

Bohr: Similarity Aware Geo-Distributed Data Analytics

Hangyu Li
City University of Hong Kong
Hong Kong, China
hangyuli3@my.cityu.edu.hk

Hong Xu
City University of Hong Kong
Hong Kong, China
henry.xu@cityu.edu.hk

Sarana Nutanong
Vidyasirimedhi Institute of Science
and Technology
Wangchan, Thailand
snutanon@vistec.ac.th

ABSTRACT

We propose Bohr, a similarity aware geo-distributed data analytics system that minimizes query completion time. The key idea is to exploit similarity between data in different data centers (DCs), and transfer *similar* data from the bottleneck DC to other sites with more WAN bandwidth. Though these sites have more input data to process, these data are more similar and can be more efficiently aggregated by the combiner to reduce the intermediate data that needs to be shuffled across the WAN. Thus our similarity aware approach reduces the shuffle time and in turn the query completion time (QCT).

We design Bohr based on OLAP data cubes to perform efficient similarity checking among datasets in different sites. We implement Bohr on Spark and deploy it across ten sites of AWS EC2. Our extensive evaluation using realistic query workloads shows that Bohr improves the QCT by up to 50% and reduces the intermediate data by up to 6x compared to state-of-the-art solutions that also use OLAP cubes.

CCS CONCEPTS

• **Information systems** → **Data analytics**; • **Computer systems organization** → *Cloud computing*; • **Networks** → Wide area networks;

KEYWORDS

Data Analytics, Cloud Computing, WAN

ACM Reference Format:

Hangyu Li, Hong Xu, and Sarana Nutanong. 2018. Bohr: Similarity Aware Geo-Distributed Data Analytics. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18)*, December 4–7, 2018, Heraklion, Greece. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3281411.3281418>

1 INTRODUCTION

Cloud service providers like Google and Microsoft deploy geo-distributed data centers (DCs) to serve their users across the world. Services running on these geo-distributed DCs constantly generate

and procure large volumes of data about users, their activities, the infrastructure, etc. [13] As a result, it is increasingly common to perform analytics over data dispersed across geo-distributed sites [27, 32, 33].

A simple solution for geo-distributed data analytics is to aggregate data to a central site and perform analytics there. This is soon deemed undesirable due to the massive bandwidth resources it requires and the excessive delay it incurs [27, 33]. A better solution is to rely on distributed data processing frameworks such as Spark to perform geo-distributed data analytics. Data are processed in-place. Yet, since WAN bandwidth is scarce and highly variable across sites, frameworks designed for homogeneous clusters do not work well out-of-the-box.

Past work has then proposed to optimize data and task placement in such a setting [27, 32, 33]. The idea is that we strategically move data out of the bottleneck sites (with low uplink bandwidth and large datasets) and assign more reduce tasks to these sites. This approach exploits the fact that many queries are *recurring*, so it is possible to know which data the query needs, and execute the data and task placement before it arrives next time. It balances the transfer times among the WAN links, and has been shown to speed up queries significantly [27].

In this work, we argue that one should carefully optimize *which* data to be moved out of the bottleneck, in addition to how much as studied before. In all previous work [27, 32, 33], it is assumed that all data are the same and they are chosen randomly for data placement. This tends to be an oversimplification. Given the high dimensionality of data [15], if we move data that are “similar” to those in the destination DC, the amount of intermediate data during the shuffle stage can be reduced even further due to the common use of combiners [17], and latency of processing queries can be further improved.

Figure 1 shows a toy example to motivate our idea. Suppose we want to execute a page rank query on our DCs in Oregon and Tokyo, and Tokyo is the bottleneck DC. The logs, which simply record the score of a website using its URL as the key, are generated and stored in each DC, or site.¹ If we process these logs in-place, the intermediate data contains four records in total. Now consider the case when we move one record from Tokyo to the other site. If we do not consider data similarity, as shown in Figure 1b we may transfer Url-B to Oregon, and end up with five records of intermediate data which is even worse than leaving data in-place. A similarity aware approach, on the other hand, moves similar data across the sites. This means that Url-A is moved to Oregon as shown in Figure 1c, resulting in just three records of intermediate data.

¹We use “sites” and “DCs” interchangeably throughout the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT '18, December 4–7, 2018, Heraklion, Greece

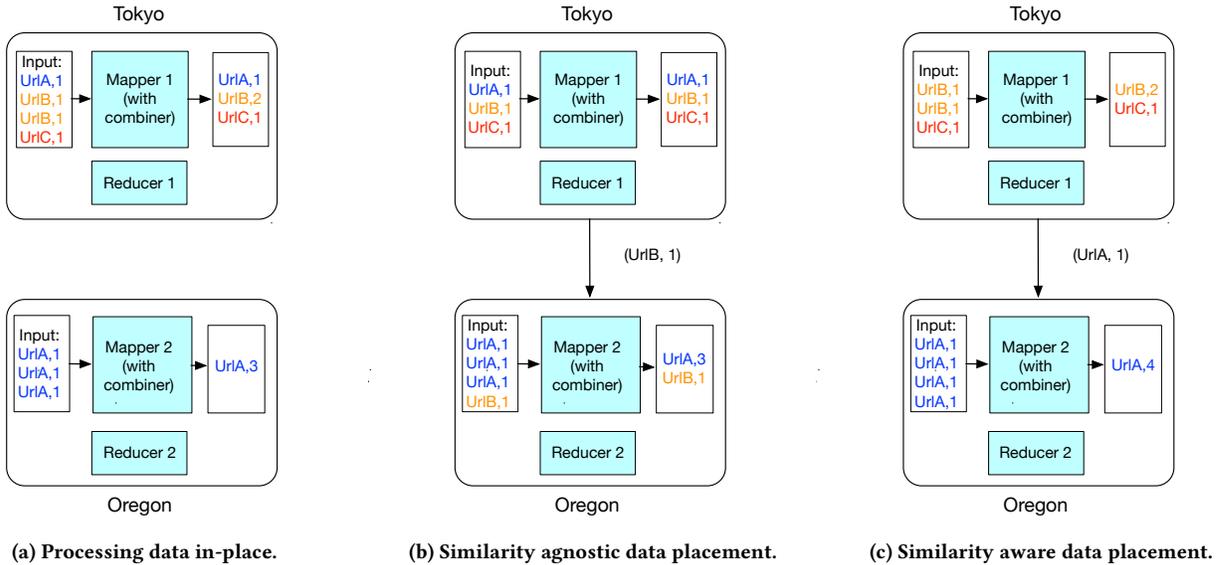


Figure 1: A motivating example. Each record here uses the website URL as the key and its score as the value.

Motivated by this intuition, we present Bohr,² a novel similarity aware geo-distributed analytics system based on Spark. In designing and implementing Bohr, we make three important contributions.

Our first contribution is an efficient similarity checking mechanism that can obtain accurate similarity information between datasets in different sites. Similarity checking should be done without having to exchange bulks of data across sites given the expensive WAN links. Meanwhile it also needs to be accurate not just for different datasets, but also for different queries accessing the same dataset, in the sense that it should reflect the very data attributes that pertain to the query of interest. We rely on OLAP (online analytical processing) cubes [14], a multi-dimensional array data structure, to perform efficient and accurate similarity checking. Bohr stores data with many attributes as OLAP cubes with many dimensions, and performs similarity search [12] using various OLAP operations based on the attributes/dimensions needed by the query to sort the data. Similarity checking can then be done using simple probes, which contain a few representative records of the dataset in question.

Our second contribution is to consider similarity aware data and task placement jointly. Similarity is used mainly for data placement, which determines how to re-balance the input data across the sites so similar data can be combined and the amount of intermediate data can be reduced. Data placement directly impacts task placement, since moving data to a site entails that we assign more reduce tasks there, so that processing and sending the shuffle data does not become the bottleneck. Prior work typically tackles the two problems separately using heuristics [27, 32, 33]. In Bohr, we consider them jointly to maximize the performance improvement. We

take a principled approach and formulate the joint optimization as a linear program, which can be solved efficiently offline during the gap of recurrent query arrivals.

