Critical Path based Performance Models for Distributed Queries

Ankush Desai
Microsoft Research India

Kaushik Rajan
Microsoft Research India

Kapil Vaswani
Microsoft Research India

Technical Report
MSR-TR-2012-121

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

Abstract

Programming models such as MapReduce and DryadLINQ provide programmers with declarative abstractions (such as SQL like query languages) for writing data intensive computations. The models also provide runtime systems that can execute these queries on a large cluster of machines, while dealing with the vagaries of distribution such as messaging, failures and synchronization. However, this level of abstraction comes at a cost – the inability to understand, predict and debug performance. In this paper, we propose a performance modelling approach for predicting the execution time of distributed queries. Our modeling approach is based on a combination of the critical path method, empirically generated black box models and cardinality estimation techniques from databases. We evaluate the models using several real world applications and find that models can accurately predict execution time to within 10% of actual execution time. We demonstrate the usefulness of the model in identifying performance bottlenecks, both during design and while debugging performance problems.

1 Introduction

Programming models such as Pig [32], Hive [20], FlumeJava [3] and DryadLINQ [23] greatly simplify the process of deploying and executing large scale batch computations on a cluster of machines. These programming models expose SQL like declarative query languages with simple, sequential semantics. A distributed query engine compiles, optimizes, schedules and executes these queries on a cluster. Query engines exploit parallelism within and across machines, and use optimizations such as pipelining to overlap computation and I/O. Query engines also deal with the pitfalls of distribution such as asynchrony and failures. The semantic simplicity coupled with high performance has resulted in widespread adoption of these models, even by novice programmers.

Problem. Despite their convenience, these programming models have a key shortcoming – the difficulty of reasoning about performance. Due to the declarative nature of query languages, users have little insight into *how* a query executes on a cluster, and consequently, how to write queries for better performance. This problem is compounded by the fact that many users of these models are not expert programmers but domains experts such as data analysts and scientists. These users find it hard to estimate how long a query may run, or how much it might cost (if resources are charged). Often, the programming models require users to specify resources (such as compute nodes, storage and network bandwidth) that a query may use a priori. In the absence of execution time estimates or knowledge of potential bottlenecks, users end up under or over allocating resources.

We illustrate this problem using a simple example. Consider a query that clusters data points using the *kmeans* algorithm (Figure 2). Figure 1 shows the execution times of this query for varying number of nodes for an input consisting of 22 million points (45GB). As one might expect, the execution time of the query decreases with increasing number of nodes. However, assigning additional resources beyond 25 nodes does not yield proportional performance benefits (and only increases cost).

The second shortcoming of these models is the difficulty in diagnosing performance problems that occur when a query executes on a cluster. For example, a common problem is *data skew*, which is caused by uneven partitioning of data across machines. Data skew causes load imbalance, low utilization of resources and in general poor performance. Even if data is evenly balanced, some operations in the query may stress a particular resource (such as the network switch or a disk) more than others. Unfortunately, many of these problems are hard to detect, diagnose and fix. The run-profile-optimize cycle traditionally used for diagnosing and optimizing performance is time consuming and impractical for large scale distributed queries. In many cases, performance problems only manifest with large workloads, and naively scaling performance from a smaller workload can grossly underestimate performance. Although performance counters collected from an actual run on the cluster can help diagnose performance issues, performance counter data is voluminous, hard to interpret and map back to the query.

Solution Overview. In this paper, we propose an approach for building predictive performance models for estimating and diagnosing performance of distributed

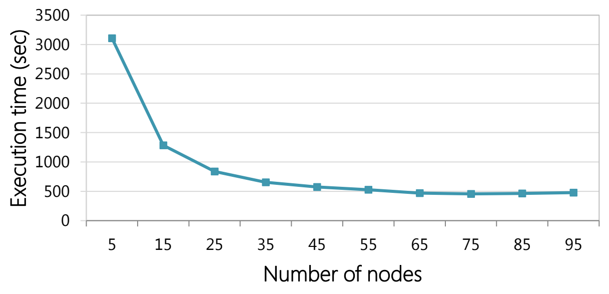

Figure 1: Trade-off between number of machines used and execution time for the *kmeans* query.

queries. Given a query, a workload and a cluster, our models predict the *ideal* execution time of the query on the cluster. The ideal execution time of a query is the execution time in ideal conditions i.e. in the absence of failures and contention from other queries. Our models also provide a fine-grained breakdown of ideal execution time and resource utilization at the granularity of each machine.

Our approach for modeling performance of distributed queries is based on the *critical path method*, a method used for modeling and scheduling parallel activities. In our model, query processing is abstractly represented by a set of *tasks*. Each task consists of three phases, a read, compute and a write phase. We represent the execution of the query as a dependence graph, where nodes in the graph correspond to phases. Edges model constraints that limit parallelism, such as data dependencies, and resource constraints such as number of available processor cores, network ports, and memory buffers. We estimate the time required for each phase using black box models for resources such as disk and network. These estimates are used to compute weights for edges in the dependence graph. We determine the size of the graph (number of phases) using cardinality estimation techniques (used in conventional databases). Finally, we estimate the ideal execution time of the query by computing the longest path in the dependence graph i.e. the critical path.

These are several reasons why the critical path method is a natural fit for modeling distributed query processing.

- The critical path method allows us to model parallelism and resource contention more precisely than other approaches (such as statistical, black box models). We use black box models only for components that are hard to model using the critical path method (such as disks and network).

- The models we build are *composable*. We first construct critical path models for each query operator individually and then compose these models for different compositions of query operators. Composability is a key attribute since it greatly simplifies the process of modeling complex queries, and permits the modeling approach to be easily re-targeted to different query engines.

- Models built using the critical path method are easily interpretable by end users. The outcome of the modeling process is a list of nodes that lie on the critical path. Often, a simple inspection of this list can help identify bottleneck tasks. The decomposition of tasks into read, compute and write phase also helps determine *why* a task is bottleneck.

- In addition to estimating query execution time, critical path based models can also be used to diagnose performance bottlenecks in actual execution. The models provide fine-grained breakdown of the execution time of the query on each machine in each stage *under ideal conditions*. A comparison of the actual execution times with estimated ideal execution times can identify the root cause of poor performance.

We evaluate the models on several real world queries and on several large datasets (most of the queries and datasets are publicly available). We find that for all queries, our model can predict execution time accurately (to within 10% of actual execution time) in a fraction of the time it takes to run the query. We also show that our models enable design time optimization, and aid in diagnosing performance problems.

This paper is organized as follows. Section 2 provides a quick overview of distributed query languages. In Section 3, we describe our hierarchical modeling approach to predict query performance. Section 5 discusses our implementation. We present a detailed evaluation of our approach in Section 6 and discuss related work in Section 7.

```
IQueryable<Point> KMeans(IQueryable<Point> points, int k) {
  var centers = GetRandomCenters(k);
  for(int i = 0; i < 3; i++)
    centers = points.GroupBy(point => NearestCenter(point, centers))
                    .Select(g => CalculateNewCenter(g));
  return centers; }
```

Figure 2: Query for k-means clustering.

