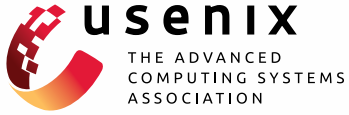


Global Analytics in the Face of Bandwidth and Regulatory Constraints

Ashish Vulimiri, ***University of Illinois at Urbana-Champaign;*** **Carlo Curino,** ***Microsoft;***
P. Brighten Godfrey, ***University of Illinois at Urbana-Champaign;***
Thomas Jungblut, ***Microsoft;*** **Jitu Padhye and George Varghese,** ***Microsoft Research***

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/vulimiri

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Global analytics in the face of bandwidth and regulatory constraints

Ashish Vulimiri[u] *Carlo Curino*[m] *Brighten Godfrey*[u]
Thomas Jungblut[m] *Jitu Padhye*[m] *George Varghese*[m]
[u]: UIUC <vulimir1,pbg>@illinois.edu [m]: Microsoft <ccurino,thjungbl,padhye,varghese>@microsoft.com

Abstract

Global-scale organizations produce large volumes of data *across* geographically distributed data centers. Querying and analyzing such data as a whole introduces new research issues at the intersection of networks and databases. Today systems that compute SQL analytics over geographically distributed data operate by pulling all data to a central location. This is problematic at large data scales due to expensive transoceanic links, and may be rendered impossible by emerging regulatory constraints. The new problem of Wide-Area Big Data (WABD) consists in orchestrating query execution across data centers to minimize bandwidth while respecting regulatory constaints. WABD combines classical query planning with novel network-centric mechanisms designed for a wide-area setting such as pseudo-distributed execution, joint query optimization, and deltas on cached subquery results. Our prototype, Geode, builds upon Hive and uses 250× less bandwidth than centralized analytics in a Microsoft production workload and up to 360× less on popular analytics benchmarks including TPC-CH and Berkeley Big Data. Geode supports all SQL operators, including Joins, across global data.

1 Introduction

Organizations operating at global scale need to analyze vast amounts of data. The data is stored in multiple data centers around the world because of stringent latency requirements for user-facing applications. The volume of data collected while logging user interactions, monitoring compute infrastructures, and tracking business-critical functions is approaching petabytes a day. These new global databases *across* data centers — as opposed to traditional parallel databases [4] *within* a data center — introduce a new set of research issues at the intersection of databases and networks, combining the traditional problems of databases (e.g., query planning, replication) with the challenges of wide area networks (e.g., bandwidth limits, multiple sovereign domains [17]). Recent work in global databases illustrates this research trend, from Spanner [12] (global consistency) to Mesa [22] (replication for fault tolerance) to JetStream [34] (analytics for data structured as OLAP cubes).

In these applications, besides the many reads and writes generated by user transactions or logging, data is frequently accessed to extract insight, using ad-hoc and recurrent analytical queries. Facebook [40, 44], Twitter [29], Yahoo! [14] and LinkedIn [3] report operating pipelines that process tens or hundreds of TBs of data each day. Microsoft operates several large-scale applications at similar scales, including infrastructures for collecting telemetry information for user-facing applications, and a debugging application that queries error reports from millions of Windows devices [19]. Many of these queries require Joins and cannot be supported using the OLAP cube abstraction of [34].

To the best of our knowledge, companies today perform analytics across data centers by transferring the data to a central data center where it is processed with standard single-cluster technologies such as relational data warehouses or Hadoop-based stacks. However, for large modern applications, the centralized approach transfers significant data volumes. For example, an analytics service backing a well known Microsoft application ingests over 100 TB/day from multiple data centers into a centralized analytics stack. The total Internet bandwidth crossing international borders in 2013 was 100 Tbps (Figure 1). Even if all this capacity were dedicated to analytics applications and utilized with 100% efficiency, it could support only a few thousand such applications.

Moreover, while application demands are growing from 100s of terabytes towards petabytes per day, network capacity growth has been decelerating. The 32% capacity growth rate in 2013-2014 was the lowest in the past decade (Figure 1). A key reason is the expense of adding network capacity: for instance, a new submarine cable connecting South America and Europe is expected

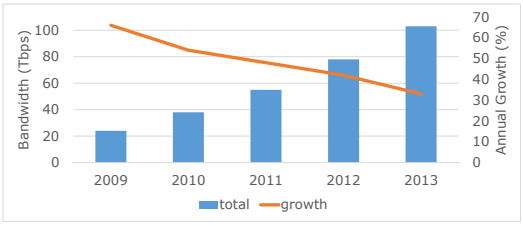


Figure 1: Sum of capacities of Internet links crossing international borders [24]

to cost $185 million. This scarcity of wide-area network bandwidth can drive applications to discard valuable data; the problem will only worsen as applications scale up and out. Our analysis of bandwidth trends is consistent with [34, 28, 21].

A second emerging difficulty is that privacy concerns (for example in the EU [16]) may result in more regulatory constraints on data movement. However, while local governments may start to impose constraints on raw data storage [35], we speculate that *derived* information, such as aggregates, models, and reports (which are critical for business intelligence but have less dramatic privacy implications) may still be allowed to cross geographical boundaries.

Thus our central thesis is: rising global data and scarce trans-oceanic bandwidth, coupled with regulatory concerns, will cause an inflection point in which centralizing analytics (the norm today) will become inefficient and/or infeasible. We consider the problem of providing wide area analytics while minimizing bandwidth over geo-distributed data structured as SQL tables, a dominant paradigm. We support the entire array of SQL operators on global data including Joins, providing exact answers. We refer to such analytics as *Wide-Area Big Data* or WABD.

Our paper proposes a solution to the WABD problem. We support SQL analytics on geo-distributed data, providing automated handling of fault-tolerance requirements and using replicated data to improve performance whenever possible. We assume that resources within a single data center (such as CPU and storage) are relatively cheap compared to cross-data center bandwidth. We target the batch analytics paradigm dominant in large organizations today [44, 29, 3], where the cost of supporting analytics execution is the primary metric of interest. Our optimizations are all targeted at reducing bandwidth cost; we currently make no attempt at minimizing analytics execution latency.

Our techniques revisit the classical database problem of query planning while adding a networking twist. In particular, while classical query planning optimizes query processing ("What's the best join order?") to minimize *computation*, in WABD we optimize the execution strategy to minimize *bandwidth* and respect sovereignity. For example, one of our techniques is based on caching previous answers to subqueries at data centers and only sending the difference, reminiscent of differential file transfer mechanisms [42, 41]. Similarly, our query optimization approach relies on the fact that analytical queries are repeated: thus simple measurement techniques common in networking can be used to measure data transfer costs across data centers. This contrasts with classical database techniques using histograms (designed to handle arbitrary queries) that are well known to be inaccurate in the face of Joins and User Defined Functions [30].

We make four main contributions:

1. Optimizer: We jointly optimize query execution plans and data replication to minimize bandwidth cost. Our solution combines a classical centralized SQL query planner (a customized version of Apache Calcite) with an integer program for handling geo-distribution.

