Uniserve: A Bolt-On Distribution Layer for Query Serving Systems

Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia
Stanford University

Abstract

Over the past few years, many organizations have developed custom distributed *query serving systems*, including full-text search systems like ElasticSearch, timeseries databases like OpenTSDB, and OLAP serving layers like Druid, to support highly demanding workloads. These systems are united by their need for scale, but often lack crucial distributed capabilities needed in cloud environments, such as load balancing, elasticity, and sometimes even fault tolerance. To ease distributing query serving systems, we propose architecturally separating their distributed concerns from their data storage and querying subsystems, creating a unifying distribution layer. This is challenging because of the diversity of their query patterns and data types, which range from SQL queries on tables to searches on indexed text. We address these challenges with Uniserve, a general distribution layer for query serving systems. The key insight of Uniserve is that most query serving systems distribute in the same way: by horizontally partitioning data into shards and distributing queries across shards. Therefore, we can develop unifying abstractions for shards and queries that support a wide range of systems. Uniserve automatically distributes systems that leverage these abstractions and implements fault tolerance, cloud-native elasticity, and a novel load balancing algorithm that maximizes query parallelism to improve performance over prior art. We evaluate Uniserve by porting three popular query systems to run over it: Druid, Solr, and MongoDB. We show that Uniserve matches the base performance of these systems and outperforms them significantly in the presence of hot shards, failures, or dynamically changing load.

1 Introduction

The last few years have seen explosive growth in the need for distributed systems that query diverse types of data [6]. These *query serving systems* use a variety of query models and data types specialized to particular problem domains. For example, online analytical processing (OLAP) systems such as Druid [46], Pinot [32], and Clickhouse [5] execute complex analytical queries over tabular data. Timeseries databases such as OpenTSDB [13] and Atlas [4] aggregate and analyze endless streams of operational logs. Full-text search systems such as ElasticSearch [7] and Solr [3] use sophisticated indexes to query text documents. Researchers and engineers design new query serving systems because no single system can satisfy all needs: specialized systems outperform general-purpose systems by orders of magnitude [44].

While query serving systems exhibit a great diversity of data types, query models, and workloads, they are united by their need for scale: the modern analytics workloads that fuel their popularity run on terabytes or petabytes of data. This scale brings with it new challenges. To maximize availability and performance, distributed query serving systems must tolerate server failure, balance load between servers, and elastically scale cluster size [18, 45]. Unfortunately, popular distributed query serving systems are missing many of these capabilities. For example, many systems, including Druid [46], Clickhouse [5], MongoDB [10], and Pinot [32], do not automatically balance load between servers. They instead ask database administrators to carefully choose shard keys or manually replicate hot shards to avoid poor performance, though these are difficult and error-prone processes [31, 47]. Moreover, although distributed query serving systems are now increasingly deployed in the cloud, most were not designed to leverage its elasticity and do not resize clusters when load changes, though this is one of the cloud's most important features [29]. Some systems can be configured with cloud auto-scalers, but others do not support them or require regular manual intervention [1, 2]. Some systems also deal poorly with server failure. For example, in Druid, a failure on any server touched by a query will fail the entire query even if data is replicated, causing problems in large deployments [34].

Given both the diversity of query serving systems and the similarity of their distributed concerns, we investigate whether it is possible to architecturally separate the distributed concerns of query serving systems from their data storage and querying subsystems. We propose separating distributed concerns into a *distribution layer* that could be "bolted on" to existing single-node query serving systems through a transparent interface. If possible, such a distribution layer would make existing and future specialized distributed query serving systems both more powerful and easier to write and maintain as they could leverage optimizations, features, and guarantees implemented in the distribution layer.

There are three core challenges inherent in designing a useful distribution layer. First, it must express the query patterns of existing query serving systems, even though these are diverse and range from SQL and NoSQL dialects to specialized OLAP queries to full-text search. Second, it must reliably distribute and manage the diverse data types used in existing systems, such as tables, NoSQL stores, time series, and text documents. Third, it should be able to maximize system performance in modern cloud environments through effective

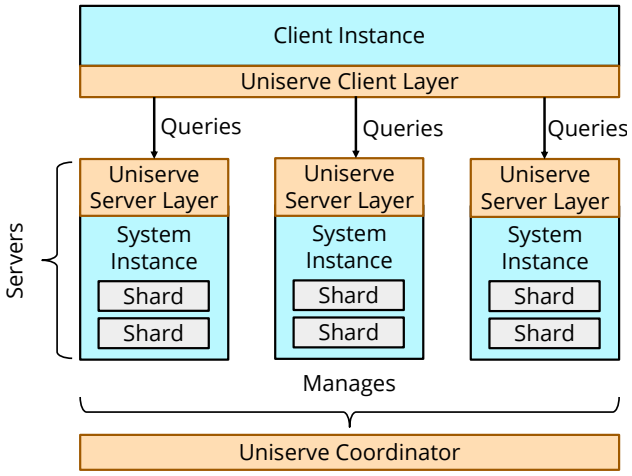

Figure 1: Uniserve distributes a single-node query serving system. It runs as a distribution layer (in orange) bolted onto instances of the underlying system (in blue), partitioning data into shards (in gray) and distributing queries across shards.

load balancing and auto-scaling algorithms for diverse query serving workloads.