2 Distributed Query Execution

This section provides an overview of how declarative queries written in high level languages (such as Pig-Latin [32], Hive [20], FlumeJava [3] and DryadLINQ [23]) are translated into a distributed computation. We illustrate this process using DryadLINQ.

Query Language. The query language underlying DryadLINQ is LINQ [1], a .NET library that provides relational constructs (Select, GroupBy, Join, Aggregation etc.) for querying arbitrary collections of objects. Figure 2 shows the the k-means algorithm (Figure 2) expressed in LINQ. The program takes as input a set of points, performs three iterations of the k-means clustering algorithm and returns the resulting center points. During each iteration, the query computes the nearest center for every point (using the function $NearestCenter$), groups all points by the nearest center, and then computes the new center for each group (using $CalculateNewCenter$).

Query Compilation and Optimization. Given a LINQ query, the DryadLINQ compiler constructs an Execution Plan Graph (EPG). The EPG is a DAG where nodes represent computation stages and edges represent data flow between stages. Each stage is composed of a set of DryadLINQ operators. The DryadLINQ compiler supports a set of static compiler optimizations including *pipelining* which merges a linear chain of operators into a single stage so that all merged operators run on a single node, and *eager aggregation*, which moves computation from downstream aggregations into upstream stages to reduce the amount of data transmitted over the network. The EPG for a single iteration of Kmeans after applying these optimizations is shown in figure 3(a).

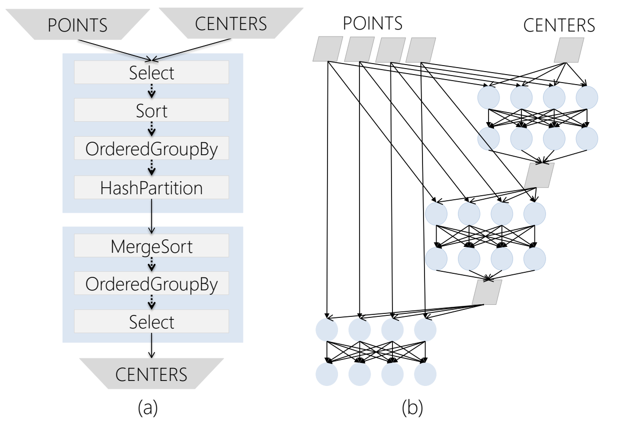

Figure 3: (a) Optimized query plan for *one iteration* of kmeans, and (b) Graph representing dataflow during execution of 3 iterations of kmeans. Computation stages are shown in blue and data inputs are shown in gray.

Query Evaluation. DryadLINQ uses the Dryad engine as the back-end for scheduling and executing queries. Dryad maps each stage to multiple nodes in the cluster, each of which operate on a partition of the data. For example, the distribution of stages to cluster nodes for KMeans is shown in figure 3(b). The *points* dataset is partitioned across four nodes. DryadLINQ stages are run sequentially one after the other. Dryad also sets up all communication within and across stages. Communication between operators within a stage occurs via in-memory structures. At the end of a stage the output data is persisted on to a file on the local machine and the next stage reads this input via remote file reads.

3 Performance Model

3.1 Response Variable

The first step in building a performance model is to identify the appropriate response variable(s). Performance of a query can be characterized by several metrics. Query optimizers in conventional databases use abstract metrics such as the number of disk accesses as a proxy for performance. However, we believe concrete measures such as execution time, bandwidth, and cost are more appropriate since we target end-users in addition to query optimizers.

The execution time of a query on a cluster depends on a number of dynamic parameters, such as the load on the cluster, and transient process and network failures. Modeling all these parameters is challenging. In this paper, we choose the *ideal query execution time* as the primary response variable. The ideal response time of a query on a given cluster is the execution time of the query *in the absence of failures and contention due to other queries*. Unlike a query's actual execution time, the ideal execution time is an *intrinsic* property of the query and the cluster alone. It accounts for resource constraints due to query's own execution, and cluster specific parameters such as the configuration of machines and the interconnect. In addition, the ideal response time is a measure that users can easily comprehend and reason about at design time. Our modeling approach can be adapted to predict other related measures such as ideal cost and ideal network bandwidth consumption.

3.2 Modeling approach

Our modeling approach is based on the *critical path method*, a technique for analyzing processes composed of a large number of tasks. Given the list of participating tasks, the dependencies between the tasks, and the time taken to complete each task, the critical path method computes the longest sequence of tasks to complete the process. All tasks on this sequence are said to be on the *critical path*, and the length of the critical path is an estimate for the overall time to completion. These tasks are ideal targets for optimization because reducing the time to complete these tasks reduces the length of the critical path and hence the overall time to completion. We now describe how this method can be adapted to build performance models for distributed queries.

As described in Section 2, query processing in a typical distributed query engine occurs in stages. A stage is a composition of one or more operators and may span several nodes. Processing in a stage involves reading inputs from nodes in the previous stage (from files or remote locations), computing results, and persisting the results (to disk or a remote location) for later stages. Our modeling approach reflects this compositional nature of query processing. We first build critical path models for each operator supported by the query engine. The models estimate the ideal execution time of the operator when executed on a single machine. We build these models by analyzing the operator's implementation and identifying steps and dependencies that influence performance. In the next section, we propose a generic framework for building operator level models. The operator level models are *composed* to obtain a performance model for each node. Finally, we compose models for each node to obtain a critical path model for query execution on the cluster. In this paper, we illustrate this approach using DryadLINQ. However, our approach can be easily retargeted to other query engines like Hive and FlumeJava, which have a similar execution model.

3.3 Operator-level Model

Query languages supported by distributed query engines support a number of relational and set theoretic operators. We now present a generic framework for building critical path models for a large class of query operators. The framework is based on the observation that most operator implementations across query engines are structurally similar. The framework captures key tasks and dependencies common to these implementations, while abstracting away low level details. We show how this framework can be instantiated to build models for operators in DryadLINQ.

Framework. In our framework, we abstractly model the execution of an operator as a *pipeline* with three phases, namely *read*, *compute* and a *write* phase. During the read phase, the operator reads input data from memory, disk or a remote location. The compute phase models operator-specific logic performed on the input data. In the write phase, the results of the compute phase are written out typically to an in-memory *output buffer* which is flushed to disk occasionally. A pipeline consists of many instances of these three phases executing in parallel; we refer to each such instance as a *task*. Figure 4 shows an instance of a pipeline as a graph with nodes representing phases. Two special nodes S and E represent the start and end of execution.

