Dynamic and Decentralized Global Analytics via Machine Learning

Hao Wang
University of Toronto
haowang@ece.toronto.edu

Di Niu
University of Alberta
dniu@ualberta.ca

Baochun Li
University of Toronto
bli@ece.toronto.edu

ABSTRACT

Operating at a large scale, data analytics has become an essential tool for gaining insights from operational data, such as user online activities. With the volume of data growing exponentially, data analytic jobs have expanded from a single datacenter to multiple *geographically distributed* datacenters. Unfortunately, designed originally for a single datacenter, the software stack that supports data analytics is oblivious to on-the-fly resource variations on inter-datacenter networks, which negatively affects the performance of analytic queries. Existing solutions that optimize query execution plans before their execution are not able to quickly adapt to resource variations at query runtime.

In this paper, we present *Turbo*, a lightweight and non-intrusive data-driven system that dynamically adjusts query execution plans for geo-distributed analytics in response to runtime resource variations across datacenters. A highlight of Turbo is its ability to use machine learning at runtime to accurately estimate the time cost of query execution plans, so that adjustments can be made 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 8 regions in the Google Cloud platform. Our experimental results have shown that Turbo can achieve a cost estimation accuracy of over 95% and reduce query completion times by 41%.

CCS CONCEPTS

• **Information systems** → **Database query processing**; *MapReduce-based systems*; • **Computing methodologies** → *Machine learning approaches*;

KEYWORDS

Data Analytics, Distributed Systems, Machine Learning

ACM Reference Format:
Hao Wang, Di Niu, and Baochun Li. 2018. Dynamic and Decentralized Global Analytics via Machine Learning. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 12 pages.
https://doi.org/10.1145/3267809.3267812

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267812

1 INTRODUCTION

All major Internet organizations and companies, such as Google, Facebook and Amazon, host their services across multiple geographically distributed datacenters to push services closer to end users. Hundreds of terabytes (TBs) or even petabytes (PBs) of data, such as user activity history and system logs, are gathered at these geo-distributed datacenters [9, 15, 17] on a regular basis. How quickly one can perform data analytics over such vast volumes of geo-distributed data is important to certain mission-critical tasks, such as activity statistics, content recommendation, and online advertising.

While data-parallel processing frameworks — such as Spark [38] and Hive [31] — have widely been adopted to handle data analytics in a single datacenter, recent studies, *e.g.*, Clarinet [34], Geode [35], Iridium [29] and JetStream [30], have pointed out that it is inefficient to first gather all the data to a single site for centralized processing, due to the excessive bandwidth cost across datacenters. Therefore, a number of decentralized solutions have been proposed to speed up data analytics in a geo-distributed setting. For example, Clarinet [34] and Geode [35] have advocated that computation (such as filter and scan operations) should be pushed closer to data in individual 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 [1, 24, 29, 34] require that the full stack of the underlying data processing system be overhauled, and possibly require re-engineering multiple mechanisms in these systems, such as data replication, reducer placement, task scheduling, and query execution strategy selection. Yet, solving joint optimization problems spanning multiple layers in the design space may lead to non-trivial operational overhead and delayed reactions to runtime dynamics, such as fluctuations in resource availability. There are alternative solutions, *e.g.*, Geode [35], that estimate the volume of the shuffling traffic by simulating the query execution in a single datacenter. However, such a sandbox-like approach incurs quite a large overhead, and may not be accurate if the same query is not recurring or if the link bandwidth is time-varying. Therefore, a large gap still exists between the existing decentralized solutions to geo-distributed analytics and their deployment in production environments with real-world data.

We argue that in global analytics, decentralized query execution must be orchestrated dynamically at runtime in order to realize its promised benefits and full potential. An "optimal" static query execution plan (QEP) predetermined through cross-layer optimization or simulation is unlikely to remain optimal during query execution over large tables, since resources in the geo-distributed system, especially the inter-datacenter bandwidth, naturally vary over time during query execution.

In this paper, we propose Turbo, a lightweight and non-intrusive system that orchestrates query planning 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 query execution. Such estimates are performed on the fly, given the size of 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 is 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. It is designed to work effectively on top of any existing underlying systems, such as Spark, Hive and Pig, through machine-learning-based runtime cost prediction and query orchestration.

Towards a real-world design and implementation of Turbo, we have made the following original contributions.

Case studies of realistic queries in a geo-distributed cloud. We measured the Google Cloud platform, and our results suggest that inter-datacenter bandwidth could fluctuate drastically within several minutes, which 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 [13] dataset, which contains realistic queries and data with broad industrial relevance. We find that, for a typical query in TPC-H executed in geo-distributed datacenters in a decentralized fashion, dynamically reordering the joins during query execution can reduce query completion times by as much as 40%.

