with Zipf-distributed join attribute, with skew parameter z. If the two inputs have different z, we include both, e.g., Zipf-5m-[1,0] indicates that one data set has $z = 1$, the other $z = 0$. For real-world data sets, skew often falls between 0.25 and 1.0, where $z = 0$ results in uniform distribution. `cloud`-5m denotes a pair of real data sets containing 5 million tuples randomly sampled from a set of cloud reports [12]. They are joined on latitude, which was quantized into 10 equi-width bins to model a climate-zone based correlation analysis. `ebird-all` is another real data set containing 1.89 million bird sightings, each with 1657 attributes describing properties of observation event, climate, landscape, etc [22]. `ebird-basic` is the same set, but with only the 953 most important columns. For both eBird data sets, we compute the self-join on three Boolean attributes, capturing presence (yes or no) of the

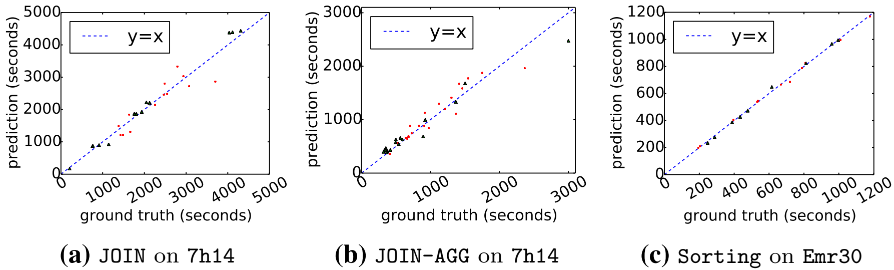


Fig. 6 General model: predicted versus measured makespan. Both training (red dots) and test cases (green triangles) are near the blue dotted perfect-prediction line (Color figure online)

top-3 most frequently reported bird species in North America. This was motivated by correlation studies exploring habitat properties based on species appearance patterns.

These data sets cover a wide range of values of $\mathcal{I}$, I_m and O_m. Specifically, for JOIN, we have $\mathcal{I} \in [10^6, 2 \times 10^9]$, $I_m \in [10^5, 5 \times 10^8]$ and $O_m \in [9 \times 10^7, 2 \times 10^9]$; for JOIN-AGG, we have $\mathcal{I} \in [5 \times 10^8, 2 \times 10^{11}]$, $I_m \in [2 \times 10^8, 4 \times 10^9]$, and $O_m \in [10^{10}, 5 \times 10^{11}]$. The profiling benchmark consists of 100 join queries (20 per parallelism degree; covering degrees 10, 20, 30, 40, and 50), executed on simple synthetic data, each generated as follows. First, we draw a value for $\mathcal{I}$, I_m, and O_m uniformly at random from the above ranges. Then we create a pair of join inputs with the following properties: There are p different join keys such that the largest group has input and output size I_m and O_m, respectively. The other $p - 1$ join groups are equal to each other in input and output per group, selected so that total input and output size reach the target values. Despite being trained on these simple data distributions, the makespan model is very accurate for the Zipf and the real data as discussed below.

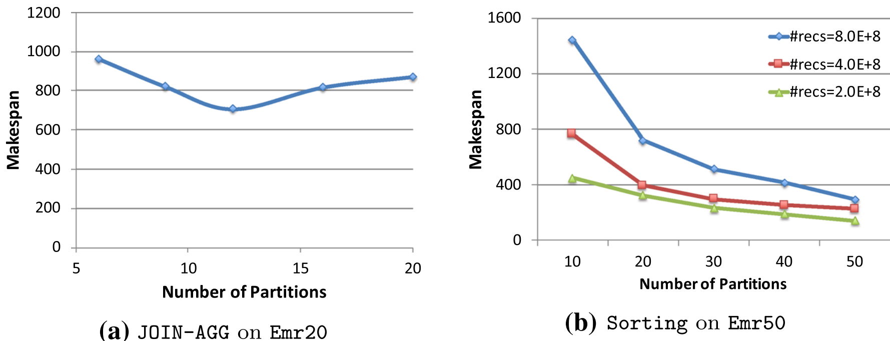
For Sorting, we create data sets consisting of 100 million to 1.2 billion randomly generated numbers of type Long (8 bytes per record) for training and testing on Emr$X$, $X \in \{10, 20, 30, 40, 50\}$. These data sets result in $\mathcal{I} \in [10^8, 1.2 \times 10^9]$ and $I_m = O_m \in [2 \times 10^6, 1.2 \times 10^8]$. The profiling benchmark is generated as discussed for joins by randomly drawing 50 configurations from the ranges (10 per parallelism degree).