Our third contribution is that we introduce, as an enhancement of Spark-based Bohr, a runtime RDD similarity checking mechanism that further exploits similarity between RDD partitions on the same machine. Conventionally RDD partitions are randomly assigned to executors for processing. Instead, we can assign similar RDD partitions that contain identical records to the same executor, so that shuffle communication across executors can be minimized, and query execution can be further accelerated. To our knowledge, RDD similarity has not been well explored. To avoid the high combinatorial complexity of comparing all possible RDD pairs, Bohr trades off accuracy for efficiency and probabilistically skips RDD pairs that are highly likely to be dissimilar based on algorithms in [34]. Similar RDDs are then clustered using for example K-means and assigned to the same executor.

We implement Bohr based on Spark to demonstrate the feasibility of our idea. We deploy the prototype across ten regions of AWS EC2 to emulate a geo-distributed setting, and evaluate its performance and overhead using empirical workloads. Our results show that Bohr improves QCT of various queries by up to 50% and reduces the intermediate data by up to 8x compared to state-of-the-art Iridium [27] enhanced with OLAP cubes. We also provide microbenchmarks for each design component, and show that the overhead of Bohr is small.

2 BACKGROUND

We start by presenting the background of geo-distributed analytics and OLAP data cubes.

²Niels Bohr is the famous Danish physicist who received the Nobel Prize in 1922. His son, Aage Bohr, also received the Nobel Prize in Physics in 1975. How similar that is!

2.1 Geo-Distributed Analytics

A geo-distributed analytics system works atop a number of WAN-connected DCs, or sites. The WAN links have heterogeneous uplink and downlink bandwidths due to different network infrastructures and applications sharing the links. Data are generated at each site and originally stored where it was generated. The entire dataset is thus geo-distributed. When a query arrives, a logically centralized controller compiles the query into a directed acyclic graph (DAG) of processing stages, each of which comprises of parallel map-reduce tasks that run on different sites. The controller coordinates all aspects of query execution across sites, including data placement, data movement, as well as task assignment [27, 32, 33].

In practice queries are usually recurring, as they need to be periodically executed to analyze new data generated during the previous period [27, 33]. For example, a search company may execute a query to identify the popular keywords every hour or even every minute to better understand the user behavior trend. It is therefore feasible to know a priori the queries that will run on a particular dataset before they arrive, and optimize the system for them.

Our goal is to minimize the average query completion times (QCT). In a geo-distributed setup, QCT is dominated by the transfer times of intermediate data during the shuffle stage, due to their sheer amount and the relatively limited WAN bandwidth [27].

2.2 OLAP Cubes

Bohr utilizes OLAP cubes to store raw data in order to facilitate similarity checking. Here we present the basics of OLAP cubes.

An OLAP (on-line analytical processing) cube is a multi-dimensional array data structure [14]. The term “cube” here refers to a multi-dimensional dataset. For the same dataset, different dimensional cubes can be generated according to the attributes in need. Figure 2 illustrates an OLAP cube which stores the sales information of all products of a company in all regions it operates in between years 2012 to 2014. There are 3 dimensions: time, region, and product.

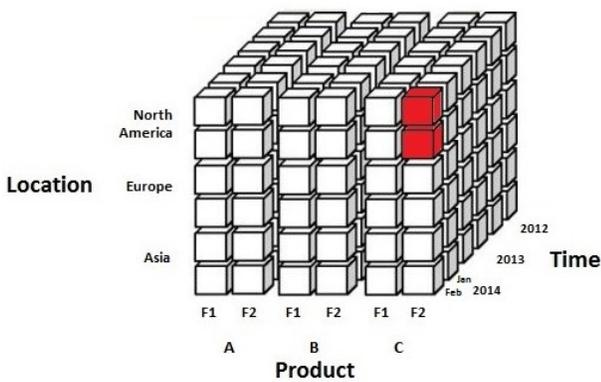


Figure 2: An example of an OLAP cube with three dimensions.

Conceiving data as a cube with hierarchical dimensions leads to conceptually straightforward OLAP operations to facilitate analysis. Common OLAP operations include slice, dice, drill down, roll up,

and pivot [14]. These operations allow us to prepare data according to the queries and facilitate similarity checking, which will be introduced in more detail in §4. For instance, the slice operation picks a rectangular subset of a cube by choosing a single value for one of its dimensions, creating a new cube with one fewer dimension. For example, we can perform a slice operation to the time dimension in Figure 2 to get the sales information of all products in all regions in 2014. The dice operation produces a subcube by allowing the user to choose specific values of multiple dimensions. For example, we can perform a dice operation to get the sales of product A in all regions in 2014.

Further, one can extract subcubes that contain only aggregated data for a subset of the dimensions of the OLAP cube. These subcubes are referred to as dimension cubes. For example, a dimension cube derived from the OLAP cube in Figure 2 may only contain information about product and time (with data aggregated along the region dimension); another dimension cube may contain only region and time information. When different queries come, we can feed the most suitable dimensional cube to the query to maximize efficiency.

3 OVERVIEW

Bohr is designed to reduce the QCT of geo-distributed analytics by: (1) identifying similarity among datasets in different sites; (2) moving similar data from bottleneck sites to those with more bandwidth to minimize the amount of intermediate data with combiners; and (3) optimizing runtime task execution according to data similarity between RDDs on the same machine. The data movement is executed in the lag between data generation and query arrival.

Figure 3 depicts the system overview of Bohr with two sites for simplicity. In each site, Bohr formats raw data that are continuously generated and stores them in OLAP cubes. It also performs pre-processing to prepare the data for similarity-aware placement when the recurring query arrives again (§4.1). Bohr then runs data similarity checking by using simple probes (§4.2). A probe contains a small number of samples for a dataset. Since queries access different datasets, multiple probes are used to detect similarity. Upon receiving the probes sent from site A, site B can quickly identify similar data in its corresponding cubes. The similarity information is aggregated to the Bohr controller.

The controller then solves a linear program that uses similarity information to calculate the optimal data and task placement, taking into account all datasets (§5). The intuition is to balance the number of tasks and amount of data according to data similarity, so each site finishes at the same time. The data placement result is dispatched to each site, which then carries out data transfer accordingly.

When the query arrives, each site executes the map and reduce tasks according to the controller’s decision. During the map stage, Bohr also runs RDD similarity checking to group similar RDD partitions on a machine to the same executor in order to further speed up the execution (§6). Lastly, the intermediate results are shuffled across the sites whenever necessary, and the final result is assembled by collecting each site’s output. This is omitted in the figure for brevity.

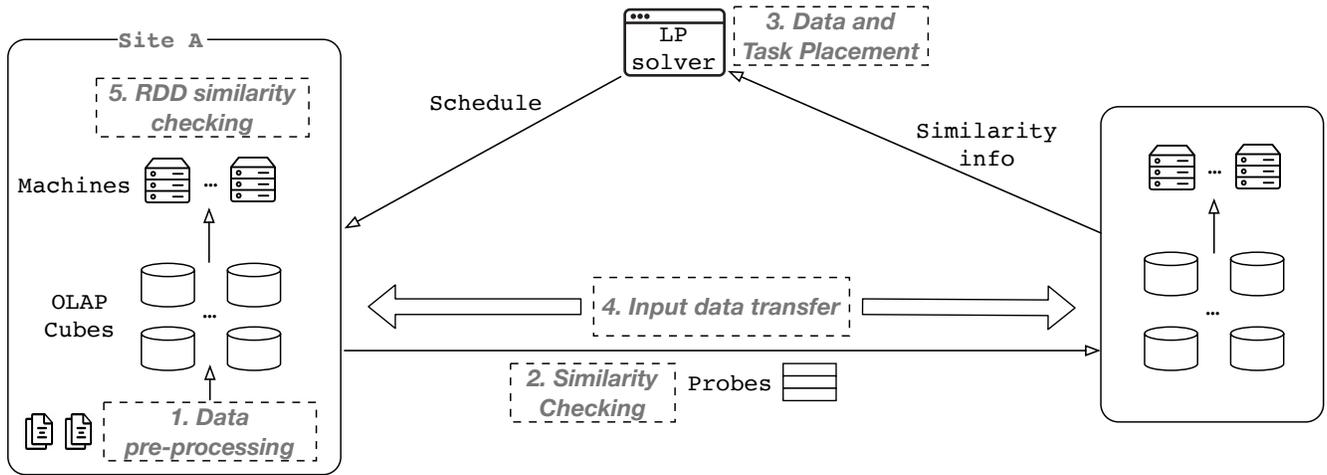


Figure 3: System overview of Bohr. We use two sites as an example.

