Parallelism-Optimizing Data Placement for Faster Data-Parallel Computations

Nirvik Baruah
Stanford University
nirvikb@stanford.edu

Peter Kraft
Stanford University
kraftp@stanford.edu

Fiodar Kazhamiaka
Stanford University
fiodar@stanford.edu

Peter Bailis
Stanford University
pbailis@cs.stanford.edu

Matei Zaharia
Stanford University
matei@cs.stanford.edu

ABSTRACT

Systems performing large data-parallel computations, including online analytical processing (OLAP) systems like Druid and search engines like Elasticsearch, are increasingly being used for business-critical real-time applications where providing low query latency is paramount. In this paper, we investigate an underexplored factor in the performance of data-parallel queries: their *parallelism*. We find that to minimize the tail latency of data-parallel queries, it is critical to place data such that the data items accessed by each individual query are spread across as many machines as possible so that each query can leverage the computational resources of as many machines as possible. To optimize parallelism and minimize tail latency in real systems, we develop a novel parallelism-optimizing data placement algorithm that defines a linearly-computable measure of query parallelism, uses it to frame data placement as an optimization problem, and leverages a new optimization problem partitioning technique to scale to large cluster sizes. We apply this algorithm to popular systems such as Solr and MongoDB and show that it reduces p99 latency by 7-64% on data-parallel workloads.

PVLDB Reference Format:
Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. Parallelism-Optimizing Data Placement for Faster Data-Parallel Computations. PVLDB, 16(4): 760 - 771, 2022.
doi:10.14778/3574245.3574260

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/stanford-futuredata/parallel-lb-simulator/tree/VLDB2023.

1 INTRODUCTION

Systems that perform highly parallel, latency-sensitive computations on data are increasingly popular. Examples of these systems include online analytical processing (OLAP) systems like Druid [32] and Clickhouse [7], search engines like ElasticSearch [9] and Solr [6], and many others. These systems are often used for real-time applications where performance is paramount, such as online fraud detection [3] or operational monitoring [2, 4].

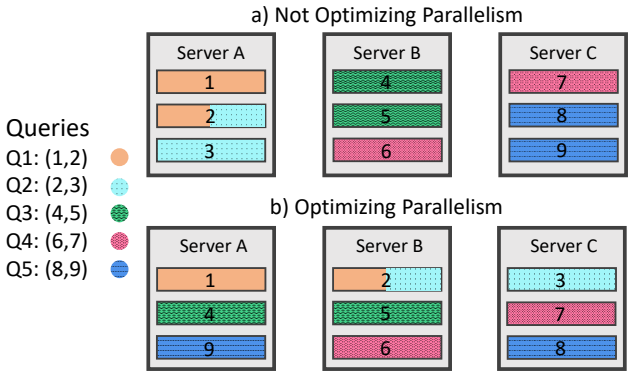


Figure 1: Each server contains three data shards colored by which queries access them. In (a), every query except Q4 accesses multiple shards on the same server, causing resource contention and high tail latency even though each server receives the same total amount of query load. In (b), all queries are processed in parallel across multiple servers, minimizing contention and tail latency.

Because low query latency is critical for many data-parallel systems, there has been much work on improving query performance. One area of focus is *data placement*. Data-parallel systems typically implement a shared-nothing architecture and horizontally partition data into many *shards*. Individual queries access data on many shards in parallel, but workloads are often heterogeneous so some shards (for example, shards storing more recent data) receive far more queries than others. Shards typically contain at least several gigabytes of data, so systems must "ship code to data" and run queries in-place on the shards they access. While it is well understood that data placement can affect query performance, most prior work only considers the importance of balancing query load across servers, leveraging techniques such as mathematical programming [31], bin packing [18], and various heuristic algorithms [21, 24, 26].

Our key insight in this paper is that even if the load is balanced in a data placement, performance can still be improved by optimizing *query parallelism*. We consider data-parallel workloads, which naturally apply the same operation to many data items typically spread over multiple shards. For example, in a Druid deployment where each shard stores one contiguous hour of data, a query for

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574260

how many events occurred in the last ten hours would access data on ten shards, running separate operations on each shard in parallel. To minimize the latency of these data-parallel queries, especially at the tail, systems should place shards that are frequently queried together (for example, shards containing consecutive time ranges) on different servers. Doing so maximizes the computational resources available to queries and minimizes resource contention by parallelizing work across as many servers as possible.

We illustrate the importance of optimizing parallelism in Figure 1. In this setup, clients issue five queries, each of which accesses data on two to three shards. In Figure 1a, load is balanced between all three servers, but shards queried together (e.g. shards 4-5, both of which are accessed by Q3) are often co-located. This can lead to high resource contention on individual servers; for example, if multiple clients issue Q3 at the same time, Server B will be momentarily overloaded and latency will spike. However, the data placement in Figure 1b optimizes query parallelism: no query accesses multiple shards on the same server, minimizing contention for computational resources.

While we are not aware of any prior work that studies the impact of data placement on query parallelism, some systems like Apache Druid [32] use domain-specific heuristics to place data in a parallelism-aware manner [1]. There has also been much research on optimizing query performance through data placement in transactional workloads. Systems such as Schism [15], Accordion [28] and Clay [29] optimize OLTP workload performance by *co-locating* data items that are frequently queried together, minimizing the number of distributed transactions. This minimizes transaction coordination costs in write-heavy OLTP workloads. However, it is counterproductive for data-parallel queries, as they are embarrassingly parallel and require minimal coordination, so their performance is maximized by parallelizing them across many servers.

In this paper, we propose Parallelism-Optimizing Data Placement (PODP), a novel data placement algorithm for optimizing query parallelism in data-parallel systems. Like prior work that focuses on load balancing or OLTP systems [28, 31] we formulate data placement as a mathematical optimization problem, specifically a mixed-integer linear program (MILP). However, adapting this approach to efficiently optimize query parallelism requires us to address two challenges. First, we must develop a linearly-computable measure of query parallelism to serve as an optimization objective. Second, we must mitigate the poor runtime performance scaling of MILP solvers, which has led prior systems to favor faster heuristic algorithms [31].

Formulating the optimization problem is difficult because we aim to minimize query tail latency by optimizing parallelism, but it is not obvious how to express either latency or parallelism as linear functions. Through experimentation, we found that worst-case latency is proportional to the maximum number of co-located shards accessed by a query, as under high load those shards may be accessed sequentially instead of in parallel. We call this the *clustering* of a query and define our MILP with a linear objective function that minimizes clustering for a given set of queries.

To scale our MILP approach to large systems, we adapt a new technique, partitioned optimization problems (POP) [27]. Instead of solving an optimization problem over the entire system, we partition it into sub-problems where each contains only a fraction of the system's shards and servers. Because solver runtime scaling is superlinear, we can solve all these sub-problems in seconds whereas solving a single large problem would take hours, while still computing a near-optimal data placement.

We evaluate PODP on several popular systems, including Solr [6] and MongoDB [5]. In an observational study, we find that differing levels of query parallelism explain as much as 82% of the variance in tail latency for parallel queries. Compared to several existing data placement strategies, PODP improves performance by 7-50% in uniform workloads where all shards are equally likely to be accessed, and by 54-64% in skewed workloads where some shards are accessed more frequently than others. We additionally find that PODP scales to large systems, computing data placements for systems with 600 servers and 6000 shards in <32 seconds.

