# Low Latency Geo-distributed Data Analytics

Qifan Pu[1,2], Ganesh Ananthanarayanan[1], Peter Bodik[1]
Srikanth Kandula[1], Aditya Akella[3], Paramvir Bahl[1], Ion Stoica[2]

[1]Microsoft Research [2]University of California at Berkeley
[3]University of Wisconsin at Madison

## ABSTRACT

Low latency analytics on geographically distributed datasets (across datacenters, edge clusters) is an upcoming and increasingly important challenge. The dominant approach of aggregating all the data to a single datacenter significantly inflates the timeliness of analytics. At the same time, running queries over geo-distributed inputs using the current intra-DC analytics frameworks also leads to high query response times because these frameworks cannot cope with the relatively low and variable capacity of WAN links.

We present Iridium, a system for low latency geo-distributed analytics. Iridium achieves low query response times by optimizing placement of both data and tasks of the queries. The joint data and task placement optimization, however, is intractable. Therefore, Iridium uses an online heuristic to redistribute datasets among the sites *prior* to queries' arrivals, and places the tasks to reduce network bottlenecks *during* the query's execution. Finally, it also contains a knob to budget WAN usage. Evaluation across eight worldwide EC2 regions using production queries show that Iridium speeds up queries by $3\times - 19\times$ and lowers WAN usage by $15\% - 64\%$ compared to existing baselines.

## CCS Concepts

**•Computer systems organization → Distributed Architectures; •Networks → Cloud Computing;**

## Keywords

geo-distributed; low latency; data analytics; network aware; WAN analytics



*SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom*
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787481

## 1. INTRODUCTION

Large scale cloud organizations are deploying datacenters and "edge" clusters globally to provide their users low latency access to their services. For instance, Microsoft and Google have tens of datacenters (DCs) [6, 11], with the latter also operating 1500 edges worldwide [24]. The services deployed on these *geo-distributed sites* continuously produce large volumes of data like user activity and session logs, server monitoring logs, and performance counters [34, 46, 53, 56].

Analyzing the geo-distributed data gathered across these sites is an important workload. Examples of such analyses include querying user logs to make advertisement decisions, querying network logs to detect DoS attacks, and querying system logs to maintain (streaming) dashboards of overall cluster health, perform root-cause diagnosis and build fault prediction models. Because results of these analytics queries are used by data analysts, operators, and real-time decision algorithms, *minimizing their response times* is crucial.

Minimizing query response times in a geo-distributed setting, however, is far from trivial. The widely-used approach is to aggregate *all* the datasets to a central site (a large DC), before executing the queries. However, waiting for such *centralized* aggregation, significantly delays timeliness of the analytics (by as much as $19\times$ in our experiments).[1] Therefore, the natural alternative to this approach is to execute the queries geo-distributedly over the data stored at the sites.

Additionally, regulatory and privacy concerns might also forbid central aggregation [42]. Finally, verbose or less valuable data (e.g., detailed system logs stored only for a few days) are not shipped at all as this is deemed too expensive. Low response time for queries on these datasets, nonetheless, remains a highly desirable goal.

Our work focuses on *minimizing response times* of geo-distributed analytics queries. A potential approach would be to leave data *in-place* and use unmodified, intra-DC analytics framework (such as Hadoop or Spark) across the collection of sites. However, WAN band-

[1]An enhancement could "sample" data locally and send only a small fraction [46]. Designing generic samplers, unfortunately, is hard. Sampling also limits future analyses.

widths can be highly heterogeneous and relatively moderate [43, 47, 48] which is in sharp contrast to intra-DC networks. Because these frameworks are not optimized for such heterogeneity, query execution could be dramatically inefficient. Consider, for example, a simple map-reduce query executing across sites. If we place no (or very few) reduce tasks on a site that has a large amount of *intermediate* data but low uplink bandwidth, all of the data on this site (or a large fraction) would have to be uploaded to other sites over its narrow uplink, significantly affecting query response time.

We build Iridium, a system targeted at geo-distributed data analytics. Iridium views a single logical analytics framework as being deployed across all the sites. To achieve low query response times, it explicitly considers the heterogeneous WAN bandwidths to optimize *data and task placement*. These two placement aspects are central to our system since the source and destination of a network transfer depends on the locations of the data and the tasks, respectively. Intuitively, in the example above, Iridium will either move data out of the site with low uplink bandwidth before the query arrives or place many of the query's reduce tasks in it.

Because durations of intermediate communications (e.g., shuffles) depend on the duration of the site with the *slowest* data transfer, the key intuition in Iridium is to *balance the transfer times among the WAN links*, thereby avoiding outliers. To that end, we formulate the task placement problem as a linear program (LP) by modeling the site bandwidths and query characteristics. The best task placement, however, is still limited by input data locations. Therefore, moving (or replicating) the datasets to different sites can reduce the anticipated congestion during query execution.

The *joint data and task placement*, even for a single map-reduce query, results in a non-convex optimization with no efficient solution. Hence, we devise an efficient and greedy heuristic that iteratively moves small chunks of datasets to "better" sites. To determine *which* datasets to move, we prefer those with high *value-per-byte*; i.e., we greedily maximize the expected reduction in query response time normalized by the amount of data that needs to be moved to achieve this reduction. This heuristic, for example, prefers moving datasets with many queries accessing them and/or datasets with queries that produce large amount of intermediate data.

Our solution is also mindful of the *bytes transferred on the WAN* across sites since WAN usage has important cost implications ($/byte) [53]. Purely minimizing query response time could result in increased WAN usage. Even worse, purely optimizing WAN usage can arbitrarily increase query latency. This is because of the fundamental difference between the two metrics: bandwidth cost savings are obtained by reducing WAN usage on *any* of the links, whereas query speedups are obtained by reducing WAN usage *only* on the bottleneck link. Thus, to ensure fast query responses *and* reasonable bandwidth costs, we incorporate a simple "knob"

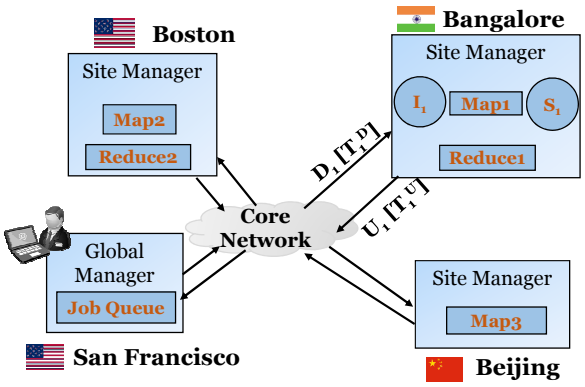



Figure 1: **Geo-distributed map-reduce query. The user submits the query in San Francisco, and the query runs across Boston, Bangalore and Beijing. We also show the notations used in the paper at Bangalore, see Table 1.**

that trades off the WAN usage and latency by limiting the amount of WAN bandwidth used by data moves and task execution. In our experiments, with a budget *equal* to that of a WAN-usage optimal scheme (proposed in [53, 54]), Iridium obtains 2× faster query responses.

Our implementation of Iridium automatically estimates site bandwidths, future query arrivals along with their characteristics (intermediate data), and prioritizes data movement of the earlier-arriving queries. It also supports Apache Spark queries, both streaming [60] as well as interactive/batch queries [59]. [2]

Evaluation across eight worldwide EC2 regions and trace-driven simulations using production queries from Bing Edge, Conviva, Facebook, TPC-DS, and the Big-data benchmark show that Iridium speeds up queries by $3\times - 19\times$ compared to existing baselines that (a) centrally aggregate the data, or (b) leave the data in-place and use unmodified Spark.

## 2. BACKGROUND AND MOTIVATION

We explain the setup of geo-distributed analytics (§2.1), illustrate the importance of careful scheduling and storage (§2.2), and provide an overview of our solution (§2.3).

### 2.1 Geo-distributed Analytics

**Architecture:** We consider the geo-distributed analytics framework to logically span all the sites. We assume that the sites are connected using a network with congestion-free core. The bottlenecks are only between the sites and the core which has infinite bandwidth, valid as per recent measurements [13]. Additionally, there could be significant *heterogeneity* in the uplink and downlink bandwidths due to widely different link capacities and other applications (non-Iridium traffic) sharing the links. Finally, we assume the sites have relatively abundant compute and storage capacity.

