Turbo: Dynamic and Decentralized Global Analytics via Machine Learning

Hao Wang, Di Niu, *Member, IEEE*, and Baochun Li, *Fellow, IEEE*

Abstract—Big data analytics are practiced in many fields to extract insights from massive amounts of data. With exponential growth in both the volume and variety of data, analytic queries have expanded from those executed in a single datacenter to those requiring inputs from multiple datacenters that are geographically separate or even globally distributed. Unfortunately, the software stack that supports data analytics is designed originally for a cluster environment and is not tailored to execute global analytic queries, where resources such as inter-datacenter networks may vary on the fly. Existing optimization strategies that determine the query execution plan before its execution are not able to adapt to resource variations at query runtime. In this article, we present Turbo, a lightweight and non-intrusive global analytics system that can dynamically adjust query execution plans for geo-distributed analytics in the presence of time-varying resources and network bandwidth across datacenters. Turbo uses machine learning to accurately predict the time cost of a query execution plan so that dynamic adjustments can be made to it when necessary. Turbo is non-intrusive in the sense that it does not require modifications to the existing software stack for data analytics. We have implemented a real-world prototype of Turbo, and evaluated it on a cluster of 33 instances across eight regions in the Google Cloud Platform. Our experimental results have shown that Turbo can achieve an accuracy of 95 percent for estimating time costs, and can reduce the query completion time by 41 percent.

Index Terms—Data analytics, distributed systems, machine learning

1 Introduction

Major global organizations host their services on datacenters that are geographically distributed across continents to reduce the service latency to end-users at the edge of the Internet. Terabytes (TBs) of data, such as user activity and system operational logs, are recorded at these geo-distributed datacenters on a daily basis. The capability of deriving insights from such vast volumes of geo-distributed data is critical to business intelligence, personalization, online advertising and system operation optimization.

Although data analytics are widely practiced within the context of a single datacenter with frameworks such as Spark [2] and Hive [3], analyzing data across multiple geo-distributed datacenters requires additional mechanisms that carefully control the bandwidth cost and latency, mostly due to the limited available bandwidth for data transmission between datacenters.

Recent studies [4], [5], [5], [6], [7] have pointed out that centralizing all data at one datacenter for data analytics may not be efficient due to the usage of excessive bandwidth. In response to solving the challenge of geo-distributed data analytics, several decentralized solutions have been proposed: Clarinet [4] and Geode [5] have advocated that computation (such as `filter` and `scan` operations) should be pushed closer to data in local datacenters and that data transfers should be carefully optimized during the reduce phases (such as joins) across datacenters.

Unfortunately, most of these existing decentralized solutions [4], [6], [8], [9] require re-engineering the full stack of the underlying data processing utility, including mechanisms for data replication, reducer placement, task scheduling, and query execution strategy selection. These schemes are not only reinventing the wheels, but solving joint optimization problems across multiple layers in the design space will also lead to significant overhead and delayed response to runtime dynamics, such as fluctuations in bandwidth availability. Geode [5] has proposed an alternative solution that estimates the volume of the shuffling traffic between datacenters by simulating the query execution in a single datacenter. However, this sandbox-like approach incurs quite a significant overhead, as well. It will not be accurate if a query is not recurring or if the inter-datacenter network bandwidth is time-varying. These facts have hindered the deployment of these state-of-the-art decentralized solutions to geo-distributed analytics in production environments.

In contrast to the existing literature, we advocate that decentralized query execution for global analytics must be orchestrated dynamically at runtime to release its promised potential. An "optimal" static query execution plan (QEP) is typically predetermined through cross-layer optimization or simulation. Such static QEPs are unlikely to remain optimal during query execution on large tables, since resources in the datacenter network, especially the inter-datacenter bandwidth, naturally vary over time during query execution.

- *H. Wang and B. Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto M5S 2E4, Canada. E-mail: haowang@ece.utoronto.ca, bli@ece.toronto.edu.*
- *D. Niu is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Alberta T6G 1H9, Canada. E-mail: dniu@ualberta.ca.*

Manuscript received 14 Nov. 2018; revised 21 Dec. 2019; accepted 1 Jan. 2020. Date of publication 7 Jan. 2020; date of current version 29 Jan. 2020. (Corresponding author: Hao Wang.) Recommended for acceptance by H. Childs. Digital Object Identifier no. 10.1109/TPDS.2020.2964667

In this paper, we propose Turbo, a lightweight and non-intrusive system that orchestrates query execution plans for geo-distributed analytics, by dynamically altering query execution plans on-the-fly in response to resource variations. At the core of our system is a carefully designed machine learning algorithm that can accurately estimate the time cost and the intermediate output size of any reduce and shuffle stage (e.g., joins) of the query execution. Such estimates are performed online, given the size of the input tables, the instantaneously measured bandwidth, hardware (e.g., memory and CPU) configurations, as well as other observable parameters in the underlying distributed computing engine. Our system offers a *non-intrusive* solution, which does not require modifications to any lower-layer functionalities regarding task placement, task scheduling, data replication or query operator optimization. Through machine-learning-based runtime cost prediction and query orchestration, Turbo works effectively on top of any existing underlying data analytics utility, such as Spark and Hive.

Toward a realistic design and implementation of Turbo, we have made the following original contributions:

Measurements of Real-World Queries Executed in a Geo-Distributed Cloud. We measured the Google Cloud platform, and our results suggest that inter-datacenter bandwidth could fluctuate dramatically within several minutes, a time span that is comparable to or even shorter than the typical time to run a global analytics query. We conduct case studies based on the TPC-H benchmark [10] dataset, which contains realistic data tables and queries with a broad industrial relevance. We observe that for a typical query in TPC-H executed on geo-distributed datacenters, dynamically reordering the joins during query execution can reduce the query completion time by as much as 40 percent.

Machine Learning for Runtime Cost Prediction. A common existing technique for estimating the time cost of data shuffling in geo-distributed analytics is to solve a reducer placement problem to minimize the longest link finish time [6] or the sum of finish times [4], under some assumed models for network transfer times. In reality, it is cumbersome to solve these optimization problems whenever bandwidth varies. The actual bandwidth used by system frameworks, such as Spark, is not the same as the bandwidth available on the link in question. Furthermore, the shuffle time also critically depends on the specific execution and sorting algorithms adopted, such as broadcast joins versus hash joins. Rather than using empirically assumed models and joint optimization, we apply a machine learning approach to predict the cost of any decentralized table joins based on the characteristics of the input tables, the underlying system as well as the network. In particular, we have designed a large nonlinear feature space for both the data and the system, and leveraged the ability of statistical and machine learning methods, such as LASSO and Gradient Boosting Regression Tree (GBRT) to sort out the most relevant features. Our machine learning based cost prediction is not only fast, but overcomes the limitation of any empirically assumed models and avoids the complications of joint optimization of task placement, scheduling, and query execution.

Non-Intrusive Query Execution Plan Orchestration. Based on the lightweight lookahead cost prediction mentioned above, in Turbo, we further design a logic-layer query orchestration mechanism to adjust the join orders in a QEP dynamically in response to changes in runtime dynamics. Turbo greedily chooses the next join it will execute to be the one with the least predicted cost, in terms of three different policies: 1) the least completion time, 2) the maximum data reduction, and 3) the maximum data reduction rate. By focusing on forward-looking join reordering based on cost prediction, Turbo is orthogonal to and complements any mechanisms in the lower-level distributed computing engine. Turbo can be used on top of off-the-shelf query optimizers, such as Calcite [11], taking advantage of the existing expertise developed for query optimization over the last few decades.

We have implemented a prototype of Turbo and deployed it on a fairly large Google Cloud cluster with 33 instances spanning eight geographical regions. Our experimental results have suggested clear evidence that Turbo can achieve a cost estimation accuracy of over 95 percent and reduce the query completion time by up to 41 percent.

2 Background and Motivation

We first review how modern data analytics frameworks, e.g., Spark [2] and Hadoop [12], execute SQL queries in a geo-distributed setting, and then use measurements and case studies based on the TPC-H benchmark [10] dataset including 15K records to show the inefficiency of all existing *static* solutions to geo-distributed analytics.

Processing and Optimizing SQL Queries. In Hive [3] and Spark SQL [13], a SQL query is first parsed into a tree called a *query execution plan*, consisting of a series of basic relational operations, such as filter, scan, sort, aggregate, and join. These relational operations are subsequently transformed by a distributed computing engine, such as Spark, to parallel map and reduce tasks, which are logically organized in a directed acyclic graph (DAG) and executed in stages following the dependencies dictated by the DAG. For instance, operators such as `SELECT`, `JOIN` and `GROUPBY` are transformed into individual map-reduce stages in the DAG.

Essentially, any SQL query involving multiple tables can be parsed into multiple feasible QEPs. Though these QEPs have the same analytic semantics, they have different orderings of joins, or different strategies and algorithms to execute joins. Then, the query optimizer selects an optimal QEP based on either predefined rules (i.e., rule-based optimization) or cost models (i.e., cost-based optimization).

Query optimizers play a critical role in database technologies and have been extensively studied for decades [14], [15], [16], [17], [18]. Modern query optimizers in state-of-the-art products, e.g., Apache Calcite [11] and Apache Phoenix [19], are well suited for centralized or parallel databases within a single datacenter. In massively parallel processing (MPP) databases, high-performance data warehouse frameworks—such as AWS RedShift [20], Apache Hive [21] and Apache HBase [22]—can select a low-latency query execution plan using a wide range of sophisticated optimization techniques involving both rule-based planning and cost-based modeling. These optimization techniques will make a wide variety of informed choices, such as between range scans and skip scans, aggregate algorithms (hash aggregate versus stream aggregate versus partial stream aggregate), as well as join strategies (hash join versus merge join).