2. Pseudo-distributed measurement: We develop a technique that *modifies* query execution to collect accurate data transfer measurements, potentially increasing the amount of (cheap) computation within individual data centers, but never worsening (expensive) cross-data center bandwidth.

3. Subquery Deltas: We take advantage of the cheap storage and computation within individual data centers to aggressively cache all intermediate results, using them to eliminate data transfer redundancy using deltas.

4. Demonstrated Gains: Our prototype, Geode, is built on top of the popular Hive [39] analytics framework. Geode achieves a 250× reduction in data transfer over the centralized approach in a standard Microsoft production workload, and up to a 360× improvement in a range of scenarios across several standard benchmarks, including TPC-CH [9] and Berkeley Big Data [6].

2 Approach Overview

We start by discussing an example inspired by the Berkeley Big-Data Benchmark [6] and use it to motivate our architecture.

Running example

We have a database storing batch-computed page metadata and a log of user visits to web pages, including information about the revenue generated by each visit:

```
ClickLog(sourceIP,destURL,visitDate,adRevenue,...)
PageInfo(pageURL,pageSize,pageRank,...)
```

Pages are replicated at multiple edge data centers, and users are served the closest available copy of a page. Visits are logged to the data center the user is served from, so that the ClickLog table is naturally partitioned across edge data centers. The PageInfo table is stored centrally

in a master data center where it is updated periodically by an internal batch job.

Now consider an analytical query reporting statistics for users (identified by their IP address) generating at least $100 in ad revenue.

```
Q: SELECT sourceIP, sum(adRevenue), avg(pageRank)
   FROM ClickLog cl JOIN PageInfo pi
     ON cl.destURL = pi.pageURL
   WHERE pi.pageCategory = 'Entertainment'
   GROUP BY sourceIP
   HAVING sum(adRevenue) >= 100
```

Supporting this query via the centralized approach requires retrieving all updates made to the ClickLog table to a central data center where the analytical query is computed. This means that the daily network bandwidth requirement is proportional to the total size of the updates to the database. Assuming 1B users, 6 pages visited per user, 200 bytes per ClickLog row, this is roughly (1B * 6 * 200) bytes = 1.2 TB per day.

By contrast, Geode provides an equivalent location-independent [33] query interface over distributed data. The analyst submits the query Q unmodified to Geode, which then automatically partitions the query and orchestrates distributed execution. Geode constructs the distributed plan in two stages:

1. Choose join order and strategies. Geode first creates a physical execution plan for the logical query Q, explicitly specifying the order in which tables are joined and the choice of distributed join algorithm for processing each join (broadcast join, semijoin etc. — see §5). In this simple query, there is only one choice: the choice of algorithm for processing the join between the ClickLog and PageInfo tables.

To make these choices we use Calcite++, a customized version we built of the Apache Calcite centralized SQL query planner. Calcite has built-in rules that use simple table statistics to optimize join ordering for a given query; Calcite++ extends Calcite to also make it identify the choice of distributed join algorithm for each join. We describe Calcite++'s design in detail in §4.1.

When Calcite++ is run on `Q`, it outputs an annotation `JOINHINT(strategy = right_broadcast)`, indicating that the join should be executed by broadcasting the (much smaller) PageInfo table to each data center holding a partition of the (larger) ClickLog table, then computing a local join at each of these data centers.

Assuming an organization that operates across three edge data centers, the physical plan Q_{opt} translates directly into the DAG in Figure 2. Each circle is a task: a SQL query operating on some set of inputs. Edges show data dependencies. Tasks can read base data partitions (e.g. q_1) and/or outputs from other tasks (e.g. q_5) as input. All the inputs a task needs must either already be present or be copied over to any data center where it is

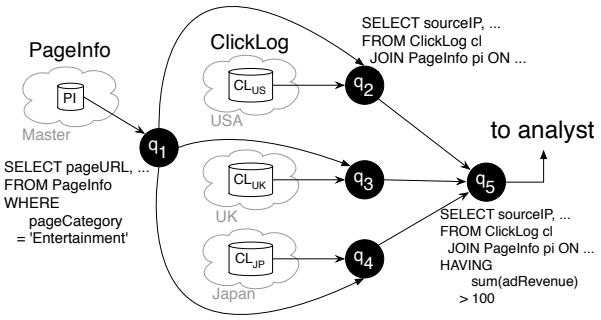


Figure 2: DAG corresponding to Q_{opt}

scheduled. While we do not consider them in this simple example, regulatory restrictions may prohibit some partitions from being copied to certain data centers, thus constraining task scheduling.

2. Schedule tasks. Geode now needs to assign tasks to data centers, taking into account task input dependencies and base data regulatory constraints.

Geode can maintain multiple copies of base data partitions, for performance and/or for fault tolerance, and potentially schedule multiple copies of tasks operating on different partition copies. For instance, it could maintain a synchronized copy of the PageInfo table at every data center and create multiple copies of task q_1 at each data center. The choice of replication strategy is controlled by a *workload optimizer*, at a much longer time scale than one individual query's execution (typically replication policy changes occur on a weekly basis or even slower). The optimizer chooses the replication policy taking various factors into account (§4).

At runtime, Geode schedules tasks for individual queries on data centers by solving an integer linear program (ILP) with variables $x_{td} = 1$ iff a copy of task t is scheduled on data center d. The constraints on the ILP specify the input dependencies for each task and the availability and regulatory constraints on copies of partitions at each data center. The ILP tries to minimize the total cost of the data transfers between tasks in the DAG if measurements of inter-task transfer volumes are available (see §4). The ILP described here is a simpler version of the more nuanced multi-query optimizer in §4.3.

Assume an initial setup where data are not replicated. Then the natural strategy is to schedule q_1 on the master data center holding the PageInfo table, push q_2, q_3, q_4 down to the edge data centers holding the ClickLog partition they operate on, and co-locate q_5 with one of q_2, q_3 or q_4. If query `Q` is submitted once a day, 1B users visit 100M distinct pages each day, 100K users have an ad revenue larger than $100, each tuple output by q_1 is 20 bytes long and by q_2, q_3, q_4 is 12 bytes long, distributed execution will transfer $3 * 100M * 20 + (2/3) * 1B * 12 + 100K * 12 = 14$ GB of data each day, compared to 1.2 TB per day for the centralized approach.

