LATENCY-BANDWIDTH TRADEOFFS IN INTERNET APPLICATIONS

BY

ASHISH VULIMIRI

DISSE表现TATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Assistant Professor Philip Brighten Godfrey, Chair
Professor Gul A. Agha
Associate Professor Indranil Gupta
Dr. George Varghese, Microsoft Research
Abstract

Wide-area Internet links are slow, expensive, and unreliable. This affects applications in two distinct ways. Back-end data processing applications, which need to transfer large amounts of data between data centers across the world, are primarily constrained by the limited capacity of Internet links. Front-end user facing applications, on the other hand, are primarily latency-sensitive, and are bottlenecked by the high, unpredictably variable delays in the wide-area network. Our work exploits this asymmetry in applications’ requirements by developing techniques that trade off one of bandwidth and latency to improve the other.

We first consider the problem of supporting analytics over the large volumes of geographically dispersed data produced by global-scale organizations. Current solutions for analyzing this data as a whole operate by copying it to a single central data center, an approach that incurs substantial data transfer costs. We instead propose an alternative geo-distributed approach, orchestrating distributed execution across data centers. Our system, Geode, incorporates two key optimizations — a low-level syntactic network redundancy elimination mechanism, and a high-level semantically aware workload optimization process — both of which operate by trading off increased processing overhead (and computation latency) within data centers for a reduction in cross-data center bandwidth usage. In experiments we find that Geode achieves an up to $360\times$ cost reduction compared to the current centralized baseline on a range of workloads, both real and synthetic.

Next, we evaluate a simple, general purpose technique for trading off bandwidth for reduced latency: initiate redundant copies of latency sensi-
tive operations and take the first copy to complete. While redundancy has been explored in some past systems, its use is typically avoided because of a fear of the overhead that it adds. We study the latency-bandwidth trade-off due to redundancy and (i) show via empirical evaluation that its use is indeed a net positive in a number of important applications, and (ii) provide a theoretical characterization of its effect, identifying when it should and should not be used and how systems can tune their use of redundancy to maximum effect. Our results suggest that redundancy should be used much more widely than it currently is.
To my parents.
I would like to thank my advisor Brighten Godfrey for his support and guidance. His door has always been open no matter what I wanted to discuss, something I, like every one of his students, have happily exploited. His insight and his uncanny ability to zero in on the right questions have always been an inspiration, and I am indebted to him for shaping my growth as a researcher.

I am fortunate to have had several amazing mentors over the course of my Ph.D. I would like to thank George Varghese for his advice and support, and for teaching me that any research agenda, no matter how dauntingly ambitious it might seem at the outset, can be thought through and pursued. This thesis is due in large part to him. I am also extremely grateful to Gul Agha and Carlo Curino for their advice and for the discussions we have had.

I am sincerely thankful to the many exceptional people I have had the opportunity to collaborate with. It has been my privilege to work with Oliver Michel, Scott Shenker, Radhika Mittal, Jitu Padhye, Konstantinos Karanasos, Thomas Karagiannis, Hitesh Ballani, Karthik Lakshminarayanan, Zitian Liu and Sri Varsha Gorge. I am also grateful to my netsys labmates for their feedback and conversation: Rachit Agarwal, Mo Dong, Fred Douglas, Soudeh Ghorbani, Chi-Yao Hong, Virajith Jalaparti, Ahmed Khurshid, Qingxi Li, Ankit Singla, and Wenxuan Zhou, thank you.

I would like to thank Indranil Gupta for serving on my committee, and for his valuable feedback.

And finally, I would like to thank my mother Padma Tata for staying a
relentlessly optimistic source of support no matter the circumstance. To her and to my late father Nageswar Rao Vulimiri I jointly dedicate this thesis.
# Table of Contents

Chapter 1  Introduction .............................................. 1
  1.1 Geo-distributed SQL analytics with Geode-SQL ............. 3
  1.2 Extending Geode to a more general computational model . 4
  1.3 Low latency via redundancy in the Internet ................. 5
  1.4 Low latency via redundancy in the data center ............. 5
  1.5 Roadmap .................................................. 6

Chapter 2  Trading latency for bandwidth in global SQL analytics:
  Geode-SQL ..................................................... 7
  2.1 Motivating example ......................................... 10
  2.2 Architecture .............................................. 14
  2.3 Command-layer interface ................................... 15
  2.4 Subquery deltas: Reducing data transfer .................... 18
  2.5 Workload optimizer ........................................ 21
  2.6 Experimental evaluation ................................... 29
  2.7 Related work .............................................. 35
  2.8 Conclusion ................................................. 38

Chapter 3  Extending Geode to a more general computational model:
  Geode-WF .................................................... 39
  3.1 Command-layer interface ................................... 41
  3.2 Architecture and optimizations ............................. 43
  3.3 Experimental evaluation ................................... 46
  3.4 Related work .............................................. 53
  3.5 Conclusion ................................................. 55

Chapter 4  Trading bandwidth for latency in the wide-area Internet . 56
  4.1 Overhead can be low ....................................... 58
  4.2 Cost-benefit analysis: Developing a latency benchmark . 59
  4.3 Experimental evaluation ................................... 65
Chapter 1
Introduction

The wide-area Internet is a critical performance bottleneck in several important applications. Internet links have limited, expensive capacity [1, 2] and frequently demonstrate high, unpredictably variable latency [3, 4]. Several recent studies show that the wide-area network is the fastest growing concern in a number of problem domains, from the back-end [5] to the front-end [4].

These performance limitations affect applications in two distinct ways. Applications such as back-end data processing are primarily bandwidth-sensitive. For instance, large organizations such as Facebook [6], Microsoft, LinkedIn [7], Yahoo [8] and Twitter [9] operate data processing pipelines that transfer 10s to 100s of TBs of data each day between data centers around the world. These pipelines are used to support batch data analysis computations, meaning processing latency is not a critical metric of interest. Instead, the primary concern is minimizing the use of scarce, expensive wide-area bandwidth [2, 5].

On the other hand, front-end user facing applications are primarily latency-sensitive. Studies from Google [10], Microsoft [11] and Amazon [12] found that reducing wide-area latencies by just tens of milliseconds could improve their annual revenue by hundreds of millions of dollars. The primary challenge here is dealing with the complex set of interdependent factors which can lead to sudden, unpredictable spikes in Internet latencies [3, 4].

Thus, both classes of applications have asymmetric requirements from the network — they are primarily concerned with optimizing one of bandwidth
and latency, and would be willing to sacrifice inflation in the other to that end. This asymmetry is what our work exploits. We develop techniques that trade off one of latency and bandwidth to improve the other in key network-centric applications.

We first consider an important bandwidth-sensitive application: supporting data analytics in large globally distributed organizations. Large organizations today collect 10s to 100s of TB of data each day across several data centers around the world [6, 7, 8, 9], and analyzing all this geographically dispersed data as a whole is necessary to expose valuable insight. Organizations currently address this problem by copying all the data to a single central location where analytics is then run, an approach that consumes a significant amount of scarce, expensive wide-area bandwidth [5, 1]. We instead build a system that leaves data at the edge (partitioned across data centers) and pushes computation down, engaging in distributed execution of complex analytical workflows. Our system, Geode, incorporates key optimizations which reduce data transfer volumes by trading off for some amount of increased processing overhead (and therefore computation latency).

Next, we consider end-user facing applications where latency, rather than bandwidth, is the primary metric of concern. Controlling latency, especially the tail of the latency distribution, is difficult: modern applications have many possible sources of high latency [13, 14, 15, 16], and even in well-provisioned systems where individual operations usually work, some amount of uncertainty is pervasive. We evaluate a natural general-purpose technique that mitigates latency uncertainty by trading it off for increased bandwidth consumption: initiate multiple redundant copies of latency sensitive operations and take the first copy to complete. Our results provide a thorough characterization of redundancy as a general technique through both empirical evaluation and theoretical analysis, and show that despite the overhead that it adds, redundancy is a net positive in a range of realistic settings, suggesting that it should be used much more widely than it currently is.
1.1 Geo-distributed SQL analytics with Geode-SQL

We start by presenting Geode-SQL, our system for supporting complex analytical SQL queries across large volumes of data (up to 100s of TBs per day) partitioned across several data centers. In contrast to the centralized approach that is the norm today, Geode-SQL operates in a geo-distributed manner, taking queries expressed against a logically centralized view of the data, automatically decomposing them and orchestrating distributed execution across data centers.

Geode-SQL implements two key optimizations. First, we use a low-level redundancy elimination mechanism to reduce data transfer volumes. We opportunistically cache all intermediate data produced during distributed query execution, and reuse cached results whenever feasible to compress data transfers. The caching and compression mechanism we employ can increase transfer latencies, but achieves significantly reduced data transfer volumes — which is the metric we are concerned with.

The form of caching we develop also serves as a novel low-level syntactic solution to the traditional view-maintenance problem in relational databases, which is conventionally solved using high-level mechanisms based on an understanding of relational semantics. Compared to these classical techniques our approach has its advantages and its disadvantages — we discuss them in Chapter 2, and argue that it is well suited to the specialized requirements of our setting.

Second, we develop a high-level workload optimizer design that jointly tunes the choice of distributed execution strategy and data replication policy to minimize overall data transfer costs, while handling sovereignty and fault-tolerance requirements. Our optimizer design combines a classical centralized relational query planner with an integer programming approach to modelling our geo-distributed setting. A key challenge here is to be able to measure the costs of the various strategies available to the system; we develop a technique we call pseudo-distributed measurement to simulate arbi-
trary alternative data replication and query execution strategies on top of the currently deployed system. Pseudo-distributed execution slow down computation within data centers, but it incurs zero bandwidth overhead and never worsens (expensive) cross-data center bandwidth consumption.

In experiments with both a real Microsoft production workload as well as several synthetic benchmarks, Geode-SQL achieves an up to $360 \times$ reduction in data transfer costs compared to the current centralized baseline.

1.2 Extending Geode to a more general computational model

While SQL analytics is a key component in large-scale data analytics pipelines, in many organizations it does not make up the entirety of the analytics workload: these pipelines can also incorporate, for example, MapReduce jobs [7, 17] or machine learning computations [18]. We therefore build an extended version of Geode-SQL, Geode-WF, supporting a much more general computational model with DAG workflows composed out of arbitrary user-provided code.

Our key ideas from Geode-SQL continue to be applicable in this expanded setting, but they need to be adapted to handle the more sophisticated computational model targeted here. In particular, the workload optimizer design needs to be substantially rearchitected: the fact that we allow arbitrary code means it is no longer possible to explore the entire optimization space exhaustively in a single phase, unlike in the SQL setting. We instead develop a dynamic evolutionary approach that iteratively searches for improvements to the currently deployed strategy, gradually transitioning the system towards better and better operating points.

We show in experiments that we continue to achieve similar levels of improvement here as in the SQL setting.
1.3 Low latency via redundancy in the Internet

We next consider latency-sensitive applications, and explore a natural, general purpose technique trading off bandwidth for latency: initiate multiple copies of latency-sensitive operations, and take the first copy to complete. Redundancy has been explored in some past systems, but these tend to be the exception, not the rule: redundancy is typically avoided because of a fear of the overhead that it adds. We argue that this is a missed opportunity, using a combination of empirical evaluation and theoretical analysis to show that redundancy is a net positive in large classes of applications despite the overhead that it adds. We make two key contributions.

First, we develop an economic framework for comparing the cost of the added overhead that redundancy induces (at all affected parties, including the client, server, and intermediate network) against the benefit that it adds. We then use this framework to derive a benchmark for identifying when redundancy is and is not useful, showing that redundancy would be cost-effective even in the most conservative (e.g. cell phone) scenarios as long as it improves latency by at least 10 ms for every kilobyte of added traffic that it sends.

Next, we empirically evaluate the effect of redundancy in three major Internet applications: the TCP handshake, DNS, and a multipath overlay network. We show that redundancy achieves significant gains in all these applications, orders of magnitude larger than the cost-effectiveness threshold. For example, querying multiple DNS servers can reduce the fraction of responses taking longer than 500 ms by 6.5×, and the fraction taking longer than 1.5 sec by 50×.

1.4 Low latency via redundancy in the data center

While most of our work targets applications operating over the wide-area Internet, the form of redundancy we described in §1.4 is a fairly general
technique. We next evaluate how it can be used in data center applications.

Unlike the wide-area Internet, data centers are typically closed systems, meaning operators can exercise a greater degree of control over system functioning. We develop and analyze a more sophisticated queueing theoretic model of how redundancy can be controlled in such systems. Our analysis provides a characterization of the conditions under which redundancy should and should not be expected to help.

We then evaluate the effect of redundancy empirically in two data center applications: a key-value storage system, and in multipath network transport. We find that in a range of realistic system configurations, redundancy achieves a substantial improvement: up to $2 \times$ better mean and $8 \times$ better tail latency in the key-value store; and up to a $38\%$ improvement in flow completion times in the data center network. We also show that our empirical results validate the predictions from our theoretical model.