In summary, our contributions are:

- We characterize the effects of query parallelism on performance in data-parallel systems. We show that query parallelism can explain as much as 82% of the variance in tail latency for highly parallel queries.
- We propose PODP, a novel data placement algorithm that uses mixed-integer linear programming to both balance load and optimize query parallelism. We show that it scales to large systems, running in <32 sec in a cluster with 6000 shards.
- We implement PODP and apply it to real systems such as Apache Solr and MongoDB, showing it reduces tail latency by 7-50% in uniform workloads and 54-64% in skewed workloads compared to several baselines.

2 PROBLEM STATEMENT AND EXPERIMENTAL EXPLORATION

In this paper, we study the impact of data placement on query tail latency. Specifically, we examine systems like Solr [6] and Druid [32] which are characterized by large data-parallel queries that access data partitioned across many *shards* stored on multiple servers. The question that interests us is: **given a set of queries, shards, and servers, how do we assign shards to servers to minimize query tail latency?**

We hypothesize that a major determinant of query tail latency is *query parallelism* and that queries perform worse when they access multiple shards co-located on the same servers. In the remainder of this section, we conduct several experiments testing this hypothesis. We find that query tail latency is significantly affected by the number of *n-clusters* in a system, where an n-cluster is a set of n shards on the same server that are accessed by a single query. We show that large numbers of n-clusters are correlated with query queuing and poor performance and that increasing query parallelism by breaking up n-clusters improves query tail latency.

Setup. In this section, we describe experiments run both in the popular full-text search system Apache Solr and in simulation. We describe our simulator in detail in Section 4.

In Solr, we use a simple query workload to demonstrate *how* parallelism-optimizing data placements reduce query latencies. We

run experiments on a cluster of five AWS EC2 m5d.xlarge servers, each with four cores. We use a workload adapted from the Lucene nightly benchmarks [25], storing 3M time-ordered Wikipedia documents and querying the documents in a time range for the number of matches to an exact search for the phrase "is also." We partition our dataset into 100 shards and set up our queries such that each accesses data in a time range stored on three consecutive shards. The performance impact of query parallelism is not unique to this workload and we show in Section 5 how PODP improves performance across a range of system parameters.

In the workload we have described, because all queries access consecutive shards, it is easy to see that placing consecutive shards round-robin across servers produces a parallelism-optimized data placement; in doing so, queries access up to one shard on each server as long as the number of servers exceeds the number of shards accessed. If there are M servers we obtain this setup by placing shard n on server $n \mod M$.

We compare this optimal setup to a range of Naive Load Balanced (NLB) data placements. NLB placements ensure that load is balanced across servers but do not consider query parallelism. In this workload, an NLB placement is one where shards are evenly divided between servers because every shard is equally likely to be queried. NLB placements are commonly used in existing systems [18, 22, 31] and provide a realistic baseline against which we can compare parallelism-optimizing data placements.

Experimental Exploration. In data-parallel systems, queries access data on several shards; each of these *shard accesses* are handled by the server on which the shard is stored. Workloads are often CPU-bound, so systems typically execute concurrent requests on multiple cores if resources are available. However, as system utilization increases and CPU resources become saturated, the server becomes unable to execute all pending requests simultaneously. As such, a queue of pending shard accesses builds up on the server. We refer to the number of pending shard accesses as the *queue size*.

We know that shard access latencies are tightly linked with the queue size of a server. Intuitively, this makes sense: if there are more shard accesses for a server to process, each shard access must wait a longer period of time before it is scheduled onto a processor. This in turn increases query latency, which is determined by the latency of the slowest shard access; that is, the end-to-end latency of a query depends on the largest queue of any server it accesses.

We postulate that large queues are especially likely to build up and increase tail latency if queries frequently access multiple shards on the same server. This is because a query accessing multiple shards on one server temporarily spikes its queue size, an effect that is compounded if multiple such queries are issued simultaneously. To investigate this, we measure in the simulator how the level of parallelism in data placement affects queue sizes, showing results in Figure 2. From this CDF, we can see that the worst-case queue sizes are much larger with NLB data placements than with parallelism-optimizing data placements. As we show later, this implies that parallelism-optimization significantly improves query tail latency.

We now quantify *how* the level of parallelism in data placement for a given workload affects queue size by investigating the number of *n-clusters* in a system. We define an n-cluster as any n shards ($n > 1$) in a server that are accessed by a single query; for example,

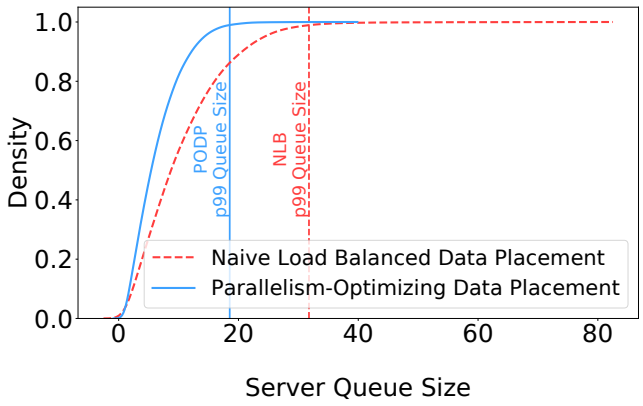


Figure 2: In simulator experiments, the CDF of server queue size using different data placement algorithms. Optimizing parallelism reduces the p99 queue size by 30%.

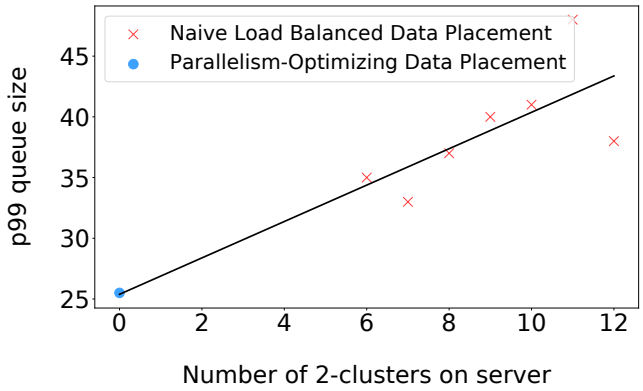


Figure 3: In simulator experiments, the relationship between the number of 2-clusters on a server and its p99 queue size. The parallelism-optimizing placement has no 2-clusters and a small worst-case queue size. NLB placements have a varying number of 2-clusters and a worst-case queue size which increases with the number of 2-clusters.

Server A in Figure 1a contains two 2-clusters: the shard set $\{1, 2\}$ (which are both accessed by Q1), and the shard set $\{2, 3\}$ (which are both accessed by Q2). Data placements that optimize query parallelism will minimize the number of n-clusters on servers; in contrast, systems that use NLB data placements do not consider n-clusters and may have many of them.