To address these challenges, we have developed Uniserve, a bolt-on distribution layer for query serving systems. Uniserve is a framework for constructing a distributed query serving system from a single-node one. The key insight of Uniserve is that most query serving systems are distributed in the same way: by horizontally partitioning data into shards and distributing queries across shards. Therefore, we can develop unifying abstractions for shards and queries that fit many diverse systems. Uniserve distributes query serving systems that implement these abstractions, horizontally scaling them while providing fault tolerance, load balancing, and elasticity. Uniserve is optimized for read-mostly analytics workloads; it targets workloads with many large read queries and a few, coarse-grained writes. We sketch Uniserve in Figure 1.

Expressing the query patterns of existing query serving systems is challenging because of their diversity. Uniserve might need to perform faceted text search in one system, anomaly detection in a second, and operational monitoring in a third. Uniserve unifies these diverse query workloads with a common abstraction: a scatter-gather or MapReduce [28]-like query model. It represents queries as a pair of functions: one from a shard to an intermediate result, and the other from a list of intermediates to the query result. It executes the former function on all relevant shards separately, then aggregates the results with the latter function. This is analogous to MapReduce, the difference being that the basic unit of computation is not an individual record, but a shard of data, which may use specialized indexes to improve query performance. For example, Uniserve distributes a full-text search query such as counting the number of documents that contain a string by computing the count on each shard separately (which is fast because each shard contains a pre-built index), then returning the sum of those counts. This query model is not universal. For example, it cannot express shuffle joins, though it can express the broadcast joins that are more common in query serving systems. However, Uniserve can efficiently express all other relational algebra operators, as well as the query languages of many popular query serving systems such as Solr, Druid, and MongoDB.

Distributing and managing data storage is difficult because of the diversity of data types used by query serving systems, including tables, text indexes, and data cubes. Uniserve unifies these diverse data types with another abstraction: the shard. A Uniserve shard is a black-box data structure that hosts a horizontal partition of data. It needs only implement four functions: create, destroy, serialize, and deserialize. It must also accept write queries, functions that write rows of data. Uniserve enables consistent, linearizable, and durable updates to shards and maintains shard availability despite server failure through replication and through backup to durable storage (such as S3 or HDFS). Crucially, the shard abstraction does not restrict how data is stored, allowing many different data structures from different systems to implement it, including relational tables, Lucene indexes [22], and Druid segments [46].

Maximizing Uniserve performance in cloud environments is difficult because it requires developing effective load balancing and auto-scaling algorithms that improve performance of query serving workloads. One major challenge in these workloads is that most queries are parallel, accessing data on many different shards. If these shards are co-located on one server, contention in that server reduces query performance. Therefore, unlike prior load balancers [31, 41, 45], Uniserve maximizes opportunities for parallel data access by placing shards frequently queried together on separate nodes, improving query tail latency by up to 2x. Uniserve does this with a novel mixed-integer linear programming formulation that also balances query load and minimizes shard movement. Moreover, Uniserve uses the load balancer's guarantees to auto-scale cluster size as total load changes, efficiently loading data onto new servers and consolidating data from removed servers while maximizing availability.

We have implemented Uniserve and used it to distribute three popular distributed query serving systems: Druid, Solr, and MongoDB, wrapping their shard and query implementations with the Uniserve abstractions in only a few hundred lines of code. We find that at worst, systems distributed with Uniserve match the performance of natively distributed systems, but at best, Uniserve dramatically outperforms them, especially under conditions such as hot shards, failures, or dynamically changing load. We also show how Uniserve can express the queries in many other popular systems, such as Clickhouse [5], ElasticSearch [7], InfluxDB [8], and TAO [23].

To summarize, our contributions are:

- We propose Uniserve, a bolt-on distribution layer for query serving systems that uses powerful shard and query

abstractions to robustly distribute many diverse systems.

- We develop novel load balancing and auto-scaling algorithms for Uniserve that apply to all systems it distributes, outperforming prior state-of-the-art.
- We apply Uniserve to three popular distributed query serving systems and find Uniserve at worst matches but often exceeds natively distributed system performance.

2 Background and Motivation

To motivate Uniserve, we examine three popular and representative distributed query serving systems in detail. All three systems often run the query serving workloads that Uniserve targets, characterized by many large reads and infrequent, coarse-grained writes. We illustrate the systems' diverse query patterns, data types, and implementations but shared need for distributed capabilities such as fault tolerance, load balancing, and elasticity. We then discuss how they can implement Uniserve's shard and query abstractions and benefit from its distributed capabilities.