6.1.1 Prediction accuracy

To test our approach, we apply it to real data and to synthetic data not used for training. Figure 6 shows that our model can accurately predict makespan, for both training and test data, and on both the local cluster 7h14 and in the cloud on Emr30. Not surprisingly, the simpler sort operation is easier to predict. Relative error tends to be around 1% and never exceeds 5%. For JOIN, the root mean squared error (RMSE) for training and test data are 332.97 and 148.66, respectively. For JOIN-AGG, they are 157.27 and 158.67, respectively. For Sorting, they are 15.81 and 6.57, respectively.

6.1.2 Optimizing operator-specific parameters

We apply the general model to determine the optimal values of operator-specific parameters for Join and Sorting on Amazon EMR clusters consisting of virtual machines



(a) `JOIN-AGG` on `Emr20` **(b)** `Sorting` on `Emr50`

Fig. 7 Optimal operator-specific partitioning

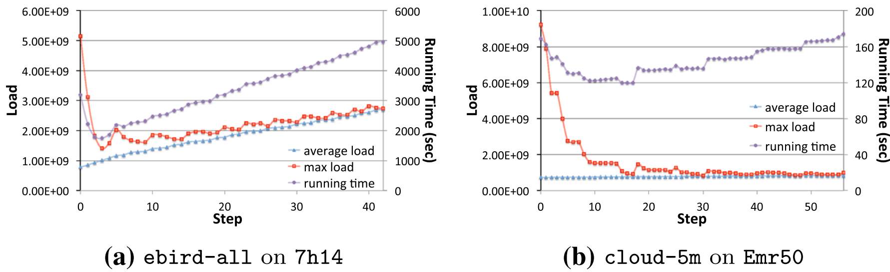
with a single core each. To isolate the effect of these parameters, we set parallelism degree and number of tasks to the number of partitions created based on the operator-specific parameters.

For `Join`, more fine-grained partitioning increases $\mathcal{I}$, while it may or may not decrease $\beta_2 I_m + \beta_3 O_m + \beta_4 C_m$. The challenge for the optimizer is to determine the right tradeoff between these two factors. Figure 7a shows results for the Cartesian product of two data sets, each containing 660, 000 tuples with 1000 integer attributes. Our model predicted the best makespan for 12 partitions, and Fig. 7a indicates that this indeed is the winning setting. Similarly, for `Sorting`, the model suggested to use 50 partitions on all 50 workers, independent of input size. Figure 7b confirms that this indeed minimizes makespan.

6.1.3 Safe pruning for distributed join

Figure 8 reports load and predicted makespan as Algorithm 1 greedily explores join-group partitionings. Max load first decreases due to improved load balancing for smaller partitions, quickly approaching average load. Hence Lemma 1 is very effective in determining a safe stopping point beyond which no better makespan can be found. In Figure 8a, in step 11, the average load per worker (blue dots) reaches 1.41×10^9, which is greater than the minimal max load (red dots) found before (1.40×10^9 in step 3). Hence it is safe to terminate the greedy algorithm in step 11. Similarly, in Fig. 8b, the average load in step 51 (8.39×10^8) surpasses the minimal max load (8.38×10^8 in step 30) and hence the greedy algorithm can safely terminate. Note that if the user is satisfied with a partitioning within 10% of optimal, then Lemma 1 makes it possible to determine that iterations can be terminated already after 16 steps.

Our experiments generally show that it is practical and efficient to use the safe termination condition. Table 3 lists the number of extra steps (from the step where the optimal partitioning was found) until safe termination was detected based on Lemma 1. Note that 40 steps take only about half a second of computation time.