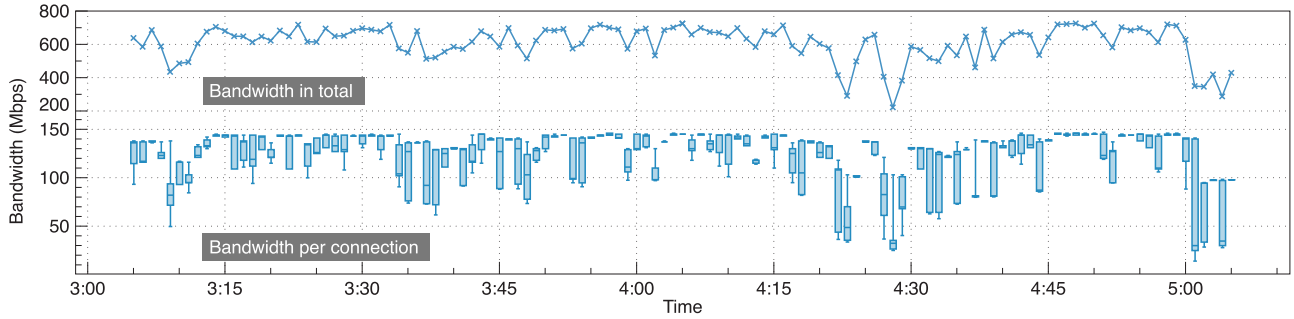


Fig. 1. Wide-area bandwidth fluctuations between two geographic regions on the Google Cloud platform.

Optimizing Geo-Distributed Data Analytics. Yet, existing query optimization technologies designed for a single datacenter, such as Calcite [11] and Phoenix [19], may not perform well in *geo-distributed* analytics, mainly due to the heterogeneous network capacity in wide-area networks.

To address such a design deficiency, Geode [5] extends Apache Calcite [11] to *Calcite++*. Geode adopts the query execution plan (including the order of joins) generated by Calcite. It incorporates additional statistics to choose the optimal join strategies (e.g., hash joins and broadcast joins) and reduce the cost of shuffle across wide-area networks. To estimate the size of network transfers in any QEPs, it uses pseudo-distributed execution to simulate running the QEP in a centralized datacenter. Geode uses measurements from the simulation to optimize site selection and data replication decisions to reduce the volume of inter-datacenter traffic. However, such a "sandbox" approach may lead to high overhead and suboptimal performance if the same queries are not recurring, or if the network link cost fluctuates at runtime.

Clarinet [4] further explicitly minimizes query execution times in geo-distributed analytics in a *static* manner, by choosing the shortest running QEP with the optimal join ordering among many feasible candidate QEPs. Since the traffic between datacenters depends on the placement of reduce tasks in each stage of a QEP, Clarinet proposes to optimize both reducer placement and task scheduling jointly. It decouples the complex optimization problem in each stage using various heuristics to minimize the QEP execution time. Clarinet greedily selects the QEP with the shortest execution time. As a "clean slate" design, Clarinet re-engineers the full spectrum of the design space of data analytics. Since existing state-of-the-art query optimizers and parallel databases cannot reuse such design, Clarinet may not be amenable to evolutionary deployment in practice.

2.1 The Need for Dynamic Query Planning

We now show through measurements that significant bandwidth variations exist on inter-datacenter links. Therefore, even an initially optimal QEP may become suboptimal during query execution. We argue that QEPs need be adjusted dynamically *at runtime* in response to bandwidth changes instantaneously, especially for those queries involving multiple shuffle stages with multiple `JOIN` operators.

A major challenge in geo-distributed analytics is that data shuffling in reduce stages must traverse links in wide-area networks. Since cloud providers do not provide performance guarantees for inter-region traffic on the public Internet [23], [24], the available bandwidth on these links is fluctuating in nature [7], especially when flows of different applications share the links. To demonstrate such variations in inter-datacenter bandwidth availability, we measured inter-region bandwidth for two hours on Google Cloud, by launching two instances in separate geographic regions, `asia-east1` (Taiwan) and `us-central1` (Iowa). Each instance has a large ingress or egress bandwidth cap that is well above 2 Gbps.

To saturate the bandwidth between the two instances and measure the amplitude and frequency of bandwidth variations, we executed `iperf -t10 -P5` for 10 seconds involving five parallel connections and repeated the command once every 50 seconds. Fig. 1 shows that both the total available bandwidth and the available bandwidth in each connection changed rapidly at the frequency of minutes. In particular, the per-connection bandwidth fluctuated between 14 Mbps and 147 Mbps, while the total bandwidth fluctuated between 222 Mbps and 726 Mbps. Moreover, we found that the round-trip time (RTT) between the two instances during the measurement was consistently between 152 and 153 milliseconds. Such stable RTTs indicate that the routing path between the two instances remained unchanged. Therefore, we conclude that the variations of available bandwidth were due to the contention and sharing among flows (of different applications), which is common in the wide-area networks.

We illustrate the benefit of adjusting QEPs dynamically with a simple example based on real-world data. We ran a SQL query, as shown in Fig. 3a, which joins four tables of data realistic data generated by TPC-H benchmark [10]. The tables are stored on four separate sites with heterogeneous inter-site bandwidth capacities, as shown in Fig. 2. Note that the bandwidth within each datacenter is 12 Gbps, which is much larger than inter-site bandwidth.

We executed this query with four different strategies for QEP selection: 1) the centralized mode, in which all the tables will be moved to DC_3 for aggregation, and the computation is only performed at DC_3;[1] 2) the distributed baseline, in which a static QEP is pre-determined by the default query optimizer of Spark SQL; 3) Clarinet[2] [4], which selects a static distributed QEP by jointly optimizing task placement and scheduling; 4) a dynamically adjusted and distributed QEP, which adjusts the QEP at runtime in response to bandwidth fluctuations.

1. It should be noted that the data movement time is part of the query completion time since data is initially distributed on different sites.

2. We have only reproduced the query selection of Clarinet since the source code of its task scheduling and network management is not open-sourced.

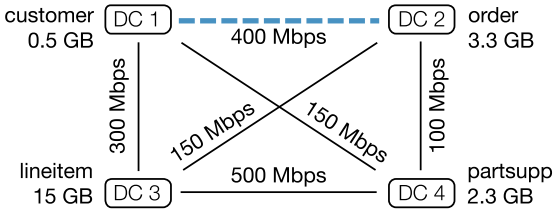


Fig. 2. The cluster setup in our example showing the benefits of runtime QEP orchestration.

For the purpose of illustration, we conducted this experiment in a controlled geo-distributed cluster of four sites [25] with stable dedicated links between the sites. To emulate bandwidth variations which would otherwise have been observed in the wide-area network, we replayed the real-world bandwidth traces collected from Google Cloud, as shown in Fig. 3c. Specifically, we periodically updated the Linux `tc` traffic control rules on both sides of the link between DC_1 and DC_2, while other links in our cluster remained stable during the experiment.

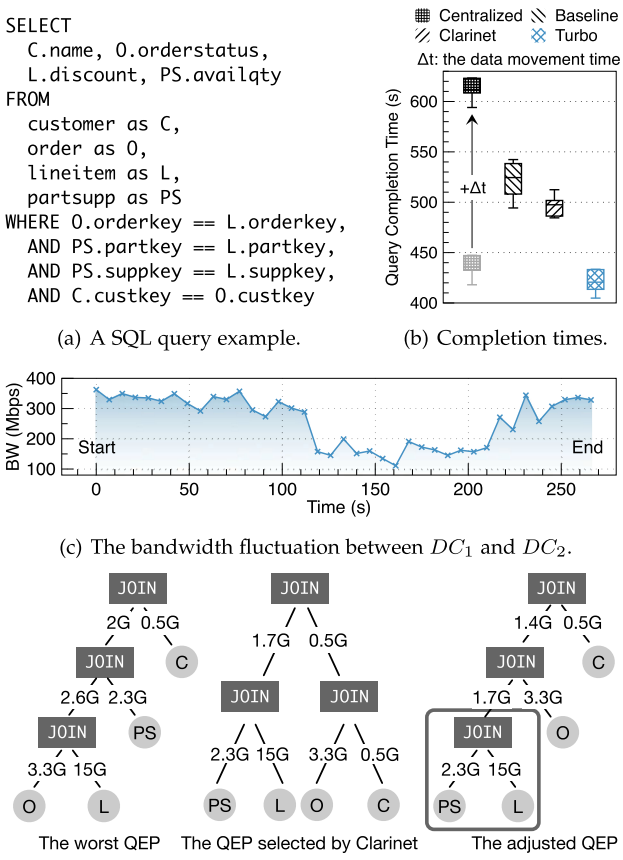
For each of the four compared strategies, we ran the same query five times under the same bandwidth configuration. The query completion times in Fig. 3b showed that the centralized mode took 611.7 seconds on average to finish. The baseline took 524.5 seconds on average to finish, while Clarinet took 497.5 seconds on average. In contrast to the other methods, adjusting QEP at runtime only took 420.7 seconds on average, which was the fastest.

Let us now analyze the QEPs chosen by the four strategies, which explain the performance gaps. The centralized mode spent as long as 176 seconds transferring data to one site and is thus inferior to other distributed methods in general. Since the baseline (default Spark) was oblivious to network bandwidth, its chosen QEP may inadvertently have joined two large tables over a narrow link such as the 150 Mbps link between DC_2 and DC_3. In contrast, Clarinet was aware of the heterogeneity of link capacities in the wide-area and selected the (static) QEP that was optimal only for the initial bandwidth availability. However, the bandwidth changes between DC_1 and DC_2 soon rendered the selected QEP a suboptimal solution, delaying the join between DC_1 and DC_2 significantly. With runtime adjustments during query execution enabled, although the initial QEP was the same as the one selected by Clarinet, after the join of 2.3 GB `partsupp` table and the 15 GB `lineitem` table, the execution plan was changed to the third one in Fig. 3d. The adjusted QEP avoided transmitting a large volume of data over the link between DC_1 and DC_2, when the bandwidth on the link dropped below 150 Mbps.