2.1 Case Studies

Apache Solr. Apache Solr [3] is a distributed full-text search system. Solr constructs Lucene indexes [22] from documents, enabling fast and powerful search queries over their contents. Solr natively shards data, building indexes on all servers and hashing documents to assign them to indexes. It executes full-text search queries by distributing them across shards, then aggregating the results. Solr replicates indexes for availability. It does not balance query load across servers, but does ensure servers host the same number of shards. Solr is not natively elastic, but it exposes an auto-scaling API providing useful information for an external auto-scaler. Solr resembles other full-text search systems, such as Elasticsearch [7].

Apache Druid. Apache Druid [46] is a high-performance analytics system. Druid ingests time-ordered tabular data, such as machine logs, and stores it in *segment* data structures optimized for fast analytics queries. Druid natively shards data, storing time ranges together in a single segment. Most Druid queries are aggregations, GROUP BYs, or approximate top-K queries, though Druid has recently added support for broadcast joins [43]. Druid replicates segments and backs them up to external durable storage for availability. However, it lacks query fault tolerance—if a server fails, all queries touching data on that server will fail even if data is replicated [34]. Druid ensures all servers host the same amount of data, but does not balance query load–if some segments are queried more than others, the user must manually replicate them or face poor performance. Druid is not natively elastic, but it moves data to new nodes if they are added and consolidates data if nodes are removed. Druid resembles OLAP systems such as Pinot [32] and Clickhouse [5], as well as timeseries databases such as OpenTSDB [13].

	Fault Tolerance	Load Balancing	Replication	Elasticity
Solr	Yes	Data Only	Manual	External
Druid	Partially	Data Only	Manual	No
MongoDB	Yes	Data Only	Manual	No
Uniserve	Yes	Queries and Data	Automatic	Yes

Table 1: Presence or absence of distributed features and guarantees in existing systems and Uniserve.

MongoDB. MongoDB [10] is a document-oriented NoSQL database. Unlike Solr and Druid, it is not primarily an analytics system, but it is often used for analytics [11]. MongoDB performs search and aggregation queries over semi-structured data. It shards data across servers using user-specified keys and distributes queries across shards with some support for broadcast joins. It replicates data for availability. Like Druid, it balances the amount of data stored on servers, but not query load. However, it is not natively elastic and requires manual intervention to add data to a new node or consolidate data if a node is removed [2].

2.2 Discussion

Although these three systems have different data types and query models, they can easily implement Uniserve's abstractions and benefit from its distributed capabilities. They all horizontally partition data into shards and their queries (including broadcast joins) follow a scatter-gather query model. All three systems only partially implement the distributed capabilities that Uniserve provides, as we show in Table 1. They have some fault tolerance features, but are not natively elastic and only balance the amount of data hosted on each server instead of the number of queries they receive. As a result, they perform poorly when faced with load skew or dynamically changing load levels. As we show later, Uniserve addresses these issues while matching performance elsewhere.

3 Uniserve Overview and Interfaces

In this section we discuss how Uniserve distributes query serving systems and describe Uniserve's architecture.

3.1 Using Uniserve

Uniserve can distribute a single-node query serving system that implements two interfaces: a shard interface for storing data and a query interface for querying data. We outline both interfaces in Figure 3. The shard interface is a shim layer describing how to create, destroy, serialize, and deserialize shards. Systems implement shards using data structures that can store horizontal partitions of data. Most commonly, these are database tables or the local equivalent, such as MongoDB collections, Lucene indexes, or Druid datasources.

The query interface is a translation layer that converts queries in the underlying system's native query language to Uniserve read and write query plans. Uniserve read query plans implement the MapReduce-like Uniserve query model, distributing queries across shards and aggregating the results.

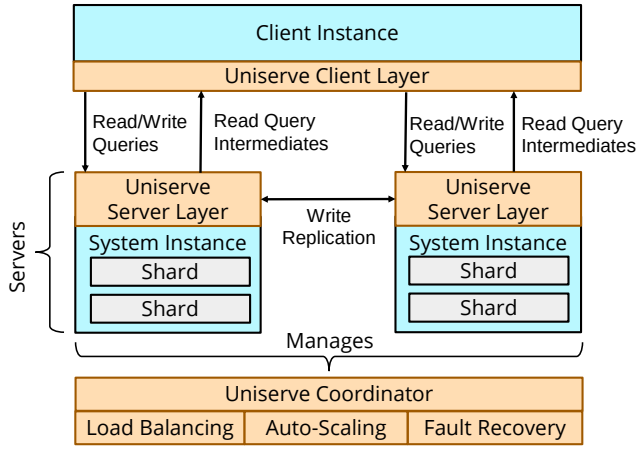

Figure 2: The Uniserve architecture. Uniserve runs as a thin layer (in orange) on client and server instances of the underlying system (in blue). Uniserve partitions data into shards (in gray) and distributes queries across shards. A coordinator manages cluster state and provides distributed capabilities.