Data can be generated on any site and as such, a dataset (such as "user activity log for application X")

[2] https://github.com/Microsoft-MNR/GDA

could be distributed across many sites. Figure 1 shows an example geo-distributed query with a logically centralized *global manager* that converts the user's query script into a DAG of *stages*, each of which consists of many parallel *tasks*. The global manager also coordinates query execution across the many sites, keeps track of data locations across the sites, and maintains durability and consistency of data; durability and consistency, though, are not the focus of our work.

Each of the sites is controlled by a local *site manager* which keeps track of the available local resources and periodically updates the global manager.

**Analytics Queries:** Input tasks of queries (e.g., map tasks) are executed locally on sites that contain their input data, and they write their outputs (i.e., intermediate data) to their respective local sites. Input stages are extremely quick as a result of data locality [37, 58] and in-memory caching of data [17, 59].

In a geo-distributed setup, the main aspect dictating response time of many queries is efficient transfer of *intermediate* data that necessarily has to go across sites (e.g., all-to-all communication patterns). In Facebook's production analytics cluster, despite local aggregation of the map outputs for associative reduce operations [9, 57], ratio of intermediate to input data sizes is still a high 0.55 for the median query, 24% of queries have this ratio $\geq 1$ (more in §6). Intermediate stages are typically data-intensive, i.e., their durations are dominated by communication times [25, 26, 52].

Queries are mostly *recurring* ("batched" streaming [60] or "cron" jobs), e.g., every minute or hour. Because of their recurring nature, we often know the queries that will run on a dataset along with any *lag* between the generation of the dataset and the arrival of its queries. Some *ad hoc* analytics queries are also submitted by system operators or data analysts. Timely completion of queries helps real-time decisions and interactivity.

**Objectives:** Our techniques for task and data placement work inside the global manager to reduce *query response time*, which is the time from the submission of a query until its completion. At the same time, we are mindful of WAN usage (bytes transferred across the WAN) [53, 54] and balance the two metrics using a simple knob for budgeted WAN usage.

## 2.2 Illustrative Examples

While Iridium can handle arbitrary DAG queries, in this section, we illustrate the complexity of minimizing the response time of a geo-distributed query using a canonical map-reduce query. As described above, efficient transfer of intermediate data across sites is the key. Transfer duration of "all-to-all" shuffles is dictated by, a) placement of reduce tasks across sites, §2.2.1, and b) placement of the input data, §2.2.2; since map outputs are written locally, distribution of the input data carries over to the distribution of intermediate data. We demonstrate that techniques in intra-DC scheduling and storage can be highly unsuited in the geo-distributed setup. We will also show that techniques to minimize WAN usage can lead to poor query response times.

| Symbol | Meaning |
|---|---|
| $I_i$ | amount of input data on site $i$ |
| $S_i$ | amount of intermediate (map output) data on site $i$ |
| $\alpha$ | selectivity of input stage, $S_i = \alpha I_i$ |
| $D_i$ | downlink bandwidth on site $i$ |
| $U_i$ | uplink bandwidth on site $i$ |
| $r_i$ | fraction of intermediate (reduce) tasks executed in site $i$ |
| $T_i^U$, $T_i^D$ | finish time of intermediate data transfer on up and down link of site $i$ |

Table 1: **Notations used in the paper.**

For ease of presentation, we consider the links between the sites and the network core as the only bottlenecks and assume that IO and CPU operations of tasks have zero duration. Table 1 contains the notations. In general, $I_i$, $S_i$, $D_i$, and $U_i$ represent the query input data, intermediate (map output) data, downlink and uplink WAN bandwidths on site $i$, respectively. The fraction of intermediate (reduce) tasks on a site $i$ is $r_i$; we use the term *link finish time* to refer to $T_i^U$ and $T_i^D$ which represent the time taken to upload and download intermediate data from and to site $i$, respectively.

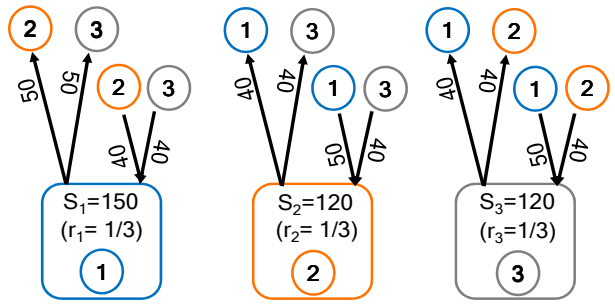
### 2.2.1 Intermediate Task Placement

Consider a log with the schema ⟨`timestamp`, `user_id`, `latency`⟩ recording latencies of user requests. A user's requests could be distributed across many sites (say, when a user represents a large global customer). Our sample query computes *exact* per-user median latency (`SELECT user_id, median(latency) GROUP BY user_id`). As we execute a user-defined and non-associative function, `median()`, the map tasks output all pairs of ⟨`user_id`, ⟨`latency`⟩⟩, and this intermediate data is shuffled across all sites. Assume the intermediate outputs are half the input size; selectivity $\alpha = 0.5$ (Table 1). Every reduce task collects `latency` values for a subset of `user_id` values and calculates the median per user.

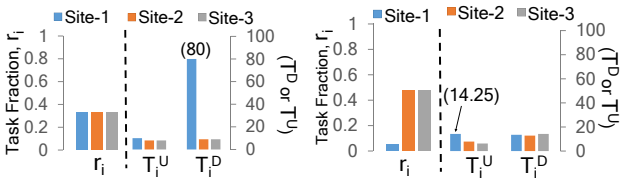
We consider execution of the above query over three sites; see Table 2a for input and intermediate data sizes and bandwidths available at the sites. State-of-the-art approaches to scheduling reduce tasks recommend equal spreading of reduce tasks across sites (or racks and machines) [19, 52]. Such an approach would result in one-third of the tasks on each of the three sites, $r = (0.33, 0.33, 0.33)$, resulting in a query response time of 80s (Figure 2b and 2c). Each of the sites has data traversing its up and down links, whose finish times ($T_i^U$ and $T_i^D$) depend on their bandwidths. The transfer duration is the maximum of all the link finish times, and we colloquially refer to the slowest site as the *bottleneck*. In this case the bottleneck is site-1 with a slow downlink (1MB/s) which has to download 1/3 of intermediate

| | Site-1 | Site-2 | Site-3 |
|---|---|---|---|
| Input Data (MB), $I$ | 300 | 240 | 240 |
| Intermediate Data (MB), $S$ | 150 | 120 | 120 |
| Uplink (MB/s), $U$ | 10 | 10 | 10 |
| Downlink (MB/s), $D$ | 1 | 10 | 10 |

(a) Setup of three sites.




(b) When tasks are *equally spread* across the three sites, $\frac{2}{3}$ of the data ($S_i$) on each site is sent out (uplink), split equally to the other two sites. The download at each site (downlink) can, thus, be correspondingly summed up.




(c) Equal Spread (d) Better Alternative

Figure 2: **Intermediate Task Placement over 3 sites (a), comparing equal spreading of tasks (b, c) and a better alternative (d). Task fractions ($r_i$) are shown on the left y-axis in ((c) and (d)) while link finish times ($T_i^D$ or $T_i^U$) are on the right y-axis.**

data from sites 2 and 3, i.e., 120MB/3 + 120MB/3 = 80MB (Figure 2b).

A better alternative, covered in §3, distributes reduce tasks with ratios $r = (0.05, 0.475, 0.475)$ across the three sites, reducing the query response time over five-fold to 14.25s, Figure 2d. The alternative, essentially, identified site-1's downlink to be the bottleneck link in Figure 2c, and hence moved tasks away from it to the other sites. For brevity, we omit the corresponding illustration similar to Figure 2b on calculating link finish times.

We cannot improve the reduce task placement much more, since site-1's up and downlinks have approximately similar link finish times now. Increasing or decreasing $r_1$ will increase $T_1^D$ or $T_1^U$, respectively, and thus increase response time. Thus, task placement needs to *carefully balance the up/down link usages* of the sites.

**WAN Usage:** To minimize WAN usage [53, 54] we need to collect the intermediate data ($S_i$) from sites-2 and 3 into site-1 since site-1 already contains the most intermediate data. This results in cross-site WAN usage of 240MB, but takes 240s (downlink $D_1 = 1$MB/s). In contrast, it can be calculated that the alternative task placement we proposed results in 268.5MB of WAN data usage; increase in WAN usage of just 12% reduces query response time 17× (from 240s to 14.25s). Thus, schemes minimizing WAN usage can be highly inefficient for query response time. This is because savings in WAN usage accrue with each link on which we reduce the amount of data transferred, whereas we reduce response time only by optimizing the bottlenecked link.