Abrupt bandwidth changes are not uncommon in a public cloud shared by many applications. Furthermore, such bandwidth changes, as illustrated by the example above, may occur multiple times during the execution of a geo-distributed query, especially for large data-intensive jobs. Therefore, an initially optimal QEP is not necessarily optimal throughout the job.

3 Overview

It is significantly challenging to adjust a distributed query execution plan at runtime dynamically. First, recomputing



(a) A SQL query example. (b) Completion times.

(c) The bandwidth fluctuation between DC_1 and DC_2.

(d) QEPs of the three cases.

Fig. 3. An example SQL query and its QEP choices.

the optimal query execution plan with complex optimization methods, such as Clarinet [4] or Iridium [6], is not feasible at runtime—once a new solution is computed, the bandwidth availability would have changed again. Moreover, since these solutions often involve a joint optimization problem of reducer placement and task scheduling, they require modification to lower layers in the data analytics engine.

We present Turbo, a lightweight *non-intrusive* layer that dynamically orchestrates distributed query execution. Fig. 5 provides an overview of Turbo's architecture. Turbo works independently on top of and complements existing distributed data analytics engines such as Spark. It reduces query completion times by switching the order at which different distributed tables should be joined, in response to network bandwidth changes. We have designed a machine learning engine judiciously to enable the online lookahead reordering of join operations. The machine learning engine predicts the time cost as well as the output data size (cardinality) of joining a pair of tables distributed on two different datacenters, based on network statistics and tables to be joined. We also introduce a lightweight bandwidth measurement scheme, which can probe instantaneous inter-datacenter link bandwidth in a non-intrusive manner. Note that Turbo leaves any lower-layer decision intact as is in existing systems, including task scheduling, reducer placement and specific join algorithms.

Typically, Hive and Spark SQL convert a QEP into a DAG of map-reduce stages. The tasks within a stage are

Fig. 4. Interpreting a pairwise join to map-reduce stages.

atomic threads executing exactly the same code, yet applying to different blocks of data. There are two types of mappings between operators and stages: an operator applied to only a single table is mapped to a single map stage, and an operator involving two tables is mapped to two map stages and a reduce stage. For example, a `SELECT` operator $\sigma_{price>100}(orders)$ is interpreted as a map stage `filter (order o => (o.price>100))`, while a natural join `customer` $\bowtie$ `orders` is interpreted as two map stages, `map (customer c => (c.custkey, c.values))` and `map (order o => (o.custkey, o.values))`, as well as a reduce stage `reduce(custkey, values)`.

In Turbo, the smallest unit for an adjustment we will focus on is a pairwise join. A geo-distributed analytics query can be parsed into a series of pairwise joins. Each pairwise join involves only two tables as well as some affiliated map stages in the join due to operators on every single table such as selection (σ) and projection (π). On an abstract level, each pairwise join is interpreted with map and reduce stages, as shown in Fig. 4, with the additional details on query optimization and execution strategy optimization hidden. Turbo aims to adjust a QEP dynamically at runtime by *reordering the pairwise joins* in it during query execution. For the sake of speeding up geo-distributed data analytics, it is critical to concentrate on operators that trigger data shuffles in the network such as `JOIN`, while leaving the optimization of map stages performing local computations to the lower-level system. We show that by focusing on smart online join reordering, query execution times can be reduced significantly for real-world workloads.

As in Fig. 5, the architecture of Turbo consists of three components:

Model Training. The abstraction of a query into a series of pairwise joins makes cost estimation feasible through machine learning. Turbo trains two machine learning models, LASSO and Gradient Boosting Regression Tree, to predict the time cost and output cardinality of a pairwise join involving two tables. The input features come from the tables, the underlying platform, hardware configurations, as well as the bandwidth probed on WAN links. The training can be done based on samples collected from past executions. In our experiment, we have created a labelled dataset of 15,000 samples containing both the target cost values and features, by running a series of realistic pairwise joins on datasets of TPC-H benchmark [10] under a variety of hardware and software settings. The models can be trained offline or updated incrementally, with new samples added to the training dataset. The underlying philosophy of our model training engine is to handcraft a range of relevant features, using feature crossing to introduce non-linearity and to extend the feature space (Section 4.2). And we filter out the irrelevant features with the model selection capability of LASSO and Gradient Boosting Regression Tree. Our machine learning model yields a prediction accuracy of 95 percent on

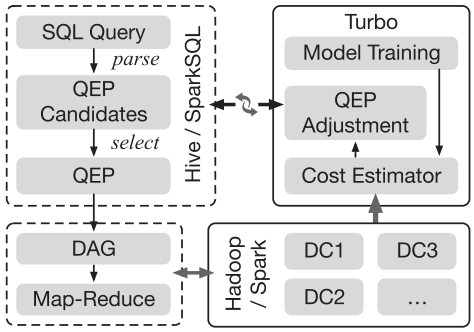


Fig. 5. The overview of Turbo.

queries of TPC-H benchmark [10], which is sufficiently accurate for online join reordering to exhibit its benefits.

Cost Estimator. During query execution, the trained machine learning models will predict the cost of a pairwise join directly. When the two tables involved in the join each reside on a different datacenter, the machine learning models predict the cost based on instantaneously measured runtime dynamics, including the available bandwidth between datacenters probed by a lightweight non-intrusive bandwidth prober. However, note that the records in a table may be distributed in several datacenters, e.g., if these records are intermediate results of a previous reduce stage, due to shuffling. If at least one input table of the pairwise join in question is distributed in more than one datacenter, we further propose a scheme in Section 5.1 to estimate the cost of this pairwise join. Our cost estimator uses the trained machine learning model as a fundamental building block and generalizes it to the case of distributed input data based on an abstract model of parallel execution.

Runtime QEP Adjustment. Turbo reuses the query processor (including a query parser and a query optimizer) of Hive or Spark SQL to parse SQL semantics. Complicated SQL query dependencies are parsed into tree-like QEPs by the query processor. Turbo first examines all QEP candidates produced by the query processor, and then only considers the pairwise joins found in the QEP candidates. The objective of runtime QEP adjustment is to minimize the overall completion time of a data-intensive query in an unstable geo-distributed environment. However, at any point in time, given the parts of the query that have already been executed, the search space for the optimal ordering of remaining joins is still exponentially large. To enable swift decision making, Turbo continuously adapts the QEP to runtime dynamics by greedily choosing the next pairwise join with the least lookahead cost. In Section 5.2, we propose three greedy policies, evaluating such lookahead cost in three different perspectives. Although the proposed greedy policies are still suboptimal—the optimal dynamic policy is impossible to be derived without knowing the entire bandwidth time series before query execution. Yet, these policies are fast and can keep up to instantaneous bandwidth changes. We show that they can effectively make positive online adjustments to reduce query completion time in real experiments based on real-world data.

4 Building Cost Models

In this section, we describe our machine learning models based on LASSO and Gradient Boosting Regression Tree.

TABLE 1
The Raw Features

Raw Features	Range
total_exec_num	$1-16$
cpu_core_num	$1-8$ per executor
mem_size	$1-4$ GB per executor
avail_bw	$5-1000$ Mbps per link
tbl1_size, tbl2_size	$0.3-12$ GB per table
hdfs_block_num	$1-90$

TABLE 2
The Handcrafted Features

Handcrafted Features
$\texttt{tbl\_size\_sum} := \text{sum}(\texttt{tbl1\_size}, \texttt{tbl2\_size})$
$\texttt{max\_tbl\_size} := \max(\texttt{tbl1\_size}, \texttt{tbl2\_size})$
$\texttt{min\_tbl\_size} := \min(\texttt{tbl1\_size}, \texttt{tbl2\_size})$
1/avail_bw, 1/total_exec_num, 1/cpu_core_num

And we introduce the extensive feature crafting for predicting the time cost and output cardinality of each pairwise join between tables, which will serve as a basis for our dynamic QEP adjustment schemes. We created a dataset of 15K samples by running the realistic TPC-H benchmark queries and collecting the corresponding statistics, which we call features.

Our basic idea is to consider all raw features relevant to the running time and output size as well as all intuitively possible nonlinear interactions across these raw features, and then rely on LASSO, a powerful dimension reduction tool, to pick out only the key (derived) features. These selected features are further input to GBRT to characterize their nonlinear contribution toward the target to be predicted. We show that the proposed models can achieve a prediction accuracy of over 95 percent on this dataset.

4.1 Dataset Creation

We built a new dataset of 15K samples, each recording the time it took to run a (possibly complex) query from TPC-H benchmark [10] and its output size, as well as several features related to the query execution. Each query in the dataset takes two tables generated by TPC-H `dbgen` [10] as the two input tables, each located on a different datacenter. Since the shuffling during reduce stages forms a major bottleneck in geo-distributed analytics, we focus on `JOIN`-like operators between the pair of tables such as Cartesian product, natural join and theta-join, which lead to heavy network shuffle traffic.

We ran different types of pairwise joins under varied combinations of input features. These features are related to the query itself, the input tables involved, and the running environment, the latter including hardware configuration, network bandwidth and parameter settings in the underlying Spark system. These features are summarized in Table 1. The feature, `total_executor_num`, represents the number of executors involved in the execution of the join and dictates the maximum number of tasks executed simultaneously. The features, `cpu_core_num` and `mem_size`, are the upper bounds of computing resources that each worker can utilize. The feature, `avail_bw`, indicates the available bandwidth between the two sites storing the two tables. During dataset creation, the varying bandwidth was obtained via `tc` rule-based bandwidth regulation. `tbl1_size`, `tbl2_size` are the actual sizes of the generated tables, ranging from 300 MB to 12 GB, as we focus on large tables and data-intensive jobs. Finally, `hdfs_block_num` indicates both the input data size and the number of parallel tasks, i.e., the parallelism of data processing.