(a) `ebird-all` on `7h14` **(b)** `cloud-5m` on `Emr50`

Fig. 8 Number of greedy partitioning steps versus load and predicted makespan (running time of the join) for `JOIN-AGG`

Table 3 Additional steps after finding optimum, until safe termination

Data set	Cluster	# Extra steps	Data set	Cluster	# Extra steps
ebird-all	7h14	8	zipf-5m-[1,0]	7h14	21
ebird-basic	7h14	16	zipf-5m-[0.25,0]	7h14	34
cloud-5m	7h14	37	ebird-all	Emr50	6
zipf-5m-1.0	7h14	10	cloud-5m	Emr50	35

6.2 Homogeneous-task model

The main purpose of these experiments is to provide a *proof of concept* that the simplified homogeneous makespan model with a “small” number of linear pieces is accurate *enough* to rank “good” above “bad” partitionings. In all experiments, the piecewise linear model for a round had between 1 and 7 pieces. To explore the feasibility of the easier-to-obtain surrogate makespan measure, we replaced in all training records true makespan with the surrogate measure (Sect. 5). This makes the model less accurate, but as our experiments will demonstrate, it is still effective for makespan optimization.

6.2.1 Sorting

We present measurements on clusters `9h36` and `Emr10`. All piecewise models for `9h36` are partitioned into ranges (0, 18] and (18, 36] on parallelism degree. Partitioning on task input and output size varies. We created 54 queries, each defined by a data set and a number of waves for execution. Data sets are drawn from a pool of 15 sets, each with a cardinality selected randomly between 100 million and 2.7 billion, containing random numbers of type Long (8 bytes per record). The number of waves is selected randomly between 1 and 10. We randomly select 41 of these queries to fit the regression-model parameters, while the other 13 are used for testing.

Figure 9 presents the relationship between input size and the value of H (Eq. 2). The y-axis reports H computed from observed $\mathcal{I}$, $\mathcal{O}$, and $\mathcal{C}$. (Degree of parallelism was set to the number of workers for all runs.) The dotted green line shows a piecewise linear model fitted to the data.

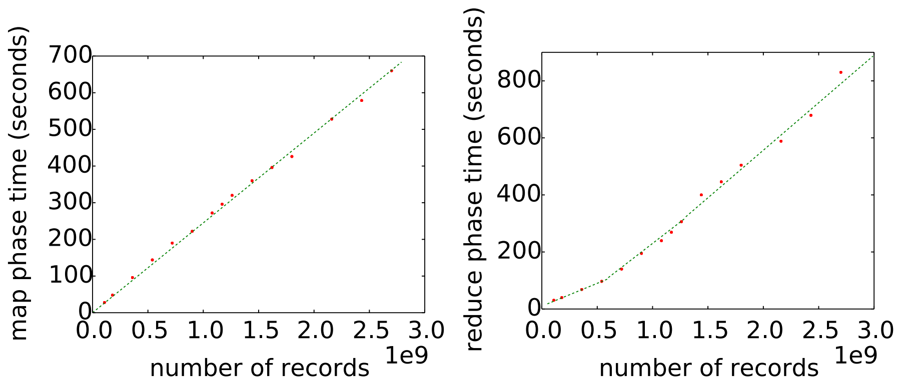


Fig. 9 Sorting: measured value of H versus input size on 9h36 for Map (left) and Reduce (right) phase

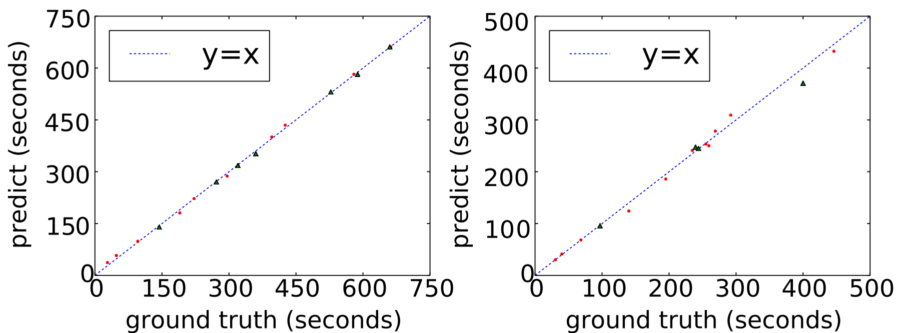


Fig. 10 Sorting: predicted versus measured value of H on 9h36 for Map (left) and Reduce (right) phase

Figure 10 compares predicted and measured values of H for map and reduce phase of sorting on cluster 9h36. The red dots are for training cases, while the green triangles are for test cases. All individual times and the overall trend are captured very accurately, as the *relative errors* are mostly around 1%, and never exceed 5%. For map phase, the RMSEs for training and testing are 0.98 and 1.16, respectively. For the reduce phase, they are 3.29 and 1.81.