1.5 Roadmap

The remainder of this thesis is structured as follows. Chapter 2 presents Geode-SQL, our geo-distributed SQL analytics system. Chapter 3 discusses how Geode can be extended to a more general computational model supporting arbitrary user-provided code. Chapters 4 and 5 show how deliberate redundancy can be used to improve latency in, respectively, the wide-area Internet and in data center applications. Chapter 6 concludes and suggests directions for future research.
Chapter 2

Trading latency for bandwidth in global SQL analytics: Geode-SQL

Organizations operating at global scale, from oil companies to social media, need to analyze vast amounts of geographically distributed data. The volume of data collected while logging user interactions, monitoring compute infrastructures, and tracking business-critical functions is approaching petabytes a day, across tens of data centers around the world. Analyzing all this data as a whole is necessary to extract business-critical insight. Handling data analytics across data centers — as opposed to traditional parallel databases [19] within a data center — introduces 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 challenge of dealing with the limited capacity of the wide-area Internet [20]. Recent work in global databases illustrates this research trend, from Spanner [21] (global consistency) to Mesa [22] (replication for fault tolerance).

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 [17, 23], Twitter [24], Yahoo! [25] and LinkedIn [26] 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 [27].

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. However, for large mod-
ern 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 2.1). 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 2.1). A key reason is the expense of adding network capacity: for instance, a new submarine cable connecting South America and Europe is expected 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 [28, 1, 5].

An additional emerging difficulty is that privacy concerns (for example in the EU [30]) may result in more regulatory constraints on data movement. However, while local governments may start to impose constraints on raw data storage [31], 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 rising global data volumes 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. This chapter focuses on data structured as SQL tables, a dominant paradigm (Chapter 3 discusses extensions to a more general computational model). We present Geode-SQL, a system supporting SQL analytics on geo-distributed data, providing automated handling of fault-tolerance requirements and using replicated data to improve performance whenever possible. Geode-SQL supports the entire array of SQL operators on global data including joins, providing exact answers.

We target the batch analytics dominant in large organizations today [23, 24, 26], where the cost of supporting analytics execution is the primary consideration, and analytics latency is not a metric of interest. In our setting, resources within a single data center (such as CPU and storage) are relatively cheap compared to cross-data center bandwidth; we focus exclusively on minimizing cross-data center bandwidth cost. Our key optimizations are designed for these requirements, and function by trading off some amount of increased computational overhead within data centers (resulting in an increase in total execution latency) for a reduction in cross-data center bandwidth usage.

We make four main contributions:

1. **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. The process of identifying and computing deltas adds a latency penalty but achieves significant bandwidth reduction.

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

3. **Optimizer**: We use these collected measurements to 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.

4. **Demonstrated Gains**: We built a prototype Geode-SQL implementation on top of the popular Hive [32] analytics framework. Geode-SQL achieves a $250 \times$ reduction in data transfer over the centralized approach in a standard Microsoft production workload, and up to a $360 \times$ improvement in a range of scenarios across several standard benchmarks, including TPC-CH [33] and Berkeley Big Data [34].

2.1 Motivating example

We start by discussing an example inspired by the Berkeley Big-Data Benchmark [34] and use it to motivate our architecture.

Consider 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:

\[
\begin{align*}
\text{ClickLog}(\text{sourceIP}, \text{destURL}, \text{visitDate}, \text{adRevenue},...) \\
\text{PageInfo}(\text{pageURL}, \text{pageSize}, \text{pageRank},...)
\end{align*}
\]

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-SQL provides an equivalent location-independent [35] query interface over distributed data. The analyst submits the query Q unmodified to Geode-SQL, which then automatically partitions the query and orchestrates distributed execution. Geode-SQL constructs the distributed plan in two stages:

1. Choose join order and strategies. Geode-SQL 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 §2.3). 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 §2.5.1.

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 2.2. 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 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-SQL now needs to assign tasks to data centers, taking into account task input dependencies and base data regulatory constraints.

Geode-SQL 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 workflow 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 (§2.5).

At runtime, Geode-SQL 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 §2.5). The ILP described here is a simpler version of the more nuanced multi-query optimizer in §2.5.3.

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 $3 \times 100M \times 20 + (2/3) \times 1B \times 12 + 100K \times 12 = 14$ GB of data each day, compared to 1.2 TB per day for the centralized approach.

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 (§2.6) contains a query with 8 joins (involving 9 different tables) for which the degrees of freedom (join order, join strategy, replication) are much higher.
2.2 Architecture

Our example motivates the architecture in Figure 2.3.

Geode-SQL 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 §2.4.

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 (§2.5.2), which may entail rewriting the analytical queries. The optimizer never initiates changes directly, but instead makes suggestions to an administrator.

We next discuss: the interface Geode-SQL presents to analysts (§2.3); an optimization we implement to reduce data transfers (§2.4); and the workload optimizer including pseudo-distributed execution (§2.5).

### 2.3 Command-layer interface

Geode-SQL presents a logically centralized view over data partitioned and potentially 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 §2.2. We discuss the basic interface Geode-SQL presents to analysts in this section.

**Describing schema and placement**

Geode-SQL 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-SQL about table schema and placement by submitting `CREATE TABLE` statements annotated with placement type and in-
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 [32]. Our architecture is not tied to Hive and can be easily adapted to work with other SQL backends instead.

Joins. By default, Geode-SQL passes user-submitted queries through Calcite++ (§2.5.1) first to optimize join order and execution strategy. However, users can enforce a manual override by explicitly annotating joins with a

\[ \text{JOINHINT(strategy = \_\_)} \]

instruction.

Geode-SQL 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 exploring adding other strategies, such as hash-joins with a special partitioning-aware hash function [36].

Nested queries. Nested queries are processed recursively\(^1\). 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

\(^1\)This simple strategy is sufficient because Hive does not support correlated subqueries.
at the master data center (hosting the Geode-SQL 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 *combined* 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 *combined* 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-SQL 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 [37], originally proposed in a CDN analytics setting and recently used by JetStream [28]; and an approximate percentile algorithm from the sensor networks literature [38]. We evaluate the benefit from these optimizations in §2.6.3.
2.4 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.

```sql
SELECT sourceIP, avg(rev)
FROM (  
    SELECT hour(visitDate), sourceIP,  
    sum(adRevenue) as rev  
    FROM UserVisits u  
    WHERE u.visitDate >= date_sub(now(), 1)  
    GROUP BY hour(visitDate), sourceIP  
    HAVING sum(adRevenue) > $10
) as T;
```

If the query is run once an hour, more than 95% of the data transfer will be wasted because every hour unoptimized Geode-SQL 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 2.4 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.
Figure 2.4: Subquery delta mechanism

Note that $DC_A$ still needs to recompute the results for $Q$ the second time around. Caching does not reduce intra-data-center computation: the process of storing cached results, computing diffs, and applying patches adds an intra-data center computation, storage, and latency penalty. 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 §2.6, 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 $3.5 \times$. 
Database parallels

In a sense, the caching mechanism is essentially an automated way of solving the view detection and maintenance problems in relational databases [39]. It materializes an implicit view when a sub-query first arrives, and it automatically refreshes the view and uses the updated results whenever a subsequent query implicitly references it. However, while the caching mechanism does provide a solution for lazy view materialization, the view selection problem (akin to a cache-aware query planning) remains challenging in our setting. Consider for example:

```sql
SELECT val - (SELECT avg(val) FROM T)
FROM T
```

Blindly caching the output of the entire query in each data center is pointless, as almost any update to val would invalidate the entire cache. On the contrary, retrieving (and caching) the base data and then subtracting the average centrally would make caching very effective. Intuitively, these two execution plans amount to “cutting” the query execution tree at two different points. Each cut-point corresponds to choosing a different implicit view to materialize, and the choice of view affects overall system performance. We leave a more sophisticated view selection mechanism for future research.

Optimizing for situations like this would necessitate implementing an intelligent view selection mechanism, but simply implementing an isolated view selection component is not sufficient for optimal performance. As we have noted several times so far, overall system performance depends on the combination of query plan, view selection, and data placement strategy chosen, all of which are interdependent, and optimizing any one component in isolation is not enough. We consider the combined problem in its full depth in §2.5.
2.5 Workload optimizer

Geode-SQL 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 [23, 26, 24]. 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 and/or fault-tolerance.

This is a challenging optimization problem, for several reasons. First, finding the optimal execution strategy for any one operation in isolation is by itself non-trivial. For example, the choice of optimal distributed strategy for processing a join between two tables is a complicated function of several parameters: the sizes of the tables, the rates at which they are updated,
the selectivity of the join predicate, etc. Figure 2.5 shows the optimal distributed strategy for joining two tables from the BigBench [40] benchmark as a function of the rates at which the two tables are updated. Depending on the update rates any of four distinct strategies can dominate: copying both tables centrally, broadcasting the updates of the least modified table, or performing a distributed hash join (i.e., re-distribute both tables via hashing). Second, the choice of strategy for any one operation can affect the performance of succeeding operations that depend on its output, since the choice of strategy can affect the partitioning and placement of its output date. Third, the choices made for different SQL queries in the workload can be coupled together, as they might leverage a shared data replication strategy, or be affected by the optimization discussed in §2.4.

The problem we face is akin to distributed database query planning. In that context, it is common [41] 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 2.6. Briefly, we start by identifying the optimal centralized plan for each query in the core workload using the Calcite++ query planner (§2.5.1). We then collect precise measures of the data transfers during each step of distributed execution for these plans using pseudo-distributed measurement (§2.5.2). 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 (§2.5.3).

### 2.5.1 Centralized query planning: Calcite++

Apache Calcite is a centralized SQL query planner currently being used or evaluated by several projects, including Hive [32]. 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 approxi-
mate 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 (§2.2) 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 (§2.6.1), 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.

2.5.2 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. Traditional database cardinality estimation techniques can be very inaccurate at generating absolute cost estimates, especially in the face of joins and user-defined functions [42]. 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-SQL deployment), modifying query execution when necessary to make it possible to collect the estimates we need. As an example consider query Q (Figure 2.2) from §2.2, 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-SQL 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 \texttt{WHERE country} = \texttt{X} clauses constraining each of \texttt{q2}, \texttt{q3}, \texttt{q4} to operate on the right subset of the data\(^2\). Figure 2.7 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 2.2.

This technique, which we call \textit{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

\(^2\)Every partitioned table in Geode-SQL has a user-specified/system-generated field identifying the partition each row belongs to (§2.3).
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-SQL 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 §2.6.2.

**Trading precision for overhead:** While Geode-SQL’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 \texttt{SUM} query operating on all this data. Retrieving one total \texttt{SUM} over all 10 partitions is sufficient, but Geode-SQL always simulates a fully distributed topology with each partition in a separate data center, thus retrieving separate \texttt{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.
2.5.3 Site selection and data replication

After identifying the logical plan (DAG of tasks) for each query (§2.5.1) and measuring the data transfer along each edge (§2.5.2), 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 2.8) 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. Constraints specify sovereignty and fault-tolerance requirements. 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 §2.4 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 due to 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-

\footnote{We schedule \emph{edges} instead of \emph{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 \emph{quadratic} (not linear) objective.}
Inputs:
$D =$ number of data centers
$P =$ number of data partitions
$G = ⟨V, E⟩ =$ union of DAGs for all core workload queries
$b_g =$ bytes of data transferred along each edge $g ∈ E$ (from pseudo-distrib.)
$\text{link}_{de} =$ cost ($$/\text{byte})$ of link connecting DCs $d$ and $e$
$f_p =$ min. # of copies of partition $p$ that need to be made for fault-tolerance
$R ⊆ P × D = \{(p, d) | \text{partition } p \text{ cannot be copied to DC } d \text{ due to regulatory constraints}\}$

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 DC $d$ and DC destination $e$
$z_{td} =$ 1 iff a copy of task $t$ in the DAGs is assigned to DC $d$

Solution:
$$\text{replCost} = \sum_{p=1}^{P} \sum_{d=1}^{D} \text{update}_p \times x_{pd} \times \text{link}_{\text{homeDC}(p),d}$$
$$\text{execCost} = \sum_{g \in E} \sum_{d=1}^{D} \sum_{e=1}^{D} y_{gde} \times b_g \times \text{link}_{de}$$

minimize $\text{replCost} + \text{execCost}$

subject to
$$\forall (p, d) \in R : x_{pd} = 0$$
$$\forall p : \sum_d x_{pd} ≥ f_p$$
$$\forall d \forall e \forall g | \text{src}(g) \text{ is a partition} : y_{gde} ≥ x_{\text{src}(g),d}$$
$$\forall d \forall e \forall g | \text{src}(g) \text{ is a task} : y_{gde} ≥ z_{\text{src}(g),d}$$
$$\forall n \forall e \forall g | \text{dst}(g) = n : z_{ne} = \sum_d y_{gde}$$
$$\forall n \forall p \forall d | n \text{ reads from partition } p \land (p, d) \in R : z_{nd} = 0$$
$$\forall n : \sum_d z_{nd} ≥ 1$$

Figure 2.8: Integer Linear Program jointly optimizing site selection and data replication
query caching.

2.6 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 (§2.6.1)? What is the runtime overhead of collecting the (pseudo-distributed) measurements needed by our optimizer (§2.6.2)? Can implementing narrow application-specific optimizations yield significant further bandwidth cost reduction (§2.6.3)?