Once a model is trained offline based on the created dataset, we can estimate the cost of executing any pairwise joins online, based on instantaneous measurements of these features in the runtime system. All the features selected can be easily measured or acquired online during query execution in a non-intrusive manner without interfering with query execution. In particular, in Section 5, we will introduce our lightweight and non-intrusive scheme for online bandwidth probing. Besides, it is also easy to incrementally expand the training dataset by including statistics from recently executed queries. And the models can easily be retrained periodically.

4.2 Crafting the Nonlinear Feature Space

Since the query completion time and output cardinality may depend on input features in a nonlinear way, we further leverage the intuitions about geo-distributed analytics to craft some derived features based on the interaction of raw features. Our additional handcrafted nonlinear features are also shown in Table 2. Furthermore, we apply *feature crossing* to both raw features and handcrafted features to obtain polynomial features, which significantly expand the dimension of the feature space. For example, the degree-2 polynomial features of a 3-dimensional feature space $[a, b, c]$ are $1, a, b, c, a^2, ab, ac, b^2, bc, c^2$. Finally, we rely on machine learning models to sort out the key features for dimensionality reduction and to characterize their nonlinear contribution toward the metrics to be predicted.

The rationale of using handcrafted features and feature crossing is to incorporate important nonlinear terms that may possibly help decide the completion time. For example, in a broadcast join, $\min(\texttt{tbl1\_size}, \texttt{tbl2\_size})/\texttt{avail\_bw}$ may decide the shuffle time, since the smaller table will be sent to the site of the larger table for join execution. Similar ideas of using such intuitive predictors have been adopted in Ernest [26], which performs a linear regression of nonlinear interactions between system parameters to predict the time to execute a data analytics job in a cluster. Similarly, the optimization-based methods in Clarinet [4] and Iridium [6] have also assumed that the data transmission time depends on the table sizes divided by the available bandwidth in a linear way. However, it is worth noting that the available bandwidth is only loosely related to data transmission time. On the one hand, the available bandwidth only defines an upper bound of bandwidth. On the other hand, the distributed computing engine can hardly saturate the bandwidth due to the reasons mentioned in Section 1.

Our statistical feature engineering and selection approach is a generalization of the above ideas—we first expand the feature space to as large as possible to incorporate all intuitively possible nonlinear interactions between relevant parameters, and then rely on the ability of LASSO to select only the

relevant ones statistically. Our machine learning approach abstracts away the excessive details in the underlying Spark system, including task placement and scheduling decisions, which would otherwise have to be considered by an optimization method like Clarinet [4] and Iridium [6].

However, the common weakness of such optimization-based approaches is the limited robustness to bandwidth fluctuation, because optimization algorithms typically depend on the estimation of available bandwidth. However, unlike the available bandwidth, the actual bandwidth consumed by a distributed computing engine such as Spark relies on several underlying factors, such as the number of concurrent connections and the data volume for each connection. Furthermore, the detailed traffic pattern also depends on the specific join algorithms used, e.g., hash join versus broadcast join, which may not be known before the query is executed. Our generalized machine model overcomes these difficulties by first expanding the feature space to as large as possible and rely on the power of models to discover the nonlinear dependencies of the completion time on different features from data, abstracting away the excessive details in the underlying Spark system.

4.3 Machine Learning Models

To keep Turbo lightweight and efficient, our chosen models must be accurate and *fast*.

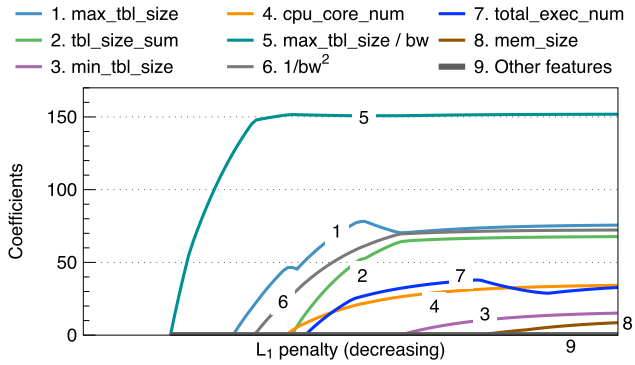
LASSO regression augments linear regression to perform sparse feature selection by introducing an L_1 norm penalty to the least-squares cost function (Eqn. (1)). Compared to linear regression, it can effectively suppress overfitting by picking out only the relevant features from the large polynomial feature space we have created. When minimizing the cost function, the L_1 penalty forces the coefficients of irrelevant features to zero. After the degree-2 polynomial feature crossing, we have obtained more than 200 derived features. We input all these features into LASSO, which automatically selects the relevant key features (usually fewer than 10) and discards the rest [27].

$$\min_{\boldsymbol{w}} \quad \sum_{i=1}^{m}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2 + \lambda\|\boldsymbol{w}\|_1, \tag{1}$$

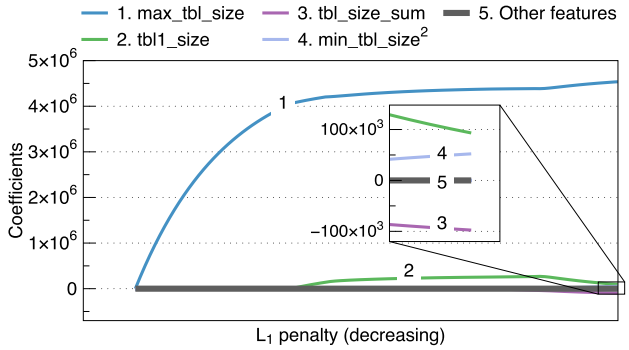
where $\boldsymbol{w}$ is the coefficients assigned to each feature, m is the number of training samples, y_i is the label of the sample $\boldsymbol{x}_i$, λ is regularization parameter and $\|\boldsymbol{w}\|_1$ is the L_1 penalty norm.

We select highly correlated features to reduce the dimensionality of ML training. We apply the LASSO path algorithm [27] to regularize the coefficient estimates. When jointly minimizing the least-squares cost function and the L_1 penalty term, some of the coefficient estimates are forced to be exactly equal to zero with the tuning parameter λ becoming sufficiently large. Hence, LASSO obtains the most relevant features and discard others. Moreover, LASSO is interpretable, stable and visualizable. The LASSO path intuitively reflects the selected features and ignored ones [28].

We plot the LASSO paths as the weight placed on the L_1 penalty decreases (Fig. 6). As more weights are placed onto the L_1 penalty, the coefficients of some features become zero, where only the most important features have non-zero coefficients. For query completion time, the key features



(a) Selected features for durations of pairwise joins.



(b) Selected features for output sizes of pairwise joins.

Fig. 6. Feature selection by LASSO path.

selected by LASSO are `max_tbl_size`, `tbl_size_sum`, `min_tbl_size`, `cpu_core_num`, `max_tbl_size/bw`, `1/bw`^{2}, `total_exec_num`, and `mem_size`. For query output size, the key features selected are `max_tbl_size`, `tbl1_size`, `tbl_size_sum`, and `min_tbl_size`^{2}.

Gradient Boosting Regression Tree (GBRT): we find that for output size prediction, LASSO can already perform well. However, for prediction on query completion time, even if the key features are selected, the completion time still depends on these selected features in a nonlinear way. GBRT is a nonlinear machine learning technique that uses an ensemble of weak regression trees to produce a single strong model [29]. GBRT improves the model generalizability and avoids overfitting by combining a large number of simple binary or ternary regression trees. GBRT algorithm seeks for finding the function $F(\boldsymbol{x})$ that is composed of weighted regression trees $h_m(\boldsymbol{x};\theta_m)$

$$F(\boldsymbol{x}) = \sum_{m=1}^{M}\gamma_m h_m(\boldsymbol{x};\theta_m),$$

where M is the number of regression trees, γ_m is the corresponding weight for each learner, θ_m is the set of parameters of a regression tree.

When building the model, GBRT progressively fits a new regression tree $h_m(\boldsymbol{x};\theta_m)$ to the residual of the previously fitted model $F_{m-1}(\boldsymbol{x})$ in a stage-by-stage fashion, and updates the current model $F_m(\boldsymbol{x})$ by adding in the new tree

$$F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \gamma_m h_m(\boldsymbol{x};\theta_m).$$

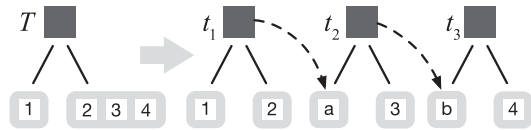
Each tree $h_m(\boldsymbol{x})$ is fitted to the error gradient of $F_{m-1}(\boldsymbol{x})$ on the training samples $\boldsymbol{x}$. The regression tree $h_m(\boldsymbol{x};\theta_m)$ and its associated weight γ_m are trained to minimize the loss function $\mathcal{L}$

$$h_m(\boldsymbol{x};\theta_m),\gamma_m = \arg\min_{h,\gamma}\sum_{i=1}^{N}\mathcal{L}(y_i, F_{m-1}(\boldsymbol{x}_i)+\gamma h(\boldsymbol{x}_i;\theta_m)),$$

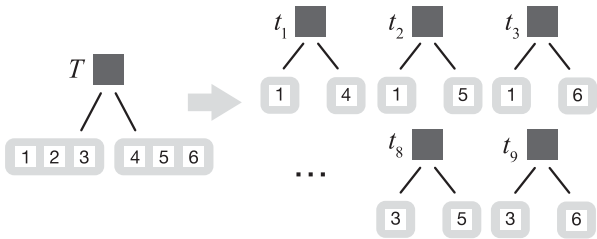
where N is the number of training samples.

In our evaluation in Section 6.1, the GBRT we build contains 500 ternary regression trees. The depth of each tree is three. And the inputs to the GBRT only contain the relevant (derived) features selected by LASSO, which can reduce the prediction variance of GBRT.

LASSO regression and GBRT are statistic models that have much fewer parameters than neural network models. The computation complexity has been further reduced after feature selection by LASSO. Training the two models on the 15K dataset only takes a few minutes on a CPU-based server.