To investigate the link between n-clusters and query latency, we measure in simulation the relationship between the number of 2-clusters on a server and its p99 queue size, showing results in Figure 3. We look specifically at the number of 2-clusters per server because there are very few 3-clusters in our workload. This graph was created by simulating 150 server setups using both NLB and parallelism-optimizing data placements and running a query workload over each of them. We compute the p99 queue size over all of these server setups and plot this against the number of 2-clusters per server. As we can see, there are many points corresponding to

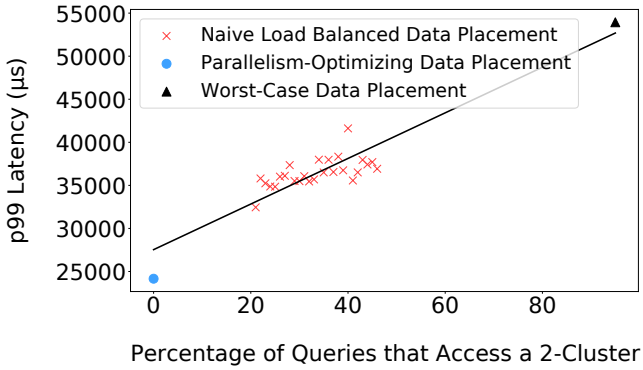


Figure 4: In Solr experiments, the relationship between the percentage of queries accessing a 2-cluster and p99 latency for different data placements. The parallelism-optimizing placement (•) contains no 2-clusters while the worst-case placement (▲) contains many 2-clusters.

NLB placements (×) because they do not consider query parallelism and may have any number of n-clusters; by contrast, the single parallelism-optimizing data placement (•) has no n-clusters. As we expect, there is a significant link between the number of shard clusters on a server and the worst-case queue size. This is because if queries with overlapping n-clusters are issued simultaneously, the servers hosting the shards containing those clusters will be momentarily overloaded, causing a spike in queue size.

We directly illustrate how n-cluster frequency impacts tail latencies in Figure 4, where we measure in Solr the relationship between the percentage of queries accessing 2-clusters and the p99 query latency of a workload. For this experiment, we began with a uniform workload and generated 25 NLB placements from random initial conditions. We also generated two special placements: a "best case" data placement that optimizes query parallelism, and a "worst case" data placement that optimizes the number of n-clusters. Across all these different data placements, we find a strong linear correlation between the proportion of queries accessing 2-clusters and the p99 query latency. Specifically, we compute an $r^2 = 0.82$, indicating that 82% of the variation in p99 latency is explained by the number of 2-clusters in our system. This demonstrates how optimizing query parallelism improves query tail latencies by reducing the number of n-clusters accessed by a workload.

One potential concern regarding parallelism-maximizing data placement is that, following a "tail at scale" argument [16], it may counter-intuitively *increase* tail latency because it increases the number of servers a query accesses and thus the probability a query may access a slow server. However, we find this effect is small because maximizing parallelism only slightly increases the number of shards that queries access compared to existing data placement techniques which do not account for query parallelism. For example, consider a query accessing 20 shards placed on 100 servers. If shards are placed without regard for query parallelism, this query accesses on average 18.2 servers with 1.8 n-clusters, but if parallelism is maximized, this query accesses 20 different servers with 0 n-clusters. While these few additional servers may slightly increase the chance of a straggler, parallelism-maximization nevertheless improves query tail latency by 7-64% in practice by eliminating n-clusters and reducing resource contention, as we show in Section 5.

Table 1: Notation used in Section 3, in order of introduction.

S	Set of all query shard sets.
c_s	Clustering of shard set s (§3.1).
f_s	Frequency of shard set s.
M	Number of data shards.
N	Number of servers.
r_{ij}	Percentage of queries for shard i to be sent to server j.
l_i	Query load on shard i.
L	Average server query load $\frac{\sum_i^M l_i}{N}$.
ϵ	Tolerance of load imbalance.
m_i	Memory usage of shard i.
C_j	Memory capacity of server j.
x_{ij}	Binary variable indicating server assignment, $x_{ij} = 1$ iff $r_{ij} > 0$.
t_{ij}	Binary variable, $t_{ij} = 0$ iff server i hosts shard j before assignment.
R	Minimum shard replication factor.

3 DATA PLACEMENT ALGORITHM

In this section we describe parallelism-optimizing data placement (PODP): a novel data placement algorithm for minimizing the latency of data-parallel computations. Developing an effective algorithm is difficult because it must efficiently optimize across multiple competing objectives: maximizing query parallelism, balancing query load between servers, and minimizing the amount of data that must be moved between servers to avoid overhead. Prior systems have formulated the data placement problem as a mixed-integer linear program (MILP) that balances query load while minimizing data movement [31], but do not consider query parallelism. To achieve all three objectives, we design PODP to execute in two stages. First, we optimize a novel linearly-computable measure of query parallelism, based on n-clusters, while keeping load balanced. Then, we minimize a measure of data movement while keeping load balanced and maintaining the optimal level of query parallelism. We additionally show how we use optimization problem partitioning [27] to scale the runtime of PODP to large cluster sizes.

3.1 Parallelism Metric

To measure the parallelism of a data placement for a given workload, we define a new metric called *clustering*: the clustering of a query is the size of the largest n-cluster it accesses. We define the shard set of a query as the set of all shards it accesses. A workload can be expressed as a list S of the most popular query shard sets s and their frequencies f_s over a recent interval. For a given shard set s, we define its clustering c_s as the maximum number of shards in the set that are placed on the same server; in other words, c_s is the size of the largest n-cluster in shard set s. For example, in Figure 1a, $c_s = 2$ for Q1 because both of the shards it accesses are on the same server; in 1b, $c_s = 1$. As discussed in Section 2, the worst-case clustering among popular queries is a faithful proxy for the query tail latency. Hence, minimizing the frequency-weighted sum of shard set clusters will improve query tail latency.

3.2 Problem Formulations

Suppose we have a cluster with M data shards assigned to N servers. The data placement algorithm constructs a shard-to-server assignment map r, where r_{ij} is the percentage of queries for shard i to be sent to server j; if $r_{ij} > 0$, server j hosts a copy of shard i. Queries are sent to servers randomly; a query for shard i may be sent to any server j where $r_{ij} > 0$ with probability r_{ij}. We define l_i as the query load on shard i, collected from the servers that host that shard. We define L as the average server query load: $L = \frac{\sum_i^M l_i}{N}$; server load is balanced if the load on each server is within a small tolerance ϵ of L. We define m_i as the memory usage of shard i and C_j as the memory capacity of server j. To model shard locations after assignment, we define a matrix x of binary variables indicating whether servers are assigned copies of shards; x_{ij} is 1 if $r_{ij} > 0$ and 0 otherwise. To model shard locations before assignment, we also define a matrix t where t_{ij} is 0 if server i currently hosts a replica of shard j and 1 otherwise. The total amount of shard movement is the sum of the element-wise product of t and x. We define R to be the minimum shard replication factor (for redundancy). For convenience, we summarize all notation in Table 1.

PODP works by solving two optimization problems, first optimizing query parallelism and then optimizing data movement.