2.6.1 Large-scale evaluation

We ran experiments measuring Geode-SQL performance on a range of workloads, on two Geode-SQL 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 [33] 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-SQL**: BigBench [40] 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 evaluate the relational component of the workload here, and consider the full workload with the non-relational components included in Chapter 3.

**Big-data**: The big-data benchmark [34], 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 [43] database and OLTP workload, configured with database schema:

```
table(key, field1, field2)
```

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` has an analytical workload consisting of the single query

```
SELECT field1, AVG(field2) FROM Table GROUP BY field1
```

Since there are only 10 distinct values of `field1`, Geode-SQL 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-SQL’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 2.9 shows results for all six workloads. (We are required to obfuscate the scale of the axes of Figure 2.9a 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-SQL and Berkeley big-data: 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 $150 - \times 360$ in four of our six workloads (YCSB-aggr,
Figure 2.9: End-to-end evaluation (contd.)
Figure 2.9: End-to-end evaluation
Microsoft prod., TPC-CH, and BigBench-SQL). The improvement was only 3× in the Big-Data with normal distributed execution, but it turns out that implementing a specialized algorithm for top-k queries [37] can improve the performance gap to 27×. This algorithm involves a sophisticated three-phase distributed computation that cannot be directly implemented using the simple MapReduce-like user-defined function interface we exposed in §2.3. It can, however, be supported using the more sophisticated workflow abstraction we expose in Chapter 3 — we discuss details in §3.3.

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.

2.6.2 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 §2.5.2). We measured the added overhead for all the queries we tested in §2.6.1.

In all our workloads, we found that the latency overhead compared to normal distributed Geode-SQL 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 2.2, 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.

2.6.3 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, by considering a concrete case study: adding approximate computation functionality to the system.

Specifically, we implemented the approximate percentile algorithm by Shrivastava et al. [38] and the approximate count-distinct algorithm by Flajolet et al. [44], and tested their performance on queries we defined on the sales table in the TPC-CH benchmark database. Figure 2.10 shows results. A substantial improvement can be achieved: the approximate percentile computation transfers $170 \times$ less data than exact computation with $< 5\%$ error, and $30 \times$ less with $< 1\%$ error; and the approximate count-distinct computation transfers $6 \times$ less data at just $0.2\%$ error, $30 \times$ less at $1\%$, and $300 \times$ at $5\%$.

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

2.7 Related work

We now discuss directly relevant prior work focusing on the relational computational model. There is additional past work targeting non-relational computation: we discuss it in §3.4.

Unlike parallel databases running in a single LAN [45, 46], 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 [47, 48] and surveyed in [41, 35], handles efficient execution of arbitrary queries assuming a fixed data partitioning and
Figure 2.10: Approximate queries on the TPC-CH database
placement. By contrast, we are able to assume a slowly evolving workload that the system can be optimized for (§2.5), 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 [35].

Spanner [21] 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-SQL. 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 [49, 28] 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 [28] 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 a simplified version of the relational model. Joins, for example, are not allowed. Further, the system relies entirely on aggregation and approximation to reduce bandwidth, techniques that are not sufficient for the analytical queries we focus on.

PigOut [50], developed concurrently with our work, supports Pig [51] 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.
2.8 Conclusion

Current data volumes and heuristics such as data reduction allow centralizing SQL 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 chapter we proposed an alternative: geo-distributed execution. Our Hive-based prototype, Geode-SQL, achieves up to a $360 \times$ bandwidth reduction at multi-TB scales compared to centralization on both production workloads and standard benchmarks. Our approach takes advantage of the latency-insensitive nature of the applications we target by developing optimizations that trade off an increase in analytics execution latency for a reduction in data transfer volumes. We next discuss how our techniques can be extended to more general (super-relational) computation.
Chapter 3

Extending Geode to a more general computational model: Geode-WF

Chapter 2 discussed the problem of supporting geo-distributed analytics with a SQL computational model. However, while SQL analytics is a key component in large-scale data analytics pipelines, in many organizations it represents only part of the analytics workload. LinkedIn [7], Twitter [18], Facebook [17] and Microsoft all report supplementing SQL analytics with other forms of computation, such as MapReduce jobs [17] and machine-learning analyses [18, 7].

Figure 3.1 shows a representative example of a super-relational analytics workflow, derived from the BigBench [40] benchmark. The Figure shows three input data sources: clickstream, storing Web server logs of user browsing activities; reviews, capturing textual representations of item reviews; and sales, a relational table storing transactional records of item purchases. Each input data source is partitioned across geographically distributed data centers. The DAG of operators shown in Figure 3.1 depicts one of the many workflows run daily to process the raw data and extract insight about user behavior, sales performance, and item reception. In particular, beside classical relational operators, this workflow includes arbitrary computations that manipulate unstructured data (session extraction and sentiment analysis) and machine learning stages (behavior modeling).

As we discussed in Chapter 2, companies deal with large-scale analytics today by copying all remotely-born data to a central location for analysis. Any such solution is destined to consume cross-data center bandwidth proportional to the volume of updates/growth of the base data. Referring back
to Figure 3.1, this consists of copying the partitions for the three base data sources clickstream, reviews, and sales from the edge data centers to a central location, and running the DAG leveraging standard single-cluster technologies. For example, using a Hadoop stack, one could use DistCP to copy data across HDFS instances in each data center, Oozie to orchestrate the workflow, Hive for relational processing, MapReduce for session extraction, OpenNLP for sentiment analysis, and Mahout for behavior modeling. We prototyped this setup and gathered initial numbers to quantify the cost of this approach. Assuming daily runs of the DAG of Figure 3.1, 1 TB daily data growth, and 10 data centers, we observe cross data center traffic of 706 GB per day. (Other base data sources in the original benchmark, not used by the DAG in Figure 3.1, make up another 318 GB per day."

This chapter presents our geo-distributed workflow execution system, Geode-WF, which significantly reduces these data transfer costs. Geode-WF extends Geode-SQL to supports directed acyclic job graphs composed out of arbitrary computations, running over geographically distributed data. The system automatically devises distributed execution plans and an accompanying data replication strategy, optimizing these two aspects concurrently to minimize WAN bandwidth utilization while respecting regulatory requirements. We show how the core techniques we developed in Geode-SQL, redundancy elimination via caching and workload optimization based
on pseudo-distributed measurements, can be extended to apply in this more general setting. The workload optimizer design, in particular, requires significant modification: the fact that we allow arbitrary code means that the precise Integer Linear Programming approach we developed in §2.8 (which operates based on an understanding of SQL semantics) no longer applies, and we instead develop an alternative, more general, greedy heuristic approach and discuss its features and limitations.

We evaluate Geode-WF on two realistic benchmarks, BigBench [40] and Voter [52], and show that we can continue to achieve the same two orders of magnitude level of improvement we saw in the more restricted SQL setting (§2.6). We also demonstrate how we can use the additional flexibility afforded to us by the expanded computational model to further optimize some of the workloads from §2.6.

3.1 Command-layer interface

We now discuss the interface Geode-WF exposes to analysts.

Instead of the SQL tables in Geode-SQL, Geode-WF operates on data stored in logical files. Each file contains one or more physical partitions stored in different data centers. Files are stored in an enhanced version of the Hadoop Distributed Filesystem file format [53], with an additional provenance field added to every tuple to track which partition (which data center) the tuple belongs to. The provenance field is used to support the pseudo-distributed measurement mechanism we use to measure data transfer costs, and serves a similar function to the partition column we described in §2.3.

Analysts define logically centralized directed acyclic graph (DAG) workflows operating on these files. Each computational node in the DAG reads from one or more files, processes its input, and produces output feeding into succeeding nodes in the DAG. Each node is of one of three kinds of jobs:
• **White-box jobs.** SQL and MapReduce computations are white-box jobs: Geode-WF is designed with an understanding of their semantics and uses that understanding when identifying distributed execution plans for these jobs, as we did in Geode-SQL (Chapter 2).

• **Grey-box jobs.** Analysts can register one or more custom distributed implementations for any job type. The Geode-WF workload optimizer will profile all provided implementations and automatically pick the best choice for the given workload.

• **Black-box jobs.** Jobs without any analyst-provided distributed implementation are treated as black boxes, and are processed by copying all their input into one data center where the provided (arbitrary) code is then run.

Figure 3.2 shows a color-coded version of the running example from Figure 3.1. The workflow reads from three logical files: *clickstream, sales* and *reviews*, and contains four white-box SQL nodes; one white-box MapReduce node (session extraction); one grey-box node (sentiment analysis, a natural-language processing task for which several distributed algorithms are available [54]); and one black-box node (behavior modeling).
3.2 Architecture and optimizations

Geode-WF shares the same basic architecture as Geode-SQL, described in Figure 2.2. Two of the key optimizations Geode-SQL incorporates, caching-based redundancy elimination (§2.4) and pseudo-distributed measurement (§2.5.2) continue to be applicable in the expanded Geode-WF setting, with only minor mechanical modifications need to support our extension to general workflows. Specifically, the caching mechanism now tags subcomputations with a signature based on the sub-DAG it represents\(^1\) rather than with the SQL representation that would have been used in Geode-SQL; and the pseudo-distributed measurement mechanism uses the provenance tag we described in §3.1 to identify which data center each tuple should belong to, rather than the partition column in §2.3.

The workload optimizer, however, requires a substantial design. The greater flexibility allowed by Geode-WF’s computational model means that, unlike in the SQL setting, it is no longer possible to explore the entire optimization space exhaustively. We instead develop a greedy heuristic that explores a reasonable subset of the space. We discuss its operation next.

3.2.1 Workload optimizer: Greedy heuristic

As we saw in §2.5, the workload optimizer faces a challenging problem. There are a number of decisions it needs to jointly optimize to minimize bandwidth usage: (1) the physical operator to use for jobs that accept multiple implementations (e.g. grey-box jobs), (2) the data center to which each task is scheduled (respecting sovereignty constraints), and (3) the set of data centers to which each partition of the base data is replicated. These decisions are difficult for several reasons.

First, finding the best execution strategy for each task in a DAG in isola-

\(^1\)signature\( (C) = \)depth-first traversal of the sub-DAG induced by \( C \). This mechanism is imperfect – e.g. changing the order in which DAG edges are listed can change the signature and cause a cache miss – but is a reasonable starting point.
tion is by itself non-trivial, as the choice of optimal join execution strategy can be a complicated function of several parameters: the size of the base data, the rates at which they are updated, the nature of the computation the job processes, etc. Second, the choice of execution strategy for a DAG node may affect the choice of strategies for other DAG nodes, as this choice determines the partitioning and placement of the node’s output data. In a workload with $n$ nodes (in one or more DAGs) and up to $k$ possible execution strategies per node, the analyzer would have to explore a $O(k^n)$ search space. Third, the choices made for nodes of different DAGs influence each other, as they might leverage a shared data replication strategy, or be affected by our redundancy elimination mechanism.

In the SQL setting, we were able to leverage our knowledge of the special structure of SQL queries to simplify a lot of this exploration. Distributed computations in SQL queries take one of a few very specific forms: joins, recursive queries, and user-defined computations with a well-defined (Map-Reduce-like) processing pattern. We were able to exploit this structure by using classical relational query planning techniques to simplify the majority of the exploration of the search space early on based on light-weight table statistics ($\S$2.5), leaving behind a manageable subspace which could be explored carefully using pseudo-distributed measurements and the integer-linear program in $\S$2.5.3.

This simplification is no longer possible in the more general workflow setting we consider here. The far greater flexibility afforded to analysts by the arbitrary workflow computational model means that space of possible execution plans is too vast to explore exhaustively. Instead, we propose a Greedy Heuristic that performs remarkably well in practice, while exploring only a small subset of the search space.

The heuristic (Algorithm 1) optimizes each node of each DAG in isolation, proceeding from the source nodes and moving greedily outward in topological order. For each node, we evaluate all strategies compatible with sovereignty constraints, using pseudo-distributed measurement to measure their costs, and greedily pick the lowest cost alternative at that node. In the
Algorithm 1 Geode-WF workload optimizer: Greedy Heuristic

for all DAG $G \in$ workload do
  for all task $t \in$ toposort($G$) do
    $t$.completed = false
    if $\exists$ parent $p$ of $t$ such that $p$.completed = false then
      assign a default strategy to $t$
    else
      if all strategies for $t$ have been evaluated then
        for all data source $S \in$ input($t$) do
          test if replicating $S$ reduces bandwidth further
          assign the lowest cost strategy to $t$, and replication strategy to $S$
          $t$.completed = true
        else
          explore next strategy for $t$
  
Figure 3.3: Placement of the running example DAG

process, the system also evaluates whether systematically replicating any of the input base tables can help amortize transfer costs among DAGs.

Figure 3.3 shows the resulting execution strategy for the DAG in our running example. The arrows in red are cross-data center data transfers, and add up to 1.07 GB. Most of the cost is incurred while broadcasting the output of sentiment analysis during join computation. The alternatives—such as using a semi-join, or redistributing via hashing—all turn out to be more expensive. In our running example, Geode-WF decides not to replicate base tables, but replication proves fundamental for all workloads in our experiments (§3.3).