4 DATA SIMILARITY CHECKING

Organizations perform geo-distributed analysis for many purposes. As a result, queries may access different attributes of a record. In this section, we introduce our similarity checking mechanism that can efficiently gauge similarity of two datasets along different attributes.

4.1 Data Pre-Processing

Data are pre-processed in Bohr to efficiently conduct similarity checking among datasets. One naive solution to pre-processing is to sort the dataset according to the attributes the recurring queries need. This does not work well in practice as there are different queries accessing different attributes even for the same dataset. Another method is to use record-level similarity scores. There are two shortcomings with this approach. First, usually the similarity between two records is not high especially given the high dimensionality of data. Second, even when two records are similar overall, it does not necessarily imply that they are indeed similar on the attributes that the query needs, which makes the accuracy of this approach questionable.

Therefore we propose to use OLAP cubes [12] introduced in §2.2 for pre-processing which we detail now. Figure 4 illustrates the pre-processing workflow.

Data Formatting. Bohr stores the datasets as OLAP cubes. We embed OLAP cube generation into the original data generation procedure in order to reduce cost. Inserting a new record into an OLAP cube only incurs little overhead to the data generation process. Note that OLAP cube generation is done offline; as queries are recurring, OLAP cubes can be generated in the interval between two consecutive queries. Thus it does not add additional processing delay. Formatting the raw data into OLAP cubes, on the other hand, does incur extra storage overhead. We show that the storage overhead is mild in §8.5.

We consider two types of data: (1) text based logs on which similarity checking can directly be performed; and (2) images which

are difficult to be aggregated and need processing such as feature extraction to perform similarity checking. We create two types of cubes correspondingly in Bohr. For logs we construct the OLAP cube according to the schema of the logs in the dataset. For images, we construct the cube according to the feature vectors of each image. In practice, vector space model (VSM) [29] is usually used to transform data into feature vectors. Then similarity between two records can be calculated by applying some distance function to their feature vectors [16].

If new data are generated during query execution, they are buffered until the query finishes. The new data may affect many dimensions and updating all of the related dimension cubes imposes too much overhead. Thus we can just update the dimension cube used by the coming query with new data first, and update the other dimension cubes in the background.

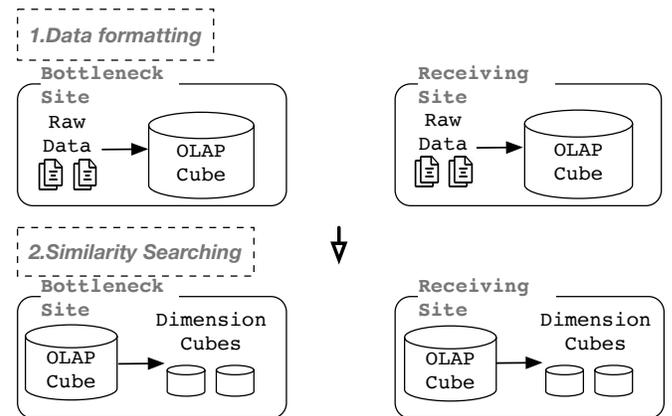


Figure 4: Pre-processing of Bohr.

Similarity Search. When a query arrives for the first time, Bohr uses OLAP operations discussed in §2.2 to retrieve the attributes

needed by the query from the corresponding OLAP cube. Bohr then performs similarity search [18] to sort the data according to their similarity on these attributes. This effectively prepares the dataset for similarity checking across different sites. It also facilitates similarity-aware data movement as similar local records have already been clustered in the cube.

When multiple queries access the same dataset, their similarity metrics can be different. This is handled by using dimension cubes within an OLAP cube [23] as explained in §2.2. As shown in Figure 4, a specific query only accesses a specific dimension cube that contains the dimensions pertinent to the query. Upon query arrivals, we analyze them and identify the attributes needed. For queries accessing the same attributes we classify them as the same type. Then we provide the specific dimension cube to each *query type*.

4.2 Similarity Checking using Probes

As shown in Figure 3, we utilize simple probes to check data similarity between two sites. The probes are sent from the bottleneck site. As mentioned in §4.1, for each query type, the queries only access the corresponding dimension cube. In the similarity search phase, all records have already been sorted and clustered in that dimension cube according to their similarity on these attributes. We then retrieve the top- k records according to the record cluster size in the bottleneck site for a query type to compose a probe.

To handle different query types for the same dataset, we choose k records in total for all query types, by considering the relative weight of each query type. The weight is defined simply as the fraction of the query type among all query types of the dataset. If there are in total 500 queries accessing the same dataset, and one query type consists of 100 queries, the weight for this query type is 0.2. For $k = 30$, this query type has 6 records in the probe.

For logs, since each attribute is low dimensional data such as characters and numbers, we can compare them directly. For images, the dimensionality of the feature vectors is high. We use locality sensitive hashing [18] to reduce the dimensionality in order to process them efficiently.

4.3 Multiple Datasets

We have discussed similarity checking for one dataset. When multiple datasets are queried, it is clear that the benefit from similarity varies depending on how we transfer different datasets out of the bottleneck site. Prior work [27] uses heuristics to identify and move the high-value datasets sequentially. High-value datasets are those that are accessed by more queries, and those whose movement results in large improvements in the intermediate data transfer of their queries.

There are two drawbacks with such an approach. First, sending datasets one after another according to the scores computed by heuristics are clearly sub-optimal. Second, the heuristics do not take data similarity into account. This implies the “best” dataset to be moved out may contain many data that differ from those in the receiving sites, which may actually result in more intermediate data in the shuffle stage and deteriorate performance.

We handle multiple datasets using a principled approach. We choose to send multiple datasets concurrently in Bohr. We determine which datasets and how many data to be moved out by utilizing a linear program formulation with similarity information. The LP also takes into account reduce task placement, which we introduce now.

5 DATA AND TASK PLACEMENT

In this section, we describe our solution for similarity aware data and task placement across the sites. As explained we consider multiple datasets.

For ease of presentation, we consider the links between the sites and the Internet backbone as the only bottleneck. This is valid based on empirical measurements [5]. We assume that the sites have abundant compute and storage capacity. We also assume that the data reduction ratio between the amount of intermediate data after the map stage and that of input data is known. This can be obtained by profiling the recurring queries that access the dataset. Table 1 summarizes the key notations.

Table 1: Notations used in the paper

Symbol	Meaning
i, j, k	index variables for sites
a	index variable for datasets
S_i^a	similarity of the data at site i for dataset a
$S_{i,j}^a$	similarity between sites i and j for dataset a
I_i^a	amount of original input data for a on i
R^a	data reduction ratio for dataset a
U_i	uplink bandwidth on site i
D_i	downlink bandwidth on site i
T	length of time lag between query arrivals
$x_{i,j}^a$	amount of data moved from i to j for dataset a
r_i	fraction of reduce tasks executed at site i
$f_i^a(x^a)$	amount of shuffle data at site i for dataset a after data movement

Initially before data movement, for a given dataset a , the amount of input data is I_i^a at site i , which results in $I_i^a R^a$ intermediate data after the map stage. After the combiners combining similar records, this amounts to $I_i^a R^a (1 - S_i^a)$ intermediate data for shuffling at site i . By the same token, after data movement, we move $\sum_{j \neq i} x_{i,j}^a$ data out from i , and receive $\sum_{k \neq i} x_{k,i}^a$ from other sites to i . The total amount of shuffle data then becomes

$$f_i^a(x^a) = \left(I_i^a - \sum_{j \neq i} x_{i,j}^a + \sum_{k \neq i} x_{k,i}^a \right) R^a (1 - S_i^a). \quad (1)$$

If we assign r_i fraction of reduce tasks to i , in the all-to-all shuffle communication pattern, i needs to upload $(1 - r_i)$ fraction of its shuffle data to other sites for a total of $(1 - r_i) f_i^a(x^a)$, and to download r_i fraction of shuffle data from each of the other sites for a total of $r_i \sum_{j \neq i} f_j^a(x^a)$.