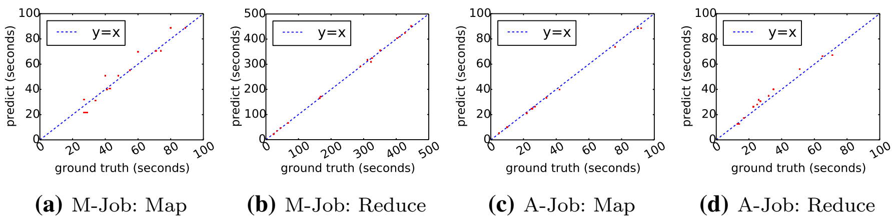
Table 4 shows that accurate estimation of H can still result in significant underestimation of makespan. This is caused by the use of the surrogate measure (number of waves times average task time, instead of true makespan, for model training), which does not capture delays caused by stragglers. However, this bias is consistent, allowing the model to capture the trend correctly, no matter if all cores or only half of them is used per machine. For large inputs, it identifies the I/O-related bottleneck: doubling the number of cores used per machine results in virtually no improvement of makespan when data size reaches 1.6 billion records.

6.2.2 Matrix multiplication

All models are partitioned into parallelism-degree ranges based on multiples of the number of machines in the cluster; partitioning on input and output size varies. The

Table 4 Degree of parallelism versus measured and predicted makespan on 9h36

Number of records	Degree of parallelism = 18		Degree of parallelism = 36	
	True (s)	Prediction	True (s)	Prediction
1.17E+9	790	601.96	698	564.21
1.26E+9	835	657.36	723	629.59
1.62E+9	1056	842.00	1050	833.66
1.80E+9	1146	928.18	1112	926.13
2.43E+9	1558	1254.39	1524	1288.04
2.70E+9	1751	1408.24	1741	1465.02



(a) M-Job: Map **(b)** M-Job: Reduce **(c)** A-Job: Map **(d)** A-Job: Reduce

Fig. 11 Matrix product: predicted versus measured value of H on 9h36. The test cases (red dots) are near the perfect-prediction line (blue dotted line) (Color figure online)

training set consists of 104 problem instances, covering 12 different matrix-size combinations (square matrices from $10k \times 10k$ to $30k \times 30k$ and also extreme rectangular ones up to $200 \times 4 \times 10^6$), each with 3 to 20 (B_0, B_1, B_2)-combinations. We randomly pick the matrix sizes and (B_0, B_1, B_2)-combinations in the above ranges. We then test the model on 57 independent problem instances, drawn from the same distribution. As Fig. 11 shows, predicted and true value of H are again very close. The training RMSEs for the four rounds (2 MapReduce jobs) are 6.99, 9.78, 1.69 and 6.82, respectively; the testing RMSEs for the four rounds are 4.95, 5.86, 1.11 and 3.00, respectively.

Like for sorting, our model underestimates true makespan due to the use of the surrogate measure, but can still correctly separate "good" from "bad" partitionings. In all cases our approach would find a near-optimal configuration. Table 5 confirms this for both synthetic and real data sets (from the UCI Machine Learning Repository [20]). There our technique is applied to the step where the data matrix is multiplied with its own transpose. Table 6 confirms that this observation also holds for Spark.

Note that for both real data sets (Table 5d, e), our model correctly discovers that setting (B_0, B_1, B_2) to (1, 18, 1) results in lower makespan than (1, 36, 1). We confirmed that due to I/O bottlenecks, it is better to only use half of the available cores per machine, even though round 2 performs a huge number of arithmetic operations (more than 11×10^9 for the Census data).

7 Related work

Structured cost models that capture execution details are essential for query optimization in relational DBMS [24], and they can be highly accurate when tuned [33]. Recent

Table 5 Ranking quality: predicted versus true makespan (sec) for matrix product (Hadoop MapReduce, (a)–(c) are synthetic data, (d) and (e) are real data)