This simple heuristic requires a limited number of measurements (as it explores just a small portion of the search space), and experimentally works
well whenever DAGs “reduce” data volumes at each subsequent stage. This seems common in practice: it is true of 98% of all the DAGs in our workloads.

However, the heuristic can fail when confronted with DAGs that “expand” the input data they consume (before optionally condensing it). Consider the DAG in Figure 3.4, a simplified version of query Q1 in the BigBench benchmark. The DAG starts from a table listing items ordered by customers (size \( n \)), performs a self-join on the table to find pairs of items that are ordered together (worst case size \( O(n^2) \)), computes frequencies of pairs, and returns frequent pairs. The heuristic would push the join down and run it distributed, thus exploding data in edge data centers, incurring unnecessarily large data transfer during the second stage. In DAGs like these (which made up less than 2% of our experimental workloads), the heuristic can identify suboptimal solutions.

In §3.3.2 we will present results from a parameter sweep evaluating how far the greedy heuristic is from the theoretical optimum in a range of workloads.

### 3.3 Experimental evaluation

We now present results evaluating a large-scale deployment of Geode-WF (§3.3.1), discuss the workload optimizer’s performance and efficiency (§3.3.2), and show how the additional flexibility allowed by the more general workflow computational model can allow further optimizing the SQL workloads from §2.6 (§3.3.3).
3.3.1 Large-scale evaluation

We evaluated Geode-WF’s performance on two benchmark workloads.

**BigBench:** This workload [40], currently on the path towards a SPEC standardization, models a large-scale retailer that sells products both online and in stores, collecting several forms of information from customers in the process, such as item reviews and click logs. The data comprises a relational portion, TPC-DS, and the two non-relational sources we show in our running example, clickstream and reviews. In our multi-data center setup, we assume data are produced at edge data centers, and that every user has a “home” data center that all their accesses are logged to.

**Voter:** The Voter benchmark [52] models a competitive reality television series in which callers phone in to vote for different contestants. The analytical workload asks for spatial and temporal aggregates over contestant popularity information.

As in our Geode-SQL evaluation (§2.6), we ran experiments on two deployments: across three geographically distributed Azure data centers (US, EU, Asia), and on a large on-premise cluster, on which we simulate a multi-data center setup. Specifically, we ran the benchmark-based experiments in both deployments for up to 25 GB of data transfers and validated that the results were identical. We ran experiments in the 25 GB to 10 TB range exclusively on the on-premise cluster, since each of the multi-terabyte runs for the baseline centralized approach would have otherwise cost thousands of dollars in cross-data center bandwidth. The on-premise cluster consists of 120 machines, each with 128 GB of RAM, 32 cores, and 12 x 3 TB of drives. The interconnect is 10 Gbps within a rack, and 6 Gbps across any two machines.

The software stack we use in these experiments is based on a combination of Oozie, Hive, MapReduce, OpenNLP, Mahout and DistCP. Since our focus is on network bandwidth consumption and not query execution performance, we expect similar results from alternative choices of stack. All network transfers, both baseline and Geode-WF, are gzip-compressed. For
the centralized baseline we always pick the best between log-shipping and batch-copying.

In both workloads we compare the total data transfer volume for both centralized and distributed execution at varying rates of change to the base data in between runs of the analytics workload. Our workload optimizer consistently picks the best available solution at every operating point; we omit these lines to avoid crowding the figures.

Figure 3.5 shows results from both workloads. We note a few key observations.

The centralized approach performs best at low update rates. There is a small range in which it outperforms the distributed approach in both workloads, when the update rates are very low. This is because at low update rates we have frequent analytics running on mostly unchanged data. The distributed approach starts to pull ahead at higher update rates.

At low update rates, caching performs well, since when the data changes slowly there is a lot of redundancy in the query answers for caching to eliminate. This advantage disappears at higher update rates once data turnover becomes more rapid. At high update rates performance with caching converges to the same rate as performance with caching.

The cost of the centralized approach grows linearly with raw data growth (note that the slope is less than 1 because of data compression). In BigBench, the cost of the distributed approach also ultimately grows linearly, at a slower rate than the centralized approach. At the largest scale we tested, the distributed approach required $257 \times$ less data transfer than the centralized approach in BigBench. In the Voter benchmark, by contrast, the data transfer volume in the distributed deployment eventually flattens out, staying constant even as data volumes increase. This is because the analytics workload in the Voter benchmark computes fixed-size summaries of the base data, which the distributed approach is able to compute by retrieving (fixed-size) partial summaries from each individual data center, then combining them centrally. The performance gap between the distributed and centralized approaches in Voter will widen indefinitely (to infinity) as data
Figure 3.5: Geode-WF vs centralized baseline: Large-scale evaluation
3.3.2 Greedy heuristic performance

We now evaluate the performance of the greedy heuristic by comparing it against the more systematic ILP-based optimizer we developed for the SQL-only special case (§2.5.3).

We first compare the optimality gap between the two solutions in: (i) the real workloads from §2.6.1, and (ii) simulations on randomly generated SQL workloads.

In all the workloads we tested in §2.6.1, 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 [40] 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 optimality gap 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 $\text{SELECT * FROM } T_1 \text{ JOIN } T_2 \ldots \text{ JOIN } T_k \text{ USING}(\text{col})$, where each table has the schema $T_i(\text{col INT})$, $k$ chosen randomly between 2 and 10. In each query we chose table sizes and join selectivities according to a statistical model by Swami and Gupta [55], 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 3.6 shows the results we obtained. The greedy heuristic and the ILP identified the same strategy in around 16% of the queries. In the re-
Figure 3.6: Greedy heuristic vs ILP: Bandwidth cost ratio on 10k randomly generated SQL queries

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 3.7 shows the running times of both approaches for workloads of the same size as the largest in §2.6.1 and §3.3.1. 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 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 [55]. The question of how well both approaches will perform on practical workloads (beyond those in §2.6.1, 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.
3.3.3 Function-specific optimizations

We close by showing how the additional flexibility allowed by the more general computational model we support here can enable further optimizing the SQL workloads in §2.6.

In the Berkeley Big-Data Benchmark [34] we evaluated in §2.6.1, we found that basic distributed execution could only achieve a roughly $3 \times$ improvement over the centralized baseline. The bulk of the data transfer in this workload turns out to be from one particular top-k query, asking for the highest revenue generating users from a specified timeframe. Cao et al. [37] proposed an efficient distributed algorithm, TPUT, for the top-k problem. TPUT runs a complex three-phase distributed computation and cannot be implemented in a straightforward fashion using the limited MapReduce-like user-defined function API exposed by Geode-SQL. However, Geode-WF is much more flexible, allowing analysts to plug in any arbitrarily structured DAG workflow implementation for any given computation. We now use this functionality to implement TPUT and evaluate how it performs on this workload.

Figure 3.8 shows the results we obtained. TPUT achieves a substantial improvement: a further $8 \times$ data transfer cost reduction over basic distributed
Figure 3.8: Berkeley Big-Data Benchmark: Improving performance using a specialized top-k algorithm

execution, leading to a net $27 \times$ improvement compared to the centralized approach.

3.4 Related work

In §2.7 we discussed relevant prior work focusing on a relational computational model. We now survey past work targeting non-relational computation.

**Single-cluster scale-out platforms** Shared-nothing parallel databases, such as Netezza [56], Vertica [57] and Greenplum [58], began as a means to accommodate ever-increasing data in data warehouses. More recently, other flavors of scale-out systems have emerged with systems like Hive [32], Impala [59], Shark [60] and SciDB [61]. Unlike our setting, all these scale-out solutions control partitioning and are deployed in single clusters. In fact, our work is complementary: Geode-WF could leverage these systems as building blocks.

**Workflow management systems** Recently, various systems for richer workflow management have been proposed, e.g., Pig [51], Spark [62], Oozie [63],
Storm [64]. These systems are typically not geo-distributed and lack a sophisticated optimizer. Conversely, the need for distribution was identified early on in scientific workflow systems [65]. Pegasus [66] is a representative example, which tailors the execution of an abstract DAG to a specific Grid environment. Although relevant, Pegasus does not consider data replication during placement of tasks.

Query optimization Many works have considered query optimization in distributed databases [41]. R* [67] was among the first to add distribution to the traditional dynamic programming optimization algorithm. Recent work on multi-query optimization [68] also relates to our multi-DAG approach. RoPE [69] and DynO [70] gather runtime data statistics, but do not change the computation to facilitate extra measurements. Babu et al. addressed the optimization of MapReduce workflows [71, 72]. Their approach can be used to optimize our initial centralized plan, but does not consider our geo-distributed scenario. Overall, optimizing arbitrary DAGs remains a hard problem; no previous work has sufficiently addressed the mix of constraints and network-focus of our settings.

Data replication Most research in distributed databases focuses on static techniques for data replication [41]. Moreover, the problems of query optimization and data replication are tackled independently despite being interrelated. On the contrary, in Geode-WF, we tackle both problems at once. Data placement in the context of Pegasus was studied in [73], assuming prior knowledge of the workflows and placing replicas asynchronously before execution.

Other architectures Sensor networks share our assumption of expensive network bandwidth with respect to compute/storage [74]. The obvious differences in scale (one micro-controller vs. one data center), and the much broader computation model we assume, make most techniques from this space not directly applicable, though relevant as an inspiration. Likewise, stream-processing databases [49, 64] consider a more restrictive model than ours, in which data are always produced at edge nodes and are not replicated. Work in the CDN setting [28] has begun to address geo-distributed
analytics, but with much simpler data models.

3.5 Conclusion

We presented Geode-WF, a system supporting complex DAG workflows across geographically dispersed data. Geode-WF extends our core techniques from Geode-SQL, adapting them to work with the more general computational model we target here. The workload optimizer design, in particular, requires a substantial redesign to handle the greater flexibility afforded to analysts by the expanded computational model. We evaluate Geode-WF on realistic workloads and find that we continue to achieve similar levels of improvement as in the SQL setting, with a more than two orders of magnitude reduction in bandwidth cost compared to the centralized approach.
We now turn our attention to applications where latency, rather than bandwidth, is the driving concern.

Low latency is critical in user-facing applications. Even slightly higher web page load times can significantly reduce visits from users and revenue, as demonstrated by several sites [11]. For example, injecting just 400 milliseconds of artificial delay into Google search results caused the delayed users to perform 0.74% fewer searches after 4-6 weeks [10]. A 500 millisecond delay in the Bing search engine reduced revenue per user by 1.2%, or 4.3% with a 2-second delay [11]. Human-computer interaction studies similarly show that people react to small differences in the delay of operations (see [75] and references therein).

Achieving consistent low latency is challenging. Modern applications are highly distributed, and likely to get more so as cloud computing separates users from their data and computation. Moreover, application-level operations often require tens or hundreds of tasks to complete — due to many objects comprising a single web page [13], or aggregation of many back-end queries to produce a front-end result [76, 15]. This means individual tasks may have latency budgets on the order of a few milliseconds or tens of milliseconds, and the tail of the latency distribution is critical. Thus, latency is a difficult challenge for networked systems: How do we make the other side of the world feel like it is right here, even under exceptional conditions?

One powerful technique for reducing latency is to trade off for some amount of increased bandwidth consumption using redundancy: Initiate an operation multiple times, using as diverse resources as possible, and use the
first result which completes. For example, a host may query multiple DNS servers in parallel to resolve a name. The overall latency is the minimum of the delays across each instance, thus potentially reducing both the mean and the tail of the latency distribution. The power of this technique is that it reduces latency precisely under the most challenging conditions: when delays or failures are unpredictable.

Redundancy has been employed in several past networked systems: notably, as a way to deal with failures in DTNs [77], and in a multi-homed web proxy overlay [14]. But beyond these specific research projects, redundancy is typically eschewed across the Internet. We argue this is a missed opportunity. The combination of interactive applications, high latency, and variability of latency in the wide-area Internet make redundancy well suited to this environment. Even in a well-provisioned network where individual operations usually work, some amount of uncertainty is pervasive and the demand for consistent low latency outweighs the need to save bandwidth which is today comparatively cheap.

To support this argument, in §4.1 we examine the cost of redundancy. Since latency-bound tasks are likely to be small, the overall overhead is small when workloads are heavy-tailed; we show that flow-size distribution measurements confirm this. We next set a benchmark for identifying when redundancy is a net positive, despite the bandwidth overhead it adds, via an economic cost vs benefit analysis (§4.2). The analysis shows that redundancy may be cost-effective even in extremely conservative (e.g. cell phone) scenarios as long as it can save more than 10 milliseconds per kilobyte of added traffic. In §4.3, we show the benefits of replication can be orders of magnitude larger than this threshold in a number of common application scenarios. For example, querying multiple DNS servers can reduce the fraction of responses later than 500 ms by 6.5×, while the fraction later than 1.5 sec is reduced by 50×. We discuss applications to web browsing, multipath routing, TCP connection establishment, and quality of service.
4.1 Overhead can be low

In general, operations that require consistent low latency are likely to be small in the work they expend per operation. Consider, for example, a flow across the wide-area Internet: if the flow is significantly larger than the network’s delay-bandwidth product, the total time it needs to complete will be dominated by its throughput. When downloading a gigabyte movie which may have minutes of buffer, a few hundred milliseconds delay or loss of a few packets will likely go unnoticed. Alizadeh et al. [15] have also noted that in a number of data center applications, latency-critical jobs are small. In such systems, the flows that are the most likely to benefit from replication are also those that are the least expensive to replicate.