Uniserve write query plans add rows of data to shards; writes are linearizable, consistent, and durable.

3.2 Uniserve Architecture

A Uniserve cluster is made up of many servers, each running a single-node query serving system instance. We diagram a Uniserve cluster in Figure 2. A thin Uniserve layer runs on each server and client. Each server layer receives queries from clients, passes them down to its underlying system instance, and returns its responses. Each client layer translates native queries into Uniserve query plans, distributes them across servers, and aggregates their responses. A single coordinator manages cluster state: it assigns shards to servers, handles failures, and adds and removes servers as cluster load changes. Uniserve also assumes the existence of durable storage (such as S3 or HDFS) on which it can back up serialized shards.

3.3 Servers and the Shard Interface

Each server in a Uniserve cluster runs both an instance of the underlying single-node query serving system and a thin Uniserve server layer. Servers host data in shards stored in the underlying system instance. Shards may be replicated across multiple servers. The Uniserve server layer passes down client queries to shards and uses the shard interface to manipulate shards in response to coordinator commands.

As shown in Figure 3, the components of the shard interface are the `Shard`, the `ShardFactory`, and the `Row`. The `Shard` and `ShardFactory` interfaces contain the core shard functions: create, destroy, serialize, and deserialize. Each underlying system instance must host many shards, so shards are typically implemented as tables or the local equivalent; for example, `createNewShard` might be implemented as a create table expression. Additionally, the `Shard` interface has a `getMemoryUsage` function used by the load balancer to ensure servers have enough memory for their shards.

```
interface Shard:
  int getMemoryUsage()
  Path serializeToDisk()
  void destroy()

interface ShardFactory:
  Shard createNewShard()
  Shard loadSerializedShard(Path path)

interface Row:
  int getPartitionKey()

interface QueryEngine:
  ReadQueryPlan planReadQuery(String query)
  WriteQueryPlan planWriteQuery(String query)

interface<I, T> ReadQueryPlan:
  List<Integer> keysForQuery()
  I queryShard(Shard s)
  T aggregateResults(List<I> results)
  int getQueryCost()

interface WriteQueryPlan:
  boolean prepareCommit(Shard s,List<Row> rows)
  void commit(Shard s)
  void abort(Shard s)
```

Figure 3: The Uniserve shard and query interfaces.

Uniserve uses the `Row` interface to assign data to shards. A row is a single unit of data, such as a SQL database row or a MongoDB or Lucene document. Each row has an integer partition key accessed through `getPartitionKey`. Uniserve assigns rows to shards based on their partition keys–all rows with the same partition key are assigned to the same shard. One limitation of Uniserve is that the key-to-shard mapping is fixed at cluster creation time; like many popular query serving systems such as Druid and ElasticSearch, Uniserve cannot move rows between shards.

3.4 Clients and the Query Interface

The Uniserve client layer uses the query interface to execute native system queries by translating them into Uniserve query plans and distributing them. As shown in Figure 3, the query interface has three components: the `QueryEngine`, the `ReadQueryPlan`, and the `WriteQueryPlan`. When the client layer receives a query, it uses the `QueryEngine` to construct a query plan, then executes it.

A `ReadQueryPlan` resembles a map-reduce. To execute it, the client layer first calls `keysForQuery` to determine the partition keys of the queried data. It then sends the query to a random replica of each relevant shard. Servers execute `queryShard` on those shards and return results to the client for aggregation with `aggregateResults`. For example, in an SQL query counting the number of rows that meet some condition, `queryShard` would count those rows on a single shard (database table) and `aggregateResults` would sum the counts. Additionally, after each `queryShard` call, the server estimates query cost with `getQueryCost` and passes the result to the coordinator for use in load balancing.

A `WriteQueryPlan` writes rows to shards. Uniserve writes each row to the shard corresponding to its partition key. Writes are replicated and are consistent, durable, and linearizable. We

	System Type	Data Type	Query Operations
Druid [46]	OLAP	Tables	Aggregate, group, filter, map, broadcast join
Pinot [32]	OLAP	Tables	Aggregate, group, filter, map
ClickHouse [5]	OLAP	Tables	Aggregate, group, filter, map, broadcast join
OpenTSDB [13]	Timeseries DB	Timeseries	Aggregate, group, filter, map
Atlas [4]	Timeseries DB	Timeseries	Aggregate, group, filter, map
InfluxDB [8]	Timeseries DB	Timeseries	Aggregate, group, filter, map
Solr [3]	Full-Text Search	Indexed text	Group (including faceting), search
ElasticSearch [7]	Full-Text Search	Indexed text	Group (including faceting), search
TAO [23]	Graph Database	Graphs	Range query edges from a node
MongoDB [10]	NoSQL	Documents	Aggregate, group, filter, map, broadcast join

Table 2: Some systems Uniserve can distribute and their properties. Systems whose ports we have implemented are in bold.

discuss writes in more detail in Section 5.