Objective 1: Optimizing Parallelism. The problem formulation for minimizing clustering, P_c, is as follows. Given M data shards, N servers, ϵ load tolerance, shard loads (l_i), shard memory usages (m_i), server memory capacities (C_j), shard-server map (t_{ij}), replication factor R, and the set of all query shard sets S and their corresponding frequencies (f_s), minimize query clustering weighted by query frequency subject to the following constraints:

$$\min_{c_s,r,x} \sum_{s \in S} c_s f_s \tag{1}$$

subject to:

$$\forall j, L - \epsilon \leq \sum_i^M r_{ij} l_i \leq L + \epsilon \tag{2}$$

$$\forall i, \sum_j^N r_{ij} = 1 \tag{3}$$

$$\forall j, \sum_i^M x_{ij} m_i \leq C_j \tag{4}$$

$$\forall ij, x_{ij} \geq r_{ij} \tag{5}$$

$$\forall ij, x_{ij} < r_{ij} + 1 \tag{6}$$

$$\forall i, \sum_j^N x_{ij} \geq R \tag{7}$$

$$\forall j, \sum_{i \in s_i} x_{ij} \leq c_s \tag{8}$$

Constraint (2) ensures that the load on each server is balanced with some tolerance ϵ. Constraint (3) ensures that all queries are assigned to a shard. Constraint (4) ensures that server memory capacities are respected. Constraints (5) and (6) ensure $x_{i,j}$ and $r_{i,j}$ are consistent. Constraint (7) ensures that the minimum replication factor is respected. Finally, Constraint (8) defines the clustering of query s as the largest number of shards in its shard set co-located on a server.

Objective 2: Optimizing Data Movement. Solving P_c, we obtain the optimal values of c_s, which are used as constraints by P_{bal} to compute the final shard-to-server assignment that minimizes shard movement. P_{bal} is formulated as follows. Given M data shards, N servers, ϵ load tolerance, shard loads (l_i), shard memory usages (m_i), server memory capacities (C_j), shard-sever map (t_{ij}), replication factor R, and the optimal clustering values c_s computed by P_k, minimize shard movement subject to the following constraints:

$$\min_{r,x} \sum_i^M \sum_j^N t_{ij} x_{ij} \tag{9}$$

subject to:

$$\forall s \in S, \forall j, \sum_i^M x_{ij} s_i \leq c_s \tag{10}$$

Constraints 2-7

Constraint (10) ensures that the level of parallelism supported by the solution to P_{bal} is equal to that found by P_c.

3.3 Scaling to Large Systems

MILP approaches like PODP often have trouble scaling to large systems because solver runtime is worst-case exponential in the number of variables in the problem, which is $O(NM)$ for both Objectives 1 and 2. Recent work [27] has shown that, in some systems, resource allocation problems can be solved quickly with a technique called Partitioned Optimization Problems (POP): partitioning a large problem into smaller sub-problems to greatly reduce solver runtime. For example, we can split a PODP problem into many sub-problems, each containing a fraction of available shards and servers. If we have P partitions, this reduces the number of variables in each MILP problem to $O(NM/P^2)$.

Applying POP to PODP requires a valid partitioning function where each partition contains an equal fraction of servers and of query load; otherwise, solutions to the partitioned problem could violate the load balancing constraint of the unpartitioned problem. While it is easy to create an initial valid partitioning, it is not obvious how to maintain it in a long-running system where query loads change over time and a partitioning valid in one round of data placement may not be valid in future rounds. If we re-partition each round of data placement naively, we will frequently move shards between partitions and introduce unnecessary data movement. To circumvent this, we make partitions *sticky*: in each round, we greedily attempt to assign shards – in ascending order by load – to the partition to which they were assigned in the previous round. If a partition is fully loaded, we replicate or transfer its shards to underloaded partitions.

We have analyzed the sticky partition heuristic and determined an upper bound on the number of shard transfers it causes. For P partitions, we define δ as the largest magnitude of the net load change across all shards in any single partition. Define M as the smallest number such that each partition has M shards whose total

load is greater than δ. Our greedy algorithm causes at most $P * M$ shards to be transferred between partitions.

Our goal is to *balance* the load of each partition: to replicate or transfer shards to or from that partition so that every partition has the same query load. Initially, each shard is assigned to exactly one partition. For any partition A with a load surplus and partition B with a load deficit, replicating or transferring at most M shards from A to B is guaranteed to balance the load of either A or B (depending on whether the surplus on A is larger than the deficit on B). Thus, each of the P partitions can be balanced in at most M shard transfers and the total number of shard transfers needed to balance all partitions is at most $P * M$. In practice, we expect both M and P to be small compared to the total number of shards.

As we show in Section 5, by partitioning PODP into sub-problems we can compute data placements for large systems >100× faster with a negligible impact on query performance. We find that as long as the number of sub-problems is small compared to the number of servers, increasing the number of sub-problems does not have a significant effect on query tail latency.

4 IMPLEMENTATION

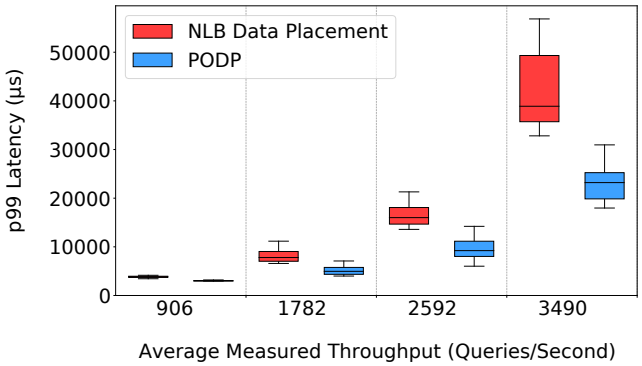
We implement PODP in Java using the CPLEX solver [8]. To run experiments, we integrate PODP into Uniserve [22], a general distribution layer for systems serving data-parallel, low-latency queries. We use PODP to replace Uniserve's default data placement algorithm, which balances load but does not account for query parallelism. We run experiments with the Uniserve ports of the full-text search system Apache Solr and the NoSQL document database MongoDB, both of which are described in detail in the original paper [22]. We make no other changes to Uniserve or its ports.

Data-Parallel System Simulator. To build intuition for how PODP performs and to model PODP performance on extremely large clusters (specifically in Figures 14 and 16), we implement a simulator in Java that emulates the behavior of a system serving data-parallel queries on a set of multi-core servers. Our simulator represents queries as sets of shard accesses. Each shard access is issued to a server and requires a set number of *ticks* (simulated discrete units of time) to complete. In each round of computation, every server in our simulated setup retrieves a shard access from a queue of pending shard accesses and decrements its remaining ticks by one, simulating it being scheduled onto a processor. To model multi-core servers, we may retrieve and decrement multiple shard accesses per server per tick. As a sanity check, we compare simulator performance to Solr performance on benchmark workloads and find they are similar, showing results in Figure 5.

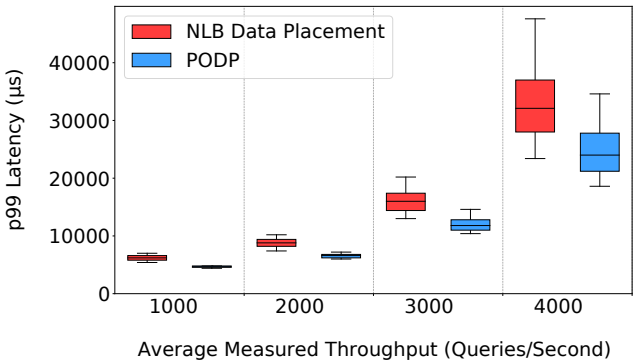
5 EVALUATION

We evaluate PODP using a variety of benchmarks simulating data-parallel query workloads. Our evaluation shows that:

- Using PODP to place data in a data-parallel system improves query tail latency by 7%-64% compared to baseline data placement strategies.
- The performance benefits of PODP are robust to varying conditions, including workloads with varying query size (Figure 11), workloads where PODP does not have accurate information on some shards (Figure 12), and workloads with non-uniform shards (Figure 13).
- PODP scales to large systems, computing data placements for thousands of shards and servers in seconds while consistently improving query tail latency relative to baselines (Figures 14-16).