Accurate cost prediction based on machine learning. A common existing technique for estimating the time cost of data shuffling in geo-distributed analytics is to solve a reducer placement problem in order to minimize the longest link finish time [29] or the sum of finish times [34], assuming some empirical models for network transfer times. In reality, it is cumbersome to solve these optimization problems whenever available bandwidth varies. In addition, the actual bandwidth used by system frameworks, such as Spark, is not the same as the bandwidth available on the link in question. The shuffle time also critically depends on the execution algorithms used, such as broadcast joins *v.s.* hash joins, and the sorting algorithms. Rather than using empirical models and joint optimization, we are the first to adopt a machine learning approach to predict the cost of any decentralized table joins based on the characteristics of the input tables, underlying system and the network. In particular, we hand-crafted a large nonlinear feature space regarding data and the system, and relied on the ability of least absolute shrinkage and selection operator (LASSO) to select the most relevant features. Our machine-learning-based cost prediction is fast, overcomes the limitation of any empirically assumed models, and avoids the complications of joint optimization regarding task placement, scheduling, and query execution strategies.

Non-intrusive query plan orchestration. We propose and implement Turbo, a logic-layer query orchestration to adjust the join orders in a QEP on-the-fly in response to changes in runtime dynamics, based on the lightweight lookahead cost prediction. Turbo greedily chooses the next join it will execute to be the one with the least lookahead 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 using machine-learning-based cost prediction and focusing on forward-looking join reordering, Turbo is orthogonal to any mechanisms in the lower-level distributed computing engine, and can be used on top of any off-the-shelf query optimizers, such as Calcite [11], taking advantage of the existing expertise in query optimization developed over decades of research. We have implemented a prototype of Turbo and deployed it on a fairly large Google Cloud cluster with 33 instances spanning 8 geographical regions. Our experimental results have shown clear evidence that Turbo can achieve a cost estimation accuracy of over 95% and reduce the query completion times by up to 40%.

2 BACKGROUND AND MOTIVATION

We first review how modern data analytics frameworks, *e.g.*, Hadoop [3] and Spark [38], execute SQL queries in a geo-distributed setting, and then use measurements and case studies based on the TPC-H benchmark [13] dataset including 15K records to show the inefficiency of all existing *static* solutions to geo-distributed analytics.

Processing and optimizing SQL queries. In Hive [31] and Spark SQL [7], a SQL query is first parsed into a tree called a *query execution plan* (QEP), 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.

Specifically, any SQL query involving multiple tables can be parsed into multiple feasible QEPs. Each QEP differs from other QEPs mainly by a different ordering of the joins of the tables and also by the particular strategies and algorithms to execute each join. Then, the query optimizer selects an optimal QEP either based on rules (*i.e.*, rule-based optimization) or based on cost models (*i.e.*, cost-based optimization).

Query optimizers play a critical role in database technologies and have been extensively studied for decades [12, 21, 23, 25, 28]. Modern query optimizers in state-of-the-art products, *e.g.*, Apache Calcite [11] and Apache Phoenix [6], 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 [10], Apache Hive [5] and Apache HBase [4] — 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 *v.s.* stream aggregate *v.s.* partial stream aggregate), as well as join strategies (hash join *v.s.* merge join).

Optimizing geo-distributed analytics. Yet, existing query optimization technologies designed for a single datacenter, such as Calcite [11] and Phoenix [6], may not perform well in *geo-distributed* analytics, mainly due to the heterogeneous network capacity in wide-area networks.

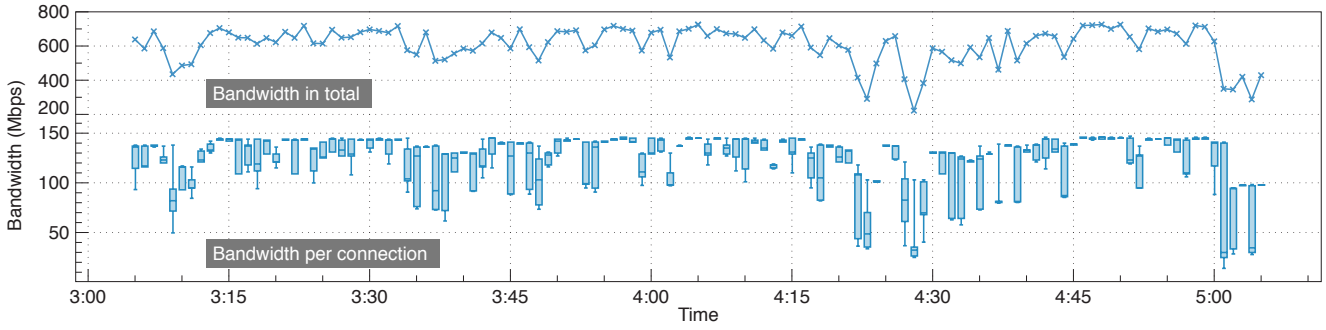


Figure 1: Wide-area bandwidth fluctuations between two geographic regions on the Google Cloud platform.