3.5 Case Study: Solr

We now describe how to create a Uniserve port of the distributed full-text search system Solr [3], hosting a single-node Solr instance on each cluster server. Solr shards text documents across Lucene indexes [22] on several machines. When Solr receives a new document, it hashes it, uses the hash to pick a shard, and adds it to that shard's index. Therefore, we can port Solr's data storage capabilities by implementing Uniserve shards as Solr collections (which consist of a single Solr shard) and Uniserve rows as Solr documents. Just like Solr, when Uniserve receives a new document, it hashes it to compute a partition key, then adds it to the shard corresponding to that key. Solr stores shard data on disk and supports two-phase commit, so all shard and write query functions map onto native equivalents; for example, `createNewShard` is implemented as `createCollection`.

All Solr queries are searches: they take in a criterion, such as a query string, and return a list of indexed documents that satisfy it. This list may be aggregated by grouping or faceting. Solr distributes queries by searching each shard separately, then combining results on a single node [14]. Uniserve matches this pattern: `queryShard` searches shards separately, aggregating as much as possible, then `aggregateResults` combines results. For example, to distribute a search for books whose title contains the word "goblin," grouped by year, Uniserve searches each index shard for "goblin" separately, groups the per-shard results by year, and then combines the per-year lists. As we show in Section 8, the Uniserve port of Solr matches or exceeds native Solr performance.

4 Generality of Uniserve

In this section, we demonstrate the generality of Uniserve by cataloging many diverse systems it can distribute, summarized in Table 2. We also discuss Uniserve's limitations.

OLAP Systems. OLAP systems are designed to rapidly answer complex multidimensional analytics queries, particularly for business use cases. Their workloads typically consist of aggregations, GROUP BYs, and top-K queries on tabular data, often pre-aggregated for efficiency. These all naturally fit Uniserve's scatter-gather query model. We have implemented a Uniserve port of one OLAP system, Druid [46]. Other OLAP systems that Uniserve can distribute include Pinot [32] and Clickhouse [5].

Timeseries Databases. Timeseries databases ingest and query time-ordered tabular data; they are typically used for business intelligence or operational monitoring. Their query workloads consist of processing, filtering, grouping, and aggregating timeseries. All of these operations fit Uniserve's scatter-gather query model, so Uniserve can distribute most timeseries databases, such as OpenTSDB [13], Atlas [4] and InfluxDB [8]. However, Uniserve is optimized for read-mostly workloads while timeseries database workloads often include many small writes; these must be batched into a few coarse-grained writes for Uniserve to work effectively.

Full-Text Search. Full-text search systems store text data in specialized data structures such as Lucene indexes [22] to answer complex text search queries. Because all their queries are searches, they easily fit Uniserve's scatter-gather query model. We have implemented a Uniserve port of one full-text search system, Solr [3]; it can also distribute ElasticSearch [7].

Graph Databases. Graph databases query graph data structures. Because many graph algorithms must iteratively process data across entire graphs, they are difficult to distribute. Uniserve can distribute some graph databases, such as TAO [23], which restrict themselves to simple queries that fit Uniserve's scatter-gather query model, such as querying the edges from a single node. However, Uniserve cannot distribute others, such as Janus [9] and Neo4j [12], whose queries require extensive communication between servers.

Limitations. Uniserve is optimized for the read-mostly analytics workloads common among query serving systems. It is not universal: it makes performance tradeoffs and lacks some database capabilities. First, Uniserve only supports scatter-gather queries. Most analytics queries fit this model, but some do not, including shuffle joins and some graph algorithms.

Additionally, Uniserve does not support transactional workloads. Its load balancer and replication system are optimized for a small number of coarse-grained writes. As we will show in Section 5, Uniserve provides serializable and linearizable writes but no read-write atomicity and is not optimized to handle write-write contention well. Transactions are rare in analytics systems, but common in other systems such as relational databases which Uniserve cannot effectively distribute.

5 Consistency and Fault Tolerance

In this section we discuss how Uniserve implements durable, consistent, and linearizable writes and mitigates server failure.

5.1 Writes and Replication

Uniserve writes add rows to shards. They can append new data or modify existing data. Writes are replicated using two-phase

commit and ZooKeeper. One replica of each shard is designated the primary. To execute a write, the Uniserve client layer sends each row to the primary of the shard corresponding to the row's partition key. The primary shards `prepareCommit` the write on the rows they receive and pass them on to their replicas. If all replicas of all shards successfully prepare, the write commits; otherwise, it aborts. The client layer writes a record to ZooKeeper to mark the decision so that even if failures occur afterwards, all surviving replicas commit or abort.