During the execution of a pipeline, data flow and re-

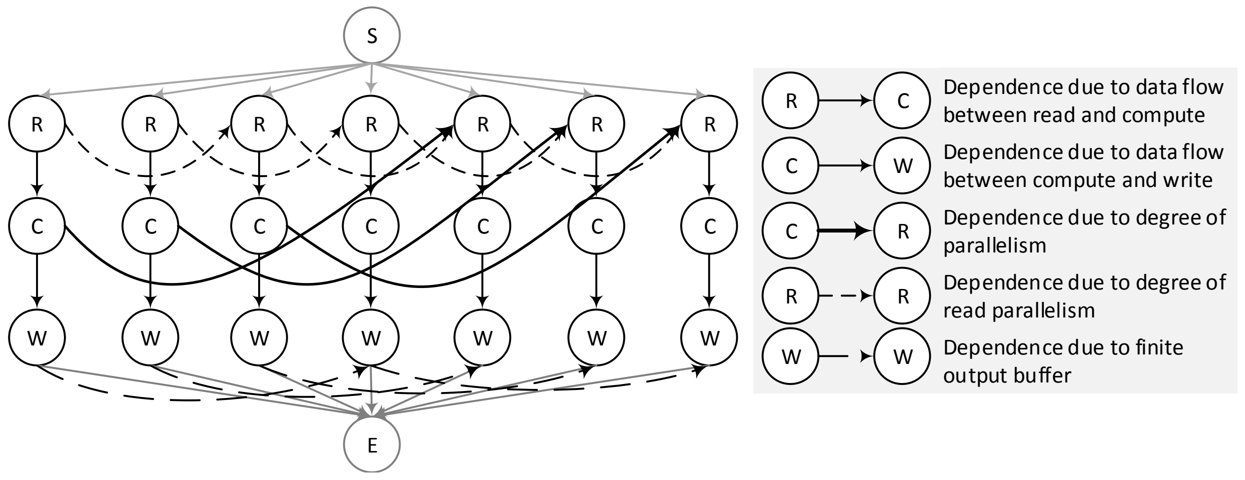
S
R
C
W
E
Dependence due to data flow between read and compute
Dependence due to data flow between compute and write
Dependence due to degree of parallelism
Dependence due to degree of read parallelism
Dependence due to finite output buffer

Figure 4: *Graph based model of a 3-stage parallel pipeline. The nodes R, C and W represent read, compute and write phases of the pipeline respectively. S and E are special nodes representing the start and end of an operator's execution.*

source constraints induce dependencies between phases (both within a task and across tasks). Our framework incorporates key dependencies that have the largest impact on performance. These dependencies can be represented as edges between phases in the pipeline graph.

- Data flow dependence between read phase and compute phase within a task ($R_i \rightarrow C_i$). This dependence limits computation of a task from starting until data has been read.
- Data flow dependence between compute and write phase of the same task ($C_i \rightarrow W_i$). This dependence limits writes for a task from starting until computation has completed.
- Dependence due to the degree of parallelism c_p ($C_i \rightarrow R_j, j > i$). This dependence constrains the read phase of task j from *starting* unless C_i completes. In data parallel operators (such as select, project, join), the degree of parallelism is limited by the number of cores available on the machine, since both read and compute phases require a dedicated processor core. For inherently sequential operators (such as ordered aggregation), the degree of parallelism is 1.
- Dependence due to read parallelism r_p ($R_i \rightarrow R_j, j > i$). The degree of read parallelism is determined by the number of input data sources an operator reads from in parallel. If an operator reads from a file on disk, r_p is 1 (assuming files on disk are read serially). If the operator reads data from multiple network locations (e.g. in the reduce operator), r_p is equal to the number of network locations. This model permits multiple reads to progress in parallel, as long as reads are from different sources.
- Dependence due to finite size of output buffer ($W_i \rightarrow W_j, j > i$). Many operator implementations use buffering to avoid expensive writes to disks and file systems. In these implementations, tasks write their results to an in-memory buffer, which is flushed to the file system or disk when it reaches close to capacity. The $W_i \rightarrow W_j$ dependence models stalls that occur due to intermittent flushes. The dependence exists between phases W_i and W_j if W_j must wait for W_i before it can write to the output buffer.

The parallel pipeline is a simplified, abstract model of execution for a large class of operators. The pipeline can be instantiated to model data parallelism and parallelism obtained by overlapping computation with I/O simply by picking the right dependencies. For example, consider the set of operators supported by DryadLINQ [23]. True data flow dependencies $R_i \rightarrow C_i$ and $C_i \rightarrow W_i$ exist in all operators. The operators *HashJoin, GroupBy, Select, Where, SelectMany, Sum* and *Count* are inherently parallel and $c_p = P$, where P is the number of available processor cores. The operators *Sort, HashPartition, Union, Merge, Take, Distinct* and *OrderedGroupBy* are sequential i.e. $c_p = 1$. The degree of read parallelism in all operators is limited by the number of distinct data sources. All DryadLINQ operators use buffering. Hence the $W_i \rightarrow W_j$ dependence applies to all operators.

Computing the critical path. The next step in building a critical path model for an operator is to estimate time required to complete each phase in the pipeline. Specifically, we require estimates for the following parameters:

1. Number of tasks in the pipeline (N),
2. Time to complete the read and compute phase for the i^{th} task ($T_r(i)$ and $T_c(i)$),
3. Amount of data written out by task i ($DW(i)$),
4. Size of the output buffer (*bufferSize*),
5. Latency of flushing the output buffer to disk (T_{flush})

We present one approach for estimating these parameters in Section 5. Assuming these parameters are available, we compute the critical path as follows. Informally, we associate every dependence with the amount of time the destination of the dependence must wait since the source has started. For example, the dependence $R_i \rightarrow C_i$ is associated with $T_r(i)$ because the compute phase of task i must wait for at least $T_r(i)$ units of time once the read phase of the task has started for input data to be available. The critical path is the *longest* path (path with the highest cumulative weight) from node S to node E.

Formally, the critical path can be interpreted as the *earliest* time at which all dependencies of the end node E have been satisfied. The length of the critical path is be obtained by solving the set of recurrence equations defined in Figure 5. Equation 1 determines the earliest time since start of execution ($EST_r(n)$) at which the read phase of a task n can begin. This depends on both the availability of a free processor core (for performing the read) and for the previous read from the same source to have completed. If the degree of parallelism for the operator is c_p, a processor core is available as soon as all but $c_p - 1$ of the previous compute phases have completed. The first three clauses of this equation model initial conditions, whereas the two cases in the final clause models these dependencies in the steady state.

Equation 2 determines the earliest start time of a compute phase, which only depends on the read phase of the same task. Equation 3 determines the earliest start time of the write phase. The write phase of a task can begin as soon as the corresponding compute phase completes, and there is space in the output buffer. We model buffering by a dependence between a write phase $W(n)$ and a previous write phase $W(i)$ such that the cumulative data written by the intermediate write phases (represented by CDW) exceeds the buffer size. We associate this dependence with the latency T_{flush}. The query terminates once all the write phases have completed (denoted by $EST_e(T)$).

3.4 Node-level Models

Each stage in distributed query processing is consists of a number of operators. Broadly, there are two approaches that query engines use for composing operator implementations, *sequential* and *nested* composition. We now describe these two composition and show how pipelines can be *composed* to model these forms of composition and build node-level models.