To address such a design deficiency, Geode [35] extends Apache Calcite [11] to *Calcite++*, which adopts the query execution plan (including the order of joins) generated by Calcite, and incorporates additional statistics to choose the optimal join strategies (*e.g.*, hash joins and broadcast joins) to lower shuffle cost across wide-area networks. In order to estimate the size of network transfers in any QEP, it uses pseudo-distributed execution to simulate running the QEP in a centralized datacenter. Such measurements from the simulation are then used 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 [34] 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 sites depends on the placement of reduce tasks in each stage of a QEP, Clarinet proposes to solve a joint problem of reducer placement and task scheduling, decoupling the complex optimization problem in each stage using various heuristics, with the objective of minimizing the QEP's execution time. The QEP with the shortest execution time will be selected. As a "clean slate" design, Clarinet requires that the full spectrum of the design space be re-engineered, and may not be amenable to evolutionary deployment in practice, since existing state-of-the-art query optimizers and parallel databases can not be reused.

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 [8, 19], the available bandwidth on these links is fluctuating in nature [30], especially when flows of different applications share the links. To demonstrate

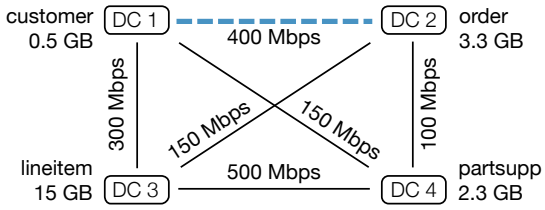


Figure 2: The cluster setup in our example showing the benefits of runtime QEP orchestration.

such variations in inter-datacenter bandwidth availability, we measured inter-region bandwidth for a period of 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 2Gbps.

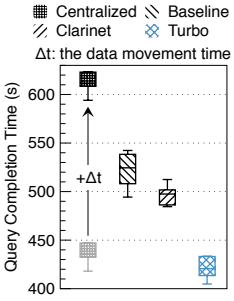
To saturate the bandwidth between the two instances and measure the amplitude and frequency of bandwidth variations, we executed `iperf -t10 -P5` for ten seconds involving five parallel connections, and repeated the command once every 50 seconds. Fig. 1 shows that both the total available bandwidth and 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. 3(a), which joins four tables of data realistic data generated by TPC-H benchmark [13]. 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 site is 12 Gbps, which is much larger than inter-site bandwidth.

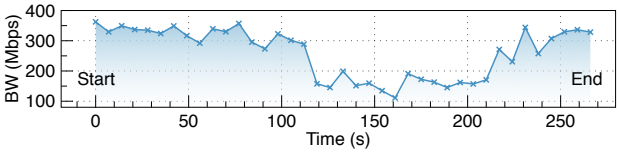
We executed this query with four different strategies for QEP selection: 1) the centralized mode, in which all the tables will be

```
SELECT
  C.name, O.orderstatus,
  L.discount, PS.availqty
FROM
  customer as C,
  order as O,
  lineitem as L,
  partsupp as PS
WHERE O.orderkey == L.orderkey,
  AND PS.partkey == L.partkey,
  AND PS.suppkey == L.suppkey,
  AND C.custkey == O.custkey
```

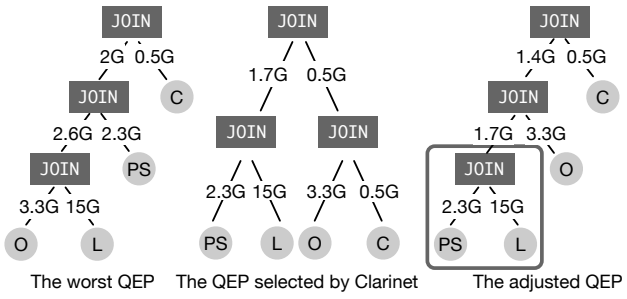
(a) A SQL query example.



(b) Completion times.



(c) The bandwidth fluctuation between DC_1 and DC_2.



(d) QEPs of the three cases.

Figure 3: An example SQL query and its different QEP choices.

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] [34], 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.

For the purpose of illustration, we conducted this experiment in a controlled geo-distributed cluster of four sites [26] 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. 3(c). 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.

[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.