Thus, the data and task placement problem can be formulated as follows:

$$\min t \quad (2)$$

$$\text{s.t.} \quad \sum_a (1 - r_i) f_i^a(x^a) / U_i \leq t, \forall i, \quad (3)$$

$$\sum_a r_i \sum_{j \neq i} f_j^a(x^a) / D_i \leq t, \forall i, \quad (4)$$

$$\sum_a \sum_{j \neq i} x_{i,j}^a / U_i \leq T, \forall i, \quad (5)$$

$$\sum_a \sum_{k \neq i} x_{k,i}^a / D_i \leq T, \forall i. \quad (6)$$

$$\sum_i r_i = 1, r_i \geq 0, \forall i. \quad (7)$$

We aim to minimize the total shuffle time for all datasets. In the formulation, constraint (3) characterizes total time to upload shuffle data at site i for all the datasets, and constraint (4) captures the total time to download shuffle data from other sites to i across all datasets. Constraint (5) characterizes the upload time to move data out from i to the other sites before shuffling, while (6) is the download time to download data to i from other DCs before shuffling. Note that data movement needs to be completed within T , the time lag between query arrivals. The decision variables are $\{x_{i,j}^a\}$ for data placement, and $\{r_i\}$ for task placement.

Note that the above formulation jointly considers data and (re-) task placement, and is different from Iridium [27] which considers them separately. Intuitively, data placement directly impacts task placement as moving data to a site means we should assign more tasks there, so that the uploading of the shuffle data to other sites do not become the bottleneck. Our formulation is a linear program, and can be efficiently solved by many centralized solvers. We evaluate the LP solving time in §8.5 to show that it does not add much overhead.

We acknowledge that our formulation is based on simplified assumptions. Extending it to consider other factors, such as the compute constraints of each site [22] is an interesting venue of future work.

6 RDD SIMILARITY

So far we have discussed how to exploit similarity for data and task placement before a query runs. In this section, we show that the same idea can also be applied at runtime for Spark-based Bohr to further speed up query execution.

Recall that the main abstraction Spark provides is the resilient distributed dataset (RDD). RDD is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs on the same node are randomly assigned to executors to process by default. Instead, we can cluster similar RDDs and assign them to the same executor as shown in Figure 5, so that communication among executors can be sped up and the QCT can be improved. Similarity among RDDs has not been well studied to the best of our knowledge.

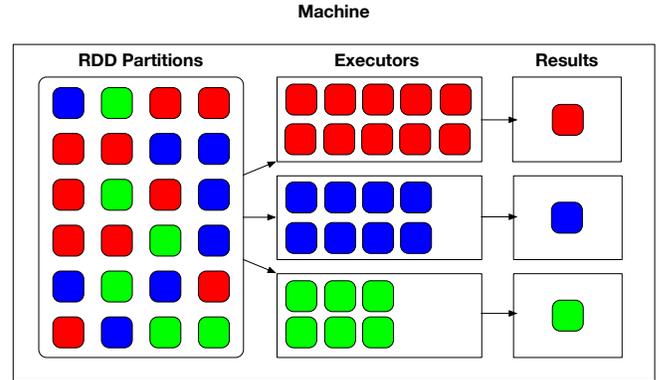


Figure 5: Bohr assigns similar RDDs to executors at runtime, so inter-executor communication can be reduced to speed up query execution.

We now explain how to perform similarity checking among RDD partitions in detail. Essentially we need to find the similarity between all pairs of the RDD partitions. This is a typical all-pair similarity problem, sometimes known as **similarity join** [30]. The naive way to check pair-wise RDD similarity is to compare the similarity score of the RDD pair to all the other RDD partitions to see if the score is larger than a threshold. To do this, first we need to extract records from the RDD partition. We use Jaccard similarity which is widely used for identifying similarity of two records as the similarity metric. Jaccard similarity for two RDD partitions X and Y is calculated by $J(X, Y) = \frac{X \cap Y}{X \cup Y}$. It measures the fraction of identical records in both RDD partitions which can be combined in the shuffle phase.

However, the computational complexity of pairwise Jaccard similarity is high, since we need to compare record by record for each pair of RDD partitions. Yet not all RDD pairs are similar; some RDD partitions are not similar at all and it is not necessary to examine every record before ruling them out. Thus in Bohr we utilize the DIMSUM algorithm [34] to efficiently calculate the pair-wise RDD similarity on the same machine.

The basic idea of the DIMSUM algorithm is to probabilistically skip checking the records when it is clear that the two sets are dissimilar. It has a single parameter (γ) to trade off computation time for accuracy. It is proposed with cosine similarity [34]. Thus we modify it to change the similarity function to Jaccard similarity. The map part of DIMSUM is used to look for similarities between two RDD partitions. Assume that a RDD partition contains L records, and for each record m hash functions are used to examine similarity. The mappers emit all pairs of records w_1 and w_2 as long as their hash values are identical with any of the m hash functions. The reducers then count, for a given pair of records, how many times their hash values are identical to output the final Jaccard between two RDD partitions. The detailed explanation of the algorithm can be found in [35].

We record the RDD similarity scores using a similarity matrix in every machine. We then utilize k-means to cluster the partitions according to the similarity matrix into n cluster, where n is the number of executors in this machine, so that we can assign similar

partitions to the same executor. In the evaluation we find the number of partitions in a machine is mild and the similarity matrix is small §8.4. Thus k-means is adequate to cluster the partitions with little overhead.

7 PROTOTYPE IMPLEMENTATION

We implement Bohr on Apache Spark v2.1.0 [4]. We utilize Apache Kylin [3] OLAP cubes on top of Hive [2] to store datasets across the DCs. We implement data generation with OLAP cubes and the similarity checking mechanism explained in §4. We modify the Spark default scheduler to implement our task and data placement algorithm. We do not disable the default replication mechanism in HDFS, and all our data movements hence only create additional copies of the data, leaving data durability unaffected. As storage is abundant, we believe this is an acceptable design. User queries are submitted through a uniform interface provided by the Spark manager. Since Bohr is built upon Spark, it can leverage Spark SQL to parse SQL queries.

We use simple techniques to do bandwidth and data reduction estimation similar to past work [27, 32, 33]. We periodically check the available bandwidth of each site, assuming it is relatively stable in the granularity of minutes. For data reduction ratio, it can be estimated with recurring queries that perform the same analytics. We use the input and actual intermediate data size of the previous query at each site to calculate the data reduction ratio to be used for the next recurring query at this site. We utilize the k-means method in Spark MLlib to cluster the RDD partitions for executor assignment in each machine at runtime. We aim to make Bohr source code open source soon.

8 EVALUATION

We present our evaluation of Bohr on AWS EC2 now.

8.1 Experimental setup

We deploy our Bohr prototype across ten EC2 regions: Seoul, Singapore, Sydney, Tokyo, Ireland, Frankfurt, London, Oregon, Virginia, and Ohio. Our experiments use m4.xlarge instances each with 16 vCPU cores and 64GB memory.

We use three commonly used analytic workloads to drive our experiments.

- (1) *AMPLab big data benchmark* [1], or *big data workload* in short, is derived from workloads studied in [26] with identical schema of the data. We use three types of queries: simple scans, aggregations, and user define functions (UDFs). The UDF here calculates a simplified version of PageRank and is implemented following [1].
- (2) *TPC-DS* [6] is an industry standard benchmark. Its underlying business model is a retail product supplier such as Amazon. The benchmark mainly consists of OLAP SQL queries that examine large volumes of data to extract business intelligence.
- (3) *Facebook workload* comes from historical Hadoop traces on a cluster at Facebook, which now has grown to 3000 machines. The trace spans 1.5 months from October 2010 to November 2010, and contains roughly 1 million jobs.

Each workload has 400GB total input data and we assign 40GB to each site as the initial data placement. The workloads are assigned in two ways: (1) uniformly at random; (2) in a locality aware fashion by clustering the input data based on attributes like date, region, etc. to the same sites to reflect the inherent data locality from the data procurement process.

We directly use the available WAN bandwidth at our VMs in the experiments. The WAN bandwidth at Singapore, Tokyo, and Oregon is about 2.5x larger than Virginia, Ohio, and Frankfurt, and 5x larger than the rest of the regions. The main performance metrics we use is query completion time (QCT) and data reduction ratio. Note here data reduction ratio is defined as how much intermediate shuffle data is saved compared to processing the query in-place.