In fact, task placements of both the policies—equal spreading and minimizing WAN usage—could result in *arbitrarily large* query response times. For example, as $S_2$ increases, response time of the equal-spread policy increases linearly, while the optimal task placement will place all reduce tasks in site-2 and keep the response time constant. Similarly, as $D_1$ gets smaller, e.g., when $D_1 = 0.1$MB/s, minimizing WAN usage requires 2400s, while we achieve a response time of 15s by placing no reduce tasks on site-1.

### 2.2.2 *Input Data Placement*

In §2.2.1, we assumed that the query inputs stayed on the sites that they were initially generated/stored. Since even the best task placements are limited by the locations of the data, it may be beneficial to move the input data to different sites before starting the query.[3] For example, when input data was generated at time $a_0$ and query is submitted at time $a_1$ ($a_1 > a_0$), we can use this *lag* of $(a_1 - a_0)$ to rearrange the input data to reduce query response time. Even when $a_0 = a_1$ but intermediate data is larger than input data ($\alpha > 1$, Table 1), moving input data would be more efficient than moving the intermediate data. Recall that since the input tasks write their outputs (intermediate data) to the local site, any change in distribution of input data carries over to the intermediate data.

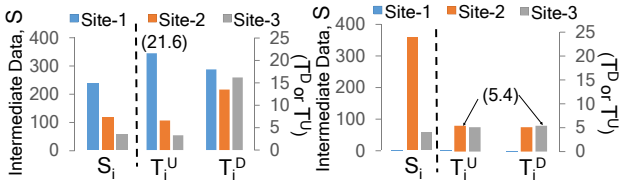
Rearranging the input data, however, is non-trivial, because as we change $S_i$'s, we have to recompute the optimal $r_i$'s as well. Consider a query with input $I = (240, 120, 60)$MB across three sites, $\alpha = 1$, and a lag of 24s between data generation and query arrival. As before, we assume that IO and CPU operations of tasks have zero duration and the WAN as the only bottleneck. Figure 3a shows the data and bandwidths at the sites along with query response time when data is left "in place" (Figure 3b). Site-1's uplink is the bottleneck link whose link finish time is 21.6s.

A better input placement will move data out of the bottlenecked site-1 in the available 24s, and Figure 3c shows the benefit of the best movement: from site-1 to site-2. The moving is gated on site-1's uplink (10MB/s) moving 240MB of data in 24s. This new data distribution reduces the response time 4× from 21.6s to 5.4s. The different $r_i$ values between Figures 3b and 3c shows that minimizing query response time indeed requires a *joint optimization over data and task placement.*

[3]While we use the term "move" in describing our solution, we in fact just *replicate*, i.e., create additional copies, §5.

| | Site-1 | Site-2 | Site-3 |
|---|---|---|---|
| Input Data (MB), $I$ | 240 | 120 | 60 |
| Intermediate Data (MB), $S$ | 240 | 120 | 60 |
| Uplink (MB/s), $U$ | 10 | 10 | 10 |
| Downlink (MB/s), $D$ | 1 | 10 | 10 |

(a) Setup of three sites.




(b) "In-place" Input Placement (c) Better Input Placement

Figure 3: **Input Data Placement for (a). Comparison of transfer durations when data is left *in place* (b) with moving data from site-1 to site-2. The query arrives with a lag of 24s after the data is available. Intermediate data ($S_i$) is shown on the left y-axis, instead of $r_i$ in Figure 2. Link finish times ($T_i^D$ or $T_i^U$) are on the right y-axis. The best $r_i$'s for (b) and (c) are $(0.1, 0.45, 0.45)$ and $(0, 0.85, 0.15)$.**

In the presence of multiple datasets, an additional challenge is determining *which* datasets to move. For example, it is advantageous to move datasets with *high number of queries* accessing them. As we will show in §6, there is a two orders of magnitude variation in access counts of datasets in Facebook's production cluster.

## 2.3 Summary and Solution Overview

In summary, we illustrated a setup of WAN-connected sites: a query's dataset is spread across sites (where they were originally generated), each site stores parts of many datasets, and each dataset is accessed by multiple queries. Our goal is to *minimize average query response time* while also being mindful of WAN usage. We achieve this by, a) moving parts of datasets across sites in the *lag* between data generation and query arrival, and b) placing intermediate tasks *during* the query's execution. The intuition is to *identify "bottleneck" sites and balance the number of tasks and/or amount of data on these sites.*

Our solution Iridium is described next.

1. We solve the problem of *task placement* for a single query (given a fixed location of data) using an efficient linear formulation (§3).
2. We devise an efficient heuristic to solve the problem of *data placement* (§4), that internally uses the formulation developed in §3.
3. We incorporate a "knob" for budgeted WAN usage in our data placement (§4.4).

For data and task placement, we ignore the (abundant) CPU and memory resources at the sites.

# 3. TASK PLACEMENT

In this section, we describe how we *place tasks of a single query to minimize its response time* given a fixed input data distribution. As we described in §2.1, input tasks that load and filter the input data involve no cross-site data movement. For such input tasks, data locality [37, 58] and in-memory caching [17, 59] is sufficient for efficient execution; input tasks write their outputs (intermediate data) locally on the site they run. Other intermediate stages of the query, such as reduce and join, communicate across the different sites and require careful task placement to minimize their duration.

As these tasks are data-intensive, i.e., their durations are dominated by the times spent on communication, our objective is to minimize the duration of the intermediate data transfer. This problem can be solved exactly and efficiently for the most common communication patterns on intermediate data—reduce or join [25]. We explain our solution for these two (§3.1 and §3.2) before extending it to arbitrary DAGs (§3.3).

## 3.1 Placement of Reduce Tasks

Consider a map-reduce query across sites, where $S_i$ is the intermediate data at site $i$ ($\sum_i S_i = S$). We decide $r_i$, the fraction of reduce tasks to place on each site $i$ ($\sum_i r_i = 1$) to minimize the longest link finish time.

For formulating the problem, we assume that the reduce tasks are infinitesimally divisible. We also assume that the intermediate data on site $i$, $S_i$, is distributed across the other sites proportionally to $r_j$'s.

The main factors involved in the decision of $r_i$'s are the bandwidths of the uplinks ($U_i$) and downlinks ($D_i$) along with the size of intermediate data ($S_i$) at the sites. In the "all-to-all" shuffle communication, given the assumptions above, each site $i$ has to upload $(1 - r_i)$ fraction of its data for a total of $(1 - r_i)S_i$, and download $r_i$ fraction of data from all the other sites for a total of $r_i(S - S_i)$. Therefore, the time to upload data from site $i$ during the intermediate data transfer is $T_i^U(r_i) = (1 - r_i)S_i/U_i$, and time to download the data is $T_i^D(r_i) = r_i(S - S_i)/D_i$. Given our assumption of a congestion-free core, the problem of reduce task placement can, hence, be formulated as a linear program (LP). The LP implicitly avoids bottlenecks; e.g., if a site has a lot of data or links with low bandwidth, the placement avoids sending too much data over the narrow link.

$$\begin{aligned} \min \quad & z \\ \text{s.t.} \quad & \forall i : r_i \geq 0 \\ & \textstyle\sum_i r_i = 1 \\ & \forall i : T_i^U(r_i) \leq z, \quad T_i^D(r_i) \leq z \end{aligned}$$

The above formulation is highly efficient and invoked (repeatedly) for data placement in §4. Our implementation, described in §5, removes some of the above approximations and uses a more general (but less efficient) MIP for task placement.

## 3.2 Placement of Join Tasks

The above approach also extends to handle joins, e.g., a join of tables A and B on a common column M. There are two join implementations: *hash* and *broadcast*, automatically chosen by query optimizers [3].

If both tables are large, they are joined using a hash join which is executed as two all-to-all shuffles of both tables (as in §3.1), followed by a pair-wise join operation on data in the same key-range. To reduce the WAN usage of the pair-wise join operation, reduce tasks of the shuffle in both tables that are responsible for the same key-range are scheduled on the same site. Thus, for our purpose, we treat the whole join as a single all-to-all shuffle and use the above LP with $S_i$ as the total amount of data of tables A and B on site $i$.