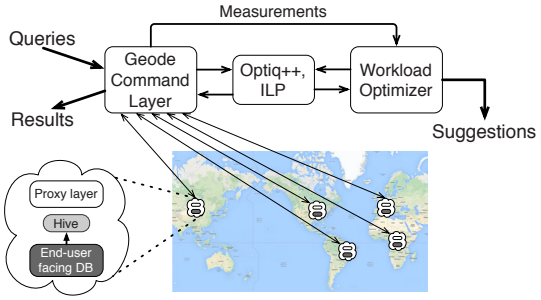


Figure 3: Geode architecture

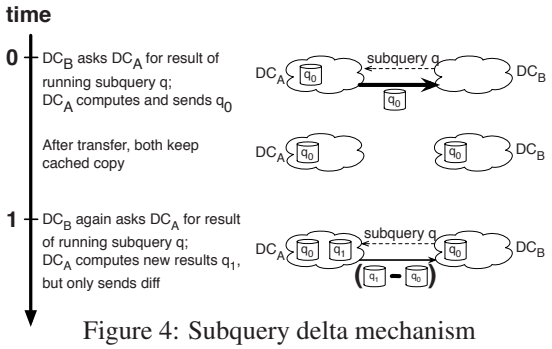


Figure 4: Subquery delta mechanism

While these numbers suggest a clear win for the distributed approach, if `Q` is submitted once every 10 minutes centralization is more efficient. The workload optimizer evaluates this tradeoff across the entire analytical workload and continuously adapts, reverting to centralized execution if needed. Analytical queries can be much more complex than `Q`; for example, the CH benchmark (§6) contains a query with 8 joins (involving 9 different tables) for which the degrees of freedom (join order, join strategy, replication) are much higher.

Architecture

Our example motivates the architecture in Figure 3.

Geode processes analytics over data split across multiple data centers, constantly updated by interactions with a set of end-users. End-user interactions are handled externally to our system, and we do not model them explicitly. We assume that at each data center all data has been extracted out into a standard single-data-center analytics stack, such as Hive or a relational database. Our current implementation is Hive-based.

The core of our system is a central command layer. The command layer receives SQL analytical queries, partitions them to create a distributed query execution plan, executes this plan (which involves running queries against individual data centers and coordinating data transfers between them), and collates the final output. At each data center the command layer interacts with a thin proxy deployed over the local analytics stack. The proxy layer facilitates data transfers between data centers and manages a local cache of intermediate query results used for the data transfer optimization in §3.

A workload optimizer periodically obtains measurements from the command layer to estimate if changing the query plan or the data replication strategy would improve overall performance. These measurements are collected using our pseudo-distributed execution technique (§4.2), which may entail rewriting the analytical queries. The optimizer never initiates changes directly, but instead makes suggestions to an administrator.

We next discuss: an optimization we implement to reduce data transfers (§3); the workload optimizer including pseudo-distributed execution (§4); and the design and implementation of our Geode prototype (§5).

3 Subquery deltas: Reducing data transfer

We first turn our attention to optimizing the mechanics of data movement. The unique setting we consider, in which each node is a full data center with virtually limitless CPU and storage, but connectivity among nodes is costly/limited, lends itself to a novel optimization for eliminating redundancy.

Consider a query computing a running average over the revenue produced by the most revenue generating IPs over the past 24 hours. If the query is run once an hour, more than 95% of the data transfer will be wasted because every hour unoptimized Geode would recompute the query from scratch, transferring all the historical data even though only the last hour of data has changed.

We leverage storage and computation in each data center to aggressively cache intermediate results. Figure 4 details the mechanism. After data center DC_B retrieves results for a query from data center DC_A, both the source and the destination store the results in a local cache tagged with the query's signature. The next time DC_B needs to retrieve results for the same query from DC_A, DC_A recomputes the query again, but instead of sending the results afresh it computes a diff (delta) between the new and old results and sends the diff over instead.

Note that DC_A still needs to recompute the results for Q the second time around. Caching does not reduce intra-data-center computation. Its purpose is solely to reduce data transfer between data centers.

We cache results for individual sub-queries run against each data center, not just for the final overall results returned to the analyst. This means that caching helps not only when the analyst submits the same query repeatedly, but also when two different queries use results from the same common sub-query. E.g. in the TPC-CH benchmark that we test in §6, 6 out of the 22 analytical queries that come with the benchmark perform the same join operation, and optimizing this one join alone allows caching to reduce data transfer by about 3.5×.

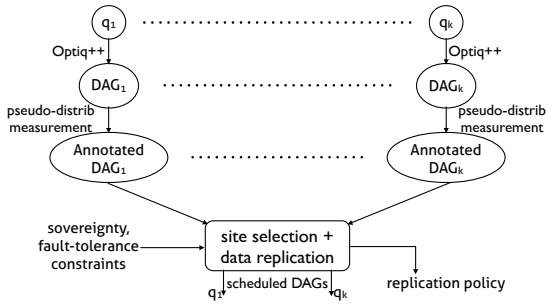


Figure 5: Optimizer architecture

4 Workload Optimizer

Geode targets analytics with a small, slowly evolving core of recurring queries. This matches our experience with production workloads at Microsoft, and is consistent with reports from other organizations [44, 3, 29]. The workload optimizer tailors policy to maximize the performance of this core workload, jointly optimizing:

1. **Query plan**: the execution plan for each query, deciding e.g. join order and the execution mechanism (broadcast join, semijoin etc.).
2. **Site selection**: which data center is used to execute each sub-task for each query.
3. **Data replication**: where each piece of the database is replicated for performance/fault-tolerance.

The problem we face is akin to distributed database query planning. In that context, it is common [27] to employ a two-step solution: (1) find the best centralized plan (using standard database query planning), and (2) decompose the centralized plan into a distributed one, by means of heuristics (often employing dynamic programming). Our approach is similar in spirit, but is faced with substantially different constraints and opportunities arising from the WABD setting:

1. *Data Birth:* We can replicate data partitions to other data centers, but have no control over where data is generated originally – base data are naturally "born" in specific data centers dictated by external considerations, such as the latency observed by end-users.
2. *Sovereignty:* We must deal with the possibility of sovereignty constraints, which can limit where data can be replicated (e.g. German data may not be allowed to leave German data centers).
3. *Fixed Queries:* We can optimize the system for a small, approximately static core workload, which means we do not have to use general-purpose approximate statistics (e.g., histograms) that yield crude execution cost estimates for one-time queries. We can instead collect narrow, precise measures for a fixed core of queries.

These features drive us to the architecture in Figure 5. Briefly, we start by identifying the optimal centralized plan for each query in the core workload using the *Calcite++* query planner (§4.1). We then collect precise measures of the data transfers during each step of distributed execution for these plans using *pseudo-distributed measurement* (§4.2). We finally combine all these measurements with user-specified data sovereignty and fault tolerance requirements to jointly solve the *site selection* and *data replication* problems (§4.3).

4.1 Centralized query planning: Calcite++

Apache Calcite is a centralized SQL query planner currently being used or evaluated by several projects, including Hive [39]. Calcite takes as input a SQL query parse tree along with basic statistics on each table, and produces a modified, optimized parse tree. Calcite++ extends Calcite to add awareness of geo-distributed execution.

Calcite optimizes queries using simple statistics such as the number of rows in each table, the average row size in each table, and an approximate count of the number of distinct values in each column of each table. All these statistics can be computed very efficiently in a distributed manner. Calcite uses these statistics along with some uniformity assumptions to optimize join order. In Calcite++ we leave the join order optimization unchanged but introduce new rules to compare the cost of various (distributed) join algorithms, passing in as additional input the number of partitions of each table. The output of the optimization is an optimized join order annotated with the lowest cost execution strategy for each join — e.g., in our running example (§2) Calcite++ chooses a *broadcast join*, broadcasting PageInfo to all ClickLog locations where local partial joins are then computed.