We compare the following schemes:

- (1) *Iridium* [27]: State-of-the-art system for geo-distributed analytics. Iridium is also based on Spark.
- (2) *Iridium-C*: Iridium with OLAP cubes as data storage to serve queries. This is to show that the improvement of Bohr is not from using OLAP cubes. This also serves as the baseline of our evaluation.
- (3) *Bohr-Sim*: Bohr without runtime RDD similarity or joint data and task placement. The bottleneck site selects the high-value datasets to transfer sequentially according to the heuristics in [27]. Essentially this system only uses similarity during the data movement process. This is to demonstrate the benefit of exploiting similarity among data.
- (4) *Bohr-Joint*: Bohr with data similarity and joint data and task placement. This is to show the benefit of joint data and task placement.
- (5) *Bohr-RDD*: Bohr with data similarity and runtime RDD similarity. This is to demonstrate the benefit of exploiting runtime RDD similarity.
- (6) *Bohr*: Our complete system.

We set the number of records in a probe to be 30 in the experiments unless otherwise stated. We use 300 datasets each with multiple queries in each experiment run. The number of queries per dataset varies uniformly at random between 2 and 10. We repeat each experiment with 5 runs to obtain the results. We evaluate how the data initial placement would affect the system performance in the overall comparison, for the rest of the evaluation, we use random assignment as the baseline.

8.2 Overall Performance

We first set out to understand the overall performance gain of Bohr compared to the-state-of-the-art, using multiple queries for multiple datasets. We use both the random assign workloads and workloads with initial locality to do the evaluations.

Figure 6 shows the average QCT comparison. When we use OLAP cubes for Iridium, we can see a small performance gain due to the use of schema. Iridium-C takes 5.83s to finish the scan query in big data workload query while Iridium takes 6.15s, which is about 5% improvement. For UDF and aggregation queries, Iridium-C is around 9% and 10% faster than Iridium, respectively. For TPC-DS workload, Iridium-C takes 3.02s for a query, which is 20% faster than Iridium. For Facebook workload, Iridium-C also provides 14% faster QCT.

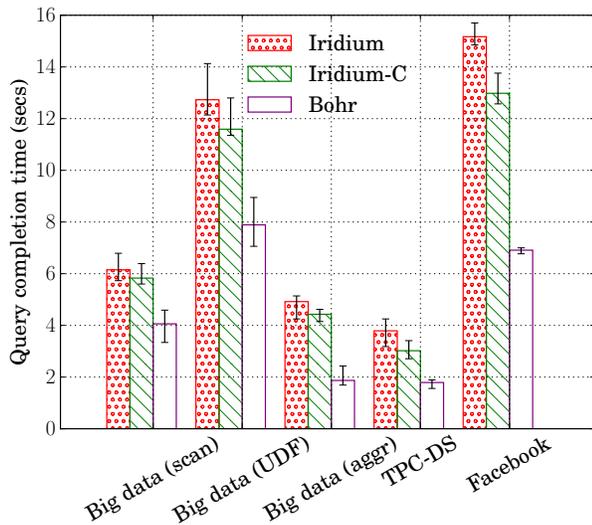


Figure 6: Query completion time comparison (random initial placement).

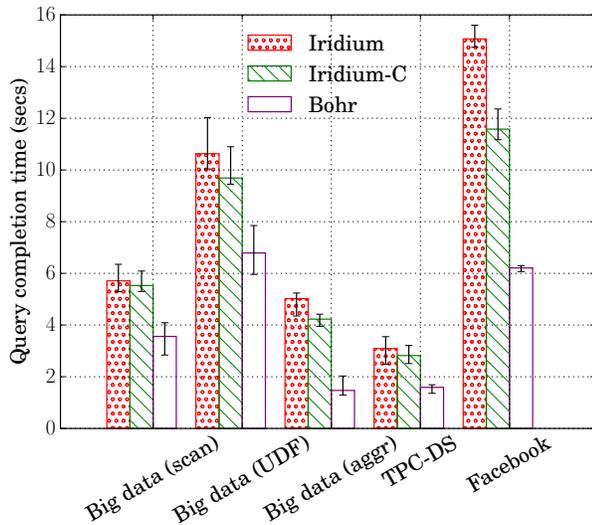


Figure 7: Query completion time comparison (locality-aware initial placement).

Bohr consistently performs better than Iridium-C in all types of queries. When processing big data workload, Bohr is 26%, 25%, and 52% faster than Iridium-C for scan, UDF and aggregation queries, respectively. For TPC-DS workload Bohr is 33% faster, and for Facebook workload 42% faster.

We also analyze Bohr’s effectiveness in reducing intermediate shuffle data. Figure 8 shows the average data reduction ratio of Bohr, Iridium and Iridium-C over original Spark. We only show the data reduction ratios with the big data workload since results with other workloads are similar. On average Bohr achieves ~30% data reduction, and is significantly better than Iridium-C and Iridium.

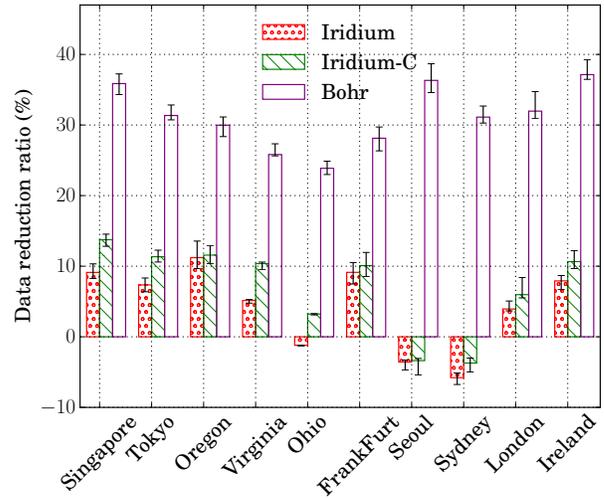


Figure 8: Intermediate data reduction comparison (random initial placement).

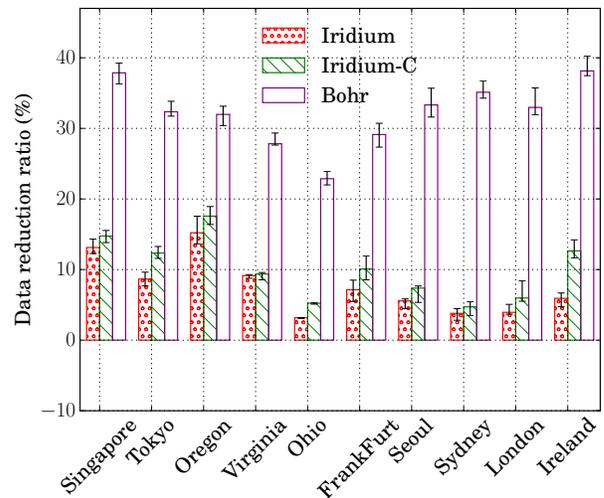


Figure 9: Intermediate data reduction comparison (locality-aware initial placement).

In the best case, for London, Bohr achieves 31.97% data reduction which is 5.3x better than Iridium-C with 5.97% data reduction and 8.1x better than Iridium with 3.94% data reduction. In the worst case of Oregon, Bohr achieves 29.98% data which is still 2.6x better than Iridium-C with 11.57%. Note that in certain locations (Ohio for Iridium, Seoul and Sydney for both Iridium and Iridium-C), Iridium and Iridium-C actually generate more intermediate data than vanilla Spark. This is because their data movement is similarity agnostic, and data transferred into these sites cannot be combined in the combine phase. Due to similarity-aware data transfer, Bohr instead achieves 23.87%, 36.32%, and 31.12% data reduction respectively at these sites. Our results thus demonstrate that Bohr effectively

improves QCT over Iridium-C by reducing the intermediate shuffle data.

We now change the initial data placement to from random to locality-aware. As shown in Figure 7, QCTs for both Iridium and Iridium-C are around 5% better compared to random initial placement. Similarly Bohr also gets 5% improvement compared to using random initial placement. We believe this gain comes from the reduced similarity searching and checking time with locality-aware initial placement. For data reduction, as shown in Figure 9, the data reduction ratio of Bohr is almost unchanged, and that of Iridium and Iridium-C is improved compared to Figure 8. They do not change the conclusion here that Bohr improves QCT and reduces intermediate data significantly.