Sequential composition. Consider two operators op_1 and op_2 that apply functions f_1 and f_2 to their inputs respectively. If the two operators have the same degree of parallelism, they can be *sequentially composed* into an operator that applies $f_2 \circ f_1$ to its inputs. For example, two *select* operators or an aggregation followed by *select* can be sequentially composed. All operators in HIVE and FlumeJava are composed this way. Sequential composition avoid the need for persisting data between operators; data produced by one operators can be directly consumed by the second.

We can obtain a model for the sequential composition of two operators by composing their compute phases. Specifically, the labels of the pipeline representing the sequential composition are as follows.

$$T_r(n) = T_r^1(n), T_c(n) = T_c^1(n) + T_c^2 n, T_w(n) = T_w^2(n)$$

$$EST_r(n) = \begin{cases} 0, & 1 <= n <= min(r_p, c_p) \\ min(\{EST_c(n-i) + T_c(n-i) \mid 1 \leq i \leq c_p\}), & r_p > c_p, c_p + 1 <= n <= r_p \\ EST_r(n - r_p) + T_r(n - r_p), & c_p > r_p, r_p + 1 <= n <= c_p \\ max(EST_r(n - r_p) + T_r(n - r_p), & \\ \quad {c_p}^{th}\ max(\{EST_c(i) + T_c(i) \mid 1 \leq i \leq N-1\})), & max(r_p, c_p) + 1 <= n <= N \end{cases} \tag{1}$$

$$EST_c(n) = EST_r(n) + T_r(n) \tag{2}$$

$$EST_w(n) = \begin{cases} EST_c(n) + T_c(n), & n = 1 \\ max(EST_c(n) + T_c(n), & \\ \quad (EST_w(n-i) + T_{flush} \mid CDW(i) >= \textit{bufferSize},\ CDW(i+1) < \textit{bufferSize})) & \\ \quad \textit{where}\ CDW(u) = \sum_{n-u}^{n-1} DW(u) & n > 1 \end{cases} \tag{3}$$

$$EST_e(N) = max(\{EST_w(i) \mid 1 <= i <= N\}) \tag{4}$$

Figure 5: *Set of recurrence equations, whose solution for the earliest start time (EST) for the* end node $EST_e(N)$ *represents the ideal execution time of N tasks of the pipeline. T_r and T_c represent the time taken by the read phase and compute phase respectively. T_{flush} represents the time to flush from from buffer to disk. c_p and r_p represent the read and compute parallelism. $DW(i)$ refers to the amount of data written by task i*

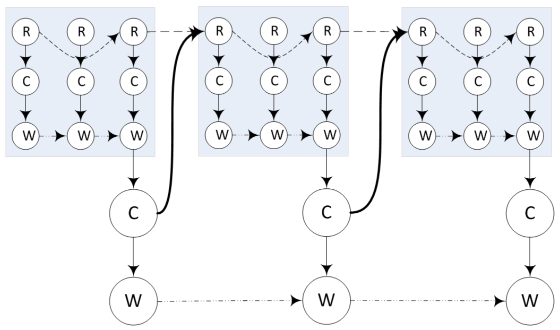

Figure 6: A graph based model of a simply nested parallel pipeline where a parallel operator is composed with an inherently sequential operator.

where $T_r^1(n), T_c^1(n)$, $T_w^1(n)$, $T_r^2(n), T_c^2(n)$, and $T_w^2(n)$ are the read, compute and writes times of the n^{th} thread of the two pipelines respectively. These labels can be used with Equation 5 to obtain the ideal execution time of the resulting pipeline.

Nested composition. Nested composition is form of operator composition where k tasks of the first operator form the read phase of the second operator. Figure 6 illustrates the graph based model of this composition for $k = 3$. This form of composition models DraydLINQ's lazy evaluation strategy where later operators in stage fetch results from earlier operators *on demand*. For example, in DryadLINQ, a *select* followed by *sort* is implemented in this fashion. This ensures that the compute phase of *sort* (which is sequential and expensive) occurs in parallel with read and compute phases of *select*. For this composition, the labels of the new pipeline can be expressed as follows.

$$T_r(n) = EST_e^1(k), T_c(n) = T_c^2(n), T_w(n) = T_w^2(n)$$

where EST_e^1 is the ideal execution time of a k-thread pipeline of the first operator.

3.5 Cluster-level Model

We now present an analytical model for predicting the ideal execution time of a query on a cluster. The basis for the model is a graph based representation of the execution of the query on the cluster, which is derived from the query plan. The graph models cluster-level parallelism in the query and the effects of data dependencies across stages. In this graph based model, each stage S^i is represented by a set of nodes $S^i_j, 1 \leq j \leq m$, where m is the number of machines participating in stage S^i. Two special nodes S_0 and S_e represent the start and end of query execution. An edge exists between nodes S^{i-1}_u and S^i_v if there is a data dependence between these

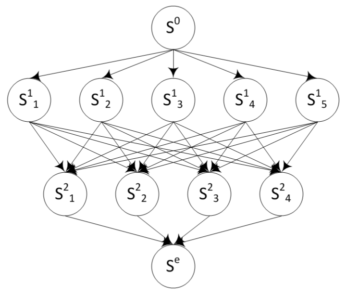

Figure 7: A graph based model of query execution on the cluster. The first stage and the second stage use 5 and 4 machines respectively.

nodes. Each edge is weighted by *ideal execution time* of the source node, obtained using node level execution models described earlier. Given this model of execution, the ideal execution time of the whole query is determined by the *critical path* i.e. the path with the largest cumulative weight amongst all paths from S^0 to S^e. An example graph for a 2-stage query is shown in Figure 7.

4 Parameter Estimation

The critical path based modeling framework we propose is parameterized by the number of tasks in the pipeline, time to complete read and compute phases, and output buffer parameters such buffer size and flush latency. We now describe an approach for estimating these parameters for a given operator on a given cluster.

Number of tasks. Many operator implementations exploit machine level data parallelism by partitioning the input into chunks and create a task for processing every chunk. The number of tasks depends on the cardinality of the input and the chunk size.

Cardinality estimation is a well-known problem in databases and many candidate solutions exist. Arguably, the most effective solutions are based on *sampling*. Informally, the basic idea is to run the query using a sample of the input, find the cardinality of the input at every operator in the sampled run, and scale up the cardinality by the sampling factor. We estimate cardinalities using a similar approach. We run the query using a random sample (with replacement) of the input and generate intermediate results at the end of every stage. We then estimate the cardinality at each node in a stage by applying the corresponding partition function to the sample input data, and scaling up the resulting cardinalities by the sampling factor. It the input to a stage is a grouping (result of a *GroupBy* operator), we use a distinct value estimator [4] to estimate the number of groups.

The second parameter that determines the number of tasks is the chunk size. In practice, we find that every operator implementation determines a chunk size (based on experiments and domain knowledge). Therefore, we leave the chunk size as a parameter for the operator level model. The number of tasks N for an operator is S/C, where S is the estimated cardinality of the input and C is the chunk size.