While both Calcite and (therefore) Calcite++ currently use only simple, rough statistics to generate estimates, in all the queries we tested in our experimental evaluation (§6.1), we found that at large multi-terabyte scales the costs of the distributed join strategies under consideration were orders of magnitude apart, so that imprecision in the generated cost estimates was inconsequential. (The centralized plan generated by Calcite++ always matched the one we arrived at by manual optimization.) Moreover, Calcite is currently under active development — for instance, the next phase of work on Calcite will add histograms on each column.

4.2 Pseudo-distributed execution

The crude table statistics Calcite++ employs suffice to compare high-level implementation choices, but for making site selection and data replication decisions we require much better accuracy in estimating the data transfer cost of each step in the distributed execution plan.

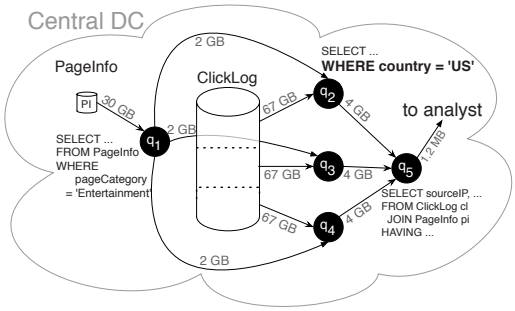


Figure 6: Pseudo-distributed execution of query `Q` (§2) in a centralized deployment. Cf. Figure 2

Traditional database cardinality estimation techniques can be very inaccurate at generating absolute cost estimates, especially in the face of joins and user-defined functions [30]. The sheer volume of data, heterogeneity network topologies and bandwidth costs, and cross-query optimizations such as the sub-query delta mechanism we propose, further complicate statistics estimation.

Instead, we *measure* data transfers when executing the plan in the currently deployed configuration (which could be a centralized deployment or an already running Geode deployment), modifying query execution when necessary to make it possible to collect the estimates we need. As an example consider query Q (Figure 2) from §2, currently running in a centralized configuration (i.e. the entire database is replicated centrally). To estimate the cost of running in a distributed fashion Geode simulates a virtual topology in which each base data partition is in a separate data center. This is accomplished by rewriting queries to push down `WHERE country = X` clauses constraining each of q_2, q_3, q_4 to operate on the right subset of the data[1]. Figure 6 depicts this process. The artificial decomposition allows us to inspect intermediate data sizes and identify the data transfer volume along each edge of the DAG in Figure 2.

This technique, which we call *pseudo-distributed execution*, is both fully general, capable of rewriting arbitrary SQL queries to simulate any given data partitioning configuration, and highly precise, since it directly executes rewritten queries and measures output and input sizes instead of attempting any estimation. We employ the technique whenever we need to evaluate an alternative deployment scenario, such as when considering moving from an initial centralized deployment to a distributed Geode deployment; or when considering adding or decommissioning data centers in a distributed deployment in response to changes in the load pattern.

The latency overhead added by pseudo-distribution is minimal and easily mitigated, as we discuss in §6.2.

[1]Every partitioned table in Geode has a user-specified/system-generated field identifying the partition each row belongs to (§5).

Trading precision for overhead: While Geode's implementation of pseudo-distributed execution measures the costs of most SQL queries accurately, including those with joins and nested queries, we deliberately introduce a limited degree of imprecision when evaluating aggregate functions to reduce measurement overhead.

Specifically, we ignore the possibility of partial aggregation within data centers. As an example, suppose 10 data partitions are all replicated to one data center, and consider a `SUM` query operating on all this data. Retrieving one total `SUM` over all 10 partitions is sufficient, but Geode always simulates a fully distributed topology with each partition in a separate data center, thus retrieving separate `SUM`s from each partition and overestimating the data transfer cost. To measure the true cost of function evaluation with partial aggregation we would need an exponential number of pseudo-distributed executions, one for each possible way of assigning or replicating partitions across data centers; one execution suffices for the upper bound we use instead.

We found this was not an issue in any of the workloads (production or benchmark) we tested. The majority of the data transfers during query execution arise when joining tables, and data transfer during the final aggregation phase after the joins have been processed is comparatively much smaller in volume. In all six of our workloads, the data transfer for tasks involved in computing combinable aggregates was $< 4\%$ of the total distributed execution cost.

4.3 Site Selection and Data Replication

After identifying the logical plan (DAG of tasks) for each query (§4.1) and measuring the data transfer along each edge (§4.2), we are left with two sets of decisions to make: *site selection*, specifying which data centers tasks should be run on and which copies of the data they should access; and *data replication*, specifying which data centers each base data partition should be replicated to (for performance and/or fault tolerance). This should be done while respecting disaster recovery requirements and sovereignty constraints.

We formulate an integer linear program (Figure 7a) that jointly solves both problems to minimize total bandwidth cost. The ILP is built from two sets of binary variables, x_{pd} indicating whether partition p is replicated to data center d; and y_{gde} identifying the (source, destination) data center pairs (d, e) to which each edge g in the considered DAGs is assigned[2]. Constraints specify sovereignty and fault-tolerance requirements.

While the ILP provides very high quality solutions, its complexity limits the scale at which it can be applied—as

[2]We schedule *edges* instead of *nodes* because (1) replication turns out to be easier to handle in an edge-based formulation, and (2) the node-based formulation would have a *quadratic* (not linear) objective.

Inputs:
D = number of data centers
P = number of data partitions
$G = \langle V, E \rangle$ = union of DAGs for all core workload queries
b_g = number of bytes of data transferred along each edge $g \in E$ (from pseudo-distrib. exec.)
update_rate$_p$ = rate at which partition p is updated by OLTP workload (bytes per OLAP run)
link_cost$_{de}$ = cost ($/byte) of link connecting DCs d and e
f_p = minimum number of copies of partition p that the system *has* to make for fault-tolerance
$R \subseteq P \times D = \{(p,d) \mid$ partition p cannot be copied to data center d due to regulatory constraints$\}$
Variables:
All variables are binary integers (= 0 or 1)
$x_{pd} = 1$ iff partition p is replicated to DC d
$y_{gde} = 1$ iff edge g in the DAGs is assigned source data center d and destination data center e
$z_{td} = 1$ iff a copy of task t in the DAGs is assigned to data center d
Solution:

$$\text{replCost} = \sum_{p=1}^{P} \sum_{d=1}^{D} \text{update\_rate}_p * x_{pd} * \text{link\_cost}_{homeDC(p),d}$$

$$\text{execCost} = \sum_{g \in E} \sum_{d=1}^{D} \sum_{e=1}^{D} y_{gde} * b_g * \text{link\_cost}_{de}$$

$$\underset{X,Y}{\text{minimize}} \qquad \text{replCost} + \text{execCost}$$

subject to

$$\forall (p,d) \in R : x_{pd} = 0$$

$$\forall p : \sum_d x_{pd} \geq f_p$$

$$\forall d \forall e \forall g \mid src(g) \text{ is a partition} : y_{gde} \leq x_{src(g),d}$$

$$\forall d \forall e \forall g \mid src(g) \text{ is a task} : y_{gde} \leq z_{src(g),d}$$

$$\forall n \forall e \forall g \mid dst(g) = n : z_{ne} = \sum_d y_{gde}$$

$$\forall n \forall p \forall d \mid n \text{ reads from partition } p \wedge (p,d) \in R : z_{nd} = 0$$

$$\forall n : \sum_d z_{nd} \geq 1$$

(a) Integer Linear Program jointly solving both problems

```
for all DAG G ∈ workload do
    for all task t ∈ toposort(G) do
        for all data center d ∈ legal_choices(t) do
            cost(d) = total cost of copying all of t's inputs to d
        if lowest cost is zero then
            assign copies of t to every data center with cost = 0
        else
            assign t to one data center with lowest cost
for all (p,d) ∉ R do
    check if replicating p to d would further reduce costs
translate decisions so far into values for x, y, z variables in ILP above
solve simplified ILP with pinned values
```