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. 3(b) 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. 3(d). 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 a significant challenge to dynamically adjust a distributed query execution plan at runtime. First, recomputing the optimal query execution plan using complex optimization methods, such as Clarinet [34] or Iridium [29], 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. 4 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 time by switching the order at which different distributed tables should be joined, in response to network bandwidth changes. The lookahead online join reordering is enabled by a judiciously designed machine learning engine which can predict the time cost as well as the output data size (cardinality) of joining a pair of tables distributed on two different datacenters, based on the table and network statistics. We also introduce a lightweight bandwidth measurement scheme which

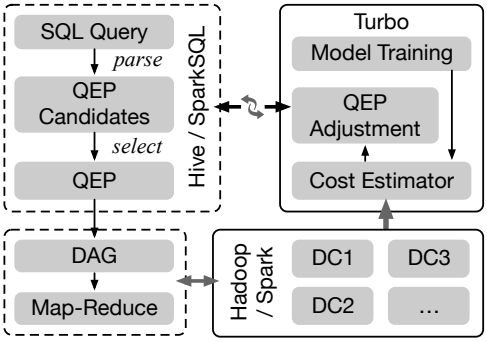


Figure 4: The overview of Turbo.

Figure 5: Interpreting a pairwise join with map-reduce stages.

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 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 executed by a single map stage, whereas an operator involving two tables is executed by 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 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 each 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. 5, 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. The rationale is that to speed 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. 4, 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, including LASSO and Gradient Boosting Regression Tree (GBRT), to predict the time cost and output cardinality of a pairwise join involving two tables given input features derived 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 labeled 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 [13] under a variety of hardware and software settings. The models can be trained offline or updated incrementally, as new samples are added to the training dataset. The basic philosophy of our model training engine is to handcraft a range of relevant features, using feature crossing to introduce non-linearity and enlarging the feature space (Sec. 4.2), and finally relying on the model selection capability of LASSO and Gradient Boosting Regression Tree to filter out the irrelevant features. Our machine learning model yields a prediction accuracy of 95% on queries of TPC-H benchmark [13], which is sufficiently accurate for online join reordering to exhibit its benefits.

Cost Estimator. During query execution, the cost of a pairwise join will be predicted directly by the trained machine learning models, if the two tables involved in this join each reside on a different datacenter, 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 might actually 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 Sec. 5.1 to estimate the cost of this pairwise join. Our cost estimator uses the trained machine learning model as a basic building block, and generalizes it to the case of distributed input data based on an abstract model of parallel execution.

Runtime QEP Adjustment. 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 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 Sec. 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.

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
tbl_size_sum = sum(tbl1_size, tbl2_size)
max_tbl_size = max(tbl1_size, tbl2_size)
min_tbl_size = min(tbl1_size, tbl2_size)
1/avail_bw, 1/total_exec_num, 1/cpu_core_num

4 BUILDING COST MODELS

In this section, we describe our machine learning models based on LASSO, Gradient Boosting Regression Tree (GBRT) and extensive feature crafting for predicting the time cost and output cardinality of each pairwise join between a pair of 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% 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 [13] and its output size, as well as a number of features related to the query execution. Each query in the dataset takes two tables generated by TPC-H dbgen [13] 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.

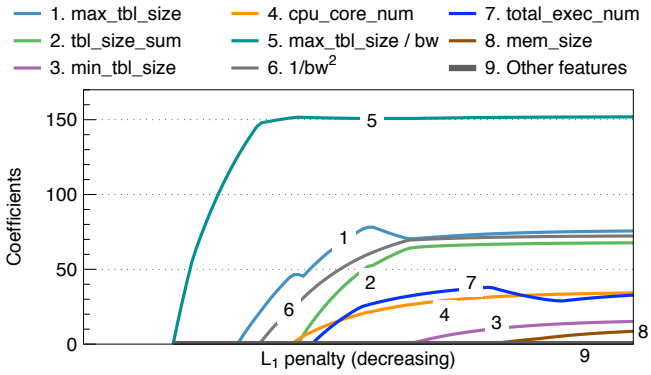
Once a model is trained offline based on the created dataset, we can estimate the cost of executing any pairwise joins online, based on instantaneous measurements of these features in the runtime system. All the features selected can be easily measured or acquired online during query execution in a non-intrusive manner without interfering with query execution. In particular, in Sec. 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

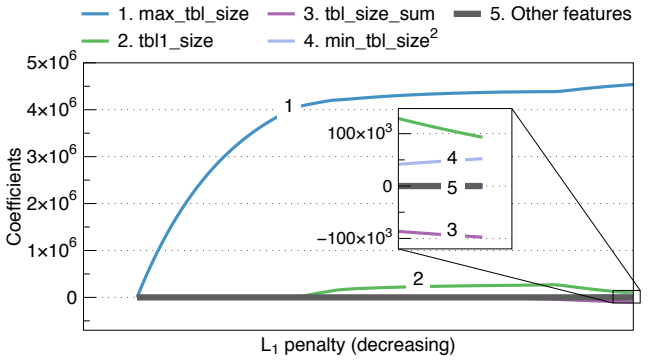
Since the query completion time and output cardinality may depend on input features in a nonlinear way, we further leverage the intuitions about geo-distributed analytics to craft some derived features based on the interaction of raw features. Our additional handcrafted nonlinear features are also shown 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 $1, a, b, c, a^2, ab, ac, b^2, bc, c^2$.