Furthermore, if latency-critical operations are small, then they will comprise a small fraction of the network’s total work if the distribution of sizes is heavy-tailed, which is a pervasive property in the Internet. Figure 4.1 shows the percentage increase in total network load that would result if the smallest $x\%$ of all flows were duplicated in four different settings: flows in a public [78] and a private [79] data center, web service requests crossing a university wide-area uplink [80], and an Internet backbone link [81]. In all

![Figure 4.1: Overhead of duplicating traffic](image)
cases it would be possible to duplicate at least the 33% smallest flows while increasing the total load by only 2%.

Several caveats may limit or expand the scenarios in which replication is useful. (1) In contrast to the general rule above, some latency-sensitive tasks are large, such as real-time video streams. Replication may still be useful if these streams have low rate (§4.3.4) or are critical. (2) If overhead is high, a replicated request could be marked as lower priority, so it can be dropped if it interferes with other work, similar to [82]. Alternately, one could replicate only the most important operations; for example, a cloud service provider could charge tenants a higher rate for consistent low latency. (3) Even if overhead is high, replication may be acceptable when the system is underutilized, as the wide-area Internet typically is. Thus, replication can reclaim value from otherwise unused resources.

4.2 Cost-benefit analysis: Developing a latency benchmark

What is the cost of the overhead added by redundancy, and when is it outweighed by the latency reduction achieved? We now develop an economic cost vs. benefit analysis to study these questions. We consider the trade-off between cost and benefit in wide-area client-server applications (such as web browsing, DNS queries, etc.) involving clients using consumer-level connectivity and service providers in the cloud. The framework we develop here serves as a baseline; it can be refined or extended for other systems. It can, indeed, be extended to reason about other techniques trading off bandwidth for latency as well, such as DNS prefetching [83], and so can be of independent interest outside of the redundancy evaluation we focus on here.

Our framework allows for various combinations of incentives at servers and clients. In the common scenario where both servers and clients care exclusively about their own benefit, we show that any technique that saves more than 10 ms of latency (in the mean or the tail, depending on the met-
ric we are concerned with) for every kilobyte of extra traffic that it sends is useful, even with very pessimistic estimates for the additional cost induced at both clients and servers. This is a conservative bound assuming the most expensive cost estimates we found; the threshold can be orders of magnitude lower in many realistic scenarios, such as when clients use DSL instead of cellular connectivity.

We develop an analytical framework for comparing the cost of and benefit from redundancy (§4.2.1) and then use this framework to derive a benchmark (§4.2.2). In the experimental evaluation (§4.3) we will demonstrate how the benchmark can be applied in practice.

4.2.1 Framework

Consider one of the forms of redundancy we evaluate in §4.3: contact multiple public DNS servers in parallel for each DNS request. Redundancy adds overhead both to the client, which sends out multiple copies of the same DNS request, and to the DNS infrastructure, which needs to service these duplicated requests. The corresponding benefit is the latency reduction achieved at the client, which also translates to an increase in expected ad revenue at the server [10]. Redundancy affects several entities, including clients, servers, and network operators. We account for the cost and benefit to all the stakeholders affected by redundancy by comparing the following five quantities:

- $\ell$ (ms/KB): the average latency savings achieved by redundancy, normalized by the volume of extra traffic it adds
- $p_s, p_c$ ($$/KB$$): the average price of processing extra traffic at the servers and the clients
- $v_s, v_c$ ($$/ms$$): the average value from latency improvement to the servers and the clients

We denote increased utilization in units of data transfer volume and measure added cost at the server and the client. Note however that these cal-
culated costs are a proxy for all the costs (not just bandwidth) incurred by all affected entities. For instance, network operator costs are accounted for via the bandwidth costs ISPs charge servers and clients, and CDN costs are accounted for via the usage fees paid by servers. One kilobyte of added client-side traffic in a web service has server-side costs including server utilization, energy, network operations staff, network usage fees, and so on. In essence, we amortize all these diverse costs over units of client- and server-side traffic.

From the perspective of a selfish client, any latency-saving technique is useful as long as the benefit it adds outweights the cost to the client: that is, \( \ell \times v_c \geq p_c \), or in other words

\[
\ell \geq \frac{p_c}{v_c}
\]

Similarly, a selfish server would need

\[
\ell \geq \frac{p_s}{v_s}
\]

We will require that both conditions be satisfied in the analysis in §4.2.2—that is, the benchmark we develop identifies latency-saving techniques that directly benefit both servers and clients. Other combinations are possible. For instance, a server might directly value both its own benefit as well as the improvement in user experience at the client, in which case we would need \( \ell \geq \max \{p_s/(v_s + v_c), p_c/v_c\} \). The analysis can be modified to account for whatever incentives are necessary in any given application scenario.

4.2.2 Analysis

Cost estimates. To estimate server-side cost \( p_s \), we use a range of advertised rates for cloud services which implement usage-based pricing, listed in the second column of Table 4.1. The most expansive (and expensive) of these is the first line, based on an Amazon Web Services sample customer
profile of a web application. The profile models a 3-tier auto-scalable web application, with a load balancer, two web servers, two app servers, a high-availability database server, 30 GB of storage, and other services, which utilizes 120 GB/month of data transfer out of EC2 and 300 GB/month out of CloudFront. The resulting amortized cost of $2.67 effectively models the cost (per transferred GB) of an average operation in this system, including the cost of all utilized services. The other services listed in the table model the cost of more limited operations, such as DNS or bandwidth alone.

On the client side, we limit this investigation to clients in which the dominant cost of incrementally added utilization is due to network bandwidth. Table 4.1 lists costs $p_c$ based on several types of connectivity. For these calculations, we assume a user who has paid for basic connectivity already, and calculate the cost of bandwidth from overage charges. Client-side bandwidth costs can be substantially higher than server-side total costs in extreme (cellular) cases but are comparable or cheaper with DSL connectivity.

Of course, there are scenarios which the above range of application costs does not model. For example, a cellular client whose battery is nearly empty may value energy more than bandwidth. But in a large class of situations, bandwidth is the most constrained resource on the client.

**Value estimates.** The value of time $v$ is more difficult to calculate, at both the client and server.

For the server, direct value may come from obtaining revenue (ads, sales). We consider the case of Google. A study by Google indicated that users experiencing an artificial 400 ms added delay on each search performed 0.74% fewer searches after 4-6 weeks [10]. Google’s revenue per search has been estimated at $0.0231; therefore, we can estimate a savings of 400 milliseconds on a single search generates, on average, an additional $0.0231 \cdot 0.0074$ in revenue, or $1.54 per hour of reduced latency. As another estimate, a 500

---

1. [http://calculator.s3.amazonaws.com/calc5.html#key=a-simple-3-tier-web-app](http://calculator.s3.amazonaws.com/calc5.html#key=a-simple-3-tier-web-app)
2. This is likely pessimistic since it includes, for example, the cost of increased storage which would not scale linearly with an increase in service operations.
millisecond delay in the Bing search engine reduced revenue per user by 1.2%, or 4.3% with a 2-second delay [11]. Using the latter (smaller) figure, combined with an estimated\(^4\) revenue per Bing search of $0.0314, we have a $2.43 per hour value. We use the more pessimistic Google value of $1.54/hr in our calculations.

On the client side, value may be obtained from a better or faster human experience. Among all the components of our analysis, this value is the hardest to estimate: it may be highly application-specific, and may depend on mean or tail latency in ways best quantified by a human user study of quality of experience. But as a first approximation, we assume the value of time is simply the US average earnings of $24.54 per hour in August 2014 [84], which implies \(v_c \approx 6.82 \cdot 10^{-6} \text{$/ms.} \)

**Finding the threshold.** We can now use our cost and value estimates to solve \(\ell \geq p/v\) to obtain the break-even point, in terms of the necessary latency savings per kilobyte of additional traffic.

Table 4.1 shows the break-even values of \(\ell\) for various scenarios. For example, the table indicates that a server replicating DNS traffic would obtain greater return in ad revenue than the cost of increased utilization with any latency-saving technique that saves more than 3.12 milliseconds per KB of traffic that it adds. The values are divided into four quadrants, one for each cost/benefit combination:

- \(p_s/v_s\) (upper-left quadrant): break-even \(\ell\) for a server making a selfish decision.
- \(p_c/v_c\) (lower-right): a client making a selfish decision.
- \(p_s/v_c\) (upper-right): a server that directly values a client’s quality of service.
- \(p_c/v_s\) (lower-left): a client that directly values the server’s benefit. This is unrealistic: a client would not typically value the server’s ad revenue yet ignore its own benefit.

\(^4\)http://www.trefis.com/company?hm=MSFT.trefis&driver=idMSFT.0817#
<table>
<thead>
<tr>
<th>Service plan</th>
<th>Cost $c$ ($/GB)</th>
<th>Break-even benefit $\ell$ (msec/KB), assuming...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>server-side value $v_s = $1.54/hr</td>
<td>client-side value $v_c = $24.54/hr</td>
</tr>
<tr>
<td><strong>Server-side plans</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon web services: “Common Customer” web app</td>
<td>2.67</td>
<td>5.95</td>
</tr>
<tr>
<td>Amazon Route 53 (DNS) assuming 0.5KB/query</td>
<td>1.40</td>
<td>3.12</td>
</tr>
<tr>
<td>Amazon CloudFront: U.S., 1 GB/mo, 1 KB/object</td>
<td>.91</td>
<td>2.03</td>
</tr>
<tr>
<td>Amazon EC2 and Microsoft Azure: bandwidth, Brazil</td>
<td>.25</td>
<td>.56</td>
</tr>
<tr>
<td>NearlyFreeSpeech.net: web hosting</td>
<td>.25</td>
<td>.56</td>
</tr>
<tr>
<td>Amazon CloudFront: U.S., 1 GB/mo, 10 KB/object</td>
<td>.20</td>
<td>.45</td>
</tr>
<tr>
<td>Amazon EC2 and Microsoft Azure: bandwidth, US</td>
<td>.12</td>
<td>.27</td>
</tr>
<tr>
<td>MaxCDN: based on “starter” plan overage fee</td>
<td>.08</td>
<td>.18</td>
</tr>
<tr>
<td>DreamHost: cloud storage, object delivery</td>
<td>.075</td>
<td>.17</td>
</tr>
<tr>
<td><strong>Client-side plans</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT&amp;T, low volume cell plan, based on overage fees</td>
<td>68.27</td>
<td>152.20</td>
</tr>
<tr>
<td>AT&amp;T, high volume cell plan, based on overage fees</td>
<td>15.00</td>
<td>33.44</td>
</tr>
<tr>
<td>O₂ mobile broadband, based on 1GB→2GB increment</td>
<td>8.02</td>
<td>17.88</td>
</tr>
<tr>
<td>AT&amp;T DSL</td>
<td>.20</td>
<td>.45</td>
</tr>
</tbody>
</table>

Table 4.1: Estimates of the cost of added utilization (in GB of data transfer), and resulting threshold benefit $\ell$ (in milliseconds saved per KB of added utilization) at which a technique becomes cost-effective. Based on providers’ publicly advertised prices as of August 2014, excluding taxes and fees.
Interestingly, excluding the last unrealistic scenario, both the server’s and
the client’s worst-case break-even benefit occurs in a similar range of 6-10
ms/KB; the client’s higher costs are roughly balanced by its greater benefit.
This analysis suggests that a given technique may be cost-effective even
in the most conservative cases as long as we can save more than ≈ 6-10
milliseconds (in the mean or tail, depending on the goal) for each kilobyte
of added traffic. Note that this is the worst-case value: the threshold can
be orders of magnitude lower in many realistic scenarios. For example, if
clients use DSL (instead of cellular) connectivity and servers use an external
web-host (instead of managing their own website on Amazon), the required
latency savings threshold drops to 0.25 milliseconds per KB of added traffic.

4.3 Experimental evaluation

4.3.1 Connection establishment

We start with a simple motivating example, demonstrating why replication
should be cost-effective even when the available choices are limited: we use
a back-of-the-envelope calculation to consider what happens when multiple
copies of TCP-handshake packets are sent on the same path. It is obvious
that this should help if all packet losses on the path are independent. In this
case, sending two back-to-back copies of a packet would reduce the proba-
ability of it being lost from \( p \) to \( p^2 \). In practice, of course, back-to-back packet
transmissions are likely to observe a correlated loss pattern. But Chan et
al. [85] measured a significant reduction in loss probability despite this cor-
relation. Sending back-to-back packet pairs between PlanetLab hosts, they
found that the average probability of individual packet loss was ≈ 0.0048,
and the probability of both packets in a back-to-back pair being dropped
was only ≈ 0.0007 – much larger than the \( \sim 10^{-6} \) that would be expected if
the losses were independent, but still 7× lower than the individual packet
loss rate.\(^5\)

As a concrete example, we quantify the improvement that this loss rate reduction would effect on the time required to complete a TCP handshake. The three packets in the handshake are ideal candidates for replication: they make up an insignificant fraction of the total traffic in the network, and there is a high penalty associated with their being lost (Linux and Windows use a 3 second initial timeout for SYN packets; OS X uses 1 second [86]). We use the loss probability statistics discussed above to estimate the expected latency savings on each handshake.