8.3 Microbenchmarks

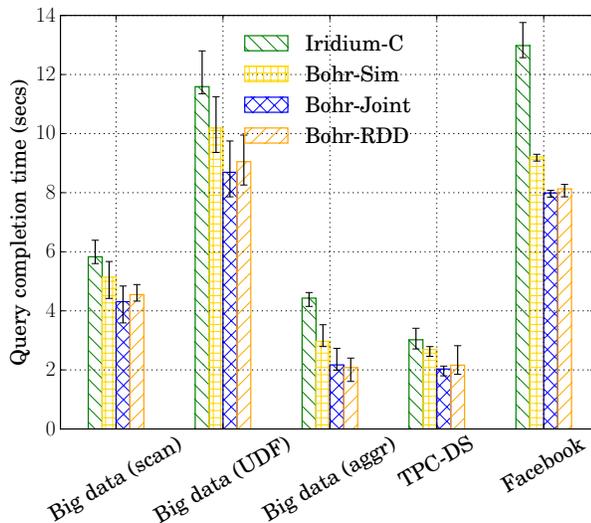


Figure 10: Benefit of different design components of Bohr in query completion time with the big data workloads.

We now would like to micro-benchmark the performance of the three key components of Bohr. Figure 10 depicts the average QCT performance of Bohr-Sim, Bohr-Joint, and Bohr-RDD compared to Iridium-C, while Figure 11 shows the average data reduction of these schemes with the big data workloads. Data reduction results with other workloads are similar.

8.3.1 Benefit of Data Similarity. First we evaluate the benefit of using data similarity, by comparing the performance between Bohr-Sim and Iridium-C. As we can see from Figure 10, for the big data benchmark Bohr-Sim is ~12% faster than Iridium-C for scan and UDF queries, and 33% faster for aggregation queries. For TPC-DS and Facebook workloads Bohr is 11% and 29% faster, respectively. As a result, across different queries and workloads, using similarity in data movement alone is able to reduce the average QCT by ~20% compared to Iridium-C.

Figure 11 shows the data reduction provided by Bohr-Sim and Iridium-C. In Seoul, Bohr-Sim delivers the highest performance

improvement: 31% data reduction compared to Iridium-C’s -3.37%. In the worst case of the Virginia site, Bohr-Sim achieves 17% data reduction, while Iridium-C achieves ~10% data reduction. Similar to the observation we made to Figure 8 in §8.2, in sites Seoul and Sydney Iridium-C achieves negative data reduction, while Bohr-Sim achieves much better data reduction by exploiting data similarity.

Therefore overall we observe that most of Bohr’s improvement comes from using data similarity, especially in data reduction ratio.

8.3.2 Benefit of Joint Data and Task Placement. We then evaluate the performance benefit of joint task and data placement using Bohr-Joint, the second key component of Bohr. Bohr-Joint is able to consider multiple datasets optimally with the optimization formulation in §5, while Iridium-C (Bohr-Sim and Bohr-RDD too) uses simple heuristics to identify high-value datasets and move them sequentially [27].

As shown in Figure 10, in the best case, Bohr-Joint provides an additional 15%–20% speedup for the average QCT compared to Bohr-Sim. Similar observation can be made to data reduction as shown in Figure 11. Bohr-Joint’s data reduction ratio is about 15%–20% higher than that of Bohr-Sim. The results confirm that similarity aware joint task and data placement helps to improve performance especially with multiple datasets.

8.3.3 Benefit of RDD Similarity. We now look at the benefit of using RDD similarity in Bohr, by comparing to Bohr-Sim.

As shown in Figure 10, Bohr-RDD provides an additional ~10% QCT benefit over Bohr-Sim, which is less significant than Bohr-Joint. In terms of data reduction, Bohr-RDD essentially has the same data reduction ratio as Bohr-Sim in Figure 11, since it works on each worker machine and does not significantly impact the amount of shuffle data. Thus, we can see that RDD similarity is able to improve QCT performance by mostly speeding up the execution at each worker.

8.4 Impact of Probing

The performance of Bohr depends on the accuracy of the similarity information, which is obtained through probes as explained in §4.2. We now benchmark the effectiveness of probing by varying the number of records in a probe k .

Figure 12 shows that data reduction of Bohr improves when we increase k . This is because we get more accurate similarity information when we increase the number of representative records for each query type. When we increase k beyond 30 (say 100), however, we only get marginal improvement while similarity checking takes much longer as will be shown in §8.5.

Figure 13 shows the QCT performance of Bohr with different values of k . Again with a larger k , QCT is consistently smaller due to more data reduction. When k is beyond 30, QCT improvement also becomes marginal. Thus we choose $k = 30$ as the default setting for Bohr.

We have also investigated the relationship between the datasets and the number of records needed by a dataset in a probe. Table 2 shows various properties of four sample datasets that the query access. We determine the number of records contained in the probe for each dataset mainly based on the dataset size.

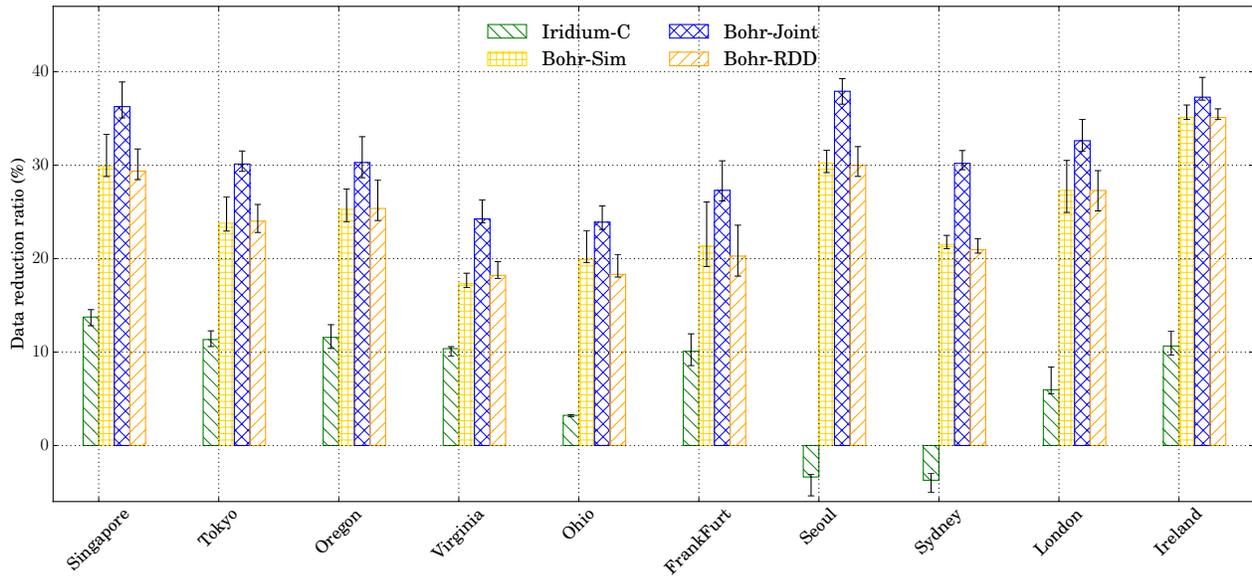


Figure 11: Benefit of different design components of Bohr in data reduction ratio with the big data workloads.

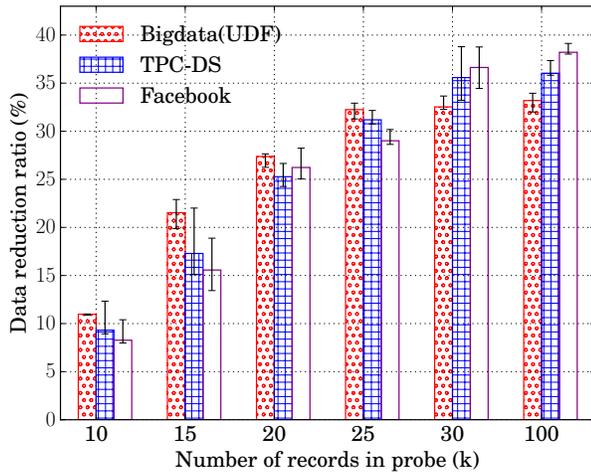


Figure 12: Effect of k on data reduction ratio.