Uniserve writes are consistent, linearizable (but not atomic), serializable and durable. Because of ZooKeeper-backed two-phase commit, either all rows are written to all replicas of all shards, or no rows are written at all. Because writes are not complete until all replicas of all shards have committed, reads made after a write completes always reflect the write, though a read made before a write completes may reflect the write on some shards but not on others. Writes, however, always occur in the same order on all shards. This is enforced optimistically. Every write is given a unique increasing transaction ID through ZooKeeper. If a shard receives a write with a lower ID than the one it is working on, it aborts its current write, completes the new one, then returns to the original. This is necessary to avoid write-write deadlock, but because Uniserve targets read-mostly workloads, we expect it to occur rarely. Because writes are replicated, they can survive server failures. Moreover, Uniserve regularly uploads serialized shards to durable storage such as S3, so Uniserve can only lose a completed write to a shard if all its replicas fail after the write completes but before the updated shard is uploaded.

5.2 Fault Tolerance

Uniserve mitigates server failures, even mid-query, using common fault tolerance techniques. Like many distributed systems [39], Uniserve assumes a fail-stop model for failures, where the only way servers fail is by crashing. It also assumes that if a server crashes, it remains crashed until restarting (when it will be treated as a new server). Moreover, it assumes that the coordinator never fails and that ZooKeeper is always available. The latter assumption is common among query serving systems that use ZooKeeper, such as Solr or Druid. These assumptions greatly simplify failure recovery.

Uniserve servers constantly ping each other in a decentralized heartbeat system inspired by RAMCloud [39]. If a server detects a failure, it notifies the coordinator, which confirms it. Upon detecting a failure, the coordinator restores shard availability. If the failed server held the primary of any shard, the coordinator promotes a random replica to primary. If no replicas exist, the coordinator orders a random server to load a new primary from durable storage. If server failure leads to imbalanced load, the load balancer corrects it.

To handle failures during write queries, Uniserve uses the procedures described in Section 5.1 to make writes consistent and durable. To handle failures during read queries, Uniserve uses a retry protocol. If a read to a shard fails, the client reloads the list of shard replicas from ZooKeeper and tries again with a different random replica. It keeps retrying until it has exhausted all replicas; this occurs only if all servers containing replicas of a shard are lost, in which case the shard will be unavailable until it can be restored from durable storage.

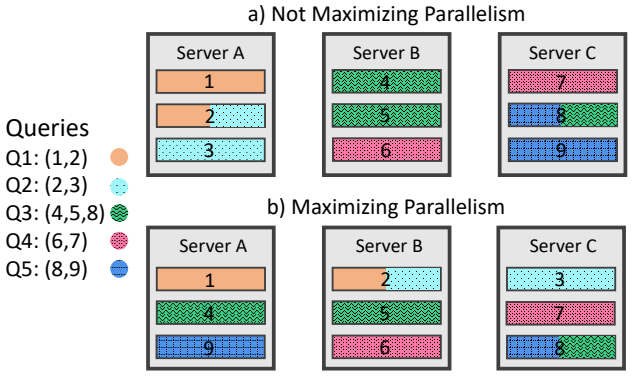

Figure 4: A diagram of the Uniserve parallelism optimization. Above, without the optimization, every query except Q4 accesses multiple shards on the same server, causing contention and reducing performance. Below, with the optimization, all queries are fully parallelized on multiple servers, improving performance.

6 Load Balancing and Elasticity

In this section, we discuss how Uniserve provides two features critical for cloud deployments of query serving systems: load balancing and elasticity. One major challenge in load balancing query serving systems is that most queries are parallel and access data on multiple shards. For example, a timeseries database might shard data by day, with queries accessing the shards for several consecutive days. Ideally, each of these shards would be hosted on a different server to maximize parallelism and hence performance; if a server hosts multiple accessed shards, contention in that server delays query completion. However, most existing load balancers, such as E-Store [45] and Accordion [41] are designed for transactional or key-value workloads and do not consider query parallelism.

Uniserve implements a novel load balancing algorithm that optimizes query parallelism, improving query tail latency by up to 2× compared to prior art. Moreover, using the guarantees provided by the load balancer, Uniserve elastically scales cluster size with a simple total-load-based algorithm. We describe Uniserve's load balancing algorithm in Section 6.1 and auto-scaling algorithm in Section 6.2.