(b) Greedy heuristic

Figure 7: Site selection + Data replication: Two solvers

we show in §6.3. For example, if we bound our optimization time to 1h (recall that this is an offline, workload-wide process), the ILP can only support up to 10 data centers for workloads of the size we test in our experiments. This is barely sufficient for today's applications. But the rapid growth in infrastructure [21] and application scales that is the norm today may soon outstrip the capabilities of the ILP. As future-proofing, we propose an alternative *greedy heuristic* (Figure 7b) that has much better scalability properties.

The heuristic approach first uses a natural greedy task placement to solve the site selection problem in isolation: identify the set of data centers to which each task can be assigned based on sovereignty constraints over its input data, and greedily pick the data center to which copying all the input data needed by the task would have the lowest cost. We are still left with the NP-hard problem of finding the best replication strategy subject to fault-tolerance and sovereignty requirements. This is tackled by a (much simpler) ILP in isolation from site selection.

The greedy heuristic scales much better than the ILP, identifying solutions in less than a minute even at the 100 data center scale. However, in some cases this can come at the cost of identifying significantly sub-optimal solutions. We evaluate the tradeoff between processing time and solution quality in §6.3.

Limitation: At this point the formulation does not attempt to account for gains due to cross-query caching (the benefit due to the mechanism in §3 when different queries share common sub-operations). The precise effect of cross-query caching is hard to quantify, since it can fluctuate significantly with variations in the order and relative frequency with which analytical queries are run. Similar to the discussion of partially aggregatable functions in the previous subsection, we would need an exponential number of pseudo-distributed measurements to estimate the benefit from caching in every possible combination of execution plans for different queries.

However, we do account for *intra*-query caching — the benefit from caching within individual queries (when the same query is run repeatedly). We always collect pseudo-distributed measurements with a warm cache and report stable long-term measurements. This means all data transfer estimates used by the ILP already account for the long-term effect of intra-query caching.

5 Geode: Command-layer interface

Geode presents a logically centralized view over data partitioned and/or replicated across Hive instances in multiple data centers. Users submit queries in the SQL-like Hive Query Language (HQL) to the command layer, which parses and partitions queries to create a distributed execution plan as in §2. We discuss the basic interface Geode presents to analysts in this section.

Describing schema and placement

Geode manages a database consisting of one or more tables. Each table is either *partitioned* across several data centers, or *replicated* at one or more data centers. Partitioned tables must have a specified *partition column* which identifies which partition any row belongs to. The partition column is used to, among other things, support pseudo-distributed execution and to automatically detect and optimize joins on co-partitioned tables. Partitioned tables can either be *value-partitioned*, meaning each distinct value of the partition column denotes a separate partition, or *range-partitioned* on an integer column, meaning each partition corresponds to a specified range of values of the partition column.

Analysts inform Geode about table schema and placement by submitting `CREATE TABLE` statements annotated with placement type and information — we omit the details of the syntax.

Supported queries

We support most standard analytics features in Hive 0.11 (the latest stable version when we started this project): nested queries, inner-, outer- and semi-joins, and user-defined aggregate functions; although we do not support some of Hive's more unusual feature-set, such as compound data structures and sampling queries [39]. Our architecture is not tied to Hive and can be easily adapted to work with other SQL backends instead.

Joins. By default, Geode passes user-submitted queries through Calcite++ (§4.1) first to optimize join order and execution strategy. However, users can enforce a manual override by explicitly annotating joins with a `JOINHINT(strategy = _)` instruction.

Geode currently supports three classes of distributed join execution strategies: (1) *co-located joins*, which can be computed without any cross-data center data movement either because both tables are co-partitioned or because one table is replicated at all of the other table's data centers; (2) left or right *broadcast joins*, in which one table is broadcast to each of the other table's data centers, where separate local joins are then computed; and (3) left or right *semi-joins*, in which the set of distinct join keys from one table are broadcast and used to identify and retrieve matches from the other table. We are currently exploring adding other strategies, such as hash-joins with a special partitioning-aware hash function [38].

Nested queries. Nested queries are processed recursively[3]. The system pushes down nested queries completely when they can be handled entirely locally, without inter-data-center communication; in this case the results of the nested query are stored partitioned across data centers. For all other queries, the final output is merged and stored locally as a temporary table at the master data center (hosting the Geode command layer). The results of nested queries are transferred lazily to other data centers, as and when needed to execute outer queries.

User-defined functions. We support Hive's pluggable interface for both simple user-defined functions (UDFs), which operate on a single row at a time, and for user-defined aggregate functions (UDAFs). Existing user code can run unmodified.

For UDAFs, note that the need is to allow users to write functions that process data distributed over multiple machines. Hive's solution is to provide a MapReduce-like interface in which users define (1) a *combine* function that locally aggregates all data at each machine, and (2) a *reduce* function that merges all the *combine*d output to compute the final answer. By default we use this interface in an expanded hierarchy to compute UDAFs by applying *combine* a second time in between steps (1) and (2) above, using it on the *combine*d output from each machine to aggregate all data within one data center before passing it on to *reduce*. Users can set a flag to disable this expansion, in which case we fall back to copying all the input to one data center and running the code as a traditional Hive UDAF.

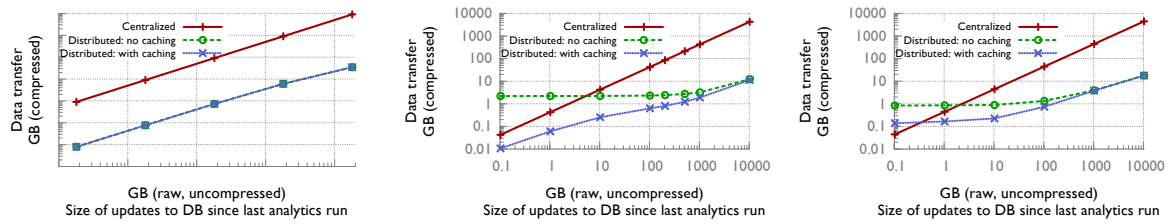
Extensibility

Geode is designed to support arbitrary application domains; as such the core of the system does not include optimizations for specific kinds of queries. However, the system is an extensible substrate on top of which users can easily implement narrow optimizations targeted at their needs. To demonstrate the flexibility of our system we implemented two function-specific optimizations: an exact algorithm for top-k queries [7], originally proposed in a CDN analytics setting and recently used by JetStream [34]; and an approximate percentile algorithm from the sensor networks literature [36]. We evaluate the benefit from these optimizations in §6.4.