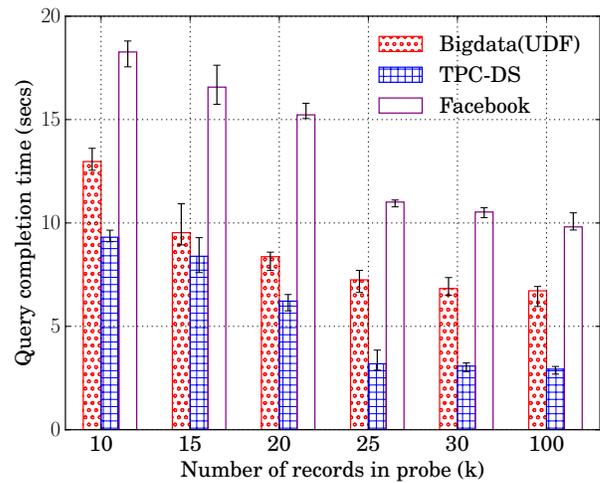


Figure 13: Effect of k on QCT.

Table 2: Dataset attributes and its impact on probing. Note the total number of records in a probe across datasets is 30.

Dataset id	1	3	7	10
# dimensions	15	42	13	8
Dataset size	0.87G	4.32G	3.21G	0.57G
# records in a probe	3	15	10	2
Similarity checking time	0.32s	1.12s	0.94s	0.15s

8.5 Overhead Analysis

In this section, we analyze the overhead brought by Bohr.

The latency overhead mainly comes from OLAP cube generation and similarity checking. OLAP cube generation is done offline during data generation. As stated in the §7, generating the OLAP cube only incur little latency overhead. We measure the average time to generate a new OLAP cube from the complete 40GB dataset from scratch, which stands at 15.05s and 8.41s, respectively, for images and text logs. Further, new data are added into the OLAP cube continuously as they become available (in batches for example),

Table 3: Data similarity checking time in pre-processing.

# records per probe	10	15	20	25	30	100
Similarity checking	0.59s	0.73s	1.29s	1.97s	2.95s	12.57s

Table 4: Overhead of RDD similarity checking in Bohr.

# Executors in a node	2	4	6	8
RDD similarity checking	0.42s	1.07s	1.59s	3.06s
QCT	16.52s	13.49s	10.27s	11.35s

Table 5: LP function solving time.

Workload	Big data(scan)	Big data(UDF)	Big data(aggr)	TPC-DS	Facebook
LP solving time	1.41s	2.15s	0.39s	0.21s	2.52s

which further reduce the latency overhead. Given a 30-second interval between queries, which are rather frequent for geo-distributed analytics, the overhead is $\sim 0.77s$ and $0.43s$, respectively for images and text logs, assuming 2GB new data become available during the interval.

The latency overhead of similarity checking mainly stems from two parts: (1) using the probes to detect data similarity; (2) RDD similarity checking. For probing, because it happens in the pre-processing stage before the query arrives, it does not affect the query completion time. Table 3 shows the similarity checking time when we vary k , the amount of records in the probe. Even with 100 records per probe, similarity checking takes less than 13 seconds on average, and can be easily done in the pre-processing stage without adding any overhead. Table 4 then summarizes the latency overhead of RDD similarity checking with varying number of executors in a node. We evaluate the overhead using the TPC-DS workload, and k is set to the default value 30. Thus the result demonstrates that the latency overhead of RDD similarity checking is mild and acceptable. Table 5 summarizes the LP solving time for the joint data and task placement with various workloads. It takes at most 2.52s to solve the LP when for the Facebook workload. Note that the LP can be used for multiple iterations of data and task placement, thus it is very efficient. Both RDD similarity checking and LP solving time are included when we measure QCT.

We now turn to the storage overhead in Bohr due to OLAP cubes. The results are shown in Table 6. Recall in §8.1 we explained that each node has 40GB data per workload. The average storage overhead per workload in Iridium is 42.32GB. Iridium-C consumes additional 17.39GB due to the OLAP cubes. Bohr enabled both OLAP cube and similarity checking so it consumes more storage, with 59.02GB on average. The overhead of OLAP cubes is 17.76GB and

Table 6: Per-node storage overhead comparison. The unit is GB. Note each node has 40GB input data per workload. Storage needed by queries is the storage space that is actually required to process queries. For Iridium-C, queries only need the OLAP cubes; for Bohr queries need the OLAP cubes and similarity metadata.

	Storage per node	Storage needed by queries	OLAP cubes	Similarity meta-data
Iridium	42.32	43.92	-	-
Iridium-C	57.97	18.52	17.39	-
Bohr	59.02	19.93	17.76	0.82

the similarity metadata is approximately 0.82GB. This represents about 43% storage overhead compared to the raw input data. We believe this is acceptable in production systems given that the cost of storage is continuously decreasing. Note that the raw input data can be moved to cold storage after pre-processing, since the queries only need OLAP cubes and similarity data. The storage needed by queries in Bohr is actually smaller than Iridium which relies on the raw data. Due to the overhead of performing OLAP operations, storage needed by queries is higher than storage for OLAP cubes and similarity metadata if any.

8.6 Highly Dynamic Datasets

In practice data processed by the recurrent queries can vary significantly mostly because of the generation of new data. In this section we evaluate Bohr’s performance under highly dynamic datasets.

To simulate highly dynamic datasets, we divide our 40GB input data on every node to two parts. The first part of 10GB is used as the initial data at the node. The rest 30GB are continuously provided to the node in batches of 2GB. Each batch of new data are available at the node every 20 seconds, which is also the query arrival interval.

We conduct the experiment as follows: (1) We utilize the first part of data to complete the initial task and data placement. (2) When a new batch of data arrives, they are pre-processed and stored in the OLAP cube and transferred to other sites if necessary according to the initial task and data placement decision before the next query arrives. (3) The next query processes using all currently available data at the node. (4) For every five queries, i.e. 10GB new data, Bohr updates the task and data placement decision by invoking the similarity checking mechanism and solving the LP again with the up-to-date information.

As shown in Table 7, query completion time with dynamic datasets is very similar to the normal setting for all workloads. The main reason is that pre-processing of the new data incurs little latency overhead as we mentioned in §8.5.

9 RELATED WORK

Geo-distributed analytics: Many systems have been developed for geo-distributed data analytics in recently years. Generally there are two main objectives: (1) minimizing wide area bandwidth usage; (2) minimizing average QCT.

Table 7: Evaluating Highly Dynamic Dataset

Workload Type	TPC-DS	Facebook	Big Data
Normal	1.79s	6.91s	7.89s
Dynamic	1.81s	6.89s	7.95s

To minimize bandwidth, Ariel et al. propose JetStream [28] that aggregates identical records in OLAP cubes for streaming systems. However JetStream aims for streaming workloads. It does not support SQL queries like Bohr does, and does not involve data or task placement. Vulimiri et al. build Geode [33] for general SQL queries. Geode optimizes query execution to minimize wide area bandwidth and provides fault-tolerance. Geode only considers data placement which limits the performance gain compared to Bohr.

As Pu et al. argue in [27], minimizing bandwidth is not always beneficial; some times it can even lead to longer QCT and worse user experience. So instead they aim at minimizing QCT directly. They propose Iridium which focuses on recurring queries. Iridium first determine which data sites are the bottleneck and then try to move data out of bottleneck sites to other data site, Iridium also adjusts the reduce task placement across the sites correspondingly. Bohr further exploits data similarity to optimize which data should be moved, and is shown to outperform Iridium even with OLAP cubes significantly. SWAG [21] optimizes QCT by coordinating the compute task scheduling across sites. The idea is to delay parts of certain jobs without degrading their response times, thus providing opportunities for other jobs to finish faster. SWAG only takes task placement into consideration.

A recent work Clarinet [31] proposes a WAN-aware query optimizer to accelerate QCT. CLARINET operates at a higher layer than all prior systems. Thus it is complementary to those systems [27, 33] and Bohr, and can be integrated into Bohr for further improvement. Tetrium [22] considers both network and computation resources, and addresses the inter-dependency of task placement and job scheduling in the geo-distributed setting where resources are heterogeneous and constrained.

Gaia [20] focuses on reducing the WAN communication overhead in machine learning systems. Gaia uses a new synchronization model called Approximate Synchronous Parallel (ASP), whose key idea is to dynamically eliminate insignificant communication (i.e. small gradients) between data centers while still guaranteeing the correctness of ML algorithms. However, Gaia is for ML workloads and does not apply to the general data analytics systems.