B_0, B_1, B_2	Prediction		Ground truth	
	Makespan	Rank	Makespan	Rank
(a) $15,000 \times 15,000$ matrices on `9h36`				
6, 1, 6	305.40	1	400.00	1
3, 3, 4	330.87	2	434.00	2
4, 1, 4	345.89	5	440.00	3
3, 3, 3	350.39	7	445.67	4
3, 4, 3	333.55	3	448.00	5
5, 1, 5	356.48	9	452.00	6
3, 2, 3	344.69	4	453.00	7
2, 6, 3	348.85	6	471.00	8
4, 2, 4	353.48	8	479.00	9
2, 9, 2	385.02	11	485.00	10
2, 6, 2	380.85	10	497.00	11
2, 4, 2	403.84	13	505.00	12
2, 8, 2	410.55	14	525.00	13
2, 7, 2	446.67	15	548.00	14
4, 1, 8	401.43	12	556.00	15
2, 2, 2	614.17	16	656.00	16
1, 18, 1	638.41	17	713.00	17
1, 36, 1	941.19	18	1,290.00	18
(b) $15,000 \times 15,000$ matrices on `2h24`				
4, 1, 6	247.93	1	325.03	1
2, 4, 3	267.90	4	366.53	2
2, 3, 4	257.58	3	384.64	3
3, 2, 4	248.79	2	388.53	4
2, 6, 2	290.78	5	408.92	5
1,12, 2	356.74	6	455.77	6
1,24, 1	765.48	7	574.52	7
(c) $10,000 \times 60,000$ matrices on `20h160`				
4,10, 4	124.57	1	186	1
4, 8, 5	128.14	2	204	2
2,20, 4	132.53	3	205	3
4, 5, 8	134.07	4	205	3
2, 8,10	137.51	5	206	5
1,20, 8	141.89	6	211	6
12, 1,12	171.08	7	238	7

Table 5 continued

B_0, B_1, B_2	Prediction		Ground truth	
	Makespan	Rank	Makespan	Rank
(d)68 × 2458285 matrices (1990 US Census data) on `9h36`				
1,18, 1	30.93	1	95.00	1
1,36, 1	37.97	2	107.25	2
3, 2, 3	59.05	4	127.00	3
2, 9, 2	51.12	3	128.5	4
3, 4, 3	64.20	5	132.25	5
6, 1, 6	85.72	6	145.00	6
(e) 481 × 191779 matrices (KDD Cup 1998 data) on `9h36`				
1,18, 1	24.12	1	94	1
1,36, 1	30.89	2	103	2
3, 2, 3	39.19	4	109	3
2, 9, 2	37.43	3	112	4
3, 4, 3	44.61	5	121	5
6, 1, 6	48.37	6	144	6

work has shown that DBMS-style optimization can also be applied to other workloads, e.g., gradient descent computation that commonly occurs in machine learning [16]. Li et al. [18] rely on DBMS optimizers, and hence low-level cost models, to determine asymmetric data partitioning for heterogeneous clusters. When applied to homogeneous clusters of equally capable machines, e.g., on the Amazon cloud, these models assign the same input share to each worker, but do not optimize the partitioning parameters discussed in this article.

DBMS cost models motivated similar approaches for MapReduce and other distributed data analysis systems [14,21,29,31,34]. Simplified cost models for Hadoop and Spark systems are also proposed [10,11], but they focus on the impact of adding more worker machines, ignoring the impact of operator-specific partitioning parameters. Our work is orthogonal to research on lowering the cost of MapReduce programs by minimizing the number of rounds [9,17].

As an alternative to structured cost models, blackbox-style machine learning techniques were explored for a variety of performance prediction problems [2,5,6,8,13]. For all previous cost models, the effect of partitioning parameters on makespan is relatively complex, hence makespan minimization would have to rely on trial-and-error style exploration of possible parameter settings. For dense matrix multiplication, this corresponds to a 10-dimensional space of $(B_0, B_1, B_2, p_0, p_1, p_2, p_3, n_1, n_3, n_4)$ combinations. (Note that Ernest [29] could possibly derive optimal settings for all p_i, $i = 0, \ldots, 3$, reducing complexity to 6 dimensions.) In contrast, our approach sacrifices some prediction accuracy to simplify model structure. This enables analytical derivation of optimal settings for most parameters, reducing complexity to 3 dimensions for dense matrix multiplication.