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(tbl1_size, tbl2_size)/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 [33], which performs a linear regression of non-linear interactions between system parameters to predict the time to execute a data analytics job in a cluster. Similarly, the optimization-based methods in Clarinet [34] and Iridium [29] 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, since it only defines an upper bound of available bandwidth, which the distributed computing engine can hardly fully saturate due to a number of reasons mentioned in Sec. 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 in a statistical way.



(a) Selected features for durations of pairwise joins.



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

Figure 6: Feature selection by LASSO path.

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 [34] and Iridium [29].

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. 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 [16]. We plot the LASSO paths in Fig. 6 as the weight placed on the L_1 penalty decreases. 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 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`, `mem_size`. For query output size, the key features selected are `max_tbl_size`, `tbl1_size`, `tbl_size_sum`, `min_tbl_size`^{2}.

Gradient boosting regression tree (GBRT): we find that for output size prediction, LASSO can already perform well. However, for query completion time prediction, 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 weaker regression trees to produce a single strong model. When building the model, GBRT progressively fits a new regression tree $h_m(x)$ to the residual of the previously fitted model $F_{m-1}(x)$ in a stage-by-stage fashion, and updates the current model $F_m(x)$ by adding in the new tree, *i.e.*, $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$. Each tree $h_m(x)$ is fitted to the error gradient of $F_{m-1}(x)$ on the training samples x. GBRT improves the model generalizability and avoids overfitting by combining a large number of simple binary or ternary regression trees.

In our evaluation in Sec. 6.1, the GBRT we build contains 500 ternary regression trees. 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 less 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.

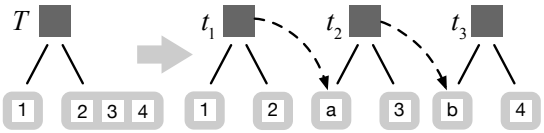
5 DYNAMIC QUERY ADJUSTMENT

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 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 polices, while the cost of each candidate join is predicted by our lookahead cost predictor to be described below.

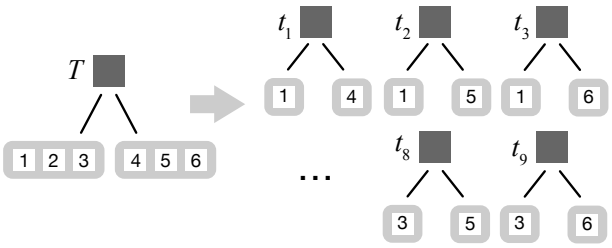
5.1 Lookahead Cost Prediction

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 Sec. 4 to predict the duration of this join, by inputting instantaneously measured features such as the bandwidth between two sites, the sizes of the two tables, as well as a number of other parameters pulled from the runtime system into the models as features.



(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 $T = \max(t_1 + t_2 + t_3,\ t_4 + t_5 + t_6,\ t_7 + t_8 + t_9)$.

Figure 7: The divide-and-conquer heuristic.

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 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(b). 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. 7(a). If, however, two input tables are both distributed across multiple sites, *e.g.*, involving $3 \times 3 = 9$ sub-joins as shown in Fig. 7(b), the total duration for the join can be predicated as $T = \max(t_1 + t_2 + t_3, t_4 + t_5 + t_6, t_7 + t_8 + t_9)$, by executing 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.

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 of predicting 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, which 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

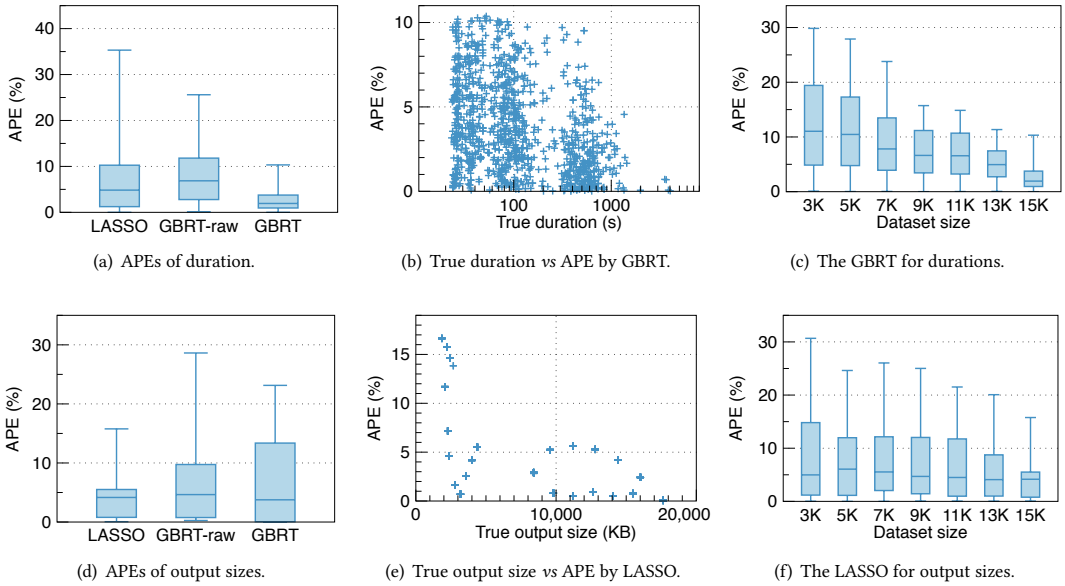
When adjusting a QEP, Turbo respects both the semantics of the SQL query and the context of the underlying distributed computing engine. The semantics of the SQL query define the set of all candidate pairwise joins. The execution context limits the choices for the next join: the next join must be chosen to preserve the part of the QEP that has already been executed. After pruning unqualified pairwise joins, Turbo explores three greedy policies to choose 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, that is its total input size less output size 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 be executed, since once the machine learning models are maintained, the predictions are instantaneous and the number of 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.