We consider an idealized network model. Whenever a packet is sent on the network, we assume it is delivered successfully after \(\frac{RTT}{2}\) seconds with probability \(1 - p\), and lost with probability \(p\). Packet deliveries are assumed to be independent of each other. \(p\) is 0.0048 when sending one copy of each packet, and 0.0007 when sending two copies of each packet. We also assume TCP behavior as in the Linux kernel: an initial timeout of 3 seconds for SYN and SYN-ACK packets and of \(3 \times RTT\) for ACK packets, and exponential backoff on packet loss [86].

With this model, it can be shown that duplicating all three packets in the handshake would reduce its expected completion time by approximately \((3 + 3 + 3 \times RTT) \times (4.8 - 0.7)\) ms, which is at least 25 ms. If we assume each packet if 50 bytes long, this implies a savings of around 170 ms/KB, which is more than an order of magnitude larger than the break-even latency savings we identified in §4.2. The benefit increases with \(RTT\), and is even higher in the tail: duplication would improve the 99.9th percentile handshake completion time by at least 880 ms, for a latency savings of around 6000 ms/KB.

### 4.3.2 DNS

An ideal candidate for replication is a service that involves small operations and which is replicated at multiple locations, thus providing diversity

\[^5\text{It might be possible to do even better by spacing the transmissions of the two packets in the pair a few milliseconds apart to reduce the correlation.}\]
across network paths and servers, so that replicated operations are quite independent. We believe opportunities to replicate queries to such services may arise both in the wide area and the data center. Here, we explore the case of replicating DNS queries.

We began with a list of 10 DNS servers and Alexa.com’s list of the top 1 million website names. At each of 15 PlanetLab nodes across the continental US, we ran a two-stage experiment: (1) Rank all 10 DNS servers in terms of mean response time, by repeatedly querying a random name at a random server. Note that this ranking is specific to each PlanetLab server. (2) Repeatedly pick a random name and perform a random one of 20 possible trials — either querying one of the ten individual DNS servers, or querying anywhere from 1 to 10 of the best servers in parallel (e.g. if sending 3 copies of the query, we send them to the top 3 DNS servers in the ranked list). In each of the two stages, we performed one trial every 5 seconds. We ran each stage for about a week at each of the 15 nodes. Any query which took more than 2 seconds was treated as lost, and counted as 2 sec when calculating mean response time.

---

6The default local DNS server, plus public servers from Level3, Google, Comodo, OpenDNS, DNS Advantage, Norton DNS, ScrubIT, OpenNIC, and SmartViper.
Figure 4.3: Reduction in DNS response time, averaged across 15 PlanetLab servers

Figure 4.4: Reduction in DNS response time, averaged across 15 PlanetLab servers, compared with best in retrospect in Stage 2
Figure 4.5 shows the distribution of query response times across all the PlanetLab nodes. The improvement is substantial, especially in the tail: Querying 10 DNS servers, the fraction of queries later than 500 ms is reduced by $6.5\times$, and the fraction later than 1.5 sec is reduced by $50\times$. Averaging over all PlanetLab nodes, Figure 4.3 shows the average percent reduction in response times compared to the best fixed DNS server identified in stage 1. We obtain a substantial reduction with just 2 DNS servers in all metrics, improving to 50-62% reduction with 10 servers. Finally, Figure 4.4 shows performance relative to the best single server in retrospect, i.e., the server with minimum mean response time for the queries to individual servers in Stage 2 of the experiment, since the best server may change over time. Even compared with this stringent baseline, we found a result similar to Fig. 4.3, with a reduction of 44-57% in the metrics when querying 10 DNS servers.

How many servers should one use? Figure 4.5 compares the marginal increase in latency savings from each extra server against the 10 ms/KB benchmark from §4.2. The results show that what we should do depends on the metric we care about. If we are only concerned with mean performance,
it does not make economic sense to contact any more than 5 DNS servers for each query, but if we care about the 99th percentile, then it is always useful to contact 10 or more DNS servers for every query. Note also that the absolute (as opposed to the marginal) latency savings is still worthwhile, even in the mean, if we contact 10 DNS servers for every query. The absolute mean latency savings from sending 10 copies of every query is 0.1 sec / 4500 extra bytes $\approx 23$ ms/KB, which is more than twice the break-even latency savings. And if the client costs are based on DSL rather than cell service, the above schemes are all more than $100 \times$ more cost-effective.

4.3.3 Using DNS redundancy to improve web browsing

We just showed that using redundant DNS requests can yield a significant improvement in raw DNS lookup times. We now measure the application-level impact of this technique, by quantifying the total improvement in web browser page load times when all DNS requests made during the page load are replicated.

At several Amazon EC2 nodes across the world, we ran a two-stage experiment similar to the one in the previous section. We started with a list of 10 DNS servers at each node, and then: (1) Ranked the DNS servers at each node in order of their average performance. (2) At different levels of DNS redundancy $r \in [1, 10]$, ran Google Chrome web browsing sessions in which our code would repeatedly either click on a link on the current web page with probability $p$, or load a new website chosen uniformly at random from the Alexa top-1000 list with probability $1 - p$. (We varied the value of $p$.) Figure 4.6 shows the results we obtained.

Figure 4.6a shows the percentage improvement in various metrics at $p = 0.0$ as a function of the number of DNS servers contacted for each DNS query. As the results show, the improvement increases with the level of redundancy, and with 10-way replication we see a 14% improvement in median browser page load time. Interestingly, the improvement is larger in the mean and median than it is in the tail. We believe this was because
(a) Fix $p = 0.0$, and vary level of redundancy

(b) Fix number of servers contacted per DNS query at 10, and vary $p$

(c) Normalized improvement in median as function of $p$ at different levels of DNS redundancy

Figure 4.6: Impact of DNS redundancy on browser-level web page load times
of the high performance variability in the small-sized EC2 VMs we used for our experiments, which led to the tail of the distribution (at every level of redundancy) being dominated by events where slow performance at the nodes caused the browser to freeze.

Next, Figure 4.6b shows results from varying the value of $p$ while fixing the level of redundancy at $r = 10$ (the optimal level according to Figure 4.6a). Overall, redundancy generally does better at lower values of $p$, with the best performance being observed at $p = 0.0$. This is not surprising: higher values of $p$ represent scenarios in which the user is likelier to keep clicking on links on the current page, meaning that at high $p$ the DNS prefetching mechanism that browsers use [83] is likelier to have already retrieved DNS results for the linked page. Redundancy thus functions as a complementary mechanism to DNS prefetching. Note also that redundancy still achieves a sizeable improvement even at high values of $p$; for instance, the improvement in the median is always at least 6%.

Finally, Figure 4.6c compares the normalized reduction in median latency against the 10 ms/KB benchmark from §4.2 at different values of $p$ and $r$. The results show that contacting two DNS servers per query is always cost-effective, but higher levels of redundancy may or may not be economically useful, depending on the value of $p$.

4.3.4 Multipath routing

We now consider replication of packets in a large-scale multipath routing setting. This setting may be available in a future Internet environment, but also mimics fairly closely (in topology and data rate) the Skype overlay network [87], where consistent low latency is beneficial.

We conducted experiments on three different overlay topologies, each consisting of a source and a destination node connected via 8 intermediate nodes, yielding 8 distinct end-to-end paths. We used two topologies spanning the US and one with the source and half the intermediate nodes in Europe and the destination and the remaining intermediate nodes in the
US. Most of these nodes were on PlanetLab, but in one topology we used exclusively reserved ProtoGENI nodes for the source and destination because the PlanetLab nodes we used at first with were too heavily loaded (see further discussion below). P2P applications such as Skype use similar overlay topologies [87], relaying traffic through intermediate nodes to circumvent NAT and firewall issues.

Over the course of roughly 48 hours, we sent UDP data packets at rates of \( r = 32 \text{kbit/s} \) and \( r = 56 \text{kbit/s} \) over the \( m \in \{1, 2, 3\} \) best paths at that point in time. The combinations of \( r \) and \( m \) were randomly chosen every minute. We used a simple-moving-average (SMA) with a window-size of 5 to rank the paths.

We observed a significant performance improvement when using replication, despite the variation in factors such as the topology, the data rate, and the shape of the latency distribution on each path. Figures 4.7 and 4.8 show that most or all of the improvement we observe comes from using two paths instead of one. While the improvement in the mean latency is relatively small, replication significantly improves the latency tail and reduces the packet loss probability. Compared to a single path we see better latencies starting at the 95th percentile, and with just one additional path the 99th percentile latency falls by about 60%.

The two data sending rates we tested, which are realistic for real-time audio traffic, yielded more or less identical behavior, but we expect that testing a larger range of rates will reveal that redundancy ceases to help when the data rate is sufficiently high. We investigate how data rates can affect redundancy in the data center evaluation in Chapter 5.

### 4.4 Related work

Redundancy is used pervasively to improve reliability, and in many systems to reduce latency. We now provide a general overview of prior uses of the technique. We discuss relevant work on using redundancy in the data center.
Figure 4.7: Multipath routing: Latency distribution in representative intra-US topology at $r = 56\text{kbit/s}$.

<table>
<thead>
<tr>
<th></th>
<th>mean latency</th>
<th>99% latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 path</td>
<td>2 paths</td>
</tr>
<tr>
<td>Intra-U.S. 1</td>
<td>103.5ms</td>
<td>7.28%</td>
</tr>
<tr>
<td>Intra-U.S. 2</td>
<td>86.0ms</td>
<td>6.63%</td>
</tr>
<tr>
<td>Trans-Atlantic</td>
<td>142.1ms</td>
<td>4.02%</td>
</tr>
</tbody>
</table>

Figure 4.8: Multipath routing: Measured RTTs in 56kbit/s experiment for 1 path and percent reduction when replicating over 2 or 3 paths (packets later than 4.5 seconds were considered lost)
center context in depth separately, in the next chapter (§5.3).

Within networking, replication has been explored to reduce latency in several specialized settings, including replicating DHT queries to multiple servers [82] and replicating transmissions (via erasure coding) to reduce delivery time and loss probability in delay-tolerant networks [77, 88]. Replication has also been suggested as a way of providing QoS prioritization and improving latency and loss performance in networks capable of redundancy elimination [89].

Andersen et al. [14]’s MONET system proxies web traffic through an overlay network formed out of multi-homed proxy servers. While the primary focus of [14] is on adapting quickly to changes in path performance, they replicate two specific subsets of their traffic: connection establishment requests to multiple servers are sent in parallel (while the first one to respond is used), and DNS queries are replicated to the local DNS server on each of the multi-homed proxy server’s interfaces. We show that replication can be useful in both these contexts even in the absence of path diversity: a significant performance benefit can be obtained by sending multiple copies of TCP SYNs to the same server on the same path, and by replicating DNS queries to multiple public servers over the same access link.

Most importantly, unlike all of the above work, our point is that replication is a general technique that can be applied in a variety of common wide-area Internet applications. We argue this point by studying the overhead associated with replicating small flows, the necessary benefit for end-users, and several use cases.

4.5 Conclusion

Low latency is critical for user-facing Internet applications. But while we know how to scale systems to increase capacity, controlling latency - especially the tail of the latency distribution - is much more difficult. We argued that the use of redundancy in the context of the wide-area Internet is an ef-
fective way to convert a small amount of extra capacity into reduced latency. By initiating redundant operations across diverse resources and using the first result which completes, redundancy improves a system's latency even under exceptional conditions. We demonstrated that redundancy can significantly reduce latency in a number of important applications, and used an economic analysis to argue that it is an effective general-purpose strategy even on devices like cell phones where bandwidth is relatively constrained.
Chapter 5
Trading bandwidth for latency in the data center

The bulk of this thesis has focused on wide-area Internet applications (Chapters 2, 3, 4). However, the form of redundancy we consider in Chapter 4 is a general purpose technique and can be applied outside of the Internet context as well. We now consider applications in the data center setting.

Controlling latency in a data center is a challenging proposition. Momentary spikes in latency can be caused by any of several unpredictable factors: disk slowdowns, virtualization delays, server overutilization, network congestion etc. Controlling for every possible source of high latency in a complex system is next to impossible. Further, many data center applications tend to be highly distributed in nature [13, 15], being structured as a combination of many small tasks. This means that a delay in any one task can have a cascading impact on the latency of the entire application. Thus a combination of factors — pervasive performance uncertainty, and an application structure that tends to magnify the effect of this uncertainty — mean that controlling latency, especially the tail of the latency distribution, is a very difficult problem.

We consider a simple, general technique for converting some amount of bandwidth and system overhead into a latency reduction: initiate multiple copies of latency-sensitive operations, ideally using distinct sets of resources, and take the first copy to complete. We saw in Chapter 4 that redundancy could achieve a significant latency improvement, far outweighing the overhead that it adds, in a number of wide-area Internet applications. We now evaluate how it performs in data center applications.