Shi et al. [25] identify four key system parameters to optimize MapReduce makespan. While similar in spirit to our approach, they do not include operator-specific

Table 6 Ranking quality: predicted versus true makespan (in seconds) for matrix product (synthetic data, Spark)

B_0, B_1, B_2	Prediction		Ground truth	
	Makespan	Rank	Makespan	Rank
(a) 800 × 80, 000 matrices on `6s12`				
2, 2, 3	73.81	1	88.8	1
2, 3, 2	74.08	3	90.67	2
1,12, 1	73.88	2	96	3
1, 3, 4	87.84	4	101	4
1, 6, 2	100.10	7	101.4	5
1, 4, 3	96.79	6	104	6
3, 1, 4	133.95	9	109.5	7
1, 6, 1	92.80	5	113	8
2, 1, 3	154.30	11	134	9
1, 2, 3	134.22	10	141	10
1, 3, 2	131.48	8	154	11
(b) 6000 × 6000 matrices on `Emr12s`				
3, 1, 4	144.73	1	149.5	1
2, 2, 3	152.50	2	152	2
2, 3, 2	156.63	3	162	3
1, 2, 6	171.79	5	170.5	4
1, 3, 4	166.27	4	171	5
1, 4, 3	180.81	6	173.5	6
1, 6, 2	184.95	7	195	7
2, 1, 3	251.14	9	254	8
1, 2, 3	268.33	8	268.5	9
1, 3, 2	277.20	11	277	10
1, 1, 6	266.92	10	304	11
1, 6, 1	365.17	12	362	12

partitioning parameters in their analysis. And due to the complexity of the model, there are no results comparable to our Lemmas that enable more efficient optimization for the key parameters.

We use dense matrix multiplication to showcase model design and makespan optimization for an analytics operator with a demanding I/O and CPU profile. Previous work explored a variety of performance-related aspects for matrix multiplication on parallel architectures. This includes load balancing [28], minimizing communication cost [1,4,15,26], and optimizing for memory hierarchy [7,27].

8 Conclusions

Starting with the goal of minimizing makespan for distributed data-intensive computation, we set out to identify the "simplest possible", "sufficiently accurate" model

to predict makespan of data analytics operators. To this end, we proposed abstract models that are piecewise linear functions depending only on input, output, and computation complexity. Our approach has two main benefits. First, it simplifies tying problem-partitioning parameters to model variables (input, output and computation) for user-defined operators, e.g., programs written in MapReduce or Spark. Second, we showed that the linear structure can be exploited for more efficient optimization algorithms. It enabled pruning of values from the optimization-parameter search space and even a significant reduction of search space dimensionality (for homogeneous tasks). For instance, optimization complexity was reduced from a search process in ten dimensions to only three for matrix product; for sorting the optimal solution was directly obtainable in closed form.

Our experiments indicated that a small number of pieces achieve sufficient prediction quality, enabling us to find near-optimal problem partitionings very efficiently. In future work, we will explore tuning of partitioning parameters along with system parameters external to user programs, by integrating our ideas into optimizers like Starfish [13].

References

1. Agarwal, R.C., Balle, S.M., Gustavson, F.G., Joshi, M., Palkar, P.: A three-dimensional approach to parallel matrix multiplication. IBM J. Res. Dev. **39**(5), 575–582 (1995)
2. Akdere, M., Cetintemel, U., Riondato, M., Upfal, E., Zdonik, S.: Learning-based query performance modeling and prediction. In: ICDE, pp. 390–401 (2012)
3. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
4. Ballard, G., Buluc, A., Demmel, J., Grigori, L., Lipshitz, B., Schwartz, O., Toledo, S.: Communication optimal parallel multiplication of sparse random matrices. In: SPAA, pp. 222–231 (2013)
5. Duggan, J., Cetintemel, U., Papaemmanouil, O., Upfal, E.: Performance prediction for concurrent database workloads. In: SIGMOD, pp. 337–348 (2011)
6. Duggan, J., Papaemmanouil, O., Çetintemel, U., Upfal, E.: Contender: a resource modeling approach for concurrent query performance prediction. In: EDBT, pp. 109–120 (2014)
7. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Rev. **46**(1), 3–45 (2004)
8. Ganapathi, A., Kuno, H.A., Dayal, U., Wiener, J.L., Fox, A., Jordan, M.I., Patterson, D.A.: Predicting multiple metrics for queries: Better decisions enabled by machine learning. In: ICDE, pp. 592–603 (2009)
9. Goodrich, M.T., Sitchinava, N., Zhang, Q.: Sorting, searching, and simulation in the mapreduce framework. In: ISAAC, pp. 374–383 (2011)
10. Gounaris, A., Kougka, G., Tous, R., Montes, C.T., Torres, J.: Dynamic configuration of partitioning in spark applications. IEEE Trans. Parallel Distrib. Syst. **28**(7), 1891–1904 (2017)
11. Gunther, N.J., Puglia, P., Tomasette, K.: Hadoop superlinear scalability. Commun. ACM **58**(4), 46–55 (2015)
12. Hahn, C., Warren, S., Eastman, R.: Extended edited synoptic cloud reports from ships and land stations over the globe, 1952–2009 (ndp-026c) (2012)
13. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of mapreduce programs. VLDB **4**(11), 1111–1122 (2011)
14. Huang, B., Babu, S., Yang, J.: Cumulon: optimizing statistical data analysis in the cloud. In: Proceedings of SIGMOD, pp. 1–12 (2013)
15. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Comput. **64**(9), 1017–1026 (2004)