(a) In Apache Solr



(b) In Simulation

Figure 5: Throughput vs. p99 latency in Apache Solr (a) and our simulator (b). We find simulator and Solr performance are similar.

5.1 Experimental Setup

We run experiments in three systems: Apache Solr, MongoDB, and our simulator.

Most of our experiments are run in Apache Solr on AWS EC2 using a cluster of m5d.xlarge instances, each with four CPUs, 16 GB of RAM, and an attached SSD. We also run experiments in MongoDB with a similar setup. Additionally, scalability experiments were run using our simulator (Section 4).

For each experiment, we compare the parallelism-optimizing data placement generated using PODP to three baseline approaches. The first baseline is the data placement algorithm used by Getafix [18], which places shards using a best-fit bin packing approach to balance load across servers while minimizing replicas. Our second baseline, Getafix+, augments Getafix with a heuristic used in Druid to account for query parallelism [1]. Specifically, it calculates the *cost* of assigning a shard to a server as the likelihood

that any two shards on the same server will be scanned together, then greedily assigns shards to servers with minimal cost. The final baseline, naive load balancing (NLB), is generated by solving a MILP that minimizes shard transfers while balancing load, but does not account for query parallelism; this closely resembles techniques used in existing systems [22, 31]. We use the same data sets and shard sizes for PODP and for all three baselines. We configure PODP and all three baselines to have a minimum replication factor of 1.

The data placements generated using all of these algorithms are sensitive to the initial placement of shards, so it is not possible to characterize their performance in a single trial. NLB placements are especially sensitive because their level of parallelism varies greatly across trials. Therefore, for each experiment, we run many trials. Each trial begins with random initial data placement, uses either NLB, Getafix, Getafix+, or PODP to generate a data placement, and then runs a query workload. We plot the distributions of p99 query latencies over these trials in box-and-whisker plots. The lower and upper whiskers of each box represent the 5th and 95th percentile p99 latencies, and the lower and upper boundaries of each box represent the first and third quartile p99 latencies.

5.2 Experiment Workloads

We evaluate each system with a different workload. Unless otherwise stated, each workload is run on a system with 100 shards and 5 servers, with an additional server acting as the system coordinator, and each query accesses 3 shards.

For Apache Solr, we use queries from the Lucene nightly benchmarks [25]. Each query is run on a dataset of 3M Wikipedia documents. This dataset is partitioned into shards by time range, and each query searches through a specified time range (and thus a particular set of shards) for the exact phrase "is also".

We benchmark MongoDB using YCSB [14], to simulate an analytics workload. Before running the workload, we insert 10M sequential items (10GB of data) into the database. We run a workload of 100% scans, where each scan retrieves 30,000 items.

Our simulator models a distributed system running data-parallel queries on many four-core servers. We run a synthetic time-series workload over this simulated server setup where queries access data stored on consecutive shards. The simulator is discussed in more detail in Section 4.

5.3 End-to-end Benchmarks

We first benchmark the performance impact of PODP in Solr and MongoDB. We issue queries asynchronously following a Poisson distribution. For each data placement algorithm, we run queries on 50 randomly-initialized data placements and plot the distribution of p99 latencies in a box-and-whisker plot.

Uniform Workload in Solr. We first measure how p99 query latency changes as we vary offered load, showing results in Figure 6. We find that as offered load increases, the p99 latency improvement from parallelism optimization increases from 7% – 24% to nearly 33% – 50%. At low utilization levels, we do not expect a significant difference in performance because there is not enough resource contention for parallelism optimization to affect performance. This is reflected in Figure 6 by the small differences in p99 latency at

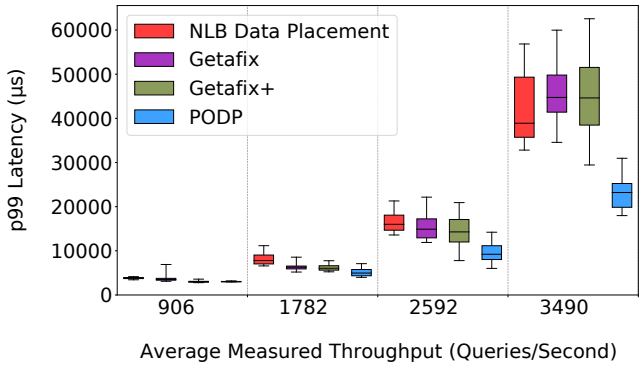


Figure 6: Measured throughput vs. p99 latency in Apache Solr. PODP significantly reduces latency and provides larger benefits at higher throughputs.

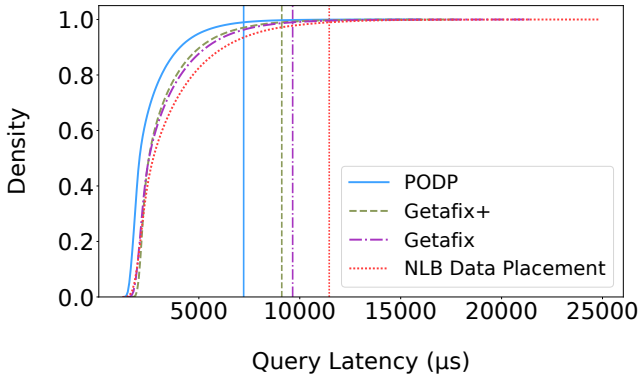


Figure 7: Query latency CDF in Solr. PODP substantially improves p99 query latency, as indicated by the vertical lines.

low offered throughputs. However, as offered load increases, resource contention in servers increases, magnifying the importance of parallelism optimization.

To further investigate the effect of parallelism optimization on latency, we run 200,000 queries in Solr at an offered throughput of 2000 QPS and plot a CDF of their latencies, showing results in Figure 7. We find that PODP substantially improves tail latency and has a smaller, though still positive, effect on median latency. This follows from our analysis in Section 2: PODP reduces maximum server queue size, which improves tail latency but should have a smaller effect on median latency

Skewed Workload in Solr. In our next set of experiments, we fix throughput at 3000 QPS and evaluate the effect of *skewness* on the performance of data placement algorithms. We vary skewness by modifying how likely the first 20 shards are to be queried relative to the other 80 shards. This setup simulates a hotspot, where some data items (e.g. more recent items) are far more likely to be queried than others.