If one of the tables is small, broadcast join will send the smaller table to all sites storing any data of the larger table. In the broadcast join, the amount of data sent over WAN is both small and constant (size of the small table). Placement of tasks does not impact join completion time.

## 3.3 DAGs of Tasks

While task placement for a single intermediate data transfer can be solved using an LP, doing so for general DAGs is a much more challenging problem. For example, placement of tasks for a query with two consecutive intermediate data transfers results in a non-convex optimization (unlike the linear one above).

As a result, Iridium adopts a greedy approach of applying the task placement LP independently in each stage of the query. Starting with the top-level stages, it applies the LP in topological order, which ensures that when placing tasks of a stage, the tasks of their parents have already been placed. While this approach is not optimal, in queries with many stages in sequence, the amount of data processed by each stage typically drops off quickly [14]. The intermediate data transfer at the query's beginning is the most important.

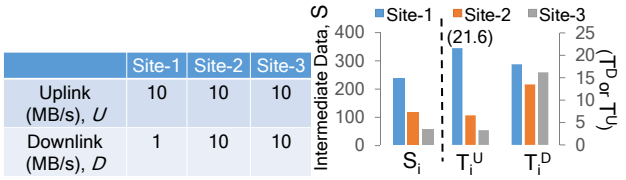
Next, we use the approach described in this section to find the best data placement, i.e., how to adjust $I_i$'s (and $S_i$'s) to minimize query response time.

# 4. DATA PLACEMENT

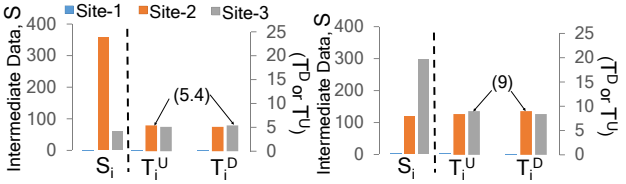
In this section, we describe our solution for distributing datasets across the sites to reduce the finishing time on the anticipated bottleneck links (motivated in §2.2.2). Changing the distribution of the input data carries over to the distribution of intermediate data since the input tasks that produce the latter write their outputs locally. Further, as the best task placement (§3) is limited by the distribution of the data ($I_i$'s or $S_i$'s), data placement is *complementary* towards reducing query response time. Again, we use "moving of data" for ease of exposition; our system just replicates additional copies that are tracked by the global manager (§2.1).

| | Site-1 | Site-2 | Site-3 |
|---|---|---|---|
| Uplink (MB/s), $U$ | 10 | 10 | 10 |
| Downlink (MB/s), $D$ | 1 | 10 | 10 |

(a) Setup of three sites.




(b) Original Input Placement




(c) Site-1 → Site-2




(d) Site-1 → Site-3

Figure 4: **Exploring destination options for data movement out of site-1 (same example as Figure 3). In the initial configuration (4b), site-1's uplink is the bottleneck. We evaluate moving 240MB from site-1 to the other two sites, site-2 (4c) and site-3 (4d). Moving to site-2 results in the lowest intermediate data transfer duration of 5.4s, from 21.6s.**

While the task placement problem alone can be solved as an LP, the joint optimization of input *and* task placement contains many local optima (see discussion in §7), making it impossible to formulate it as a linear or a quadratic program, or solve using convex optimization tools. Thus, we proceed to devise an efficient heuristic.

We first provide intuition for a single dataset and query (§4.1), generalize to multiple datasets and queries in §4.2, provide important enhancements in §4.3, and finally describe the knob to budget WAN usage in §4.4.

## 4.1 Basic Intuition

The LP in §3 provides a useful starting point. It computes the optimal query response time, $z$, but also identifies the bottleneck link, where link finish time is equal to $z$. Our heuristic rests on the intuition of moving data *out* of the bottleneck site, thereby reducing this maximum link finish time (and hence the query duration). Two natural questions arise in doing so: $(i)$ *where* to move the data? $(ii)$ *how much* data to move out? We answer both questions next.

$(i)$ Figure 4 illustrates the effect of picking different destinations to move data *to*, using the same example from §2.2.2. Recall that query's input across the three sites was $I = (240, 120, 60)$MB with selectivity $\alpha = 1$, i.e., $S = I$ (see Figure 4a). Figure 4b shows the result of running the query leaving the original data unmoved. Site-1 will be the bottleneck during the intermediate data transfer ($r_i$'s and bottleneck site derived using §3). Options for moving data from site-1 are to the other two sites, 2 and 3. Figure 4c and 4d show the potential effect of both these moves, with $r_i$'s recalculated based on the new $S_i$'s. Moving the data from site-1 to site-2

```
    class Move
        double cost
        ⟨QueryID, double⟩ timeReduction
        Site bottleneck
1:  procedure AllocateMoves(List⟨Dataset⟩ D)
2:      for each Dataset d in D do
3:          Move d.m ← FindMove(d)
4:          lag ← Σ_{q∈d.Queries} q.lag / d.Queries.Count
5:          d.value ← Σ_{q∈d.Queries} d.m.timeReduction[q ] /
    lag
6:          d.score ← d.value / d.m.cost
7:      for each Dataset d in D.SortedByDesc(d.score) do
8:          if d.m.bottleneck.canMove() then
9:              execute d.m
```

Pseudocode 1: Iridium **Solution. The function takes the set of all datasets $D$ and assigns each site to move out part of a dataset. For simplicity, we do not present the calculation of the *destination* site to be moved.**

is the best move as the transfer duration is 5.4s compared to 9s in Figure 4d. While not applicable in this example, we ignore moves to those sites that *increase* the transfer duration.

(*ii*) On the second question of *how much* data to move out of the bottleneck site, the above example ended up moving all the data from site-1 to site-2 because such a move happened to result in the lowest duration for intermediate transfer. In our system, we use a "what-if" calculation to assess moving data out of the bottleneck site in increments of $\delta$ (say, 10MB), i.e., move 10MB, 20MB and so forth. We pick the increment of $\delta$ that provides the smallest transfer duration.[4]

Iridium's heuristic can be summarized as follows: *iteratively* identify bottlenecked sites and move data out of them to reduce the duration of intermediate data transfer (considering all potential destinations and increments of $\delta$).[5] We extend this simple intuition to a workload of *multiple* competing datasets in §4.2. We then enhance the solution in §4.3 with techniques to predict future query arrivals, minimize contention between data movement and query traffic, etc.

## 4.2 Prioritizing between Multiple Datasets

In prioritizing between datasets, Iridium seeks to identify and move the high-valued datasets. High-valued 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.

In the example above, the "value" of moving data out of site-1 to site-2 is $(21.6 - 5.4) = 16.2$s. The relative value of moving a dataset also increases if its queries are to arrive sooner, i.e., smaller *lag*. The "cost" of the move is the amount of data that needs to be moved over the WAN to improve the query, 240MB in the example. We select the move that achieves the highest "score", i.e., (value/cost).

Pseudocode 1 lists the two main steps in our heuristic. We defer estimation of future query arrivals to §4.3.

**Step a)**, lines $2-5$, first calls FindMove() that returns the *Move* object that contains the bottlenecked site, data to be moved out, and the reduction in query durations ($\geq 0$) due to the move. The query durations and bottleneck site are calculated using §3. If there are multiple bottlenecked sites, we arbitrarily pick one.

The value of the move is calculated using the reduction in query durations and query lags (described shortly). The "score" of the proposed move is $\frac{value}{cost}$.

**Step b)**, lines $6-8$, processes datasets in descending order of their score. To prevent new low-value dataset moves from slowing down ongoing high-value moves, we allocate a site's uplink and downlink to *only one* dataset at a time (justified in §6.4). The *canMove* function performs this check.

**Query Lag:** For two datasets A and B that arrived at 1:00, all else being equal, if dataset A's queries arrive at 1:05 and dataset B's queries at 1:10, we should prefer to move dataset A at 1:00 since we can move B starting at 1:05. This is analogical to the "earliest-deadline-first" scheduling approach.

We adopt this approach by calculating the query *lag* for a dataset, i.e., time between dataset availability and the query's arrival, as the average of the lag of all the queries accessing the dataset. The value for the dataset is then multiplied by $\frac{1}{lag}$. Thus, the smaller the average lag, the higher its value and urgency in moving the dataset. In §6.4, we also evaluate other metrics of arrival lag (e.g., median, earliest, latest) and see that using the average works best.