16. Kaoudi, Z., Quiane-Ruiz, JA., Thirumuruganathan, S., Chawla, S., Agrawal, D.: A cost-based optimizer for gradient descent optimization. In: SIGMOD, pp. 977–992 (2017)
17. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for mapreduce. In: SODA, pp. 938–948 (2010)
18. Li, J., Naughton, J.F., Nehme, R.V.: Resource bricolage and resource selection for parallel database systems. VLDB J. **26**(1), 31–54 (2017)
19. Li, R., Riedewald, M., Deng, X.: Submodularity of distributed join computation. In: (Upcoming) SIGMOD (2018)
20. Lichman, M.: UCI machine learning repository (2013)
21. Morton, K., Balazinska, M., Grossman, D.: Paratimer: a progress indicator for mapreduce dags. In: SIGMOD, pp. 507–518 (2010)
22. Munson, A.M., Webb, K., Sheldon, D., Fink, D., Hochachka, W.M., Iliff, M., Riedewald, M., Sorokina, D., Sullivan, B., Wood, C., Kelling, S.: The Ebird Reference Dataset, Version 2014. Cornell Lab of Ornithology and National Audubon Society, Ithaca, NY (2014)
23. Quinlan, J.R., et al.: Learning with continuous classes. Aust. Jt. Conf. Artif. Intell. **92**, 343–348 (1992)
24. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw-Hill, New York (2003)
25. Shi, J., Zou, J., Lu, J., Cao, Z., Li, S., Wang, C.: Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. In: VLDB, pp. 1319–1330 (2014)
26. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5d matrix multiplication and LU factorization algorithms. In: Euro-Par 2011 Parallel Processing, pp. 90–109 (2011)
27. Valsalam, V., Skjellum, A.: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurr. Comput. **14**(10), 805–839 (2002)
28. van de Geijn, R.A., Watts, J.: Summa: Scalable Universal Matrix Multiplication Algorithm. University of Texas at Austin, Tech. rep. (1995)
29. Venkataraman, S., Yang, Z., Franklin, M., Recht, B., Stoica, I.: Ernest: efficient performance prediction for large-scale advanced analytics. In: NSDI, pp. 363–378 (2016)
30. Vieth, E.: Fitting piecewise linear regression functions to biological responses. J. Appl. Physiol. **67**(1), 390–396 (1989)
31. Wang, G., Chan, CY.: Multi-query optimization in mapreduce framework. In: VLDB, pp. 145–156 (2013)
32. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C, Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: OSDI, pp. 255–270 (2002)
33. Wu, W., Chi, Y., Hacígümüş, H., Naughton, J.F.: Towards predicting query execution time for concurrent and dynamic database workloads. VLDB **6**(10), 925–936 (2013)
34. Zhang, X., Chen, L., Wang, M.: Efficient multi-way theta-join processing using mapreduce. In: VLDB, pp. 1184–1195 (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Rundong Li[1] · Ningfang Mi[2] · Mirek Riedewald[1] · Yizhou Sun[3] · Yi Yao[2]

Ningfang Mi
ningfang@ece.neu.edu

Mirek Riedewald
m.riedewald@northeastern.edu

Yizhou Sun
yzsun@cs.ucla.edu

Yi Yao
yyao@ece.neu.edu

[1] CCIS, Northeastern University, Boston, USA

[2] ECE, Northeastern University, Boston, USA

[3] Department of Computer Science, UCLA, Los Angeles, USA