Read Times. The read time of a task depends on the amount of data read by the task, and the data transfer rate. The amount of data read in turn depends on the chunk size and the size of the records in the partition. In some cases, size of input records can be determined statically from the size of the *data type* use to store input records. However, this approach does not work if the size of the type is variable. As an example, consider the k-means query where each point is an n-dimensional vector. If each point is represented by an array of integers, the size of each record if fixed. However, if adjacency lists are used, the size cannot be statically determined. In such cases, we use sampled inputs to estimate the distribution of record sizes RS_i of each task i. We partition the sampled input to the node into N contiguous sets of records $s_1, \ldots, s_N$ (where N is the number of tasks). We then compute a histogram of the size of records in set s_i, and scale up the frequencies in the histogram by the sampling rate to obtain the distribution RS_i.

Determining the ideal *data transfer rate* is also challenging because the rate depends on the source of the data (disk, network location in the same rack or different rack) and the amount of data read. We use *micro-benchmarking* to build black box models for the disk and network. Our micro-benchmarks are read intensive workloads that are parametrized by the record size, amount of data to read, and the source of the data (local or remote). We run these workloads on an unloaded cluster for different record sizes and measure data transfer rates (bytes/sec) for each record size. We obtain one such distribution for local reads, intra-rack reads, inter-rack reads. Since it is not possible to tabulate rates for all record sizes, we measure rates for selected data points and use linear interpolation to estimate data transfer rates for a given record size.

Let RT represent the data transfer rate distribution for reads applicable to the operator. The ideal read time of task i, the amount of time to complete the read phase of task i (represented by $T_r(i)$) is determined as follows.

$$T_r(i) = \sum_{j \in dom(RS)} j \times RS_i(j)/RT(j) \tag{5}$$

Note that if the record size is statically determined to be a constant and the input data is partitioned equally across threads, then the ideal read time $T_r(i)$ for each task is the same. Also note that due to the use of black-box models and sampling, we can only *estimate* the ideal read times. We cannot guarantee that the ideal read times we compute are always less than actual read times, although this tends to be the case in practice.

Compute times. The compute phase of each task applies an operator-specific function to each input record. The time to complete the compute phase depends on the cardinality of the input and the complexity of this function. Query languages such as LINQ permit arbitrary user-defined functions to be used in conjunction with relational operators. In general, estimating the complexity user-defined functions is hard. One option for estimating complexity is static analysis [15]. However, static analysis tends to be conservative and is not designed to estimate constants in complexity, which are important for estimating execution time. Instead, we choose a simple, sampling based approach to this problem. During the sampling run, we measure the function's execution time, along with input record size. Let ET_f be the distribution of execution times of a function f for different record sizes. Given RS_i, sampling rate s, we estimate the ideal compute time of a task i (represented by $T_c i$) as follows.

$$T_c(i) = \sum_{j \in dom(RS)} RS_i(j) \times ET(j) \tag{6}$$

Write Parameters. The last operator in every stage writes the results of the stage out to a buffer and eventually to persistent storage. The parameters we must estimate for the write phase are the amount of data output by each task, the buffer size, and the rate at which the persistent storage medium can commit writes. The amount of data to be written depends on the cardinality of the input, size of output records, and the *selectivity* of the operator. The selectivity of an operator is the factor by which the operator *scales* its input.

Estimating the selectivity of an operator is challenging because it is a function of both the inputs and the operator. We use sampling to determine selectivity. For operators with a single input and output, the estimated selectivity is simply the ratio of the sampled input and output. For natural joins (which takes multiple inputs and produces a single output), we use an approach proposed by Chaudhuri et al [5] (which also describes limitations of this approach). We first perform a join over the sampled inputs and compute the sample output size(out). If the sampling rate is s, and k is the number of inputs to the join, then we use out/s^k as an estimate for query output size. To determine the output record size distribution ($RSout$) we use a technique similar to the one used to estimate input record size distributions.

Given the selectivity of the operator and the output record size distribution, we can estimate the amount of data ($DW(i)$) written by the task i.

$$DW(i) = \sum_{j \in dom(RSout)} j \times sel \times RSout_i(j) \tag{7}$$

We estimate $bufferSize$ and the latency to flush the buffer T_{flush} using micro-benchmarking. In this case, our micro-benchmarks are write intensive queries that are parametrized by the amount of data to write. We run these querying while varying the amount of data written and measure the write latencies. We observe that the data transfer rate distribution has a step curve i.e. there exists a threshold such that writing amounts less than threshold has virtually no latency, whereas writing data larger than the threshold has a high latency. We estimate *bufferSize* to be this threshold, and the buffer flush latency T_{flush} to be the latency of writing $bufferSize$ bytes.

5 Implementation

We have implemented a tool that estimates ideal execution time for DryadLINQ queries using the approach described above. The tool consists of a micro-benchmarking component, which estimates cluster and machine specific parameters. We expect the cluster administrator to run the micro-benchmarks once initially and subsequently when the cluster configuration changes.

The estimation component of the tool consumes query plans generated by the DryadLINQ compiler. Given a query plan and an input, the tool generates a sample of the input, and runs an instrumented version of the query on the sample. The instrumented version generates statistics such as input cardinalities. These statistics, along with the output of micro-benchmarking, are used to compute parameters for the model. The tool uses the query plan to build a cluster level model. The tool outputs an estimate for the ideal execution time of the query and the entire critical path.

6 Experimental Evaluation

In this section, we present a detailed evaluation of the accuracy and cost of our modeling approach. We evaluate our models using 7 real-world queries operating on representative data sets. We also evaluate the utility of the model in debugging actual executions.

Experimental Setup and Methodology. For our evaluation, we use an in-house cluster with 240 machines. Each machine has two dual-core AMD Opteron CPUs with clock speeds of 2.6 GHz, 16 GB of DDR2 RAM, four 750 GByte SATA hard drives, and 2 high speed (1 Gbps) ethernet cards. Each machine runs Windows Server 2003 (64-bit version). The machines are placed on 9 racks, each rack has a dedicated switch with three 10Gbps links. Each of these rack switches are directly connected to a central core-switch via a 30 Gbps link. The cluster uses TidyFS [11] as the distributed file system that provides necessary abstractions for data parallel computations, enabling features like replication, fault tolerance and high throughput data access.

We evaluate the modeling approach by comparing predicted ideal execution time with actual execution time for each query/input pair. We measure actual execution times by running queries directly on the cluster. We run each query 5 times and report the mean and 95% confidence intervals. We use a sampling rate of 10% for all queries, this is consistent with the sampling rate used by databases [4].