(a) APEs of duration. (b) True duration *vs* APE by GBRT. (c) The GBRT for durations.

(d) APEs of output sizes. (e) True output size *vs* APE by LASSO. (f) The LASSO for output sizes.

Figure 8: Analysis of model test errors.

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 [18]. Each of the instances has 4 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.

Our experiments show Turbo can effectively adjust QEPs in corresponding to fluctuating WAN conditions and reduces the query completion times from 12.6% to 41.4%.

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.

We use randomly selected 10% of the 15K samples as the test set and the remaining 90% as the training 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 $APE_i = |y_i - h(x_i)|/y_i \times 100\%$. Fig. 8(a) and Fig. 8(d) 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; and for the output size prediction, LASSO effectively keeps APEs under 16% and performs better than the other two models, though its RMSE is bit higher than that of GBRT. By jointly considering the RMSEs and APEs in Fig. 8, 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. 8(b), and Fig. 8(e), we observe a decreasing trend of the absolute percentage error (APE) in general as the true duration or the true output size of the pairwise join increases, while 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. 8(c) and Fig. 8(f), as the training dataset becomes larger, the APEs of both models decrease significantly.

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

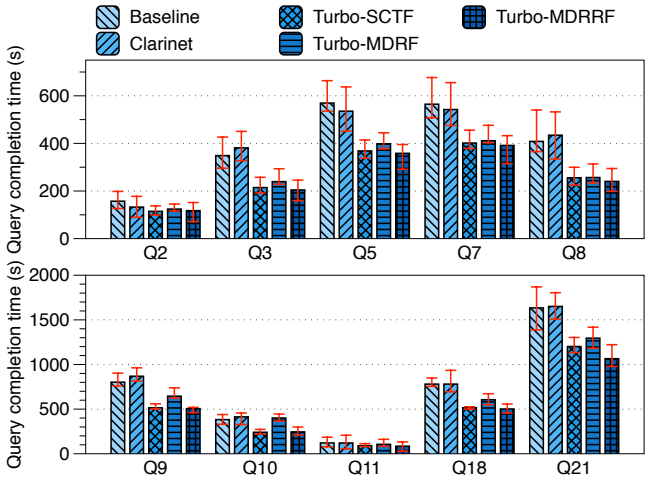


Figure 9: Query completion times.

6.2 Turbo Performance

We use the dataset from the TPC-H benchmark [13], which contains eight relational tables. The tables are stored as structured data in Hive with the HDFS as the storage backend. The tables are distributed as `lineitem` (Taiwan), `customer` (Frankfurt), `region` (Singapore), `orders` (Sao Paulo), `supplier` (Sydney), `nation` (Northern Virginia), `part` (Belgium), and `partsupp` (Oregon). The data distribution is maintained as a metadata attribute in Hive.

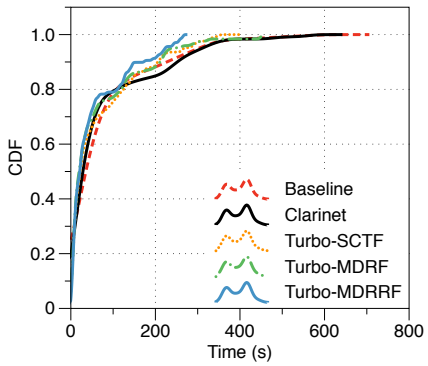
TPC-H benchmark [13] contains 22 queries with broad industry-wide relevance. Among the 22 queries, we ignore those queries that process only one or two tables, as there is no alternative joins when performing the QEP adjustment. We run the remaining 10 TPC-H queries (Q2, Q3, Q5, Q7, Q8, Q9, Q10, Q11, 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 [7] 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 [34].
- **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.