We show the effects of workload skewness on query tail latency in Figure 8. As we can see, parallelism-optimizing data placements provide p99 latency reductions of 54%-64%, increasing slightly with

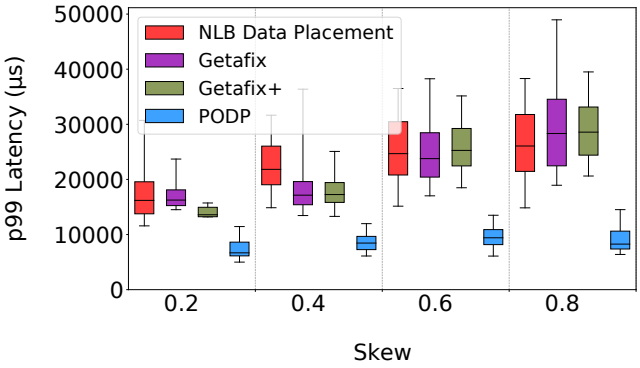


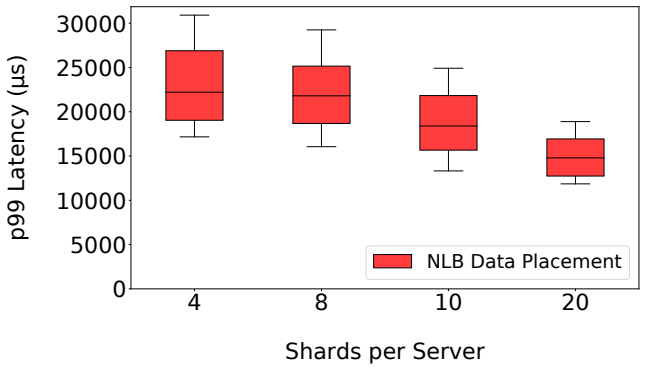
Figure 8: Query skewness vs. p99 latency in Apache Solr. Skewness is defined as the percentage of queries that access the "hot" 20% of shards in our system. PODP significantly reduces latency and provides larger benefits at higher skewness.

skew. Additionally, they have a lower variance in p99 latency compared to the baselines. This is because, as we will show, skewed workloads amplify the performance impacts of queries on a smaller number of shards, increasing tail latencies in setups that do not maximize parallelism.

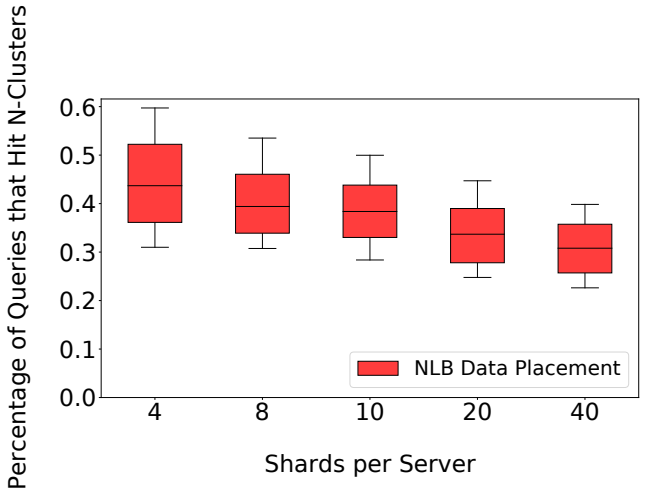
When skewness is high, the placement of a small number of shards has an outsized impact on query tail latencies since the majority of accesses are to those shards. For example, in our experiments, when skewness is 0.8, this means that 20 shards receive 80% of shard accesses so the placement of these 20 shards is especially important for optimizing query performance.

To demonstrate the importance of this effect, we run a new experiment using NLB data placements where we vary the number of shards per server while proportionally varying offered throughput to hold constant the rate of shards accessed per server. We find that median latency remains unchanged, but as shown in Figure 9a, tail latency increases as the number of shards per server decrease. We further explore this effect in the same setup in Figure 9b, finding that as the number of shards per server decreases, the percentage of queries accessing n-clusters increases. These findings are consistent with our skewness experiment; optimizing parallelism is especially important when the placement of a small number of shards dominates system performance.

MongoDB Benchmarks. We next evaluate the effects of data placement on a YCSB workload in MongoDB. As in our Solr experiments, we vary offered throughput and graph the p99 latency of queries across a range of cluster setups, showing results in Figure 10. Due to the large amount of data accessed by each query in our scan-heavy YCSB workload, query tail latencies are much higher and offered throughputs are much lower than in our Solr experiments. Nevertheless, we find PODP provides similar tail latency improvements of 17%-36%, demonstrating that it consistently improves performance across a range of systems.



a) Shards per Server vs. Tail Latency



b) Shards per Server vs. N-Cluster Proportion

Figure 9: Graph from Solr showing NLB data placements of the number of shards per server vs. p99 latency of queries (a) and the percentage of queries that access n-clusters (b). p99 latency is higher in setups with fewer shards because queries are more likely to access n-clusters.

5.4 Robustness Benchmarks

We now run a set of microbenchmarks evaluating how the performance impact of data placement is affected by conditions such as varying query size, missing information on shards, non-uniform shard size, and system scale.

Query Size in Solr. In this experiment, we measure how the performance impact of data placement is affected by varying query size, finding that PODP improves performance even for workloads that scan large portions of a dataset. We fix offered load at 3000 queries/second and vary how many shards each query accesses. Each shard contains 30,000 documents. We graph how query tail latency changes with increasing query size in Figure 11. As we can see, parallelism-optimizing data placements provide consistent p99 latency reductions of around $24\% - 39\%$ regardless of query sizes.

Withholding Information from PODP. We now investigate how PODP performs when it has incomplete information on queries.

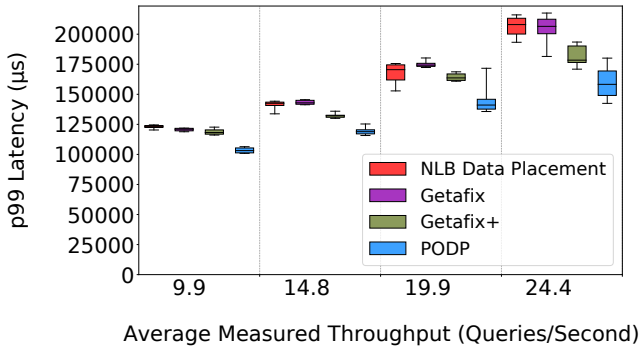


Figure 10: Measured throughput vs. p99 latency in MongoDB. PODP provides consistent performance improvements.

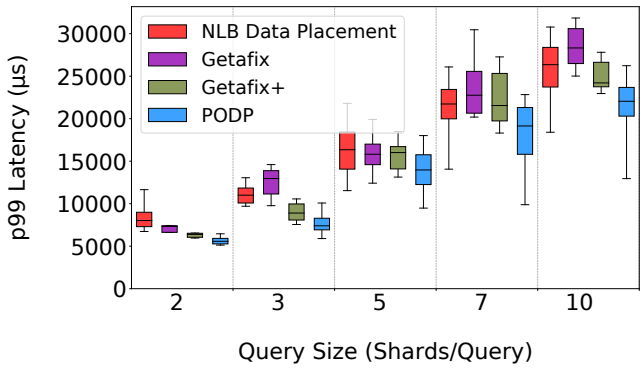


Figure 11: Number of shards per query vs. p99 latency in Apache Solr. PODP provides consistent performance improvements across a range of query sizes.

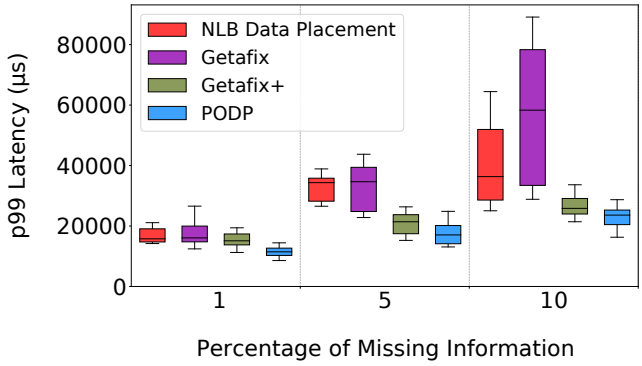


Figure 12: Percentage of shard information withheld from the data placement algorithm vs. p99 latency in Apache Solr. PODP performance degrades gracefully when it is missing information, consistently outperforming all three baselines.