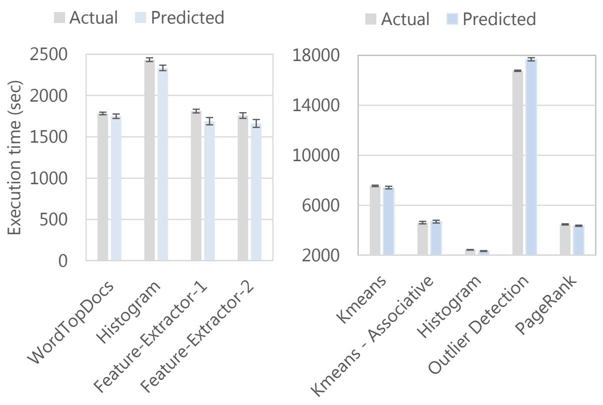

Figure 8: Prediction accuracy

Benckmarks. Table 1 describes the queries and datasets we used to evaluate the modeling approach. All queries and datasets are obtained from a production DyradLINQ cluster.

Prediction accuracy. Figure 8 shows the actual and predicted execution times with confidence intervals. For several queries, the difference in predicted and actual execution times is not statistically significant. The largest percentage error in the mean values is 7% for the feature extraction query. Also note that the predicted execution times are lower than the actual times for almost all queries. This is expected since the disk and network read and write time distributions used for predicted are computed under ideal conditions.

Next, we evaluate the prediction accuracy of the model at the stage level.

k-means. Table 2 shows the predicted ideal execution time and actual runtime for two stages, nearest center computation (NC) and new cluster center computation (CC) in 5 iterations of the k-means query (with k = 24) for two datasets. We report mean actual and predicted execution

k-means Clustering	We evaluate two versions of the query shown in Figure 2, one in which the method *computeNewCenters* is marked associative (enables eager aggregation) and another in which it is not. We use two publicly available data sets, the bag of words (pubmed) dataset [29] with 22 million articles (12GB, 5 partitions) and a data set of 47,000 books from project Gutenberg (45GB, 5 partitions).
WordTopDocs	WordTopDocs is a query that counts the frequency of words in a set of documents and returns the top k words per document. We use the Gutenberg data set (45GB, 5 partitions).
SkyServer Q18	The query [22] computes the gravitational lens effect by comparing locations and colors of stars in a large astronomical table. We use data from the Sloan Digital Sky Survey database [14] (5 billion objects, 100GB, 6 partitions).
Histogram computation	This query finds the frequency distribution of a given attribute in a dataset. We use the Gutenberg dataset (45GB, 5 partitions).
AVFScore Outlier Detection	Finds top K outliers using AVFScore[25]. We use the Gutenberg dataset (45 GB, 100 partitions).
Feature Extraction	We use two proprietary queries that perform feature extraction. They extract relevant information from phone call logs. The queries use two tables, a subscriber table and a service usage table. Experiments were conducted on a real world dataset with subscriber table (756MB) and service usage table (110GB, 15 partitions).
Page Rank	PageRank is a query for scoring web pages based on random walks over the web graph. The query keeps page ranks for each page and iteratively updates these scores using power iteration. The dataset we use contains 10 billion pages (225GB, 90 partitions).

Table 1: Queries and datasets used for evaluating the predictive model.

times since the variance across executions is low (as seen in Figure 8). The table also shows the % difference in mean values, and the overall weighted mean % difference obtained by weighting stage level difference by their actual execution time.

From a modeling perspective, k-means is a challenging benchmark for several reasons. With each iteration of k-means, the mapping of points to clusters changes, which can be hard to predict. Also note that the actual execution time of the first stage decreases over iterations for both data sets. This is because in successive iterations, the number of *active* centers i.e. centers with some points associated with them reduces as points cluster around a small number of centers. Since the number of centers reduces, the amount of time it takes to compute the nearest center for each point also reduces. Our performance model is able to predict this variation across iterations, as observed by small stage level differences. We also observe that the model is reasonably precise across two data sets. Furthermore, the model is able to predict the reduction in execution time of the second stage of kmeans when the function *CalculateNewCenter* is declared associative. This annotation allows the query engine to perform partial aggregation in the first stage and reduce the amount of data that must be transferred between stages. The experiment illustrates the utility of an accurate performance model in identifying optimization opportunities quickly at design time.

Outlier detection. The outlier detection query has six stages. The first four stages compute the frequency of all possible bi-grams across all input documents. The fifth stage computes the AVFscore per document using bi-gram frequencies. This stage is computationally expensive due to a user-defined function which computes a sum of the frequency of each bi-gram in the document. The complexity of the function is linear in the size of the document with a large constant due to the cost of looking up the bi-gram frequency distribution, which is maintained as a map in memory.

Stage	Actual Time(s)	Predicted Time(s)	Diff
Merge + GroupBy + HashPartition	291	207	28.88%
Merge + GroupBy	136	189	38.90%
Merge	24	38	58.33%
Select + Sort + Take	16200	17193	6.12%
MergeSort + Take	29	17	41.37%
Total	16755	17604	6.93%

Table 3: Stage level breakdown of the actual and predicted execution times for the outlier detection query.

Table 3 shows a per stage breakdown for this query. Our black box approach is able to estimate compute times with 6% difference on average, which eventually determines the weighted % difference.

Feature extraction. The feature extraction query is a very complex query with 13 stages. Unlike most queries where stages occur in a sequence, the stages of this query form a DAG with results from multiple sequences combined two stages, once via a join and another with a union. Table 4 shows a stage level breakdown of the execution time. Again, our approach is able to estimate the execution time of long running stages with high accuracy. The % difference for short running stages are higher due to high errors in cardinality estimation.

Stage	Iteration	Bag Of Words Actual Time(s)	Bag Of Words Predicted Time(s)	Bag Of Words Diff	Gutenberg Actual Time(s)	Gutenberg Predicted Time(s)	Gutenberg Diff	Gutenberg(Associative) Actual Time(s)	Gutenberg(Associative) Predicted Time(s)	Gutenberg(Associative) Diff
NC	1	420	413	1.67%	1489	1412	5.17%	1612	1567	2.79%
CC		316	293	7.28%	445	418	6.07%	39	35	10.25%
NC	2	529	607	14.74%	874	805	7.89%	879	905	2.95%
CC		351	311	11.40%	887	857	3.38%	33	35	6.00%
NC	3	133	115	13.53%	455	442	2.86%	695	742	6.76%
CC		327	312	4.59%	801	784	2.12%	39	35	10.25%
NC	4	135	113	16.30%	500	487	2.60%	776	762	1.80%
CC		331	310	6.34%	847	812	4.13%	37	35	5.40%
NC	5	135	114	15.56%	437	486	11.21%	537	571	6.30%
CC		333	309	7.21%	789	811	2.79%	32	35	9.30%
Total		3009	2856	9.26%	7524	7314	4.68%	4679	4722	3.87%

Table 2: Stage level breakdown of the actual and predicted execution times for the k-means query.