(a) The left table is located on Site-1, while the right table is distributed across Site-2, Site-3 and Site-4. The join duration is estimated by $T = t_1 + t_2 + t_3$.



(b) The left table is distributed across Site-1, Site-2 and Site-3, while the right table is distributed across Site-4, Site-5 and Site-6. The duration is estimated by $T = \max(t_1+t_2+t_3, t_4+t_5+t_6, t_7+t_8+t_9)$.

Fig. 7. The divide-and-conquer heuristic.

5 Dynamic Query Adjustment

In this section, we present the detailed policies used by Turbo to adjust query execution plans dynamically at runtime. During the execution of a QEP, given the parts of the query that have already been executed, adjusting the remaining part of the QEP still involves exponentially many possibilities of join reordering. To avoid a large decision space and to make the system respond fast to resource availability fluctuations, Turbo greedily selects the pairwise join with the lowest estimated cost to be executed next, according to various proposed policies. The cost of each candidate join is predicted by our lookahead cost predictor as described below.

In a real-world deployment of geo-distributed data analytics, the geographical data sources and the data dependencies jointly determine the database design, including logical schemas and storage structures. The logical schema defines the rows and columns of tables, and the storage structure determines the storage partitioning of tables. There are mainly two table partitionings: a table can be partitioned either within a single datacenter as in [4] and [5] or across multiple geo-distributed datacenters as in [30] and [6]. We argue that partitioning a table across multiple geo-distributed datacenters makes it thorny to keep data consistent and up-to-date. Turbo supports pairwise joins of tables partitioned by both partitionings. A divide-and-conquer heuristic is proposed for tables partitioned geo-distributedly.

5.1 Lookahead Cost Prediction

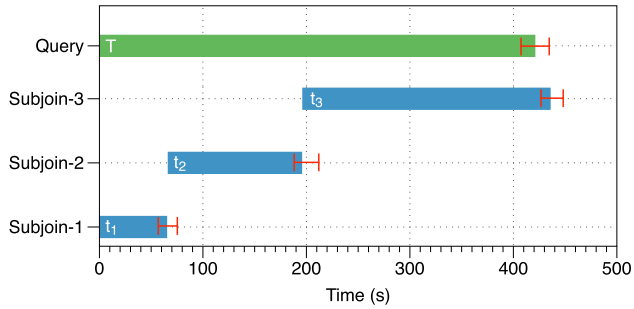
Let us first explain how the cost of each candidate join operation in the remaining QEP can be predicted. Note that when a series of joins are to be executed, the output results from a current join, which serve as the input to the next join operation, may not reside on a single datacenter. An intermediate table is usually spread across multiple sites because the reduce tasks that generated such intermediate results were placed on multiple sites by the system.

If the two input tables of a pairwise join to be evaluated are indeed located on two sites, respectively, we can directly use the trained machine learning models as described in Section 4 to predict the duration of this join. The instantaneously measured features of the trained models include the bandwidth between two sites, the sizes of the two tables, as well as several other parameters pulled from the runtime system into the models as features.

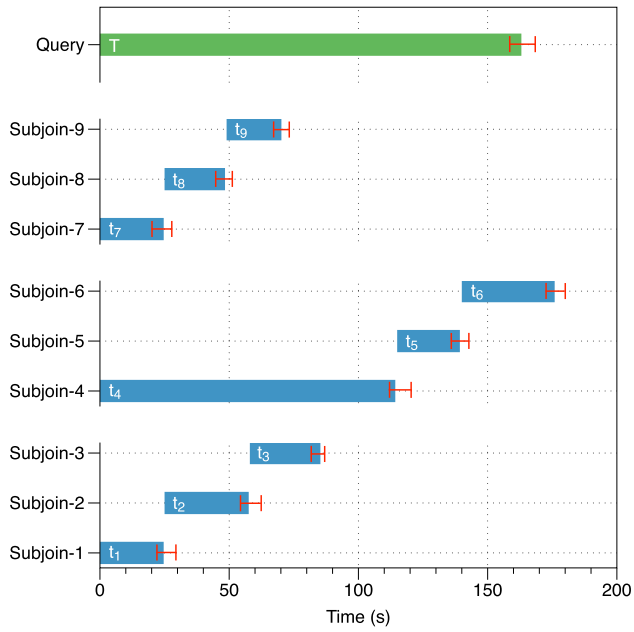
On the other hand, however, if at least one of the two input tables of the pairwise join is spread across multiple sites, as a result of the previous computation, the way that this join is physically executed will not strictly match the joins logged in the training dataset. In this case, we need to introduce additional mechanisms to be able to leverage the trained models in the presence of distributed input data. To address this issue, we use a *divide-and-conquer* approach that splits the pairwise join that involves distributed input data into multiple *sub-joins*, each between a subset of the left table and a subset of the right table, where each subset only contains a part of the left table or the right table stored on a single site.

In particular, our divide-and-conquer approach handles two cases, as shown in Fig. 7. If only one table is distributed across multiple sites, we cascade the sub-joins sequentially to predict the duration of the join, as shown in Fig. 7a. If, however, two input tables are both distributed across multiple sites, e.g., involving $3 \times 3 = 9$ sub-joins as shown in Fig. 7b, the total duration for the join can be predicated as $T = \max(t_1+t_2+t_3, t_4+t_5+t_6, t_7+t_8+t_9)$. Turbo executes the sub-joins involving the same subset of the left table sequentially in a cascaded way while executing sub-joins involving different subsets of the left table in parallel.

We perform a real-world query across multiple nodes to verify the query duration approximated by the divide-and-conquer heuristic. The query joins two tables `lineitem` and `partsupp`, which are generated by TPC-H `dbgen`. For the case that only one table is distributed across multiple sites, we split the `lineitem` data into three partitions and store them separately on different nodes. As Fig. 8a, we



(a) One table is distributed to three sites as Fig. 7(a).



(b) Both two tables are distributed to three sites as Fig. 7(b).

Fig. 8. Justifying the divide-and-conquer heuristic.

obtain the query duration T and the duration t_1, t_2 and t_3 of sub-joins. The query duration T is 421.2 seconds, and the total duration of sub-joins, $t_1 + t_2 + t_3$, is 436 seconds. $t_1 + t_2 + t_3$, the duration approximated by the divide-and-conquer heuristic, is larger than T because we iteratively save the result of a sub-join as a new table and join the new table with the next table partition, which enforces the sequential sub-joins but introduces extra I/O time.

For the case that both two tables are distributed to multiple sites, we split both `partsupp` and `lineitem` into three partitions and store them separately on different nodes, as in Fig. 7b. Similarly, as in Fig. 8b, we obtain the query duration T is 163.1 seconds, while the duration of the sub-joins, $\max(t_1 + t_2 + t_3, t_4 + t_5 + t_6, t_7 + t_8 + t_9)$, is 175.9 seconds, which also involves the extra I/O time. The duration of pairwise joins predicted by our GBRT model does not involve the extra I/O time, with which the divide-and-conquer heuristic estimates closer durations.

5.2 Runtime QEP Adjustment

By focusing on join reordering, Turbo's query optimizer is a shim layer that wraps the default query optimizer (or any advanced query optimizer) of the underlying distributed computing engine such as Hive and Spark SQL, deployed on the master node of the distributed computing engine. During query execution, the cost of each candidate join operation to be executed next can be estimated according to the lookahead cost predictor described above. Such estimates are fed into the query optimizer that chooses the next join in terms of three different policies, to greedily reduce a query's overall completion time. The greedy nature of decision making, together with the ability to predict the costs of pairwise joins online prior to their execution, enables Turbo to make fast and timely decisions regarding QEP adjustments, to respond to runtime dynamics.

Turbo performs the iterative process of cost prediction and QEP adjustment in a stage-by-stage pipelined fashion. During the map stage of an ongoing pairwise join, Turbo probes the available WAN bandwidth in the network to avoid bandwidth contention, since map stages are not bandwidth-intensive and can also finish fast. During the reduce stage of the ongoing pairwise join, Turbo collects the distribution of the reduce tasks. It can be used to estimate the input data distribution for the next join, since the input of a next join consists of the output data from the ongoing join (s) and possibly some new table. The measured available bandwidth information, as well as the estimated input data distribution, are used to estimate the time cost of a next join operation using the method mentioned above.

5.3 Adjustment Policies

When adjusting a QEP, Turbo respects both the semantics of the SQL query and the context of the underlying distributed computing engine. The semantics of the SQL query define the set of all candidate pairwise joins. The execution context limits the choices for the next join: the next join must be chosen to preserve the part of the QEP that has already been executed. After pruning unqualified pairwise joins, Turbo explores three greedy policies to select the next pairwise join, based on the estimated durations and/or output sizes of all candidate joins:

Shortest Completion Time First (SCTF) selects the next pairwise join to be the one that is predicted to have the least completion time. This policy is intuitive because the overall query completion time is the summation of the completion time of each pairwise join.

Maximum Data Reduction First (MDRF) selects the next pairwise join to be the one that is predicted to lead to the maximum difference in volume between the input data and output data. The maximum data reduction implies that less data will be transferred over the network later on, thus saving the query execution time in a geo-distributed setting.

Maximum Data Reduction Rate First (MDRRF) selects the next pairwise join to be the one that is predicted to maximize the data reduction rate, which is defined as the volume of data reduced per unit time for the operation. The rate is calculated by the difference in volume between the input data and output data, then divided by the predicted join completion duration. This policy takes into account both data reduction and the time needed to achieve that amount of data reduction.

Turbo makes extremely fast decisions, in fact within less than one second, for the choice of the next join to be executed. Once the machine learning models are maintained, the predictions are instantaneous, and the number of

TABLE 3
The Absolute Test Errors

Model	RMSE	
	Duration (s)	Output Size (KB)
LASSO	54.14	301.49
GBRT	9.41	282.4

candidate joins to be compared is not large due to the SQL semantic and execution context constraints. In an environment where significant bandwidth fluctuations are only observed over minutes, Turbo is perfectly competent to generate valid QEP adjustments dynamically.