We specifically examine the case where new shards are added to the system and it has no information on how they are queried. We model this by withholding information on a fraction of shards from the data placement algorithm (PODP or the baselines). We vary this fraction from 1% – 10%, where a fraction of 5% means that we withhold information on 5% of shards from the data placement algorithm, so it assumes those shards receive zero load and are not part of any query's shard sets. We show results in Figure 12 and find that PODP performance degrades gracefully when it is missing information, consistently outperforming all three baselines.

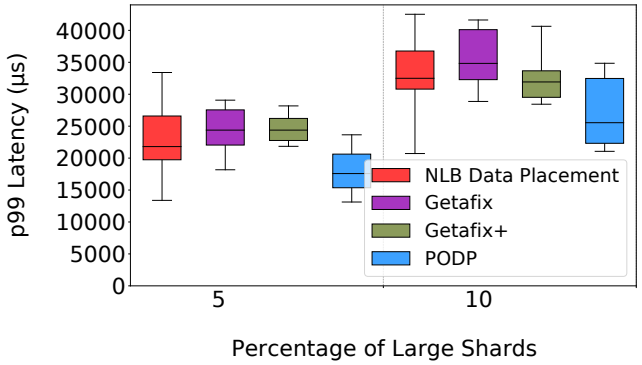


Figure 13: Percentage of expensive-to-query (double-sized) shards vs p99 latency in Solr. PODP performance improvements remain consistent despite variance in shard scan times.

Non-Uniform Shard Scan Time. Next, we investigate how PODP performs when shard scan times are non-uniform, for example because queries access different amounts of data in different shards. We model this by making a small number of shards hold double the number of documents as the others, thus doubling the scan time for any queries which access those shards. We vary the fraction of large shards and measure query tail latency, showing results in Figure 13. We find that PODP consistently outperforms all three baselines with different fractions of large shards.

Scalability. Finally, we evaluate how the performance impact of parallelism optimization changes as we scale the size of our system. We run this experiment in simulation so we can experiment with very large cluster sizes. We vary the number of servers in our system setup from 50 to 1000. Every server stores 10 shards. We scale the size of queries such that every query accesses 1% of the shards.

In order to efficiently solve the MILP, we partition it into smaller sub-problems (as discussed in Section 3) which can be solved quickly in parallel. The number of sub-problems is set to the number of servers divided by the number of shards accessed per query.

We show the effects of system size on query tail latencies in Figure 14. The performance improvements provided by PODP are consistent regardless of system size. PODP provides p99 latency reductions of 22% – 35% across a range of system sizes.

We now investigate the performance impact of partitioning PODP into sub-problems. In Figure 15, we graph how the number of sub-problems used by PODP affects end-to-end optimization time at different problem scales. We measure optimization time on an AWS EC2 m5d.xlarge instance with four CPUs, 16 GB of RAM, and an attached SSD. As this figure shows, using POP to split our optimization problem into smaller sub-problems allows us to place shards on significantly larger systems in a fraction of the time. Importantly, we find that for any number of servers there exists

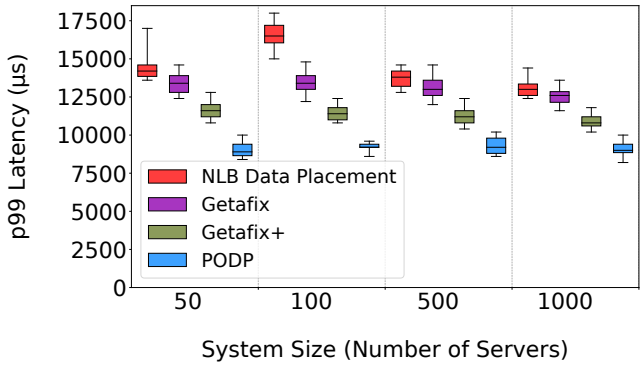


Figure 14: Number of servers in a system vs. p99 query latency in simulation. PODP provides consistent performance improvements across a range of system sizes.

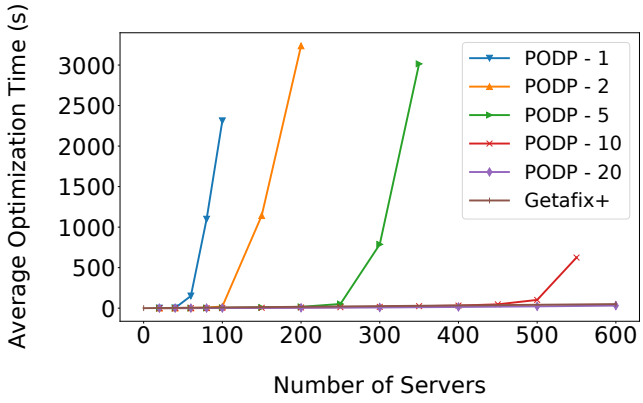


Figure 15: Number of servers in a system vs. average optimization time for PODP with a varying number of sub-problems, as well as Getafix+. Splitting PODP into sub-problems dramatically reduces optimization time.

some number of partitions such that PODP solves in comparable time as the best-performing baseline, Getafix+, while (as we show in Figure 16) substantially improving query tail latency.

We also investigate how partitioning PODP into sub-problems affects solution quality. In Figure 16, we graph how the number of sub-problems used by PODP affects query tail latency at different problem scales. We find that as long as the number of sub-problems is small compared to the number of servers, increasing the number of sub-problems does not have a significant effect on query tail latency. Additionally, solutions are of a significantly higher quality than Getafix+, the best-performing baseline.

6 RELATED WORK

Load Balancing and Data Placement. Many systems use data placement algorithms to balance query load across servers in a cluster. Getafix [18] proposes a load balancer for Druid that treats load balancing as a bin-packing problem. NashDB [24] proposes a general-purpose OLAP load balancer using a greedy algorithm inspired by economic models which has users assign a monetary

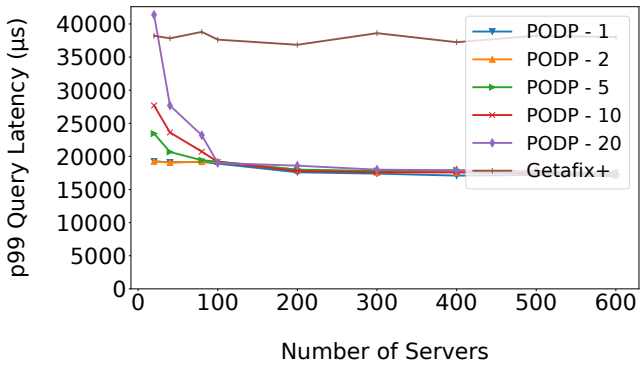


Figure 16: Number of servers in a system vs. p99 query latency in simulation for PODP with a varying number of sub-problems, as well as Getafix+. As long as the number of sub-problems is small compared to the number of servers, splitting the optimization problem does not have a significant effect on query tail latency.