6.1 Load Balancing

The goal of the Uniserve load balancer is to construct an assignment of shards to servers that maximizes query parallelism while balancing query load. In query serving workloads, queries frequently access multiple shards and run slower when these shards are co-located on the same server. For example, in Figure 4a, queries Q1, Q2, Q3, and Q5 each access multiple shards on the same server, which would cause

contention and slow them down. Unlike existing load balancers, Uniserve carefully assigns shards to servers in a way that minimizes this contention and maximizes query performance. We illustrate such an assignment in Figure 4b, where each query's shards are hosted by different servers.

Minimizing query contention is challenging. In a real deployment, the number of shards, servers, and distinct shard access patterns would likely be large, the sets of shards accessed by different queries would overlap (for example, both Q1 and Q2 access Shard 2), and shard access patterns would change over time, increasing problem complexity. Moreover, any shard placement algorithm would have to respect additional constraints such as balancing query load and respecting server memory capacities and would also have to minimize shard movement to reduce shard transfer overhead.

The Uniserve load balancer addresses these challenges with a mixed-integer linear programming (MILP) formulation that constructs a shard-to-server assignment that optimizes query parallelism, balances server load, satisfies memory constraints, and minimizes shard movement. This formulation is effective and practical: as we show in Section 8.4, it improves query tail latency by up to 2× over prior art and executes in under three minutes on over a thousand shards.

The key insight of the load balancer is that we can represent the query parallelism of a workload as a linear function of each query's *contention*, the maximum number of co-located shards it can access, and its frequency. Let us define the shard set of a query as the set of all shards it touches. For example, in Figure 4, the shard set of Q1 is $\{1,2\}$. Uniserve constructs a list S of the k most popular query shard sets s and their frequencies f_s over the past load balancing period; k is configured by the user. For a given shard set s, we define its contention c_s as the maximum number of shards in the set which have co-located replicas. For example, in Figure 4a, $c_s = 2$ for Q1 because both the shards it touches are on the same server; in 4b, $c_s = 1$. The load balancer minimizes the sum of shard set contentions weighted by frequency.

The load balancer operates in two steps by solving two MILP problems: P_k and P_b. P_k computes the shard-to-server assignment that minimizes contention in a representative set of k queries – thereby maximizing query parallelism – while balancing server load. P_b then computes the assignment that matches the optimal level of contention computed by P_k and balances server load while minimizing shard movement. The load balancer runs periodically, where its period is defined by the user. Each time it runs, it collects load, memory, and query statistics, solves P_k and P_b, then implements the assignment computed by P_b by moving, replicating and removing shards.

Assume we have a cluster with M data shards assigned to N servers. The load balancer 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. 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 1 if server i does not currently host a replica of shard j and 0 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).

P_k 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 locations (t_{ij}), replication factor R, and the set of all query shard sets S and their corresponding frequencies (f_s), minimize query contention weighted by query frequency subject to constraints:

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

subject to:

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

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

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

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

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

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

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

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) ensures that shard loads go to servers that host the shard. Constraint (7) ensures that the replication factor is respected. Finally, Constraint (8) defines the contention of query s as the maximum number of shards in its shard set co-located on a server.

Solving P_k, we obtain the optimal values of c_s, which are used as constraints by P_b to compute the final shard-to-server assignment that minimizes shard movement. P_b 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 locations (t_{ij}), replication factor R, and the optimal contention values c_s computed by P_k, minimize shard movement subject to 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 3-7

where Constraint (10) ensures that the level of parallelism supported by the solution to P_b is equal to that found by P_k.

6.2 Elasticity

When deployed in an elastic cloud environment such as Amazon EC2, Uniserve automatically scales cluster size in response to load changes. The Uniserve auto-scaler runs on the coordinator after each load balancing event. Its algorithm is simple: it collects average cluster CPU utilization over the last load balancing period and compares it to upper and lower thresholds. If utilization exceeds the upper threshold, Uniserve adds a server; if it is below the lower threshold, Uniserve removes a server. To avoid oscillation, Uniserve employs hysteresis, setting a large gap between the upper and lower thresholds. In our experiments, the upper threshold is 70% utilization and the lower threshold is 30% utilization. Uniserve can use a simple average-utilization-based algorithm because it has an effective load balancer: all servers receive the same amount of load, so if average utilization is high, the entire cluster is overburdened. This approach is similar to existing elastic database systems such as E-Store [45] and to some cloud auto-scalers [21]. However, unlike a cloud auto-scaler, the Uniserve auto-scaler benefits from integration with the Uniserve load balancer and shard interface, allowing it to more easily and effectively add and remove servers.

To add a server, Uniserve asks the cloud environment to create a new server using a specified image and launch script. Once the new server comes online, the coordinator runs the load balancer with the new server included to determine what shards to load onto it. To maximize availability, the server loads its newly-assigned shards sequentially, prioritizing the shards that are expected to receive the most load; we found sequential loading was only slightly slower than parallel loading but improved overall performance by loading some shards earlier. A system using a cloud auto-scaler must implement a similar capability to automatically transfer data to a new server; many systems such as MongoDB [2] lack this.