The AllocateMoves() function in Pseudocode 1 is invoked every time a new dataset or a query arrives or when a scheduled data movement completes. Arrival of queries aborts any incomplete movements of their data.

## 4.3 Enhancements

We now present two important enhancements.

### *Estimating Query Arrivals*

For *recurring* workloads ("batch" streams [60] or "cron" jobs), we estimate arrivals based on past executions.

However, this is hard to do for ad hoc queries. For a dataset, we care about the number of queries that will access it and their arrival times, i.e., lag. To that end, we make the following simple assumption that works well in our evaluation (§6.4). We assume the dataset's future query arrivals will repeat as per the query arrivals so far (from the time the dataset was generated). For instance, if the dataset was generated at time $t$ and two queries arrived at times $(t+2)$ and $(t+3)$, we will assume at $(t+3)$ that two more queries would arrive at

[4]This approach brings the transfer duration down to, at least, the second-most bottlenecked link. Thus, this avoids a "loop" of the same data being moved *back* to the site in the next step. In general, fixing $\delta$ avoids jointly calculating new values for $r_i$'s and $S_i$'s.

[5]If none of the moves out of the bottleneck site help, we consider analogical moves of data *into* the bottleneck site.

times $(t+3)+2$ and $(t+3)+3$. We use these arrival lags in Pseudocode 1. In general, at the end of $n$ queries, it would assume $n$ *more* queries will arrive.

Such a scheme *under-estimates* the number of accesses initially. But the estimate grows quickly, and it estimates correctly at the "half-way" number of accesses. Beyond this half-way point, it *over-estimates* future accesses, which could lead to unnecessary data movement. In practice however, for even moderate number of accesses, data movements mostly stabilize by the time the over-estimation starts, thus limiting any fallout.

### *Queries/Data Contention*

In an online system, our heuristic makes its data movement decisions even as (tasks of) queries are executing on the sites. This results in contention between the network flows of the tasks and data movement. When we schedule a data movement out of a site, we measure the impact, i.e, increase in duration of the running tasks and the corresponding queries. An increase in task duration need not necessarily increase the query's duration because the latter is bound by its slowest task. In measuring the increase in duration, we assume fair sharing of uplink/downlink bandwidth among all the flows.

We evaluate if the slowdown of the other running queries due to contention is worth the speedup of the queries whose data will be moved. Data is moved *only* if the trade-off is beneficial, otherwise we ignore this dataset and move to the next dataset in the ordered list (not included in Pseudocode 1 for simplicity).

## 4.4 WAN Usage Budget

WAN usage across sites is an important operational cost ($/byte) for datacenters [53, 54]. Also, third-party services running on AWS or Azure across regions pay based on WAN usage [5]. As described so far, Iridium does not account for WAN usage. If there is enough lag for a query's arrival or if there are not many competing datasets, it will be free to move the datasets *even if* they only marginally improve the response times.

To avoid wasting WAN bandwidths on such movements, we incorporate a *budget* for WAN usage that forces our heuristic to balance between the speedup of queries and WAN costs. The challenge in setting the budget is to ensure it is neither *too low* (and moving very few datasets leading to limited query speedups), nor *too high* (and causing wasted usage of WAN links).

As a baseline for our budget, we start with the WAN consumption, $W$, of a (data and task placement) scheme that *optimizes* for WAN usage [53, 54]. We set the budget for our heuristic to be $(B \cdot W)$, $B \geq 1$. $B = 1$ implies a strict WAN usage budget, while higher values of $B$ trade it for faster query response.

How do we calculate and keep track of the WAN budget over time? We adopt the following greedy approach. We start with a counter $M = 0$. Every time a new dataset arrives, we compute the $W$ for this dataset and increment $M += W \cdot B$. Every time we execute a data move, we decrement $M$ by amount of data moved. If $M = 0$, we do not execute any new data moves.

Setting the knob $B$ is a matter of policy but our results indeed highlight the presence of a "sweet spot". With $B = 1.3$, Iridium's gains are nearly 90% of the gains with an unconstrained budget. In fact, even with WAN usage equal to a WAN-usage optimal policy, i.e., $B = 1$, its query speedups are 2× more, §6.5.

# 5. SYSTEM IMPLEMENTATION

Our prototype implementation of Iridium is on top of the Apache Spark [59] framework. The source code is available here: https://github.com/Microsoft-MNR/GDA

To implement our task placement, we override the default scheduler of Spark and plug-in our module that internally uses the Gurobi solver [7]. We would like to note that we solve the task placement problem as a Mixed Integer Problem (in contrast to the simple LP in §3). The MIP uses the exact amount of intermediate data read by every task from each site, thus handles any intermediate communication pattern, and outputs a specific site to place each task. Even though the MIP is less efficient, it is invoked only once per job for task placement. The LP is an efficient approximation and used in the many iterations of data placement decisions.

We incorporate our data placement heuristic inside the Hadoop Distributed File System (HDFS) [8] that Spark uses as its data store. We do not disable the default replication mechanism in HDFS, and all our data movements hence only create *additional* copies of the data, thus leaving data durability unaffected. As storage at the sites is abundant, we believe this to be an acceptable design.

User queries and analytics jobs are submitted through an uniform interface provided by the Spark manager. Because Iridium is built upon Spark, it can leverage two Spark extensions, Spark SQL and Spark Streaming [60], for parsing SQL queries and running streaming jobs.

We use simple techniques to estimate the bandwidths at sites and intermediate data sizes (or $\alpha$) of queries.
*Estimating Bandwidths:* Our experiments at the eight EC2 sites (described in §6) indicate that the available bandwidth is relatively stable in the granularity of minutes. Thus, we use a simple test that checks the available bandwidth every few minutes. However, we also get continuous fine-grained measurements by piggybacking measurements on the throughputs of the data movement and task flows. Given our focus on recurring queries, such piggybacking provides a sufficiently rich source of bandwidth values that automatically considers non-Iridium traffic. We plug these in to our heuristics.
*Estimating Intermediate Data Sizes:* Unlike input sizes of queries, intermediate data sizes are not known upfront. Again, we leverage the recurring nature of our workloads to infer the intermediate data sizes. Repeated queries, even on newer parts of the same dataset, often produce similar filtering of data. We are able to es-

timate the ratio of intermediate to input data of queries ($\alpha$) with an accuracy of 92% in our experiments.

# 6. EVALUATION

We evaluate Iridium using a geo-distributed EC2 deployment as well as trace-driven simulations. The highlights of our evaluation are as follows.

1. Iridium speeds up workloads from Conviva, Bing Edge, TPC-DS [12] and Big-data benchmarks [4] by 64% to 92% (3× to 19×) when deployed across eight EC2 regions in five continents.

2. Iridium saves WAN bandwidth usage by 15% to 64%. Even with usage equal to a WAN-usage optimal policy, its query speedups are 2× more.

## 6.1 Methodology

We begin by describing our evaluation setup.

**EC2 Deployment:** We deploy Iridium across eight EC2 regions in Tokyo, Singapore, Sydney, Frankfurt, Ireland, Sao Paulo, Virginia (US) and California (US) [2]. We use c3.4xlarge instances in each region [1] and the WAN connecting them is a more constrained resource than the local CPU/memory/disk. In addition, we also mimic a larger geo-distributed setup of 30 sites within one region.

**Workloads:** We tested our system using four analytics workloads from Conviva, Bing Edge, TPC-DS and AMPLab Big-data benchmark (§6.2). These workloads consist of a mix of Spark [59] and Hive [49] queries.

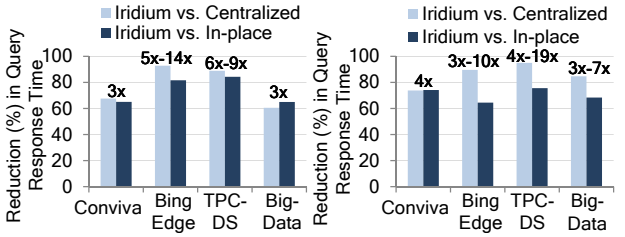
**Trace-driven Simulator:** We evaluate Iridium over longer durations using a trace-driven simulator of production traces (one month, 350K jobs) from Facebook's Hadoop cluster. The simulator is faithful to the trace in its query arrival times (lag), task input/output sizes, and dataset properties of locations, generation times and access patterns. We mimic 150 sites in our simulator; slots within sites are unconstrained.

We predict future query arrivals (lags) using the technique in §4.3, and evaluate its accuracy in §6.4.