However, severe bandwidth fluctuations might happen between datacenters during one pairwise join operation, and Turbo will not revoke the ongoing pairwise join operation, even though it is not the optimal plan. We argue that revoking the ongoing operation and running a new QEP will lead to a waste of resources, including computation and bandwidth. It is still an open question to deal with highly frequent bandwidth variations during the execution of a geo-distributed data analytic task, such as a pairwise join operation.

6 Implementation and Evaluation

We implemented the prototype of Turbo in Scala 2.11 and Python 2.7. The machine learning module is developed with `scikit-learn` 0.19 and the query optimizer is built on Spark SQL 2.0.2. The interfaces between the machine learning module, the query optimizer and the Spark scheduler are implemented by Apache Thrift (`scrooge` 4.13 for Scala and `thrift` 0.10 for Python). We have developed a toolkit[3] for collecting training data with the RESTful APIs of the Spark's history server. We have also extended the HDFS command `put` to specify data nodes for data partitions of different tables.[4]

We launch a 33-instance cluster across the eight regions of Google Cloud Compute Engine [31]. Each of the instances has four vCPU cores, 15 GB memory and 120 GB SSD and runs Ubuntu-16.04 with Oracle Java 1.8.0. We build the data analytic frameworks with HDFS 2.7.3 as the persistent storage backend, Hive 1.2.1 as the structured data warehouse and Spark 2.0.2 as the data processing engine.

The duration prediction of our tuned GBRT model on the testing dataset is up to 94.5 percent and the output size prediction by LASSO on the testing dataset is up to 95.8 percent. Our experiments with trained GBRT and LASSO models show that Turbo can effectively adjust QEPs in corresponding to fluctuating WAN conditions and reduces the query completion times from 12.6 to 41.4 percent.

6.1 Model Evaluation

We evaluate both the machine learning models on the 15K dataset created and select the most accurate model for predicting durations and output sizes, respectively.

3. https://github.com/chapter09/sparkvent
4. https://github.com/chapter09/hadoop/tree/putx

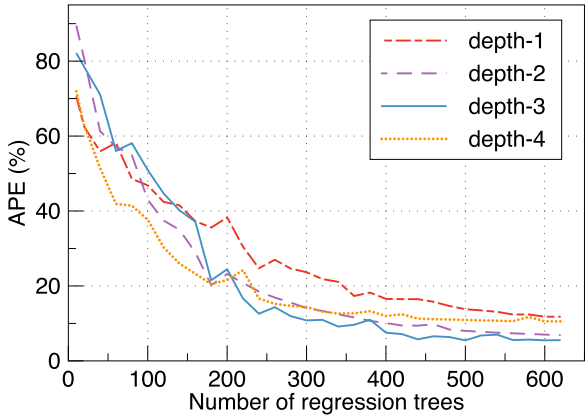


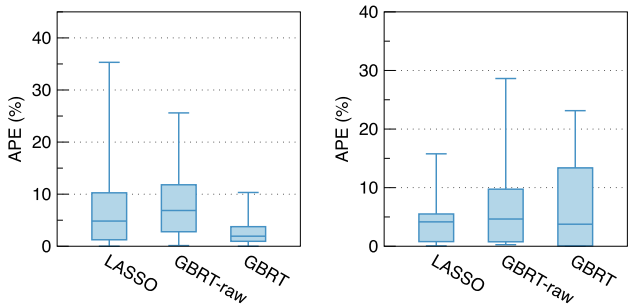
Fig. 9. Tune the number of regression trees and max tree depth in GBRT.

We use randomly selected 10 percent of the 15K samples as the test set and the remaining 90 percent as the training set. Table 3 presents the root-mean-squared error (RMSE) of each model for predicting the durations and output sizes of pairwise joins in the test set. The root-mean-squared errors (RMSEs) of LASSO are 54.14 seconds for predicting the durations and 301.49 KB for predict output sizes, while the RMSEs of GBRT are 9.41 seconds for durations and 282.4 KB for output sizes.

We then analyze the test error distribution of different models in terms of the absolute percentage error (APE), which is calculated as $APE_i = |y_i - h(x_i)|/y_i \times 100\%$.

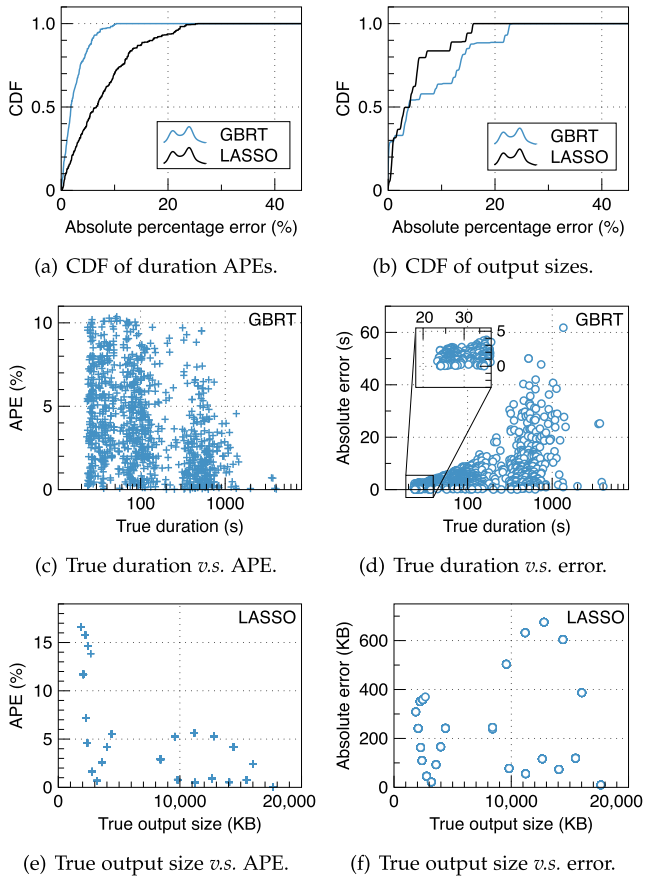
We show the tuning process of GBRT in Fig. 9. The two major hyper-parameters of GBRT are the number of regression trees and the max tree depth. By performing grid search over the hyper-parameter space, we find that the GBRT model with 500 ternary regression trees and max depth as three achieves the lowest APE of 5.5 percent. Predicting duration with an accuracy of 94.5 percent is sufficiently accurate to make online join reordering decisions for query performance benefits.

Fig. 10a and 10b present the box plots of the test APEs for duration predictions and output size predictions. GBRT-raw denotes the GBRT model taking all raw features as input, which have not been selected by LASSO. By comparing the average APEs achieved by GBRT and GBRT-raw, it demonstrates that using features selected by LASSO improves the model accuracy. As we can see, for duration prediction, GBRT achieves much lower errors compared to the other two models. For the output size prediction, LASSO achieves



(a) APEs of duration. (b) APEs of output sizes.

Fig. 10. Model test errors.



(a) CDF of duration APEs. (b) CDF of output sizes.

(c) True duration *v.s.* APE. (d) True duration *v.s.* error.

(e) True output size *v.s.* APE. (f) True output size *v.s.* error.

Fig. 11. Analysis of model test errors.

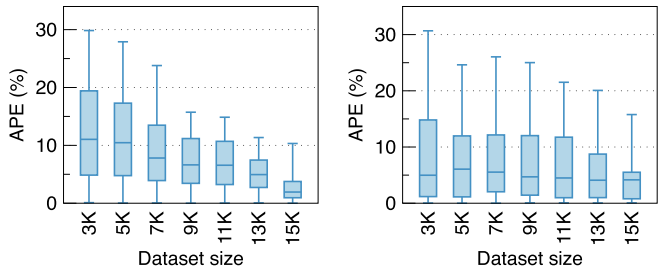
an average APE of 4.2 percent and effectively keeps APEs under 16 percent, which performs better than the other two models, though its RMSE is bit higher than that of GBRT. Fig. 11a and 11b present the CDFs of test APEs for duration predictions and output size predictions respectively. By jointly considering the RMSEs and APEs in Fig. 11, we choose GBRT to predict durations of pairwise joins and LASSO to predict output sizes in Turbo.

We further investigate the correlation between test errors and the predicted targets. Through the scatter plots in Fig. 11c, 11d, 11e, and 11f, we observe a decreasing trend of the absolute percentage error in general as the true duration or the true output size of the pairwise join increases. However, the absolute errors do not increase much as the true durations and output sizes scale up to large values. This fact indicates that our machine-learning-based method has a higher accuracy when handling large tables, which are common in big data analytics.

Finally, we note that the models can achieve higher accuracy as the size of the training set increases. We test the GBRT model and the LASSO model trained on datasets of different sizes, which are subsets of the 15K dataset. As shown in Fig. 12a and 12b, as the training dataset becomes larger, the APEs of both models decrease significantly.

6.2 Turbo Performance

Benchmark and Metrics. We use the dataset from the TPC-H benchmark [10], which contains eight relational tables. The tables are stored as structured data in Hive with the HDFS



(a) The GBRT for durations. (b) The LASSO for output sizes.

Fig. 12. The test errors of models trained on datasets of different sizes.

as the storage backend. The data distribution is maintained as a metadata attribute in Hive. We evaluate Turbo with both table partitionings: tables partitioned within a region and across multiple regions.

TPC-H benchmark [10] contains 22 queries with broad industry-wide relevance. Among the 22 queries, we ignore those queries that process only one or two tables, as there is no alternative joins when performing the QEP adjustment. We run the remaining 10 TPC-H queries (Q2, Q3, Q5, Q7, Q8, Q9, Q10, Q11, Q18 and Q21) under the following five schemes to evaluate Turbo. The first two schemes are used for comparison. The other three schemes are Turbo configured with the three greedy policies, respectively.