"value" to each query. Several information retrieval systems use greedy algorithms to assign terms to servers to balance query load [21, 26]. General-purpose sharding systems such as Slicer [10], Shard Manager [23], and Uniserve [22] also provide load balancing as a service, using a variety of customizable algorithms.

We do not know of any general-purpose query parallelism-optimizing algorithm like PODP. However, some systems, like Apache Druid [32], use domain-specific heuristics to place data in a parallelism-optimizing manner (e.g., spreading new data across many servers because it is likely to be queried), finding this improves performance at scale [1]. Additionally, some proposed data placement algorithms for transactional systems do the opposite: they *co-locate* data items which are frequently queried together to minimize the frequency of distributed transactions. For example, Schism [15] models a database workload as a graph where tuples are vertices and transactions are edges, then partitions the graph to determine tuple placements that minimize the number of distributed transactions. Accordion [28] uses linear programming to determine the placement of static data partitions while considering their affinity–how frequently they are accessed by the same transactions. Clay [29] builds on Accordion to consider dynamically-changing data partitions, automatically creating new partitions from hot tuples.

Optimization Problems in Resource Management. Systems resource management problems like parallelism-optimizing data placement can often be expressed as mathematical optimization problems. For example, E-Store [31] proposes a linear program for load balancing similar to the load balancing component of PODP, although they do not consider parallelism. As mentioned earlier, Accordion [28] and Clay [29] also use linear programming to minimize distributed transactions in transactional databases. Because linear programs scale poorly, these systems have difficulty managing large problem sizes and some fall back to heuristics at scale [31]. However, newly developed techniques like partitioned optimization problems (POP) [27] allow some mathematical optimization

problems for resource management to be solved quickly at large scales with minimal loss of optimality.

A related problem to data placement is the online scheduling of queries over a given placement, where a schedule can reduce latency by taking advantage of replicas to avoid server contention. This is a form of online job-shop scheduling, a classical optimization problem where jobs (queries) are scheduled to shops (servers) while optimizing a global objective, such as makespan (latency); such problems have been extensively studied in existing literature [19, 20, 33]. Existing scheduling algorithms do not address the data placement problem, which can be viewed as the set-up stage that determines the allocation of *equipment* (data shards) to shops, whereby jobs can then be scheduled across shops that hold the necessary equipment. Our approach to data placement with PODP complements an online scheduling algorithm by minimizing the clustering of known queries, which we have shown to reduce server contention.

Optimization in Batch Processing Systems. Batch processing systems such as MapReduce [17], Hadoop [30] and Spark [34] operate at multi-second timescales and so can feasibly move data between servers as part of query execution, unlike the low-latency systems we target. This allows them to obtain high parallelism by moving data even if the original data placement was not highly parallel, thus reducing the necessity of optimizations like PODP. It also enables many optimizations based on data movement, such as straggler mitigation [13], speculative execution [35], and task stealing [11] which are not practical for systems that require low query latency.

One batch processing system that leverages ideas similar to ours is the caching service PACMan [12]. PACMan recognizes that batch processing jobs run as fast as their slowest task, so caching task inputs only improves performance if all tasks in a job can be cached and sped up. Thus, it provides cache access in an all-or-nothing manner, either caching the inputs of every task in a job or none of them. This idea of providing cache access for every task in a job in parallel resembles PODP's objective of minimizing clustering by parallelizing queries across as many servers as possible, although unlike PODP, PACMan does attempt to spread out cache locations or consider task load balancing.

7 CONCLUSION

In this paper, we analyze the importance of query parallelism to performance for data-parallel query workloads. We demonstrate both in principle and empirically that if queries frequently access co-located data items, resource contention on individual machines causes significant spikes in query tail latency. We propose PODP, a novel *parallelism-optimizing* data placement algorithm which formulates data placement as a mixed-integer linear program. To make this problem tractable, we define a new linearly-computable measure of query parallelism to optimize, and adapt an approach for partitioning the optimization problem into smaller sub-problems to scale to large system sizes. We apply our algorithm to popular systems such as Solr and MongoDB and show that it improves performance by 7-64% compared to several baselines.

ACKNOWLEDGMENTS

This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, and VMware—as well as Toyota Research Institute, Cisco, SAP, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Toyota Research Institute ("TRI") provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

REFERENCES

[1] 2016. Distributing data in druid at scale. https://metamarkets.com/2016/distributing-data-in-druid-at-petabyte-scale.
[2] 2018. Why Architecting for Disaster Recovery is Important for Your Time Series Data. https://www.influxdata.com/customer/capital-one/.
[3] 2019. How Walmart is Combating Fraud and Saving Consumers Millions. https://www.elastic.co/elasticon/tour/2019/dallas/.
[4] 2020. Enterprise Scale Analytics Platform Powered by Druid at Target. https://imply.io/virtual-druid-summit.
[5] 2021. MongoDB. https://www.mongodb.com/.
[6] 2022. Apache Solr. https://lucene.apache.org/solr/.
[7] 2022. ClickHouse. https://clickhouse.tech/.
[8] 2022. CPLEX. https://www.ibm.com/analytics/cplex-optimizer.
[9] 2022. Elasticsearch. www.elastic.co.
[10] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 739–753. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya
[11] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. 2012. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 61–74. https://doi.org/10.1145/2150976.2150984
[12] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated Memory Caching for Parallel Jobs. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 267–280. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan
[13] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 265–278.
[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 48–57. https://doi.org/10.14778/1920841.1920853
[16] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[17] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[18] Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta, and Himanshu Gupta. 2018. Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 40, 14 pages. https://doi.org/10.1145/3190508.3190542
[19] Klaus Jansen and Ralf Thöle. 2010. Approximation algorithms for scheduling parallel jobs. *SIAM J. Comput.* 39, 8 (2010), 3571–3615.
[20] Berit Johannes. 2006. Scheduling parallel jobs to minimize the makespan. *Journal of Scheduling* 9, 5 (2006), 433–452.
[21] Yubin Kim, Jamie Callan, J. Shane Culpepper, and Alistair Moffat. 2016. Load-Balancing in Distributed Selective Search. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*

(Pisa, Italy) *(SIGIR '16)*. Association for Computing Machinery, New York, NY, USA, 905–908. https://doi.org/10.1145/2911451.2914689

[22] Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. 2022. Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/nsdi22/presentation/kraft

[23] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. 2021. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 553–569. https://doi.org/10.1145/3477132.3483546

[24] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1253–1267. https://doi.org/10.1145/3183713.3196935

[25] Michael McCandless. 2020. Lucene nightly benchmarks. (2020).

[26] Alistair Moffat, William Webber, and Justin Zobel. 2006. Load Balancing for Term-Distributed Parallel Retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Seattle, Washington, USA) *(SIGIR '06)*. Association for Computing Machinery, New York, NY, USA, 348–355. https://doi.org/10.1145/1148170.1148232

[27] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 521–537. https://doi.org/10.1145/3477132.3483588

[28] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1035–1046. https://doi.org/10.14778/2732977.2732979

[29] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (nov 2016), 445–456. https://doi.org/10.14778/3025111.3025125

[30] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[31] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[32] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A Real-Time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 157–168.

[33] Deshi Ye, Xin Han, and Guochuan Zhang. 2009. A note on online strip packing. *Journal of Combinatorial Optimization* 17, 4 (2009), 417–423.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[35] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 29–42.