**Baselines:** We compare Iridium to two baselines: (*i*) leave data "in-place" and use stock Spark's scheduling and storage policies, and (*ii*) "centralized" aggregation of data at a main DC whose in-bandwidth is generously and conservatively set to be practically-infinite, i.e., it is rarely the bottleneck during the aggregation. We again use stock Spark's scheduling/storage within the main DC that they are optimized well for.

**Metric:** Our primary metric is *reduction (%) in average response time* of queries. For a query whose response times with the baseline and Iridium are $b$ and $x$, we calculate $100 \times \frac{(b-x)}{b}$; maximum is 100%. We also quote $b/x$, the factor of reduction in response time when appropriate. In §6.5, we measure WAN usage.

We describe our EC2 deployment results in §6.2 and simulation results in §6.3. We assess Iridium's design decisions in §6.4 and the WAN usage knob in §6.5.




(a) Inter-Region (b) 30 sites

Figure 5: **EC2 Results across eight worldwide regions (a): Tokyo, Singapore, Sydney, Frankfurt, Ireland, Sao Paulo, Virginia (US) and California (US). The figure on the right (b) is on a larger 30-site setup. Iridium is 3× − 19× better compared to the two baselines.**

## 6.2 EC2 Deployment

We used four workloads to evaluate Iridium on EC2.

**(1) Conviva Video Analytics:** We use queries from Conviva, a video delivery and monitoring company. Data from clients (e.g., the edge/CDN serving them, their ISP and network characteristics) are analyzed to modify the parameters of video sessions (e.g., codec, buffer sizes) to improve performance (re-buffering ratio [21]). The queries contain a mixture of aggregation ("reduce") and table-joins. Every query has 160GB input.

**(2) Microsoft Bing Edge Dashboard:** Microsoft's Bing service maintains a running dashboard of its edge servers deployed worldwide. The queries aggregate data from 40,000 raw counters filtered by a range of location (lat/long, city), user-id, etc. values to produce average and $90^{\text{th}}$ percentiles. This is also an example of a *streaming* query that we execute using Spark Streaming's "mini-batch" model [60] in every time period.

**(3) TPC-DS Benchmark:** The TPC-DS benchmark is a set of decision support queries [12] based on those used by retail product suppliers such as Amazon. These OLAP queries examine large volumes of data (215GB) each, and are characterized by a mixture of compute and disk/network load, the latter of relevance to us.

**(4) AMPLab Big-Data:** The big-data benchmark [4] is derived from workloads and queries from [45] with identical schema of the data. The suite contain a mix of Hive and Spark queries. The queries contain simple scans, aggregations, joins, and UDF's.

In our inter-region EC2 experiment, we use bandwidths naturally available to the instances on the sites. In our 30-site setup, we vary the bandwidths between 100Mb/s to 2Gb/s (Linux Traffic Control [10]), hoping to mimic the heterogeneous bandwidths across edge clusters and DCs available for analytics frameworks.

Figure 5a plots our average gains for the four workloads across eight-region EC2 regions. Gains compared to the in-place and centralized baselines range from 64% to 92% (3× to 14×). Singapore, Tokyo and Oregon (US) had 2.5× higher bandwidth than Virgina (US) and Frankfurt, and 5× higher bandwidth than Sydney,

| | Iridium vs. In-place | Iridium vs. Centralized |
|---|---|---|
| Core | 26% | 32% |
| Core + Query Lag | 41% | 46% |
| **Core + Query Lag + Contention** | 59% | 74% |
| Core + Contention | 45% | 53% |

Table 2: **Progression of Iridium's gains as additional features of considering query lag and contention between query/data movements are added to the basic heuristic. (Facebook workload)**

Sao Paulo and Ireland. Iridium automatically adjusts its data and task placement away from these sites to avoid unduly congesting their links during query execution. Our gains are similar (but a bit higher) with our 30-site setup at $3\times - 19\times$. Note that since the Bing Edge query is a streaming operation executed as a series of "mini-batch" queries, we report the gains per batch.

Gains compared to the centralized baseline is higher than with the in-place baseline for all but the Conviva workload. This is because the intermediate data size is closer to the input size ($\alpha$) in the Conviva workload, which makes central aggregation less hurtful. In addition, the Conviva and Big-data queries also have a more intensive map stage (during which we do just as good as our baselines via data locality) that relatively brings down the opportunity and gains for Iridium. Finally, the Conviva and Bing Edge queries have lesser skew in their map outputs which limits the value of Iridium's task placement compared to the in-place baseline.

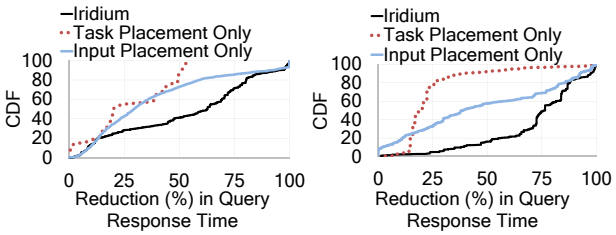
**Overheads:** Micro-benchmarks show that our data placement iterations are efficient to implement, finishing in under 100ms for up to 200 sites; the LP used (§3.1) calculates fraction of tasks ($r_i$), thus the number of tasks do not matter in its calculation.

## 6.3 Trace-driven Simulation

In this section, we present simulation results based on the production trace from Facebook's Hadoop cluster. We use bandwidth values 100Mb/s to 2Gb/s in our simulator, similar to §6.2, but we also present a result when the bandwidths are higher with lower heterogeneity later ($\{10, 50\}$ Gb/s), indicative of only large DCs.

Compared to baselines of leaving data in place and central aggregation, Iridium improves average response time by 59% and 74%, respectively.

Table 2 shows the progression in gains as we start with the basic data placement heuristic, and incrementally add the usage of query lag in the score for datasets, and consideration of contention between query/data traffic. The basic heuristic itself starts with fairly moderate gains, but jumps by a factor of 1.5× when we consider the lag, and a further 1.5× with modeling contentions with query traffic. The final result is also significantly better than adding either one of the features. These results seek to underline the use of query lag in differentiating between datasets, and avoiding contention with running queries.




(a) In-place baseline (b) Centralized baseline

Figure 6: **CDF of Iridium's gains with the Facebook workload. We also compare our two techniques—task placement and data placement—standalone.**

Also, keeping data in place is a more stringent baseline than the common approach of central aggregation. This is because reduction in data in intermediate stages ($\alpha < 1$) of many queries results in unnecessary delays using the centralized approach. Iridium automatically makes the right call on placing data and tasks depending on the intermediate data and other factors.

**Distribution of Gains:** Figure 6 plots the distribution of gains. Gains compared to the in-place baseline are more uniform (and lower) compared to the centralized baseline where the gains are steep overall. The curves converge at the third quartile at $\sim 80\%$. Importantly, Iridium does not make any query worse. This is because its decisions on data and task placements automatically subsume the corresponding decisions by the baselines.

We also compare the effect of our two techniques—task and data placements—*standalone*. With the former, we leave data in-place, and with the latter, we use Spark's stock intermediate task placement. With the in-place baseline, using Iridium's task placement alone moderately outperforms using Iridium's data placement alone (outside of the final quartile); Figure 6a. However, compared to the centralized baseline, it reverses and significantly under-performs. This is roughly intuitive given that smart data movements mainly assuage the problem in moving all the data (centralized baseline) while smart task placement mainly solves the congestion issues due to naive task placement (in-place baseline).

**Bucketing the Gains:** Given the above variation in gains, *which are the queries that gain more*? Figure 7 buckets queries by different characteristics and plots the average gains in each bucket (along with the fraction of queries). We use the stricter in-place baseline here.

(*i*) *Intermediate/Input Ratio* ($\alpha$): Among queries with same input size, Iridium's data movement prioritizes those with higher intermediate data as they provide higher value. While Iridium's task placement works for both, queries with higher intermediate data present more opportunity for smart task placement. Consequently, gains are skewed towards queries with high $\alpha$ (Figure 7a). However, even queries with $\alpha < 1$ see significant gains because the times spent on input and intermediate stages is also dictated by the vastly different processing speeds of these stages (in-memory vs. WAN). Thus, optimizing intermediate stages is still valuable.