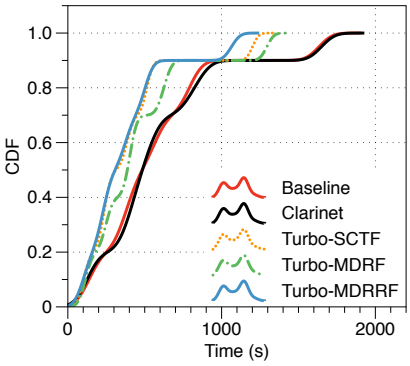
As in Fig. 9, we run the ten queries on the cluster including 33 instances across the eight regions under the five schemes. Each region contains four instances, and the extra instance is configured as the master node. For each of the five schemes, we run the ten queries for five times and record the query completion times.

Fig. 10(a) shows baseline and Clarinet both have severe long tail delay on pairwise join completion times. As in Fig. 10(b), compared

[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. The original Clarinet should have better performance with such capabilities, though which are closed source.



(a) CDF of pairwise join completion times.



(b) CDF of query completion times.

Figure 10: CDF of completion times.

to the baseline, the overall query completion times is reduced by 25.1% to 38.5% for Turbo-SCTF (32.6% on average), 12.6% to 37.1% for Turbo-MDRF (27% on average) and 25.2% to 41.4% for Turbo-MDRRF (34.9% on average).

We plot all the stage completion times in Fig. 11. Compared to the baseline and Clarinet, the three policies of Turbo have reduced the maximum stage completion times 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 a in-depth case study on query Q21. We run the 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.

In Fig. 12, we plot a Gantt chart to show the progress of the query Q21 running under the five schemes, dealing with WAN fluctuations. Two colors are used to distinguish different stages of the running query. We also plot the bandwidths between each two

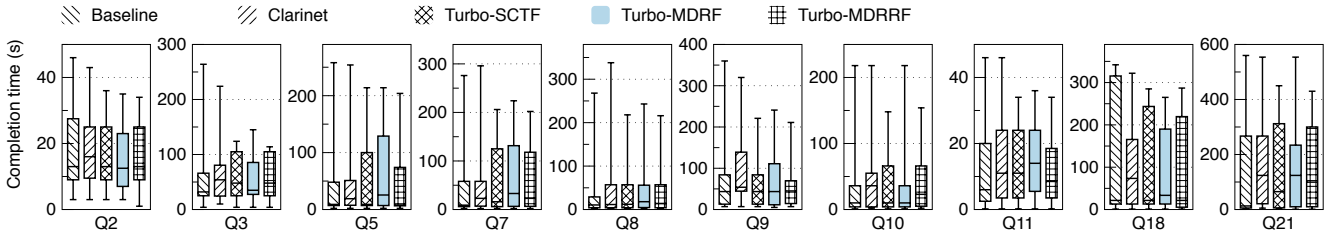


Figure 11: The completion time distributions of pairwise joins.

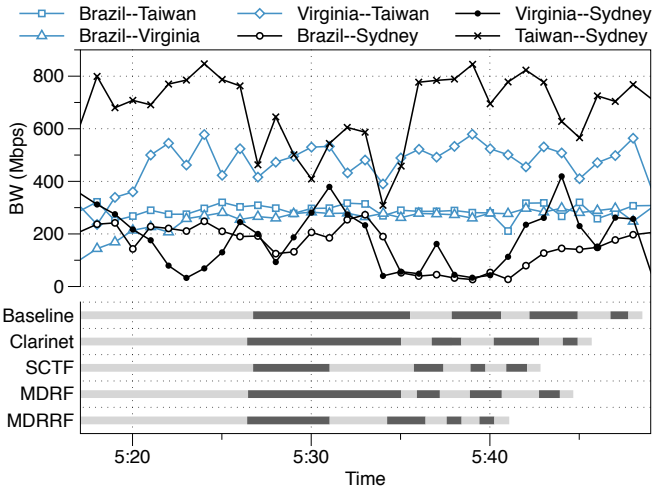


Figure 12: The Gantt chart of the query Q21.

of 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 both 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.

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) [21, 28] has been proposed to detect cardinality estimation errors at runtime. For data analytics within a single datacenter, PILR [24] executes part of the query as a "pilot run" for dynamic cost estimation. RoPE [1] enables re-optimization of query plans by interposing instrumentation code into the job's dataflow. 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 [29] seeks a tradeoff between data locality and WAN bandwidth usage by data movement and task scheduling. Geode [35] jointly exploits input data movement and join algorithm selection to minimize WAN bandwidth usage. WANalytics [36] optimizes and replicates data to minimize total bandwidth usage. JetStream[30] uses data aggregation and adaptive filtering to support data analytics. SWAG [22] coordinates job scheduling across datacenters to take advantage of data locality and improves GDA performance. Graphene [20] packs and schedules tasks to reduce job completion times and increases cluster throughput. Lube [37] incorporates the awareness of underlying resource bottlenecks into task placement and job scheduling.