Stage	Actual Time(s)	Predicted Time(s)	Diff
Where	1136	1089	4.14%
Sort + HashPartition	65	45	30.77%
MergeSort + GroupBy + Select + HashPartition	35	27	22.86%
Sort + OrderedGroupBy + HashPartition	305	325	6.56%
DryadMerge	15	2	86.67%
HashJoin + Partition	78	55	29.45%
Merge	9	0	100.00%
Where + Sort +Partition	89	78	12.36%
Sort + GroupBy + Partition	45	33	26.67%
Merge	9	0	100.00%
Union	45	34	24.44%
Sort + Partition	67	55	17.91%
Merge + GroupBy + Where	78	66	15.38%
Total :	1733	1626	8.48%

Table 4: Stage level breakdown of the actual and predicted execution times for feature extraction query.

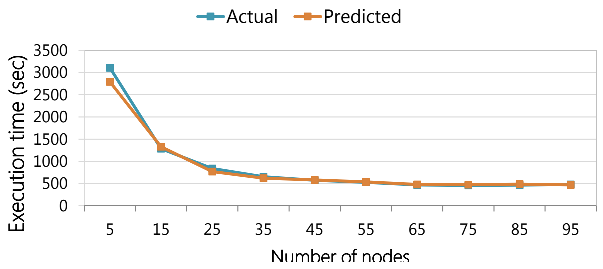

Figure 9: Prediction accuracy for *kmeans* query for varying number of machines.

Sensitivity Analysis. A key decision in using the critical path method is identifying dependencies between tasks. The experiments above show that our choice of dependencies leads to simple and precise models. However, are all dependencies we model required to achieve high precision, or can the framework be simplified further by eliminating some dependencies? To answer this question, we perform a simple experiment. For each dependence, we build an alternate model that ignores the dependence. We also build a model where a dependence exists between every C_i and R_{i+1}, modeling a processor with one core. The column *All* represents the baseline model that considers all dependencies as described above.

Observe that the prediction accuracy is impacted significantly if any of the dependencies are dropped. Modeling constraints on read and compute parallelism is particularly important. The dependence due to finite output buffer is relatively the least significant. However, ignoring this dependence can cause the model to under-approximate execution time by as much as 10%.

Modeling Time. An important parameter for any prediction tool is the time taken to predict the execution time. Table 6 shows the time taken by our modeling tool to predict execution time of various queries. We report time spent in running the query with the sample input, and in building critical path models and computing the critical path. Observe that the total modeling time is a fraction of the query execution time (4-5% on average).

Design space exploration. We used our modeling approach to explore the effect of varying number of machines on the performance of the *kmeans query*. Figure 9 shows the mean actual and predicted execution times for

Benchmark	Actual Time(s)	Sample Time(s)	Modeling Time(s)	Total Time(s)	% Total
kmeans (Gutenberg)	7521	41	55	96	1.32%
WordTopDoc	1768	41	143	184	10.24%
SkyServer	5104	73	178	251	4.19%
Histogram	2414	44	34	78	3.06%
Feature-Extractor-1	1821	25	56	81	4.44%
Feature-Extractor-2	1733	38	67	105	6.05%
Outlier Detection	16755	245	149	394	2.34%
Page Rank	4448	89	189	278	6.25%

Table 6: Total prediction time for benchmark applications.

the *kmeans* query running the Gutenbeg data set. We observe that the predicted execution times closely tracks the actual execution time. Thus, our model can help end users make crucial design time decisions with reasonable accuracy without performing expensive runs.

Debugging. We performed a simple experiment to evaluate the use of our models for debugging. During an execution of k-means, we simulated a slow disk on one of the machines in the first stage (by injecting delays in calls to disk reads). Not surprisingly, injecting these delays slows the execution. We then compute the critical path with and without the delay (both at the stage level and the cluster level). This slowdown causes a change in the critical path. Specifically, the machine with the simulated delays exists on the actual critical path but not on the predicted critical path.

Figure 10 shows both predicted and actual critical path of the slow machine (some dependences and weights not shown for clarity). Note the change in the critical path from one that alternates between read and compute phases to a path that goes through read phases exclusively. Therefore, one can infer that the performance problem is due to increases read latencies. Finding out why some phases move to the critical path requires a deeper investigation. However, narrowing the potential cause can help isolate the problem and drive optimizations. In this case, it is possible to improve performance simply by excluding the slow machine.

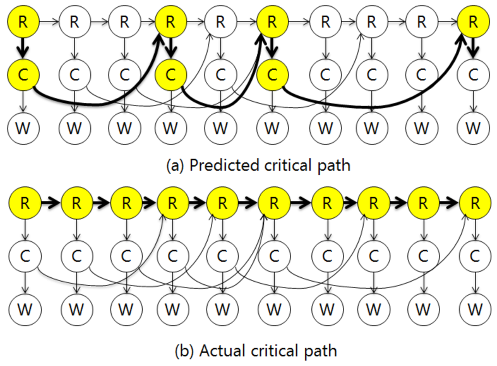

Figure 10: The difference in critical paths of predicted and actual executions of k-means due to injected delays.

7 Related Work

Parallel database engines utilize performance models during query optimization to cost query plans and compare alternatives. Early optimizers [18, 21] adopted a two step strategy where the optimal single machine query plan is identified as the first step and the chosen plan is parallelized as a second step. Such optimizers do not account for parallelism during costing. The cost metrics used by most single step optimizers [13, 27] are typically a vector containing one cost per resource considered. A single cost metric is then computed using a simplified weighting scheme [34]. Many models are built for specific parallel systems (like shared disk and shared memory) that have not found wide spread adoption [26]. RoPE [2] revisits the problem of query optimization in a modern distributed query engine. RoPE utilizes the fact that in such systems queries are often executed repeatedly with little change in workloads. They propose a adaptive query optimization technique that piggybacks information collected over previous executions to optimize and improve the query plan over multiple runs. Cost estimates in query optimization are only used to compare alternatives. None of the models take into account inter and intra machine parallelism within and across resources to the extent that we do in this paper and hence their cost estimates may not be accurate representations of execution time.

Recently a few optimizers have also been proposed for map reduce style jobs written in imperative languages.

Query	All	No $R_i \rightarrow R_j$	No $C_i \rightarrow R_j$	$C_i \rightarrow R_{i+1}$	No $W_i \rightarrow W_j$
k-means	7741	2298	5672	9589	7206
SkyServer	5011	1784	4321	7751	4981
Feature-extraction	1722	756	1439	2157	1688
Page rank	4562	1876	4125	4785	4012
Outlier	17604	6345	11563	21005	18013

Table 5: Sensitivity analysis for dependencies.

Manimal [24] is a tool to analyze low level map-reduce programs and identify opportunities for data centric optimizations (some of which are also used by database systems) like data filtering, projection and data specific compression. Herodotou et al. [19] propose profile driven tool that enables what-if analysis to identify the best configuration parameters for a given query. In cotrast, we target declarative query languages and exploit the semantics of the operators to build detailed white-box models. We also do not rely on information collected from full runs of the job.