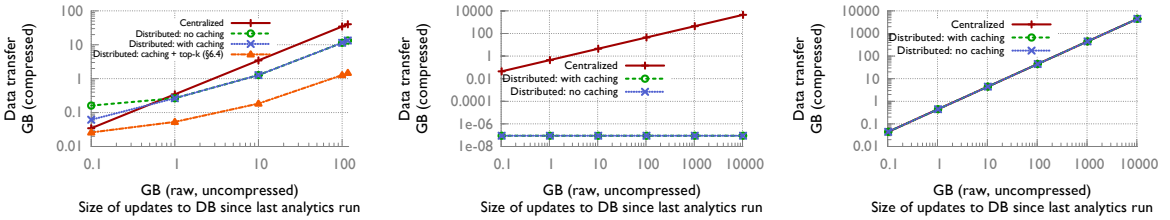
6 Experimental Evaluation

We now investigate the following questions experimentally: How much of a bandwidth savings does our system actually yield on real workloads at multi-terabyte scales (§6.1)? What is the runtime overhead of collecting the (pseudo-distributed) measurements needed by our optimizer (§6.2)? What is the tradeoff between solution quality and processing time in the optimizer (§6.3)? Can implementing narrow application-specific optimizations yield significant further bandwidth cost reduction (§6.4)?

[3]This simple strategy is sufficient because Hive does not support correlated subqueries.



(a) Microsoft production workload (b) TPC-CH (c) BigBench



(d) Berkeley big-data (e) YCSB-aggr (f) YCSB-getall

Figure 8: End-to-end evaluation of all six workloads

6.1 Large-scale evaluation

We ran experiments measuring Geode performance on a range of workloads, on two Geode deployments: a distributed deployment across three data centers in the US, Europe and Asia, and a large centralized cluster on which we simulated a multi-data center setup. Specifically, we ran experiments on both deployments up to the 25 GB scale (and validated that the results were identical), but used the centralized cluster exclusively for all experiments on a Microsoft production workload and all experiments larger than 25 GB on other workloads. This was because running experiments at the multi-terabyte scale we evaluate would have otherwise cost tens of thousands of dollars in bandwidth in a fully distributed deployment.

We tested six workloads.

Microsoft production workload: This use case consists of a monitoring infrastructure collecting tens of TBs of service health/telemetry data daily at geographically distributed data centers. The data are continuously replicated to a central location and analyzed using Hive. The bulk of the load comes from a few tens of canned queries run every day producing aggregate reports on service utilization and infrastructure health.

TPC-CH: The TPC-CH benchmark [9] by Cole et al. models the database for a large-scale product retailer such as Amazon, and is a joint OLTP + OLAP benchmark constructed by combining the well-known TPC-C OLTP benchmark and the TPC-H OLAP benchmark.

BigBench: BigBench [18] is a recently proposed benchmark for big-data systems modeling a large scale product retailer that sells items online and in-store, collecting various information from customers (including reviews and click logs) in the process. Analytics consists of a core of Hive queries along with some non-relational machine learning operations that further process the relational output. We do not implement the non-relational component explicitly, but model it as a black box analyst interacting with Geode — Geode's task is to compute the results the non-relational black box needs as input.

Big-data: The big-data benchmark [6], developed by the AMPLab at UC Berkeley, models a database generated from HTTP server logs. The analytical queries in this benchmark are parametric: each has a single parameter that can be adjusted to tune the volume of data transfer that would be required to process it. In our experiments we set the normalized value ($\in [0,1]$) of each parameter to 0.5, to make each query require median data transfer.

YCSB-aggr, YCSB-getall: We defined these two very simple benchmarks to demonstrate the best- and worst-case scenarios for our system, respectively. Both benchmarks operate using the YCSB [11] database and OLTP workload, configured with database schema:

```
table(key, field1, fleld2)
```

The OLTP workload is constituted by transactions that add a single row with `field1` a randomly chosen digit in the range $[0,9]$ and `field2` a random 64-bit integer. The difference between the two benchmarks is solely in their analytical workload.

YCSB-aggr's analytical workload consists of the single query `SELECT field1, AVG(field2) FROM Table GROUP BY field1`. Since there are only 10 distinct values of `field1`, Geode achieves significant aggregation, requiring only 10 rows (partial sum and count for each distinct `field1`) from each data center.

YCSB-getall's analytical workload is a single query asking for every row in the table (`SELECT * FROM Table`). Here no WABD solution can do better than centralized analytics.

We evaluate all six workloads by measuring the data transfer needed for both centralized and distributed execution for varying volumes of changes to the base data in between runs of the analytical workload. Our workload optimizer consistently picks among the best of the centralized and distributed solutions at each point, so that Geode's performance would be represented by the min of all the graphs in each plot. We omit the min line to avoid crowding the figures.

Figure 8 shows results for all six workloads. (We are required to obfuscate the scale of the axes of Figure 8a due to the proprietary nature of the underlying data.) We note a few key observations.

In general, the centralized approach performs relatively better when update rates are low, actually outperforming distributed execution at very low rates in 2 of the 6 workloads. This is because low volumes mean frequent analytics running on mostly unchanged data. Distributed execution performs better at higher update rates.

Caching significantly improves performance at low update rates in TPC-CH, BigBench and Berkeley bigdata: for instance, performance with caching always outperforms centralized execution in the TPC-CH benchmark, while performance without caching is worse for volumes $<$ 6 GB per OLAP run. However, at high update rates, caching is ineffective since redundancy in the query answers is minimal. Caching does not help in the YCSB workloads because small changes to the base data end up changing analytics results completely in both benchmarks, and in the Microsoft production workload because every query tagged all output rows with a query execution timestamp, which interacts poorly with the row-based approach we use to compute deltas (more sophisticated diffs can overcome this limitation).

At the largest scales we tested, distributed execution outperformed the centralized approach by $150-360\times$ in four of our six workloads (YCSB-aggr, Microsoft prod., TPC-CH, and BigBench). The improvement was only $3\times$ in the Big-Data with normal distributed execution, but when we implemented the special optimization for top-k queries [7] we discussed in §5, the improvement went up to $27\times$ — we discuss details in §6.4. Finally, YCSB-getall was deliberately designed so that distributed execution could not outperform the centralized approach, and we find that this is indeed the case.

6.2 Optimizer: Runtime overhead

The pseudo-distributed execution method we use to collect data transfer measurements can slow down query execution (although it never worsens bandwidth cost, as we discussed in §4.2). We measured the added overhead for all the queries we tested in §6.1.