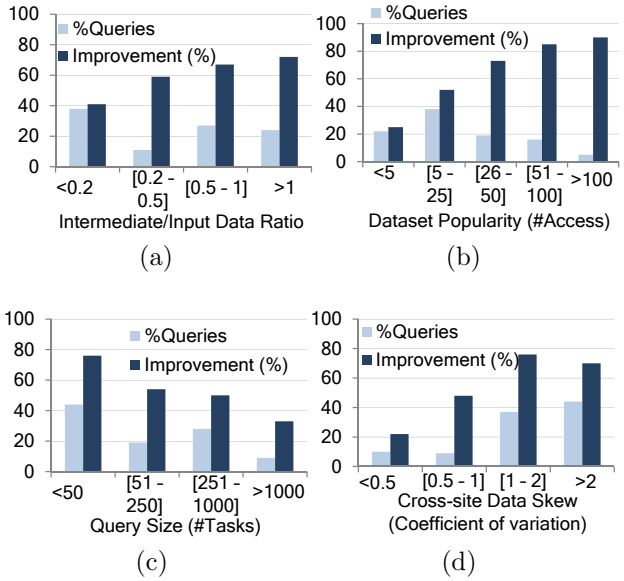



Figure 7: **Iridium's improvements (and % queries), bucketed by various query characteristics: (a) intermediate/input data ratio, (b) dataset access count, (c) query size (# tasks), and (d) cross-site skew in intermediate data.**

(*ii*) *Dataset popularity*: As described in §4.2, Iridium prefers moving datasets that are accessed by many queries because the costs of moving is amortized better; our cost and value in Pseudocode 1 aptly capture this aspect. This is consequently reflected in the gains (Figure 7b). Queries of oft-accessed datasets see $4\times$ the gains compared to queries whose whose input datasets are less popular; the trend is also strictly monotonic.

(*iii*) *Query Size (in number of tasks)*: Somewhat surprisingly, our solution favors smaller queries (Figure 7c). We believe this is due to two reasons. First, their datasets happen to be oft-accessed as these queries are often interactive and exploratory, resulting in repeat accesses. Second, moving their datasets not only has a lower cost but also higher value than those with many tasks. This is due to the property of wave-based gains of parallel jobs [17]. Speeding up a wave of simultaneous parallel tasks, regardless of the *number of* parallel tasks (wave-width), results in the *same gain* in query response time.

(*iv*) *Cross-site skew of intermediate data*: Iridium's task placement, and to a lesser extent, data placement, is most effective when there is substantial skew across sites in the intermediate data of a query. This is a trend we observe in Figure 7d too where we bucket queries by the coefficient-of-variation of their intermediate data sizes across sites; smaller values of COV represent less skew. However, the lower COV buckets have $< 20\%$ of queries.

**Bandwidth of $\{10, 50\}$ Gb/s:** By making the bandwidths relatively higher and less heterogeneous, there is lesser overlap of flows (flows finish faster due to higher bandwidths), and the baseline task placement is better off. While Iridium's gains continue to be substantial, an interesting aspect is that their values compared to the centralized baseline drops down to 56%, which is also roughly where the gains with the in-place baseline land. Higher and lesser heterogeneous bandwidths *slightly* soften the inefficiencies of the centralized baseline's data aggregation and in-place baseline's task placement, respectively.

| Lag Metric | Vs. In-place | Vs. Centralized |
|---|---|---|
| Iridium **(Avg.)** | 59% | 74% |
| Iridium (Median) | 56% | 75% |
| Iridium (Earliest) | 38% | 42% |
| Iridium (Latest) | 24% | 40% |
| **Oracle** | 66% | 81% |

Table 3: **Effectiveness of estimating query lag. Iridium's approach of using the average lag outperforms other options and crucially, has gains of $\sim 90\%$ of an oracle that has full knowledge about query arrivals.**

## 6.4 Iridium's Design Decisions

In this section, we evaluate the design decisions made in Iridium's data placement heuristic in §4.

**Query Lag:** In calculating the score for datasets to rank them, we use the inverse of *average* lag of the queries accessing them (§4.2). We now compare using alternate metrics of query lag than the average—median, earliest and latest (Table 3). Using the median lag results in nearly similar results indicating that the arrival of queries is not long-tailed, but using the earliest or latest results in substantially poorer performance due to significant under- and over-estimation. They make Iridium either too aggressive and mis-prioritize datasets or too lethargic in its movement of datasets.

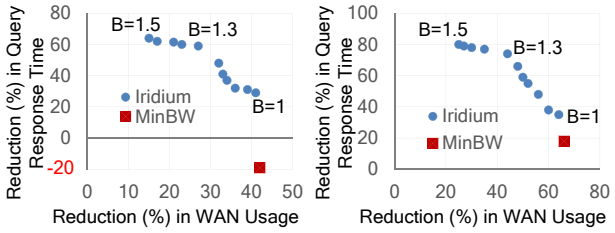
An encouraging aspect is the closeness of the gains using the average lag to an "oracle" ($\sim 90\%$) that knows all the query arrivals and uses the exact query lags in its decisions. Our simple predictor that assumes that the arrival of queries henceforth will mimic the query arrivals thus far, in fact, works well in practice.

**Dataset at a time:** The final design decision we evaluate is to move only one dataset at a time out of a site (step b) in §4.2). We compare it to two alternatives. The first natural alternative is at the other extreme of having no cap. All the data movement flows sharing the link obtain their fair share of bandwidth. The other alternative is to allow many flows but allocate bandwidths between them in proportion to the "value" they are estimated to obtain.

Iridium outperforms both these alternatives whose gains compared to the baselines are 41% and 55% for the first, and 48% and 61% for the second alternative.

## 6.5 WAN Bandwidth Usage

Finally, we evaluate the functioning of Iridium's knob to budget WAN bandwidth usage (§4.4); our results so far were with the WAN budget knob $B = 1.3$. Figure 8 plots the results as $B$ varies. "MinBW" is the scheme that optimizes for WAN bandwidth usage proposed in [53, 54]. While Iridium's bandwidth gains are lower than those of MinBW, they are still appreciable.




(a) In-place baseline (b) Centralized baseline

Figure 8: **WAN Bandwidth Usage knob, $B$. MinBW is the scheme that optimizes for WAN bandwidth usage. Even with same WAN usage as MinBW ($B = 1$), Iridium's gains in query response time are significantly higher. MinBW slows down queries against the in-place baseline.**

With just a small value of $B = 1.3$ (i.e., 30% higher WAN usage than MinBW), Iridium's query speedups of 59% and 74% are $\sim$ 90% of those without any WAN usage budget (64% and 80%). This shows that Iridium smartly uses the bandwidth budget to balance gains in bandwidth usage with gains in query response time. This also shows that over long periods, arrival of "high-valued" and "low-valued" datasets overlap sufficiently in the workload. This is an important characteristic for our greedy budgeted scheme to function well.

Even for $B = 1$ (i.e., same WAN bandwidth usage as MinBW), Iridium's gains in query response time are appreciable. Crucially, MinBW results in an *increase* in query response time (negative gains) with the in-place baseline. While MinBW's query gains are positive compared to the centralized baseline, Iridium query gains are handily 2× better *for the same WAN usage*.

# 7. DISCUSSION AND LIMITATIONS

We now discuss some limitations of our solutions.

**Compute and Storage Constraints:** Our work did not consider limitations in compute and storage at the sites as we believed that to be reasonable for datacenters. However, as geo-distributed analytics moves to "edge" clusters, it is conceivable that compute and storage are also limitations. Under such a scenario, compute and storage capacity have to be comprehensively considered for task and data placement. A simple approach could do the following. To the task placement formulation in §3, we add the following constraint on every site $i$: $r_i \cdot D \leq C_i$, where $D$ is the compute required by the stage and $C_i$ is the capacity. In our data placement heuristic, when a site is running out of storage capacity, we will simply not consider moves into that site.

**WAN Topologies:** How do our heuristics change when the core network connecting the sites is not congestion-free? One could model pair-wise connectivity between sites, say $B_{ij}$ as the available bandwidth from site $i$ to site $j$. To optimize task placement, we formulate an LP to determine the $r_i$'s, similar to §3.1. Given a distribution of intermediate data $S_i$, let $T_{ij}(r_j)$ be the time it takes to send data from site $i$ to site $j$; $T_{ij}(r_j) = S_i r_j / B_{ij}$. The LP to compute $z$, the minimal shuffle duration, and the corresponding $r_i$'s is as follows: $\min z$, s.t. $\sum_i r_i = 1$ and $\forall i \neq j : T_{ij}(r_j) \leq z$. Redesigning the data placement heuristic, however, is more challenging and requires careful consideration.