Osman et al. [33] propose analytical models based on queuing networks to compare different database design alternatives. Disk I/O cost is the response variable tracked by these queuing models. STEADY [8] uses queuing networks to estimate the response time of a query in a parallel database. They map queries to pre-identified resource usage profiles and then use heuristic rules to labels resources as M/M/1 or M/G/1 queuing systems. Our model differs from these as it targets query response time and models resource parallelism directly using critical path models. An alternate approach for predicting query performance is the use of black-box statistical and machine learning models [12, 7]. In general, black box approaches suffer from the problems of finding the right feature vectors. Here, we take a mixed approach to the use of black box models, preferring white box models that expoit the semantics of relational operators as far as possible.

A key component of a performance model for database engines is a cardinality estimator [17, 5, 9, 4, 16]. It predicts the size of the output of a given operator in a query. Cardinality estimation is especially challenging for queries with joins and aggregations. At a high level, approaches for cardinality estimation can be classified into four categories, namely histograms, sampling, curve fitting and parametric methods. Harangsri et al provide a complete survey of query size estimation techniques in database systems. Chaudhuri et al. [5] study sampling in the context of joins and investigate the limits of sampling based approaches. Haas et al. [16] survey many distinct value estimators (used to estimate number of groups in group-by/aggregation queries) and propose a few new ones. Charikar et al. [4] improve on these and propose a hybrid estimator with some theoretical guarantees. We utilize cardinality estimation in our models to estimate the number of tasks that an operator will employ.

Another problem closely related to query time estimation is the progress estimation [28, 6]. Here the goal is to roughly estimate the progress of a query. These papers propose analytical models for progress estimation while assuming the presence of an accurate cardinality estimator. Parallax [31] extends these approaches to estimate the progress of MapReduce Jobs. ParaTimer [30] and Jockey [10] further adapt progress estimators for distributed frameworks by providing multiple estimations, each accounting for different possible future behaviors (failures, reduced resources etc). Progress estimators have the advantage that they can use feedback from the execution in progress to refine estimates, an option not available at design time. Progress estimates cannot be used for making design time optimizations like query refactoring, estimating resource requirements and evaluating the cost of running a query.

References

[1] Linq : Language integrated queries http://msdn.microsoft.com/en-us/library/bb397926.aspx.

[2] Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. Re-optimizing data-parallel computing. In *Networked Systems Design and Implementation* (2012), NSDI'12.

[3] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. Flumejava: easy, efficient data-parallel pipelines. In *Programming language design and implementation* (2010), PLDI '10.

[4] Charikar, M., Chaudhuri, S., Motwani, R., and Narasayya, V. Towards estimation error guarantees for distinct

values. In *symposium on Principles of database systems* (2000), PODS '00.

[5] Chaudhuri, S., Motwani, R., and Narasayya, V. On random sampling over joins. In *SIGMOD Rec.* (June 1999), vol. 28, ACM, pp. 263–274.

[6] Chaudhuri, S., Narasayya, V., and Ramamurthy, R. Estimating progress of execution for sql queries. SIGMOD '04.

[7] Chun, B.-G., Huang, L., Lee, S., Maniatis, P., and Naik, M. Mantis: Predicting system performance through program analysis and modeling. *CoRR abs/1010.0019* (2010).

[8] Dempster, E. W., Williams, M. H., Tomov, N., Pua, C. S., Burger, A., and King, P. J. B. Steady - a tool for predicting performance of parallel dbmss. In *Computer Performance Evaluation: Modelling Techniques and Tools* (2000), TOOLS '00.

[9] Dobra, A. Histograms revisited: when are histograms the best approximation method for aggregates over joins? In *symposium on Principles of database systems* (2005), PODS '05.

[10] Ferguson, A. D., Bodk, P., Kandula, S., Boutin, E., and Fonseca, R. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*.

[11] Fetterly, D., Haridasan, M., Isard, M., and Sundararaman, S. Tidyfs: a simple and small distributed file system. In *USENIX annual technical conference* (2011), USENIX-ATC'11.

[12] Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., and Patterson, D. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *International Conference on Data Engineering* (2009), ICDE '09.

[13] Ganguly, S., Hasan, W., and Krishnamurthy, R. Query optimization for parallel execution. *SIGMOD Rec. 21*, 2 (June 1992), 9–18.

[14] Gray, J., Szalay, A. S., Thakar, A., Kunszt, P. Z., Stoughton, C., Slutz, D. R., and vandenBerg, J. Data mining the sdss skyserver database. Tech. rep., 2002.

[15] Gulwani, S., Mehra, K. K., and Chilimbi, T. Speed: precise and efficient static estimation of program computational complexity. *POPL '09*.

[16] Haas, P. J., Naughton, J. F., Seshadri, S., and Stokes, L. Sampling-based estimation of the number of distinct values of an attribute. In *International Conference on Very Large Data Bases* (1995), VLDB '95.

[17] Harangsri, B. *Query Result Size Estimation Techniques in Database Systems*. PhD thesis, The University of New South Wales School of Computer Science, 1998.

[18] Hasan, W., and Motwani, R. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Very Large Data Bases* (1994), VLDB '94.

[19] Herodotou, H., and Babu, S. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB, 2011*.

[20] Hive. http://hive.apache.org/.

[21] Hong, W. Exploiting inter-operation parallelism in xprs. In *international conference on Management of data* (1992), SIGMOD '92.

[22] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys European Conference on Computer Systems 2007* (2007), EuroSys '07.

[23] Isard, M., and Yu, Y. Distributed data-parallel computing using a high-level programming language. SIGMOD '09.

[24] Jahani, E., Cafarella, M. J., and Ré, C. Automatic optimization for mapreduce programs. *PVLDB*.

[25] Koufakou, A., Ortiz, E. G., Georgiopoulos, M., Anagnostopoulos, G. C., and Reynolds, K. M. A scalable and efficient outlier detection strategy for categorical data. In *International Conference on Tools with Artificial Intelligence - Volume 02* (2007), ICTAI '07.

[26] Kremer, M., Kremer, M., Gryz, J., and Gryz, J. A survey of query optimization in parallel databases. Tech. rep., 1999.

[27] Lanzelotte, R. S. G., Valduriez, P., and Zaït, M. On the effectiveness of optimization search strategies for parallel execution spaces. In *Very Large Data Bases* (1993), VLDB '93.

[28] Luo, G., Naughton, J. F., Ellmann, C. J., and Watzke, M. W. Toward a progress indicator for database queries. In *international conference on Management of data* (2004), SIGMOD '04.

[29] Machine Learning Repository, U. http://archive.ics.uci.edu/ml/datasets/bag+of+words.

[30] Morton, K., Balazinska, M., and Grossman, D. In *SIGMOD Conference, 2010*.

[31] Morton, K., Friesen, A., Balazinska, M., and Grossman, D. In *International Conference on Data Engineering*, ICDE '10.

[32] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *international conference on Management of data* (2008), SIGMOD '08.

[33] Osman, R., Awan, I. U., and Woodward, M. E. Performance evaluation of database designs. In *International Conference on Advanced Information Networking and Applications* (2010), AINA '10.

[34] Spiliopoulou, M., Hatzopoulos, M., and Cotronis, Y. Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline. *IEEE Trans. on Knowl. and Data Eng.* (1996).