The closest work to us is Clarinet [34], which selects the optimal query execution plan based on the WAN condition before the query is executed. Once a plan is selected, Clarinet leaves it oblivious to the varying runtime environment.

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 module to enable online query planning non-intrusively. A few efforts have been made to perform resource management with machine learning techniques [14, 27], workload classification [33], cluster configuration [2] and database management system tuning [32].

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, that 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%. 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% 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.

9 ACKNOWLEDGMENTS

This work is supported by a research contract with Huawei Technologies Co. Ltd. and an NSERC Collaborative Research and Development (CRD) grant.

REFERENCES

[1] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-optimizing data-parallel computing. In *Proc. the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[3] Apache. 2016. Apache Hadoop Official Website. http://hadoop.apache.org/. (2016). [Online; accessed 1-May-2016].

[4] Apache. 2016. Apache HBase Official Website. https://hbase.apache.org. (2016). [Online; accessed 1-May-2016].

[5] Apache. 2016. Apache Hive Official Website. https://hive.apache.org. (2016). [Online; accessed 1-May-2016].

[6] Apache. 2017. Apache Phoenix: OLTP and Operational Analytics for Apache Hadoop. https://phoenix.apache.org. (2017). [Online; accessed 1-September-2017].

[7] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proc. ACM International Conference on Management of Data (SIGMOD)*.

[8] AWS. 2016. Amazon EC2 and Amazon Virtual Private Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-vpc.html. (2016). [Online; accessed 30-January-2018].

[9] AWS. 2016. AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/. (2016). [Online; accessed 1-July-2016].

[10] AWS. 2016. AWS RedShift Official Website. https://aws.amazon.com/redshift/. (2016). [Online; accessed 1-May-2016].

[11] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proc. the ACM International Conference on Management of Data (SIGMOD)*.

[12] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proc. the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

[13] Transaction Processing Performance Council. 2017. TPC-H Benchmark Specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf. (2017). [Online; accessed 1-July-2017].

[14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proc. the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[15] Facebook. 2016. Facebook Latest Datacenter Construction. https://www.facebook.com/notes/clonee-data-center/breaking-ground-at-clonee/675009992639648. (2016). [Online; accessed 1-July-2016].

[16] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Springer series in statistics New York.

[17] Google. 2016. Google Datacenter Locations. http://www.google.com/about/datacenters/inside/locations/. (2016). [Online; accessed 1-July-2016].

[18] Google. 2017. Google Cloud Compute Engine Regions and Zones. https://cloud.google.com/compute/docs/regions-zones/regions-zones. (2017). [Online; accessed 1-September-2017].

[19] Google. 2018. Google Cloud Platform for Data Center Professionals: Networking. https://cloud.google.com/docs/compare/data-centers/networking. (2018). [Online; accessed 30-January-2018].

[20] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[21] Wook-Shin Han, Jack Ng, Volker Markl, Holger Kache, and Mokhtar Kandil. 2007. Progressive Optimization in A Shared-Nothing Parallel Database. In *Proc. the 2007 ACM International Conference on Management of Data (SIGMOD)*. ACM.

[22] C. Hung, L. Golubchik, and M. Yu. 2015. Scheduling Jobs Across Geo-Distributed Datacenters. In *Proc. ACM Symposium on Cloud Computing (SoCC)*.

[23] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *ACM Computing surveys (CSUR)* (1984).

[24] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically optimizing queries over large scale data platforms. In *Proc. the 2014 ACM International Conference on Management of Data (SIGMOD)*.

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. the VLDB Endowment* (2015).

[26] Alberto Leon-Garcia and Vladimiro Cirillo. 2018. The SAVI Testbed. http://www.savinetwork.ca/about-savi/. (2018). [Online; accessed 1-Jan-2018].

[27] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proc. 15th ACM Workshop on Hot Topics in Networks (HotNet)*.

[28] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *Proc. the ACM International Conference on Management of Data (SIGMOD)*.

[29] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency Geo-Distributed Data Analytics. In *Proc. the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[30] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman. 2014. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[31] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. 2009. Hive: A Warehousing Solution Over a Map-Reduce Framework. *VLDB Endowment* (2009).

[32] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proc. the ACM International Conference on Management of Data (SIGMOD)*.

[33] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proc. 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[34] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. Clarinet: WAN-Aware Optimization for Analytics Queries. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[35] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[36] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese. 2015. WANalytics: Analytics for A Geo-Distributed Data-Intensive World. In *Proc. Conference on Innovative Data Systems Research (CIDR)*.

[37] Hao Wang and Baochun Li. 2017. Lube: Mitigating Bottlenecks in Wide Area Data Analytics. In *Proc. the 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.