**Local minima and greedy approach:** As we alluded to in §4, the joint problem of data and task placement is non-convex. This means that the greedy approach adopted by our heuristic may get stuck in local minima. Overcoming them requires exploring potential options that could increase query response time temporarily before bringing it down. While our gains are significant even with the greedy solution, depending on the lag and bandwidth available for moving data, one could conceive a larger move with much more significant gain. Extending our heuristic to overcome local minima is part of future work.

# 8. RELATED WORK

*1) Distributed databases:* While big-data frameworks currently operate only within a single cluster, work on distributed databases has been a popular topic [22, 27]; see surveys in [40, 44]. Google Spanner [28] is an instance of a distributed database deployed at scale. Our problem is simpler because we do not have to deal with concurrency and deadlocks; data processing systems are typically *append-only*. This gives us more freedom to move data across sites. JetStream [46], a stream processing system for OLAP cubes, uses data aggregation and adaptive filtering to support data analytics. However, unlike Iridium, JetStream does not support arbitrary SQL queries and does not optimize data and task placement. Recent work [53, 54] optimize for WAN bandwidth usage across sites. As we showed, this can lead to poor query response times. In contrast, Iridium optimizes for query response time *and* WAN usage using a budget (§4.4 and §6.5).

*2) Reducing data analytics responses:* There is a large body of work on improving query response time in data parallel systems [20, 23, 29]. These systems improve data locality of input tasks and fairness [37, 58], and minimize outliers in task execution [16, 19, 61]. While these systems optimize task placement, they do not consider network contentions (which matter less within a DC), and they do not move data around to relieve potential network bottlenecks [33]. Further, Iridium is complementary to approximation techniques [15, 18].

*3) Optimizing communication patterns:* Flow schedulers like D3 [55], PDQ [35], DeTail [62], and D2TCP [51] aim to improve flow completion times or guarantee deadlines. However, they operate inside a single DC and do not consider complex communication patterns. Orchestra [25], Varys [26], and Baraat [30] are network flow schedulers that optimize for completion time of *coflows*, i.e., collections of flows. However, because the endpoints of the coflows are fixed (e.g., source and des-

tination specified by location of input data and tasks), these cannot schedule around network bottlenecks.

*4) Scheduling on the WAN:* There has been much work on optimizing WAN transfers including tuning ECMP weights [32] and adapting allocations across pre-established tunnels [31, 39]. Also, both Google [38] and Microsoft [36] recently published details on their production WAN networks. All this work improves the efficiency of the WAN by scheduling network flows *inside the WAN*. Instead, we optimize end-to-end application performance, i.e., reducing response time of big-data jobs, by placing data and tasks to explicitly reduce load on congested WAN links. Other works optimize data placement to improve WAN latencies and utilization [41, 50]. Iridium optimizes much more complex communication patterns, such as shuffles, that require coordination of a large number of flows across many sites. Moreover, most of the above could be used to improve the individual WAN transfers in Iridium.

# 9. CONCLUSION

Cloud organizations are deploying datacenters and edge clusters worldwide. The services deployed at these sites, first-party and third-party, produce large quantities of data continuously. Results from analyzing these geo-distributed data is used by real-time systems and data analysts. We develop Iridium, a system that focuses on minimizing response times of geo-distributed analytics queries. Our techniques focus on data transfers in these queries that happen across the WAN. By carefully considering the WAN's heterogeneous link bandwidths in the placement of data as well as tasks of queries, we improve query response times in workloads derived from analytics clusters of Bing Edge, Facebook and Conviva by $3\times - 19\times$. However, we would like to point out that our approach is greedy in nature (not optimal) and we offer only a partial solution to optimizing complex DAGs of tasks, both of which we aim to improve.

## Acknowledgments

We would like to thank Kaifei Chen, Radhika Mittal and Shivaram Venkataraman for their feedback on the draft. We also appreciate the comments from our shepherd Mohammad Alizadeh and the anonymous reviewers. This work was partially supported by NSF grants CNS-1302041, CNS-1330308 and CNS-1345249.

## References

[1] Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/.
[2] Amazon Web Services. http://aws.amazon.com/about-aws/global-infrastructure/.
[3] Apache Calcite. http://optiq.incubator.apache.org/.
[4] Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.
[5] EC2 Pricing. http://aws.amazon.com/ec2/pricing/.
[6] Google Datacenter Locations. http://www.google.com/about/datacenters/inside/locations/.
[7] Gurobi Optimization. http://www.gurobi.com/.
[8] Hadoop Distributed File System. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[9] How Map and Reduce operations are actually carried out. http://wiki.apache.org/hadoop/HadoopMapReduce.
[10] Linux Traffic Control. http://lartc.org/manpages/tc.txt.
[11] Microsoft Datacenters. http://www.microsoft.com/en-us/server-cloud/cloud-os/global-datacenters.aspx.
[12] TPC Decision Support Benchmark. http://www.tpc.org/tpcds/.
[13] Measuring Internet Congestion: A preliminary report. https://ipp.mit.edu/sites/default/files/documents/Congestion-handout-final.pdf, 2014.
[14] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-Parallel Computing. In *USENIX NSDI*, 2012.
[15] S. Agarwal, B. Mozafari, A. Panda, M. H., S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, 2013.
[16] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.
[17] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.
[18] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. USENIX NSDI, 2014.
[19] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX OSDI*, 2010.
[20] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[21] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *ACM SIGCOMM*, 2013.
[22] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 1981.
[23] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.
[24] M. Calder, X. Fan, Z. Hu, E. Katz-Bassett, J. Heidemann, and R. Govindan. Mapping the Expansion of Google's Serving Infrastructure. In *ACM IMC*, 2013.
[25] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM*, 2011.
[26] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *ACM SIGCOMM*, 2013.

[27] W. W. Chu and P. Hurley. Optimal Query Processing for Distributed Database Systems. *IEEE Transactions on Computers*, 1982.

[28] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *USENIX OSDI*, 2012.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[30] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *ACM SIGCOMM*, 2014.

[31] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: Multipath Adaptive Traffic Engineering. *Computer Networks*, 2002.

[32] B. Fortz, J. Rexford, and M. Thorup. Traffic Engineering with Traditional IP Routing Protocols. *Communications Magazine, IEEE*, 2002.

[33] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM*, 2014.

[34] A. a. Gupta *et al.* Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. In *VLDB*, 2014.

[35] C.-Y. Hong, M. Caesar, and B. Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM*, 2012.

[36] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *ACM SIGCOMM*, 2013.

[37] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[38] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. *ACM SIGCOMM*, 2013.

[39] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. *ACM SIGCOMM*, 2005.

[40] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computer Survey*, 2000.

[41] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter Bulk Transfers with Netstitcher. *ACM SIGCOMM*, 2011.

[42] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. GUPT: Privacy Preserving Data Analysis Made Easy. In *ACM SIGMOD*, 2012.

[43] E. Nygren, R. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-Performance Internet Applications. In *ACM SIGOPS OSR*, 2010.

[44] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. 2011.

[45] A. Pavlo, E. Paulson, A. Rasin, d. Abadi, Daniel an dDeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *ACM SIGMOD*, 2009.

[46] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *USENIX NSDI*, 2014.

[47] R. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain. Overlay Networks: An Akamai Perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*, 2014.

[48] S. Sundaresan, W. d. Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet Performance: A View From the Gateway. In *ACM SIGCOMM*, 2011.

[49] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse using Hadoop. In *ICDE*, 2010.

[50] S. Traverso, K. Huguenin, I. Trestian, V. Erramilli, N. Laoutaris, and K. Papagiannaki. TailGate: Handling Long-tail Content with a Little Help from Friends. In *WWW*, 2012.

[51] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM*, 2012.

[52] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. Franklin, and I. Stoica. The Power of Choice in Data-Aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[53] A. Vulimiri, C. Curino, B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *USENIX NSDI*, 2015.

[54] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. WANalytics: Analytics for a Geo-distributed Data-intensive World. In *CIDR*, 2015.

[55] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. *ACM SIGCOMM*, 2011.

[56] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. Madhyastha. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *ACM SOSP*, 2013.

[57] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *ACM SOSP*, 2009.

[58] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *ACM EuroSys*, 2010.

[59] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[60] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.

[61] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *USENIX OSDI*, 2008.

[62] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *ACM SIGCOMM*, 2012.