In all our workloads, we found that the latency overhead compared to normal distributed Geode was contained in the $<$**20%** range. Given the scale-out nature of the Hive backend, this is easily compensated for by increasing parallelism. Note also that this overhead is only occasionally felt, since in our architecture the optimizer operates on a much slower timescale than normal query execution. E.g. if queries are run once a day and the optimizer runs once a month, pseudo-distributed execution only affects $1/30 = 3.3\%$ of the query runs.

Further, this overhead could be reduced in many cases by using separate lightweight statistics-gathering queries to estimate transfers, instead of full-fledged pseudo-distributed runs. For instance, for the query in Figure 2, we could instead run a `SELECT sum( len(pageURL) + len (pageRank)) FROM PageInfo WHERE ...` query to estimate the size of the join, and then determine the size of the final output by executing the query using a normal (as opposed to a pseudo-distributed) join.

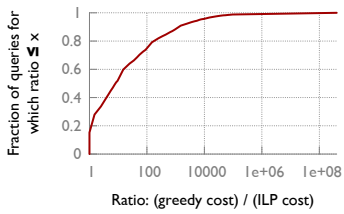
6.3 Optimizer Performance, Running time

The optimizer consists of two components, the Calcite++ centralized SQL query planner, and a site selection + data replication solver. Calcite++ is responsible for a very small proportion of the optimizer's running time, completing in $<$ 10 s for all the queries in §6.1. The majority of the time spent by the optimizer is in the site selection and data replication phase, for which we defined two solutions: a slower but optimal integer linear program, and a faster but potentially suboptimal greedy heuristic (§4.3). We now investigate the relative performance of these two approaches.

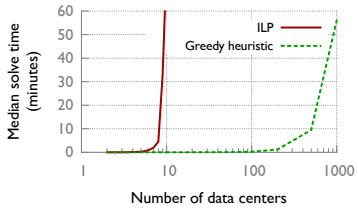
We first compare the optimality gap between the two solutions by evaluating their performance on: (i) the real workloads from §6.1, and (ii) simulations on randomly generated SQL workloads.

In all the workloads we tested in §6.1, the optimality gap is small. The greedy strategy performs remarkably well, identifying the same solution as the ILP in over 98% of the queries we tested. It does fail in some instances, however. For example, the BigBench [18] benchmark has a query which joins a sales log table with itself to identify pairs of items that are frequently ordered together. The heuristic greedily pushes the join down to each data center, resulting in a large list of item pairs stored partitioned across several data centers. But it is then forced to retrieve the entire list to a single data center in order to compute the final aggregate. By contrast, the ILP correctly detects that copying the entire order log to a single data center first would be much cheaper.

In order to compare the strategies' performance in a more general setting, we simulated their performance on randomly generated SQL queries. We generated 10,000 random chain-join queries of the form `SELECT * FROM` T_1 `JOIN` T_2 `... JOIN` T_k `USING(col)`, where each table has schema T_i`(col INT)`, k chosen randomly



(a) Bandwidth cost ratio on 10k randomly generated queries



(b) Running time for workloads of the same scale as §6.1

Figure 9: ILP and greedy heuristic comparison

between 2 and 10. In each query we chose table sizes and join selectivities according to a statistical model by Swami and Gupta [37], which tries to cover a large range of realistic query patterns, generating e.g. both queries which heavily aggregate the input they consume in each step, as well as queries which "expand" their inputs.

Figure 9a shows the results we obtained. The greedy heuristic and the ILP identified the same strategy in around 16% of the queries. In the remaining 84% the ILP performs better: 8× better in the median, and more than 8 orders of magnitude better in the tail. The worst performance generally arises when the heuristic compounds multiple errors of the kind described in the example above. The results show that the gap between the true optimum and the greedy strategy can be substantial.

However, this optimality gap turns out to be difficult to bridge at large scales. Figure 9b shows the running times of both approaches for workloads of the same size as the largest in §6.1. The ILP's running time grows very quickly, taking more than an hour with just 10 data centers. By contrast, the greedy heuristic takes less than a minute even at the 100 data centers scale, although as we have seen this can come at the expense of a loss in solution quality.

We are actively evaluating a hybrid strategy that first uses the greedy heuristic to generate an initial solution, and then uses the ILP for a best-effort search for better alternatives starting from the initial greedy solution, until a specified running time bound. We defer reporting results until a thorough evaluation.

We note again that many of the results reported in this section were based on simulating synthetic workloads, albeit ones that were designed to be realistic [37]. The question of how well both approaches will perform on practical workloads (beyond those in §6.1, where we saw that the greedy heuristic was competitive) remains open, and can only be answered in the future, as analytical workloads rise in sophistication to take advantage of the cost reduction achieved by geo-distributed execution.

6.4 Function-specific optimizations

We close by showing how performance could be improved even further by leveraging optimizations targeted at specific classes of queries from past work. We evaluate the two optimizations we discussed in §5: for top-k queries [7] and for approximate percentile queries [36].

Both algorithms proved quite effective on applicable queries. The top-k algorithm directly benefited the most data-intensive query in the Berkeley big-data benchmark (Figure 8d), achieving a further 9× reduction in data transfer over normal distributed execution. And in a sales-value percentile query we defined on the TPC-CH benchmark database, the approximate percentile algorithm achieved170× less data transfer than exact computation with $< 5\%$ error, and 30× less with $< 1\%$ error.

There is a vast range of optimizations from several related fields one can leverage in the WABD setting — Geode serves as a convenient framework on which these optimizations can be layered.

7 Limitations and Open Problems

Several considerations arise when designing a global analytics framework. We chose to focus solely on minimizing bandwidth costs, while handling fault-tolerance requirements and respecting sovereignty constraints. We did not attempt to address:

Latency. Our focus was entirely on reducing data transfer volume and large scale, and we made no attempt to optimize analytics query latency. It is likely that in many cases the problems of minimizing bandwidth usage and minimizing latency coincide, but effort characterizing the differences is necessary.

Consistency. We support a relaxed eventual consistency consistency model. This suffices for many use cases which only care about aggregates and trends, but the problem of building a WABD solution for applications requiring stronger consistency guarantees remains open.

Privacy. Geode addresses regulatory restrictions by limiting where base data can be copied. However, we allow arbitrary queries on top of base data, and make no attempt to proscribe data movement by queries. While this suffices for scenarios where all queries are carefully vetted before they are allowed to execute, an automated solution, which would necessitate a differential privacy [15] or privacy-preserving computation [26] mechanism, would be interesting to pursue.

Other bandwidth cost models. We assumed each network link has a constant $/byte cost. Supporting other cost models, such as ones based on 95th %ile bandwidth usage, would require modifying the workload optimizer.

Other data models. We have concentrated on WABD for a relational model, but similar issues of bandwidth minimization and latency/regulatory constraints arise in other data models as well, such as Map-Reduce or even computational models that go beyond querying such as machine learning. The fundamental issues, limited bandwidth and the choice between various levels of distributed and centralized computing, remain the same. We discuss these challenges further in [43].