A key distinction in the data center setting is that data centers are typi-
cally *closed* systems, meaning the system operator has a much greater degree of control over how a mechanism such as redundancy is deployed and configured than in wide-area Internet applications. We start by developing and analyzing a queueing model to understand how redundancy can best be controlled in a closed data center-like environment. Our analysis provides a characterization of the system conditions under which redundancy should and should not be expected to be a net positive.

Next, we empirically evaluate how redundancy performs in two key applications: a distributed key-value storage system, and a multipath packet transport protocol. We show that redundancy achieves a significant net improvement in a range of realistic scenarios in both applications: up to $2 \times$ better mean and $8 \times$ better tail latency in the key-value store, and up to a 38% reduction in flow completion times in the data center network. We also find that our empirical results match the predictions from our theoretical model.

5.1 Analysis: Queueing analysis

Two factors are at play in a system with redundancy. Replication reduces latency by taking the faster of two (or more) options to complete, but it also worsens latency by increasing the overall utilization. In this section, we study the interaction between these two factors in an abstract queueing model.

We assume a set of $N$ independent, identical servers, each with the same service time distribution $S$. Requests arrive in the system according to a Poisson process, and $k$ copies are made of each arriving request and enqueued at $k$ of the $N$ servers, chosen uniformly at random. We will assume that $N$ is large enough compared to $k$ that the states of the queues at the individual servers can be assumed to be independent of each other. In simulations, when $k = 2$, we found that $N \approx 10$ was enough for this assumption to be tenable. We measure the system’s performance with *mean response time,*
Figure 5.1: A first example of the effect of replication, showing response times when service time distribution is deterministic and Pareto ($\alpha = 2.1$)
which is queueing delay plus service time.

This simple model assumes that replication is “free” for the clients — that it adds no appreciable penalty apart from an increase in server utilization. We consider the effect of client-side overhead at the end of this section.

Figures 5.1a and 5.1b show results from a simulation of this queueing model, measuring the mean response time as a function of load with two different service time distributions. Replication improves the mean, but provides the greatest benefit in the tail, for example reducing the 99.9th percentile by 5× under Pareto service times. Note the thresholding effect: in both systems, there is a threshold load below which redundancy always helps improve latency, but beyond which the extra load it adds overwhelms any latency reduction that it achieves. The threshold is higher — i.e., redundancy helps over a larger range of loads — when the service time distribution is more variable.

The threshold load, defined formally as the largest utilization below which replication always helps mean response time, will be our metric of interest in this section. Our goal is to understand how the service time distribution affects the threshold load.

The special case when the service times at each server are exponentially distributed is analytically tractable. A closed form expression for the response time CDF exists in this case, and it can be used to establish the following result.

**Theorem 1.** If the service times at every server are i.i.d. exponentially distributed, the threshold load is 33%.

**Proof.** Assume, without loss of generality, that the mean service time at each server is 1 second. Suppose requests arrive at a rate of \( \rho \) queries per second per server.

Without replication, each server evolves as an M/M/1 queue with departure rate 1 and arrival rate \( \rho \). The response time of each server is therefore exponentially distributed with rate \( 1 - \rho \)[90], and the mean response time is \( \frac{1}{1-\rho} \).
With replication, each server is an M/M/1 queue with departure rate 1 and arrival rate $2\rho$. The response time of each server is exponentially distributed with rate $1 - 2\rho$, but each query now takes the minimum of two independent samples from this distribution, so that the mean response time of each query is $\frac{1}{2(1-2\rho)}$.

Now replication results in a smaller response time if and only if $\frac{1}{2(1-2\rho)} < \frac{1}{1-\rho}$, i.e., when $\rho < \frac{1}{3}$.

In the general service time case, two natural (service-time independent) bounds on the threshold load exist.

First, the threshold load cannot exceed 50% load in any system. This is easy to see: if the base load is above 50%, replication would push total load above 100%. It turns out that this trivial upper bound is tight — there are families of heavy-tailed high-variance service times for which the threshold load goes arbitrarily close to 50%. See Figures 5.2 and 5.3.

Second, we intuitively expect replication to help more as the service time distribution becomes more variable. Figures 5.2, 5.3 and 5.4 validate this trend in three different families of distributions. Therefore, it is reason-
Figure 5.3: Pareto service times. Variance increases with $\beta$, and is infinite for $\beta > 0.5$.

Figure 5.4: A simple two-point discrete distribution: service time = 0.5 with probability $p$ and $\frac{1-0.5p}{1-p}$ with probability $(1 - p)$. Mean is always 1, and variance increase with $p$ and goes to infinity as $p \rightarrow 1$. 
able to expect that the worst-case for replication is when the service time is completely deterministic. However, even in this case the threshold load is strictly positive because there is still variability in the system due to the stochastic nature of the arrival process. With the Poisson arrivals that we assume, the threshold load with deterministic service time turns out to be slightly less than 25% — more precisely, \( \approx 24.93\% \) — based on simulations of the queueing model, as shown in the leftmost point in Figure 5.4.

We conjecture that this is, in fact, a lower bound on the threshold load in an arbitrary system.

**Conjecture 1.** Deterministic service time is the worst case for replication: there is no service time distribution in which the threshold load is below the (\( \approx 25\% \)) threshold when the service time is deterministic.

The primary difficulty in resolving the conjecture is that general response time distributions are hard to handle analytically, especially since in order to quantify the effect of taking the minimum of two samples we need to understand the shape of the entire distribution, not just its first few moments. However, we have two forms of evidence that seem to support this conjecture: analyses based on approximations to the response time distribution, and simulations of the queueing model.

The primary approximation that we use is a recent result by Myers and Vernon [91] that only depends on the first two moments of the service time distribution. The approximation seems to perform fairly well in numerical evaluations with light-tailed service time distributions, such as the Erlang and hyperexponential distributions (see Figure 2 in [91]), although no bounds on the approximation error are available. However, the authors note that the approximation is likely to be inappropriate when the service times are heavy tailed.

As a supplement, therefore, in the heavy-tailed case, we use an approximation by Olvera-Cravioto et al. [92] that is applicable when the service times are regularly varying\(^1\). Heavy-tail approximations are fairly well es-

\(^{1}\)The class of regularly varying distributions is an important subset of the class of heavy-
tablished in queueing theory (see [90]); the result due to Olvera-Cravioto et al. is, to the best of our knowledge, the most recent (and most accurate) refinement.

**Theorem 2.** Within the approximation due to Myers and Vernon [91] of the response time distribution, the threshold load is minimized when the service time distribution is deterministic.

**Proof.** Myers and Vernon [91] approximate the queue length distribution (which is a proxy for the response time distribution) by a geometric distribution. The geometric distribution is the discrete analog to the exponential distribution we used in Theorem 1, and a similar analysis can be used to derive closed form expressions for the mean response time with and without replication as a function of the total load and the variance of the service time distribution. Comparing these expressions shows that the threshold load has to be smallest when the service time variance is zero — that is, when the service time distribution is deterministic. □

The approximation by Olvera-Cravioto et al. [92] applies to arbitrary regularly varying service time distributions, but for our analysis we add an additional assumption requiring that the service time be sufficiently heavy. Formally, we require that the service time distribution have a higher coefficient of variation than the exponential distribution, which amounts to requiring that the tail index $\alpha$ be $< 1 + \sqrt{2}$. (The tail index is a measure of how heavy a distribution is: lower indices mean heavier tails.)

**Theorem 3.** Within the approximation due to Olvera-Cravioto et al. [92], if the service time distribution is regularly varying with tail index $\alpha < 1 + \sqrt{2}$, then the threshold load is $> 30\%$.

**Proof.** The approximation represents the response time distribution for any given load $\rho$ piecewise, as the combination of an Exponential(1) distribution up to a certain phase-change latency value $\phi(\alpha, \rho)$, and by a heavy-tailed tailed distributions that includes as its members the Pareto and the log-Gamma distributions.
distribution with tail index \( \alpha - 1 \) for all latencies larger than \( \phi(\alpha, \rho) \). Thresholding effects are easy to quantify in the second, heavy-tailed part of the distribution – for instance, replication always helps performance beyond the 99th percentile when the base load is < 44%, and it always helps beyond the 92nd percentile when the base load is < 30%. And when \( \alpha < 1 + \sqrt{2} \) a numerical calculation shows that the part of the distribution beyond the 92nd percentile is responsible for enough of the mass of the entire distribution that the improvement due to redundancy here far outweighs any added cost in the part of the distribution below the 92nd percentile.

Simulation results also support the conjecture. We generated a range of service time distributions by, for various values of \( N \), sampling from the space of all unit-mean discrete probability distributions with support \( \{1, 2, ..., N\} \) in two different ways — uniformly at random, and using a symmetric Dirichlet distribution with concentration parameter 0.1 (the Dirichlet distribution has a higher variance and generates a larger spread of distributions than uniform sampling). Figure 5.5 reports results when we generate a 1000 different random distributions for each value of \( N \) and look at the minimum and maximum observed threshold load over this set of samples.
Effect of client-side overhead

As we noted earlier, our analysis so far assumes that the client-side overhead (e.g. added CPU utilization, kernel processing, network overhead) involved in processing the replicated requests is negligible. This may not be the case when, for instance, the operations in question involve large file transfers or very quick memory accesses. In both cases, the client-side latency overhead involved in processing an additional replicated copy of a request would be comparable in magnitude to the server latency for processing the request. This overhead can partially or completely counteract the latency improvement due to redundancy. Figure 5.6 quantifies this effect by considering what happens when replication adds a fixed latency penalty to every request. These results indicate that the more variable distributions are more forgiving of overhead, but client side overhead must be at least somewhat smaller than mean request latency in order for replication to improve mean latency. This is not surprising, of course: if replication overhead equals mean latency, replication cannot improve mean latency for any service time distribution — though it may still improve the tail.
5.2 Experimental evaluation

5.2.1 Disk-backed database

Many data center applications involve the use of a large disk-based data store that is accessed via a smaller main-memory cache: examples include the Google AppEngine data store [93], Apache Cassandra [94], and Facebook’s Haystack image store [95]. In this section we study a representative implementation of such a storage service: a set of Apache web servers hosting a large collection of files, split across the servers via consistent hashing, with the Linux kernel managing a disk cache on each server.

We deploy a set of Apache servers and, using a light-weight memory-soaking process, adjust the memory usage on each server node so that about half the main memory is available for the Linux disk cache (the other half being used by other applications and the kernel). We then populate the servers with a collection of files whose total size is chosen to achieve a preset target cache-to-disk ratio. The files are partitioned across servers via consistent hashing, and two copies are stored of every file: if the primary is stored on server \( n \), the (replicated) secondary goes to server \( n + 1 \). We measure the response time when a set of client nodes generate requests according to identical Poisson processes. Each request downloads a file chosen uniformly at random from the entire collection. We only test read performance on a static data set; we do not consider writes or updates.

Figure 5.7 shows results for one particular web-server configuration, with

- Mean file size = 4 KB
- File size distribution = deterministic, 4 KB per file
- Cache:disk ratio = 0.1
- Server/client hardware = 4 servers and 10 clients, all identical single-core Emulab nodes with 3 GHz CPU, 2 GB RAM, gigabit network interfaces, and 10k RPM disks.
Figure 5.7: Base configuration

Figure 5.8: Mean file size 0.04 KB instead of 4 KB
Figure 5.9: Pareto file size distribution instead of deterministic

Figure 5.10: Cache:disk ratio 0.01 instead of 0.1. Higher variability because of the larger proportion of accesses hitting disk. Compared to Figure 5.7, 99.9th percentile improvement goes from $2.3\times$ to $2.8\times$ at 10% load, and from $2.2\times$ to $2.5\times$ at 20% load.
Figure 5.11: EC2 nodes instead of Emulab. x-axis shows unnormalised arrival rate because maximum throughput seems to fluctuate. Note the much larger tail improvement compared to Figure 5.7.

Figure 5.12: Mean file size 400 KB instead of 4 KB
Figure 5.13: Cache:disk ratio 2 instead of 0.1. Cache is large enough to store contents of entire disk

Figure 5.14: memcached
Disk is the bottleneck in the majority of our experiments — CPU and network usage are always well below peak capacity.

The threshold load (the maximum load below which replication always helps) is 30\% in this setup — within the 25-50\% range predicted by the queueing analysis. Redundancy reduces mean latency by 33\% at 10\% load and by 25\% at 20\% load. Most of the improvement comes from the tail. At 20\% load, for instance, replication cuts 99th percentile latency in half, from 150 ms to 75 ms, and reduces 99.9th percentile latency 2.2×.

The experiments in subsequent figures (Figures 5.8-5.13) vary one of the above configuration parameters at a time, keeping the others fixed. We note three observations.

First, as long as we ensure that file sizes continue to remain relatively small, changing the mean file size (Figure 5.8) or the shape of the file size distribution (Figure 5.9) does not significantly alter the level of improvement that we observe. This is because the primary bottleneck is the latency involved in locating the file on disk — when file sizes are small, the time needed to actually load the file from disk (which is what the specifics of the file size distribution affect) is negligible.