- *Baseline:* the default query optimizer [13] of Spark SQL, which is agnostic to the fluctuating bandwidths. It only considers the cardinalities of tables when selecting the join algorithms.
- *Clarinet:* the optimal query plan is determined by the bandwidths and data distribution when the query is submitted. This is an approximation[5] to Clarinet [4].
- *Turbo-(SCTF, MDRF and MDRRF):* With awareness of network bandwidth and data cardinalities, Turbo applies three different greedy policies to choose the next join to run.

Tables Partitioned Within a Datacenter. The data distribution is as Table 4 that each of the eight tables is partitioned within a single region. As in Fig. 14, we run the ten queries on the cluster including 33 instances across the eight regions under the five schemes. Each region contains four instances, and the extra instance is configured as the master node. For each of the five schemes, we run the ten queries for five times and record the query completion times.

Fig. 15a shows baseline and Clarinet both have severe long tail delay on pairwise join completion times. As in Fig. 15b, compared to the baseline, the overall query completion times is reduced by 25.1 to 38.5 percent for Turbo-SCTF (32.6 percent on average), 12.6 to 37.1 percent for Turbo-MDRF (27 percent on average) and 25.2 to 41.4 percent for Turbo-MDRRF (34.9 percent on average).

We plot all the stage completion times in Fig. 13. Compared to the baseline and Clarinet, the three policies of Turbo have reduced the maximum stage completion times

5. It should be noted we do not perform bandwidth reservation and task placement. The bandwidth reservation is performed by Clarinet's WAN manager, a component that is privileged to operate MPLS-based or SDN-based WAN.

TABLE 4
Benchmark Data Locations

Table	Location	Table	Location
lineitem	Taiwan	customer	Frankfurt
region	Singapore	orders	Sao Paulo
supplier	Sydney	nation	Northern Virginia
part	Belgium	partsupp	Oregon

for most stages, which indicates there are less delayed stages. Turbo-MDRF fails to choose the right join when running query Q10.

Then we perform an in-depth case study on query Q21. We run query Q21 from the TPC-H benchmark under the five schemes to show how Turbo adapts a QEP to the fluctuating WAN. The query Q21 processes four tables, lineitem, orders, nation and supplier. We launch six clusters of the same hardware configuration, as mentioned. Each cluster is composed of four instances from four regions, respectively, i.e., Brazil, Taiwan, Sydney and Virginia. Five of the clusters run query Q21 simultaneously in terms of the five schemes. The remaining one cluster runs iperf to periodically measure bandwidths between the four regions, which avoids contending bandwidths with the five clusters running Q21.

As shown in Fig. 16, we plot a Gantt chart to show the progress of the query Q21 running under the five schemes, dealing with WAN fluctuations. Two colours are used to distinguish different stages of the running query. We also plot the bandwidths between the four regions, according to the timeline of the query execution. The fluctuating links are marked in black. As we can see from the Gantt chart, Turbo-SCTF and Turbo-MDRRF adjust the QEP plan to react the bandwidth fluctuation between Taiwan and Sydney around 5:25. Turbo-MDRF does not change the QEP since it only considers the volume of data reduction.

From this case study, we have observed that the bandwidth between datacenters is changing at a frequency of sub-seconds. Turbo reorders pairwise joins based on the relative bandwidths between different datacenters, which change slowly at minutes or hours. If the relative bandwidths remain unchanged within one pairwise join operation as in Fig. 16, the QEP by Turbo will still be beneficial.

Tables Partitioned Geo-Distributedly. We run the query Q21 to evaluate Turbo performance when tables are partitioned geo-distributedly. We evaluate two partitioning scales: each of the four tables is partitioned across two regions without overlap as lineitem (Taiwan and Singapore), orders (Sao Paulo and Belgium), nation (Northern Virginia and Frankfurt) and supplier (Sydney and Oregon), where each region runs four instances, and each table is hosted by

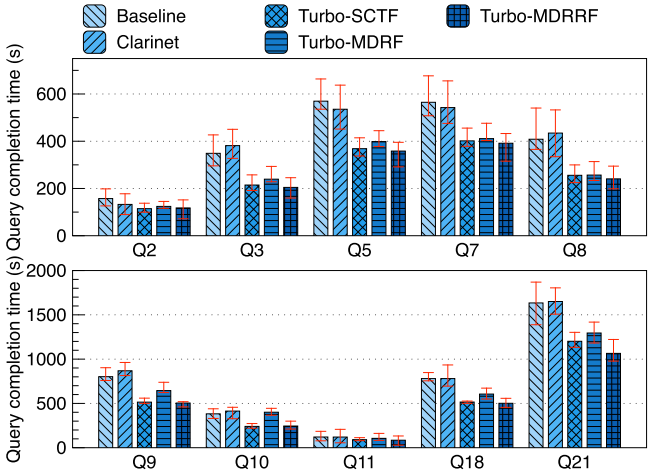


Fig. 14. Query completion times.

the eight instances of the two regions; the other scale is that all tables are partitioned across all regions, and each table is also hosted by eight instances from the eight regions respectively.

We compare the Q21 completion time achieved by different QEP adjustment schemes under different partitioning scales in Fig. 17, where Clarinet is absent, due to its lack of support for tables partitioned across multiple regions. Clarinet assumes a table is hosted within a single region. Compared to the baseline, the Q21 completion time is reduced by 27.1 percent (13.1 percent) for Turbo-SCTF, 14.7 percent (14.6 percent) for Turbo-MDRF and 37 percent (25.9 percent) for Turbo-MDRRF when each table is partitioned to two regions (all eight regions). We have also observed a higher performance variation when tables are partitioned to all eight regions.

7 Related Work

A number of recent studies have attempted to improve the performance of geo-distributed data analytics (GDA). Turbo adds to the rich literature on query optimization in both distributed database systems and big data analytics frameworks. Essentially, Turbo shows how to enable the query optimizer to react to runtime dynamics.

The sub-optimality of static query execution plans has been a thorny problem. For traditional databases, progressive optimization (POP) [16], [17] has been proposed to detect cardinality estimation errors at runtime. For data analytics within a single datacenter, PILR [9] executes part of the query as a "pilot run" for dynamic cost estimation. RoPE [8] enables the re-optimization of query plans by interposing instrumentation code into the job's dataflow.

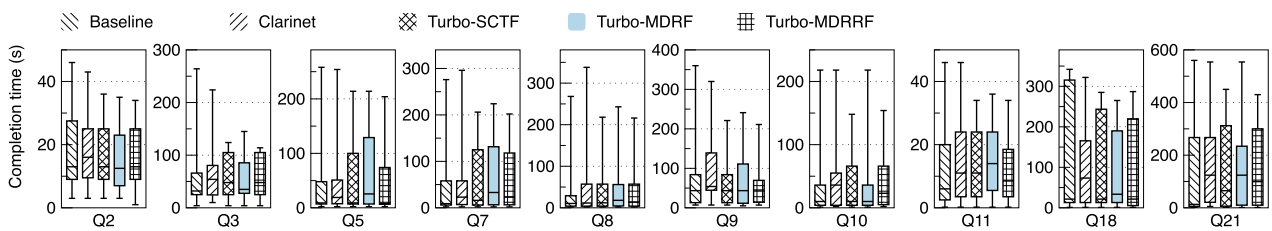
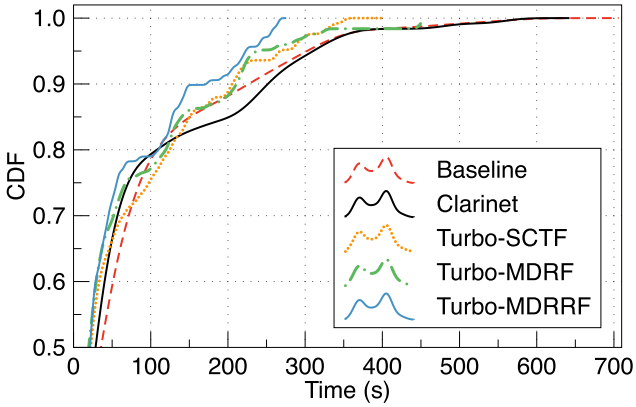
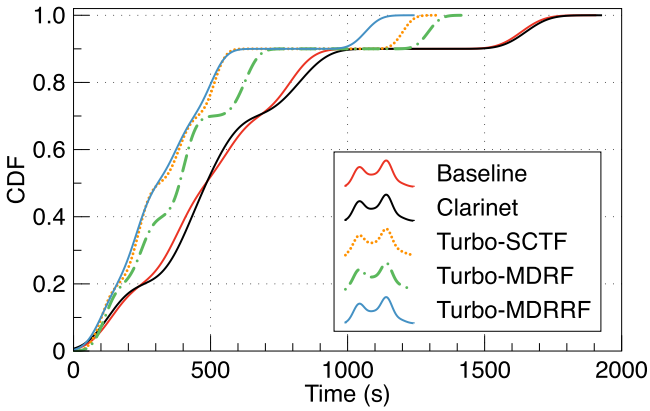


Fig. 13. The completion time distributions of pairwise joins.



(a) CDF of pairwise join completion times.



(b) CDF of query completion times.

Fig. 15. CDF of completion times.

Turbo leverages the interpretation from pairwise joins to map-reduce stages and orchestrates query execution plans across datacenters without refactoring the existing data analytic frameworks.

Most existing work has explored low-layer optimizations to improve GDA query performance, such as data placement and task scheduling. Iridium [6] seeks a tradeoff between data locality and WAN bandwidth usage by data movement and task scheduling. Geode [5] jointly exploits input data movement and join algorithm selection to minimize WAN

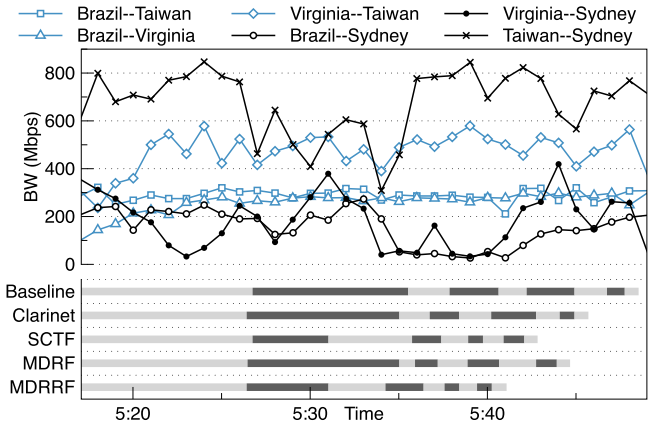


Fig. 16. The Gantt chart of the query Q21.