8 Related Work

Unlike parallel databases running in a single LAN [13, 20], where latencies are assumed to be uniform and low, we have non-uniform latency and wide-area bandwidth costs. Work on distributed databases and view maintenance, starting as early as [8, 5] and surveyed in [27, 33], handles efficient execution of arbitrary queries assuming a fixed data partitioning and placement. By contrast, we are able to assume a slowly evolving workload that the system can be optimized for (§4), and automatically replicate data for performance and fault-tolerance while handling regulatory constraints. The focus on analytics instead of transactions, the much larger scale of WABD, and the focus on bandwidth as a measure further differentiates WABD from distributed databases [33].

Spanner [12] focuses on consistency and low-latency transaction support, and is not designed to optimize analytics costs. A complete solution would complement Spanner-like consistent transactions with cost-efficient analytics as in Geode. The Mesa [22] data warehouse geo-replicates data for fault tolerance, as we do, but continues to process analytical queries within a single data center. Stream-processing databases [23, 34] process long-standing continuous queries, transforming a dispersed collection of input streams into an output stream. The significant focus in this area has been on relatively simple data models with data always produced at the edge, with (typically degraded) summaries transmitted to the center, in contrast with the relational model we consider.

Jetstream [34] is an example of stream processing for data structured as OLAP cubes that focuses, as we do, on bandwidth as a metric; however, its data model is not as rich as a relational model. Joins, for example, are not allowed. Further, the sytem relies enitirely on aggregation and approximation to reduce bandwidth, techniques that are not sufficient for the analytical queries we focus on.

Sensor networks [31] share our assumption of limited network bandwidth, but not our large scale or the breadth of our computational model. However, some sensor network techniques can be of interest in WABD: for instance, the approximate percentile algorithm we tested in §6.4 was originally proposed for a sensor network.

Hive [39], Pig [32], Spark [45] and similar systems can provide analytics on continuously updated data, but to the best of our knowledge have not been tested in multi-data center deployments (and are certainly not optimized for this scenario). PNUTS/Sherpa[10] does support geograhically distributed partitions but lays out data to optimize latency (by moving a "master" copy close to where it is commonly used) and not to minimize analytics cost.

Volley [2] addresses placement issues for data while accounting for wide-area bandwidth and user latencies, but without our additional constraint of handling rich analytics. RACS [1] distributes a key-vaue store, not a database, across data centers and focuses on fault-tolerance, not bandwidth. Distributed file systems share our assumption of limited bandwidth, and the caching mechanism we use can be viewed as operating on cached files of answers to earlier analytical queries. However, distributed file systems do not share our relational data model or the query planning problem we face.

PigOut [25], developed concurrently with our work, supports Pig [32] queries on data partitioned across data centers, but targets a simpler two-step computational model than ours and focuses on optimizing individual queries in isolation.

In a recent paper [43] we discussed the vision of geo-distributed analytics for a more general computational model with DAGs of tasks. This was a vision paper that focused on non-SQL models and did not include the detailed description or evaluation of our techniques we present here.

9 Conclusion

Current data volumes and heuristics such as data reduction allow centralizing analytics to barely suffice in the short term, but the approach will soon be rendered untenable by rapid growth in data volumes relative to network capacity and rising regulatory interest in proscribing data movement. In this paper we proposed an alternative: Wide-Area Big Data. Our Hive-based prototype, Geode, achieves up to a 360× bandwidth reduction at multi-TB scales compared to centralization on both production workloads and standard benchmarks. Our approach revisits the classical database problem of query planning from a networking perspective, both in terms of constraints such as bandwidth limits and autonomous policies, as well as solutions such as sub-query deltas and pseudo-distributed execution.

Acknowledgements

We would like to thank our shepherd Hakim Weatherspoon and the anonymous reviewers for their valuable suggestions. We gratefully acknowledge the support of an Alfred P. Sloan Research Fellowship.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *SoCC 2010*.

[2] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI 2010*.

[3] A. Auradkar, C. Botev, S. Das, D. D. Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang. Data infrastructure at LinkedIn. *2014 IEEE 30th International Conference on Data Engineering*, 0:1370–1381, 2012.

[4] C. Ballinger. Born to be parallel: Why parallel origins give Teradata database an enduring performance edge. `http://www.teradata.com/white-papers/born-to-be-parallel-teradata-database`.

[5] 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.

[6] The Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[7] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC '04*, PODC '04.

[8] W. W. Chu and P. Hurley. Optimal query processing for distributed database systems. *IEEE Trans. Comput.*

[9] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *DBTest '11*, DBTest '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10*.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, 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 *OSDI 2012*.

[13] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*.

[14] A. Dey. Yahoo cross data-center data movement. `http://yhoo.it/1nPRImNl`, 2010.

[15] C. Dwork. Differential privacy: A survey of results. In M. Agrawal, D. Du, Z. Duan, and A. Li, editors, *Theory and Applications of Models of Computation*, volume 4978 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2008.

[16] European Commission press release. Commission to pursue role as honest broker in future global negotiations on internet governance. `http://europa.eu/rapid/press-release_IP-14-142_en.htm`.

[17] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw.*, 9(6):733–745, Dec. 2001.

[18] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013. ACM.

[19] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[20] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*

[21] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *ACM CCR*, 39(1), 2008.

[22] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[23] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Data Engg. Workshop 2007*.

[24] T. Inc. Global Internet Geography. `http://www.telegeography.com/research-services/global-internet-geography/`, 2014.

[25] K. Jeon, S. Chandrashekhara, F. Shen, S. Mehra, O. Kennedy, and S. Y. Ko. Pigout: Making multiple hadoop clusters work together. In *Big Data, 2014 IEEE International Conference on*, pages 100–109, Oct 2014.

[26] F. Kerschbaum. Privacy-preserving computation. In B. Preneel and D. Ikonomou, editors, *Privacy Technologies and Policy*, volume 8319 of *Lecture Notes in Computer Science*, pages 41–54. Springer Berlin Heidelberg, 2014.

[27] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, Dec. 2000.

[28] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *SIGCOMM 2011*.

[29] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at Twitter. *PVLDB*, 2012.

[30] G. Lohman. Is query optimization a "solved" problem? `http://wp.sigmod.org/?p=1075`, April 2014.

[31] S. Madden. Database abstractions for managing sensor network data. *Proc. of the IEEE*, 98(11):1879–1886, 2010.

[32] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD 2008*.

[33] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[34] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *NSDI 2014*.

[35] M. Rost and K. Bock. Privacy by design and the new protection goals. *DuD, January*, 2011.

[36] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 239–249, New York, NY, USA, 2004. ACM.

[37] A. Swami and A. Gupta. Optimization of large join queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 8–17, New York, NY, USA, 1988. ACM.

[38] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 102–113, Washington, DC, USA, 2012. IEEE Computer Society.

[39] A. Thusoo, J. S. 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*.

[40] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. SIGMOD, 2010.

[41] A. Tridgell and P. Mackerras. The rsync algorithm, 1996.

[42] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *Trans. Storage*, 5(4):14:1–14:28, Dec. 2009.

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

[44] J. Wiener and N. Boston. Facebook's top open data problems. `https://research.facebook.com/blog/1522692927972019/facebook-s-top-open-data-problems/`, 2014.

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.