Second, as predicted in our queueing model (§5.1), increasing the variability in the system causes redundancy to perform better. We tried increasing variability in two different ways — increasing the proportion of access hitting disk by reducing the cache-to-disk ratio (Figure 5.10), and running on a public cloud (EC2) instead of dedicated hardware (Figure 5.11). The increase in improvement is relatively minor, although still noticeable, when we reduce the cache-to-disk ratio. The benefit is most visible in the tail: the 99.9th percentile latency improvement at 10\% load goes up from 2.3× in the base configuration to 2.8× when we use the smaller cache-to-disk ratio, and from 2.2× to 2.5× at 20\% load.

The improvement is rather more dramatic when going from Emulab to EC2. Redundancy cuts the mean response time at 10-20\% load on EC2 in half, from 12 ms to 6 ms (compare to the 1.3 − 1.5× reduction on Emulab). The tail improvement is even larger: on EC2, the 99.9th percentile latency at
10-20% load drops $8 \times$ when we use redundancy, from around 160 ms to 20 ms. It is noteworthy that the worst 0.1% of outliers with replication are quite close to the 12 ms mean without replication!

Third, as also predicted in §5.1, redundancy ceases to help when the client-side overhead due to replication is a significant fraction of the mean service time, as is the case when the file sizes are very large (Figure 5.12) or when the cache is large enough that all the files fit in memory (Figure 5.13). We study this second scenario more directly, using an in-memory distributed database, in the next section.

### 5.2.2 memcached

We run a similar experiment to the one in the previous section, except that we replace the filesystem store + Linux kernel cache + Apache web server interface setup with the memcached in-memory database. Figure 5.14 shows the observed response times in an Emulab deployment. The results show that replication seems to worsen overall performance at all the load levels we tested (10-90%).

To understand why, we test two versions of our code at a low (0.1%) load level: the “normal” version, as well as a version with the calls to memcached replaced with stubs, no-ops that return immediately. The performance of this stub version is an estimate of how much client-side latency is involved in processing a query.

Figure 5.15 shows that the client-side latency is non-trivial. Replication increases the mean response time in the stub version by 0.016 ms, which is 9% of the 0.18 ms mean service time. This is an underestimate of the true client-side overhead since the stub version, which doesn’t actually process queries, does not measure the network and kernel overhead involved in sending and receiving packets over the network.

The client-side latency overhead due to redundancy is thus at least 9% of the mean service time. Further, the service time distribution is not very variable: although there are outliers, more than 99.9% of the mass of the en-
Figure 5.15: memcached: stub and normal version response times at 0.1% load

tire distribution is within a factor of 4 of the mean. Figure 5.6 in §5.1 shows that when the service time distribution is completely deterministic, a client-side overhead greater than 3% of the mean service time is large enough to completely negate the response time reduction due to redundancy.

In our system, redundancy does not seem to have that absolute a negative effect – in the “normal” version of the code, redundancy still has a slightly positive effect overall at 0.1% load (Figure 5.15). This suggests that the threshold load is positive though small (it has to be smaller than 10%: Figure 5.14 shows that replication always worsens performance beyond 10% load).

5.2.3 Data center network

Replication has always added a non-zero amount of overhead in the systems we have considered so far (even if that overhead was mitigated by the response time reduction it achieved). We now consider a setting in which this overhead can be essentially eliminated: a network in which the switches are capable of strict prioritization.

Specifically, we consider a data center network. Many data center net-
work architectures [78, 15] provide multiple equal-length paths between each source-destination pair, and assign flows to paths based on a hash of the flow header [96]. However, simple static flow assignment interacts poorly with the highly skewed flow-size mix typical of data centers: the majority of the traffic volume in a data center comes from a small number of large elephant flows [97, 15], and hash-based flow assignment can lead to hotspots because of the possibility of assigning multiple elephant flows to the same link, which can result in significant congestion on that link. Recent work has proposed mitigating this problem by dynamically reassigning flows in response to hotspots, in either a centralized [98] or distributed [99] fashion.

We consider a simple alternative here: redundancy. Every switch replicates the first few packets of each flow along an alternate route, reducing the probability of collision with an elephant flow. Replicated packets are assigned a lower (strict) priority than the original packets, meaning they can never delay the original, unreplicated traffic in the network. Note that we could, in principle, replicate every packet — the performance when we do this can never be worse than without replication — but we do not since unnecessary replication can reduce the gains we achieve by increasing the amount of queueing within the replicated traffic. We replicate only the first few packets instead, with the aim of reducing the latency for short flows (the completion times of large flows depend on their aggregate throughput rather than individual per-packet latencies, so replication would be of little use).

We evaluate this scheme using an ns-3 simulation of a common 54-server three-layered fat-tree topology, with a full bisection-bandwidth fabric consisting of 45 6-port switches organized in 6 pods. We use a queue buffer size of 225 KB and vary the link capacity and delay. Flow arrivals are Poisson, and flow sizes are distributed according to a standard data center workload [79], with flow sizes varying from 1 KB to 3 MB and with more than 80% of the flows being less than 10 KB.

Figure 5.16 shows the completion times of flows smaller than 10 KB when
Figure 5.16: Median and tail completion times for flows smaller than 10 KB
we replicate the first 8 packets in every flow.

Figure 5.16a shows the reduction in the median flow completion time as a function of load for three different delay-bandwidth combinations (achieved by varying the latency and capacity of each link in the network). Note that in all three cases, the improvement is small at low loads, rises until load \( \approx 40\% \), and then starts to fall. This is because at very low loads, the congestion on the default path is small enough that replication does not add a significant benefit, while at very high loads, every path in the network is likely to be congested, meaning that replication again yields limited gain. We therefore obtain the largest improvement at intermediate loads.

Note also that the performance improvement we achieve falls as the delay-bandwidth product increases. This is because our gains come from the reduction in queuing delay when the replicated packets follow an alternate, less congested, route. At higher delay-bandwidth products, queueing delay makes up a smaller proportion of the total flow completion time, meaning that the total latency savings achieved is correspondingly smaller. At 40% network load, we obtain a 38% improvement in median flow completion time (0.29 ms vs. 0.18 ms) when we use 5 Gbps links with 2 us per-hop delay. The improvement falls to 33% (0.15 ms vs. 0.10 ms) with 10 Gbps links with 2 us per-hop delay, and further to 19% (0.21 ms vs. 0.17 ms) with 10 Gbps links with 6 us per-hop delay.

Next, Figure 5.16b shows the 99th percentile flow completion times for one particular delay-bandwidth combination. In general, we see a 10-20% reduction in the flow completion times, but at 70-80% load, the improvement spikes to 80-90%. The reason turns out to be timeout avoidance: at these load levels, the 99th percentile unreplicated flow faces a timeout, and thus has a completion time greater than the TCP minRTO, 10 ms. With redundancy, the number of flows that face timeouts reduces significantly, causing the 99th percentile flow completion time to be much smaller than 10 ms.

At loads higher than 80%, however, the number of flows facing timeouts is high even with redundancy, resulting in a narrowing of the performance
Finally, Figure 5.16c shows a CDF of the flow completion times at one particular load level. Note that the improvement in the mean and median is much larger than that in the tail. We believe this is because the high latencies in the tail occur at those instants of high congestion when most of the links along the flow’s default path are congested. Therefore, the replicated packets, which likely traverse some of the same links, do not fare significantly better.

Replication has a negligible impact on the elephant flows: it improved the mean completion time for flows larger than 1 MB by a statistically-insignificant 0.12%.

5.3 Related work

§4.4 in the previous chapter was a broad overview of prior work using redundancy in a number of different settings. In this section we detail directly relevant work in the data center environment.

Redundancy has been used to work around loss and latency uncertainty in task schedulers. Distributed job execution frameworks, for example, have used task replication to improve response time, both preemptively [100, 101] and to mitigate the impact of stragglers [102].

Dean and Barroso [103] discussed Google’s use of redundancy in various systems, including a storage service similar to the one we evaluated in §5.2.1, but they studied specific systems with capabilities that are not necessarily available in general (such as the ability to cancel outstanding partially-completed requests), and did not consider the effect the total system utilization could have on the efficacy of redundancy. In contrast, we thoroughly evaluate the effect of redundancy at a range of loads both in various configurations of a deployed system (§5.2.1, §5.2.2), and in a large space of synthetic scenarios in an abstract system model (§5.1).

Subsequent to our work [104, 16], there has been recent effort on pro-
viding theoretical characterizations of the effect of redundancy in specific systems, such as job execution frameworks [105] and distributed storage systems [106]. These analyses focus on specific systems with special capabilities such as job cancellation [105] and erasure coding [106], as opposed to the general model we consider in §5.1.

5.4 Conclusion

Achieving consistent low latency in data center applications is challenging: some amount of latency uncertainty is unavoidable in any large scale system. We showed how redundancy can be used to mitigate this uncertainty by trading off for some amount of bandwidth and system overhead. We studied an abstract queueing-theoretic characterization of the tradeoff between the latency reduction achieved by redundancy and the cost of the overhead it induces to demonstrate that redundancy should be expected to have a net positive impact in a large class of systems. We then confirmed empirically that redundancy offers a significant benefit in a number of realistic scenarios in key practical data center applications. We believe our results demonstrate that redundancy is a powerful technique that should be used much more commonly in networked systems than it currently is. Our results also will guide the judicious application of redundancy within only those cases where it is a win in terms of performance.
Chapter 6

Conclusion

Many Internet-bottlenecked applications place asymmetric demands on the network: they are primarily concerned with either minimizing their bandwidth consumption or achieving low latency, but not both. Our work exploits this asymmetry by developing techniques that trade off one of bandwidth and latency to improve the other. We showed how to trade off (a) latency for bandwidth in global-scale data analytics, an important problem, developing mechanisms operating at both a low/syntactic level and a high/semantic level, and (b) bandwidth for latency in a range of applications using redundant requests. Our evaluation showed that in a number of important applications — geo-distributed analytics, DNS, web browsing, a key-value store etc. — our techniques achieve a substantial improvement over currently deployed systems, orders of magnitude in many scenarios.

We now conclude with a brief discussion of ways in which our work could be further extended.

6.1 Future Work

Low latency global analytics. We focused on supporting bandwidth sensitive batch global data analytics in our work, but many organizations are also interested in solutions for latency sensitive streaming analytics systems [28], providing rapidly updated lightweight summaries of data as it is collected. It would be interesting to leverage our work on low latency techniques in the geo-distributed analytics context, to build a complete global data an-
alytics solution — a single system providing joint support for both complex data-intensive batch computation as well as lightweight time sensitive streaming computation.

**Improved analytics workload optimization.** The simulations in §3.3.2 predict that although the Geode workload optimizer is capable of tuning the system to near-optimal performance in today’s workloads, its efficacy could start to degrade if application structures become significantly more complex. The optimization problem we are faced with is difficult, combining as it does joint multi-query optimization (which is known to be a challenge even in centralized systems supporting purely relational workloads) with a geo-distributed setting, and extending existing approaches to query optimization, cardinality estimation, and view maintenance and selection to our more-than-relational geo-distributed environment will be a challenging direction for future work to pursue.

**Leveraging approximate computation.** Allowing the system to return inexact answers can significantly expand the gains our techniques can achieve. We saw a brief demonstration in §2.6.3, where we saw how deploying approximate percentile and count-distinct algorithms could reduce the bandwidth cost for solving these problems by orders of magnitude with minimal loss in accuracy. This only scratches the surface of the possibilities here, and significant further work is needed to understand how approximate computation can best be used to optimize performance while meeting the system’s latency and bandwidth objectives.

**More sophisticated forms of redundancy.** Our evaluation of redundancy studied the most basic form of the technique — initiate multiple copies of all operations, and let them all run to completion — and evaluated it in depth, across a range of system models and deployed applications. We leave it to future work to study more sophisticated forms of the technique, such as: initiating redundant requests after waiting for a timeout instead of immediately; tuning the level of redundancy in response to measurements of system conditions; or choosing the targets of redundant requests intelligently.
Beyond latency and bandwidth. We focused on bandwidth and latency requirements in this thesis, but there are other considerations applications are faced with. Consistency and fault-tolerance guarantees, for example, are a critical factor in data storage and retrieval applications, and prior work has shown how to navigate consistency/latency and fault-tolerance/bandwidth [107] tradeoffs in these systems. There is a rich space of problems that remain to be explored in the area of jointly optimizing over all the possible requirements applications can have — bandwidth, latency, sovereignty, fault-tolerance, consistency, privacy.
References


structure at facebook,” in Proceedings of the 2010 ACM SIGMOD
International Conference on Management of Data, ser. SIGMOD

[18] J. Lin and A. Kolcz, “Large-scale machine learning at Twitter,” in SIG-

com/white-papers/born-to-be-parallel-teradata-database.

2001. [Online]. Available: http://dx.doi.org/10.1109/90.974527

S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kan-
thak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle,
S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymniak, C. Taylor,
R. Wang, and D. Woodford, “Spanner: Google’s globally-distributed
database,” in 10th USENIX Symposium on Operating Systems Design
and Implementation (OSDI 12). Hollywood, CA: USENIX Association,

[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. Wiener and N. Boston, “Facebook’s top open data prob-
lems,” https://research.facebook.com/blog/1522692927972019/

infrastructure for data analytics at Twitter,” PVLDB, 2012.

1nPRLmNl, 2010.