Finally, the idea and basic design of using similarity for geo-distributed analytics is sketched in an earlier workshop paper [11]. This paper introduces the comprehensive design, implementation, and evaluation of a new system Bohr, and is substantially different. **Accelerating analytics systems:** Another related line of work is on accelerating data analytics systems in the common setting

of a local cluster. Many approaches have been studied, ranging from improving data locality [7, 24, 36] to mitigating stragglers [8–10]. Mantri [10] is a system that monitors tasks and culls outliers using cause and resource aware techniques, apply strategies including restarting outliers, network-aware placement of tasks and protecting outputs of valuable tasks to improve the job completion time. Dolly [8] utilize cloning of small jobs and avoiding waiting and speculation altogether to solving the problem exist in former mitigation technicals, which is the element of waiting and speculation; GRASS [9] carefully uses speculation to mitigate the impact of stragglers in approximation jobs. These techniques can still be applied at each site to locally accelerate the processing. They however do not addresses challenges specific to geo-distributed data analytics.

Some other work [19, 25] proposes to accelerate query processing by formatting the data storage. Rhea [19] is a system that automatically generates and runs storage-side data filters for unstructured and semi-structured data. It uses static analysis of application code to generate filters that remove the unused data to reduce the WAN bandwidth usage. Rhea does not consider data similarity as Bohr does. MANIMAL [25] is an analyzer for MapReduce jobs. It also applies static analysis techniques to generate an indexed and column-projected version of the data. Index-generation programs must be run to completion on the entire data set to show any benefit, and must be re-run whenever additional data is appended. The entire data set must be read by Hadoop compute nodes and then the index written back to storage. This is not suitable for geo-distributed analytics scenario because MANIMAL does not consider recurrent jobs that are common in reality. Instead, Bohr is able to deal with the scenario when new data are appended, as we have shown in §8.6.

10 CONCLUSION

We developed Bohr, a new geo-distributed data analytics system that minimizes the query completion time (QCT) over geo-distributed datasets. Our key idea is to exploit the data similarity in transferring data out of the bottleneck sites. By moving data that are highly similar, the destination site enjoys a larger data reduction ratio and produces less intermediate data after the combiner, even though its input data size actually increases. We implement Bohr on Spark and deploy it across ten sites of AWS EC2. Our extensive evaluation using realistic query workloads shows that Bohr improves the QCT by up to 50% and reduces the intermediate data by up to 6x compared to state-of-the-art solutions that also use OLAP cubes.

11 ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd Ganesh Ananthanarayanan for their valuable comments. This work was supported in part by the Research Grant Council, University Grants Committee of Hong Kong (award numbers 11216317, 11202315, and C7036-15G).

REFERENCES

- [1] AmpLab Big Data Benchmark. <https://amplab.cs.berkeley.edu/benchmark/>.
- [2] Apache Hive. <https://hive.apache.org/>.
- [3] Apache Kylin. <http://kylin.apache.org/>.
- [4] Apache Spark. <http://spark.apache.org/>.
- [5] Measuring Internet Congestion: A preliminary report. <https://ipp.mit.edu/sites/default/files/documents/Congestion-handout-final.pdf>.
- [6] TPC-DS. <http://www.tpc.org/tpcds/>.
- [7] ANANTHANARAYANAN, G., AGARWAL, S., KANDULA, S., GREENBERG, A., STOICA, I., HARLAN, D., AND HARRIS, E. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proc. ACM EuroSys* (2011).
- [8] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective straggler mitigation: Attack of the clones. In *Proc. USENIX NSDI* (2013).
- [9] ANANTHANARAYANAN, G., HUNG, M. C.-C., REN, X., STOICA, I., WIERMAN, A., AND YU, M. GRASS: Trimming stragglers in approximation analytics. In *Proc. USENIX NSDI* (2014).
- [10] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proc. ACM OSDI* (2010).
- [11] AUTHORS, A. Bohr: Similarity Aware Geo-Distributed Data Analytics. In *Proc. USENIX HotCloud* (2017).
- [12] CHÁVEZ, E., NAVARRO, G., BAEZA-YATES, R., AND MARROQUÍN, J. L. Searching in metric spaces. *ACM Comput. Surv.* (2001).
- [13] CORBETT, J. C., AND ET. AL. Spanner: Google's Globally-Distributed Database. In *Proc. USENIX OSDI* (2012).
- [14] CUZZOCREA, A., BELLATRECHE, L., AND SONG, I.-Y. Data warehousing and olap over big data: Current challenges and future research directions. In *Proc. AMC DOLAP* (2013).
- [15] FERREIRA CORDEIRO, R. L., TRAINA, JUNIOR, C., MACHADO TRAINA, A. J., LÓPEZ, J., KANG, U., AND FALOUTSOS, C. Clustering very large multi-dimensional datasets with mapreduce. In *Proc. ACM SIGKDD* (2011).
- [16] FROME, A., SINGER, Y., AND MALIK, J. Image retrieval and classification using local distance functions. In *Proc. NIPS*. 2007.
- [17] GATES, A. F., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAMURTHY, S. M., OLSTON, C., REED, B., SRINIVASAN, S., AND SRIVASTAVA, U. Building a high-level dataflow system on top of map-reduce: The pig experience. *Proc. VLDB Endow.* (2009).
- [18] GIONIS, A., INDYK, P., AND MOTWANI, R. Similarity Search in High Dimensions via Hashing. In *Proc. ACM VLDB* (1999).
- [19] GKANTSIDIS, C., VYTINIOTIS, D., HODSON, O., NARAYANAN, D., DINU, F., AND ROWSTRON, A. Rhea: automatic filtering for unstructured cloud storage. In *Proc. USENIX NSDI* (2013).
- [20] HSIEH, K., HARLAP, A., VIJAYKUMAR, N., KONOMIS, D., GANGER, G. R., GIBBONS, P. B., AND MUTLU, O. Gaia: Geo-distributed machine learning approaching LAN speeds. In *Proc. USENIX NSDI* (2017), pp. 629–647.
- [21] HUNG, C., GOLUBCHIK, L., AND YU, M. Scheduling jobs across geo-distributed datacenters. In *Proc. ACM Symposium SoCC* (2015).
- [22] HUNG, C.-C., ANANTHANARAYANAN, G., GOLUBCHIK, L., YU, M., AND ZHANG, M. Wide-area analytics with multiple resources. In *Proc. ACM EuroSys* (2018).
- [23] HURTADO, C. A., MENDELZON, A. O., AND VAISMAN, A. A. Maintaining data cubes under dimension updates. In *Proc. IEEE ICDE* (1999).
- [24] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *Proc. ACM SOSP* (2009).
- [25] JAHANI, E., CAFARELLA, M. J., AND RÉ, C. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.* (2011).
- [26] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A Comparison of Approaches to Large-scale Data Analysis. In *Proc. ACM SIGMOD* (2009).
- [27] PU, Q., ANANTHANARAYANAN, G., BODIK, P., KANDULA, S., AKELLA, A., BAHL, P., AND STOICA, I. Low Latency Geo-distributed Data Analytics. In *Proc. ACM SIGCOMM* (2015).
- [28] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., AND FREEDMAN, M. J. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proc. USENIX NSDI* (2014).
- [29] SALTON, G., WONG, A., AND YANG, C. S. A vector space model for automatic indexing. *Commun. ACM* (1975).
- [30] VERNICA, R., CAREY, M. J., AND LI, C. Efficient parallel set-similarity joins using mapreduce. In *Proc. ACM SIGMOD* (2010).
- [31] VISWANATHAN, R., ANANTHANARAYANAN, G., AND AKELLA, A. CLARINET: WAN-Aware Optimization for Analytics Queries. In *Proc. ACM OSDI* (2016).
- [32] VULIMIRI, A., CURINO, C., GODFREY, P. B., JUNGBLUT, T., KARANASOS, K., PADHYE, J., AND VARGHESE, G. WANalytics: Geo-Distributed Analytics for a Data Intensive World. In *Proc. ACM SIGMOD* (2015).
- [33] VULIMIRI, A., CURINO, C., GODFREY, P. B., JUNGBLUT, T., PADHYE, J., AND VARGHESE, G. Global analytics in the face of bandwidth and regulatory constraints. In *Proc. USENIX NSDI* (2015).
- [34] ZADEH, R. B., AND CARLSSON, G. Dimension independent matrix square using mapreduce. *CoRR abs/1304.1467* (2013).
- [35] ZADEH, R. B., AND GOEL, A. Dimension independent similarity computation. *Journal of Machine Learning Research* (2013).
- [36] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMEELEGGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. ACM EuroSys* (2010).