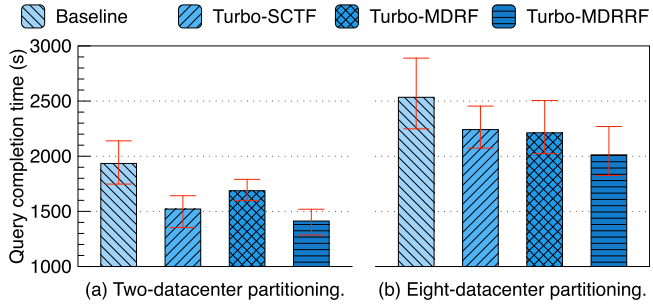


Fig. 17. The Completion time of query Q21 on tables partitioned in two geo-distributed scales.

bandwidth usage. WANalytics [30] optimizes and replicates data to reduce total bandwidth usage. JetStream [7] uses data aggregation and adaptive filtering to support data analytics. SWAG [32] coordinates job scheduling across datacenters to take advantage of data locality and improves GDA performance. Graphene [34] packs and schedules tasks to reduce job completion times and increases cluster throughput.

The solutions based on storage systems improve the performance of GDA in terms of data locality, such as [5], [6], [7], [30]. They seek a tradeoff between data locality and WAN bandwidth usage by making data placement decisions. The solutions based on data processing engines such as [6], [30], [32], [33], [34], [36] try to incorporate the awareness of underlying resources into task placement and job scheduling. The solutions based on query optimizers such as [4], [5], select an optimal QEP from candidate equivalent QEPs that differ in, e.g., their ordering of joins in the query, according to the utilization and availability of key resources such as network bandwidth. We summarize the existing work into the three categories, as in Table 5.

The closest work to us is Clarinet [4] and QOOP [35]. Clarinet selects the optimal query execution plan based on the WAN condition before the query is executed. Once a plan is selected, Clarinet leaves it unchanged even under a varying runtime environment. QOOP performs dynamic query replanning in reaction to resource changes of a single cluster, while Turbo focuses on queries across multiple datacenters. Turbo is a lightweight and non-intrusive system, while QOOP refactors the whole data analytics stacks—cluster scheduler, execution engine and query planner.

However, most of the existing solutions require the full stack of the original data processing frameworks to be re-engineered. Turbo has carefully designed a machine learning

TABLE 5
Related Work on Distributed Wide-Area Data Analytics

Related Work	Data Placement	Task Scheduling	Plan Optimization	Working Mode
Geode [5]		✓	✓	static
WANalytics [30]	✓	✓		static
Iridium [6]	✓	✓		static
SWAG [32]		✓		static
JetStream [7]	✓			static
Clarinet [4]		✓	✓	static
Lube [33]		✓		dynamic
Graphene [34]		✓		static
QOOP [35]		✓	✓	dynamic
Turbo			✓	dynamic

module to enable online query planning non-intrusively. A few efforts have been made to perform resource management with machine learning techniques [36], [37], workload classification [26], cluster configuration [38] and database management system tuning [39].

8 Conclusion

In this paper, we have presented our design and implementation of Turbo, a lightweight and non-intrusive system that orchestrates query planning for geo-distributed analytics. We argue that, in order to optimize query completion times, it is crucial for the query execution plan to be adaptive to runtime dynamics, especially in wide-area networks. We have designed a machine learning module, based on careful choices of models and fine-tuned feature engineering. The model can estimate the time cost as well as the intermediate output size of each reduce and shuffle stage (including joins) during query execution given a number of easily measurable parameters, with an accuracy of over 95 percent. Based on quick cost predictions made online in a pipelined fashion, Turbo dynamically and greedily alters query execution plans on-the-fly in response to bandwidth variations. Experiments performed across geo-distributed Google Cloud regions show that Turbo reduces the query completion times by up to 41 percent based on the TPC-H benchmark, in comparison to default Spark SQL and state-of-the-art optimal static query optimizers for geo-distributed analytics.

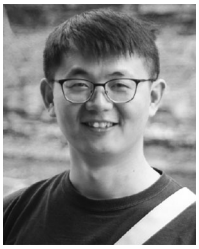
Acknowledgments

This work was supported by a research contract with Huawei Technologies Co. Ltd., and a NSERC Discovery Research Program. Preliminary results have been presented in ACM SoCC'18 [1].

References

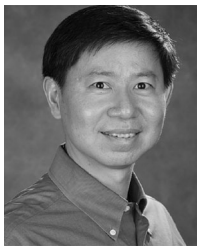
[1] H. Wang, D. Niu, and B. Li, "Dynamic and decentralized global analytics via machine learning," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 14–25.
[2] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, Art. no. 2.
[3] A. Thusoo *et al.*, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endowment*, vol. 2, 2009, pp. 1626–1629.
[4] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-aware optimization for analytics queries," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 435–450.
[5] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 323–336.
[6] Q. Pu *et al.*, "Low latency geo-distributed data analytics," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 421–434.
[7] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman, "Aggregation and degradation in JetStream: Streaming analytics in the wide area," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 275–288.
[8] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, Art. no. 21.
[9] K. Karanasos *et al.*, "Dynamically optimizing queries over large scale data platforms," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 943–954.
[10] TPC-H benchmark specification, 2017, Accessed: Jul. 1, 2017. [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf
[11] Apache calcite: Dynamic data management framework, 2017, Accessed: Sep. 1, 2017. [Online]. Available: https://calcite.apache.org
[12] Apache hadoop official website, 2016, Accessed: May 1, 2016. [Online]. Available: http://hadoop.apache.org/
[13] M. Armbrust *et al.*, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[14] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surveys*, vol. 16, pp. 111–152, 1984.
[15] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst.*, 1998, pp. 34–43.
[16] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 659–670.
[17] W.-S. Han, J. Ng, V. Markl, H. Kache, and M. Kandil, "Progressive optimization in a shared-nothing parallel database," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 809–820.
[18] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endowment*, vol. 9, 2015, pp. 204–215.
[19] Apache phoenix: OLTP and operational analytics for apache hadoop, 2017, Accessed: Sep. 1, 2017. [Online]. Available: https://phoenix.apache.org
[20] AWS RedShift official website, 2016, Accessed: May 1, 2016. [Online]. Available: https://aws.amazon.com/redshift/
[21] Apache hive official website, 2016, Accessed: May 1, 2016. [Online]. Available: https://hive.apache.org
[22] Apache HBase official website, 2016, Accessed May 1, 2016. [Online]. Available: https://hbase.apache.org
[23] Google cloud platform for data center professionals: Networking, 2018, Accessed: Jan. 30, 2018. [Online]. Available: https://cloud.google.com/docs/compare/data-centers/networking
[24] Amazon EC2 and Amazon virtual private cloud, 2018, Accessed: Jan. 30, 2018. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-vpc.html
[25] The SAVI testbed, 2018, Accessed: Jan. 1, 2018. [Online]. Available: http://www.savinetwork.ca/about-savi/
[26] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 363–378.
[27] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*. New York, NY, USA: Springer, 2001.
[28] R. Tibshirani, "Regression shrinkage and selection via the LASSO," *J. Roy. Statist. Soc.. Series B (Methodological)*, vol. 58, pp. 267–288, 1996.
[29] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, pp. 1189–1232, 2001.
[30] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a geo-distributed data-intensive world," in *Proc. 7th Biennial Conf. Innovative Data Syst. Res.*, 2015, pp 1–15.
[31] Google cloud compute engine regions and zones, 2017, Accessed: Sep. 1, 2017. [Online]. Available: https://cloud.google.com/compute/docs/regions-zones/regions-zones
[32] C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 111–124.
[33] H. Wang and B. Li, "Lube: Mitigating bottlenecks in wide area data analytics," in *Proc. 9th USENIX Workshop Hot Top. Cloud Comput.*, 2017, Art. no. 1.
[34] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 81–97.
[35] M. Kshiteej, C. Mosharaf, A. Aditya, and C. Shuchi, "Dynamic query re-planning using QOOP," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 253–267.
[36] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Top. Netw.*, 2016, pp. 50–56.
[37] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 127–144.

[38] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 469–482.
[39] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1009–1024.



Hao Wang received both of the BE degree in information security and the ME degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2012 and 2015, respectively. His research interests include large-scale data analytics, distributed computing, machine learning and datacenter networking.

Di Niu received the BEng degree from Sun Yat-sen University, Guangzhou, China, in 2005 and the MSc and PhD degrees from the University of Toronto, Toronto, Canada, in 2009 and 2013, respectively. He is currently an associate professor with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada, specialized in the interdisciplinary areas of distributed systems, data mining, machine learning, text modeling, and optimization algorithms. He was the recipient of the Extraordinary Award of the CCF-Tencent Rhino Bird Open Grant 2016 for his invention of the story forest system for news document understanding at scale. He is a member of the IEEE.



Baochun Li (F) received the BE degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and the MS and PhD degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000, respectively. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, where he is currently a professor. He holds the Nortel Networks junior chair in Network Architecture and Services from October 2003 to June 2005, and the Bell Canada Endowed chair in computer engineering since August 2005. His research interests include cloud computing, large-scale data processing, computer networking, and distributed systems. In 2000, he was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the field of communications systems. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He is a fellow of IEEE and a member of ACM.