To remove a server, Uniserve asks the cloud environment to terminate one. Uniserve carefully chooses and sets up this server to ensure its loss does not affect availability. Specifically, it removes the server that hosts the fewest primary shards. If a server hosts no primary shards at all (only replicas), Uniserve can safely remove it without affecting availability; the load balancer will resolve any resulting load skew. If there are primary shards, Uniserve promotes a replica to primary before removing the original. If a primary lacks replicas, Uniserve instead loads and promotes a new replica. A system using a cloud auto-scaler needs a similar capability to consolidate data from a server being removed or it will lose availability; many systems such as Druid [34] and MongoDB [2] lack this.

7 Integrating Uniserve into Existing Systems

We evaluated Uniserve by using it to distribute three popular query serving systems: Solr, Druid, and MongoDB. Our Uniserve ports of these systems support their main query types, but also provide fault tolerance, load balancing, and elasticity as discussed in previous sections. In this section, we describe how we implemented each port.

Solr. We have already described the Uniserve port of Solr in Section 3.5. Interestingly, because Solr separates its distribution layer (Solr itself) from its data storage and querying subsystems (Lucene), we can easily count the number of lines of code used to distribute Solr: over 300,000. For comparison, the Uniserve port of Solr provides most of Solr's query functionality in fewer than 500 lines. The Druid and MongoDB ports are of similar length.

Druid. In our Uniserve port of Druid [46], each server runs a single-node Druid instance. We implement shards as Druid datasources. These are analogous to database tables and are backed by Druid segments, which are indexes for timeseries data. We implement most `Shard` interface functions, including creating, destroying, and writing, using the Druid API for manipulating datasources. To implement `Shard` serialization, we copy the on-disk segments to a target directory, then export Druid's segment metadata table. To implement `Shard` deserialization, we copy the segments into the Druid data directory, then load the exported segment metadata into Druid.

All Druid queries aggregate filtered and/or grouped data from potentially JOINed datasources (broadcast joins only). The Uniserve port of Druid currently only supports simple Druid queries: sums, counts, or rankings of filtered and/or grouped data from a single datasource. In our `ReadQueryPlan` implementations, `queryShard` sums, counts, or ranks data from each shard separately, then `aggregateResults` combines the results. Our port could easily be extended to support other aggregations and groupings, such as sketches and data cubes, as well as broadcast joins.

MongoDB. In our Uniserve port of MongoDB [10], each server runs a single-node MongoDB database. We implement shards as MongoDB collections, analogous to database tables. We implement most `Shard` interface functions, including creating, destroying, and writing, using the MongoDB API for manipulating collections. We implement the `Shard` serialization and deserialization functions using the `mongodump` and `mongorestore` tools to create and load collection backups.

MongoDB queries apply an "aggregation pipeline" of operators to a collection. These operators perform many tasks,

including filtering, grouping, and accumulating documents. Uniserve can support all MongoDB operators, but we have only implemented operations for filtering, projecting, summing, counting, and grouping data. Our `ReadQueryPlan` implementations are similar to those in our Druid port: aggregating on each node separately, then combining results.

8 Evaluation

We evaluate Uniserve using the ports of the popular query serving systems Solr, Druid, and MongoDB discussed in Section 7. We demonstrate that:

1. Under ideal conditions—static workloads without load skew or failures—systems distributed with Uniserve match the performance of natively distributed systems on representative workloads.
2. Under less ideal conditions–workloads that change, have load skew, or have server failures–systems distributed with Uniserve outperform natively distributed systems.
3. The parallelism-maximizing Uniserve load balancer improves query tail latency by up to 2× over prior art.

8.1 Experimental Setup

We run most benchmarks on a cluster of five m5d.xlarge AWS instances, each with four CPUs, 16 GB of RAM, and an attached SSD. We evaluate using Apache Solr 8.5.0, Apache Druid 0.17.0, MongoDB 4.2.3, and Apache ZooKeeper 3.5.6.

When benchmarking each system natively, we place the master (Solr ZooKeeper instance, Druid coordinator, MongoDB config and `mongos` servers) on a machine by itself and a data server (SolrCloud node, Druid historical, MongoDB server) on each other node. On all systems, we disable query caching and set the minimum replication factor to 1.

When benchmarking each system with Uniserve, we use the implementations described in Section 7, disabling query caching and setting a minimum replication factor of 1. We place the Uniserve coordinator and a ZooKeeper server on a machine by themselves and data servers on the other nodes.

8.2 Benchmarks

We evaluate each system with a representative workload taken when possible from the system's own benchmarks. We benchmark Solr with queries from the Lucene nightly benchmarks [38]. We run each query against a dataset of 1M Wikipedia documents taken from the nightly benchmarks. We use two of the nightly benchmark queries–an exact query for the number of documents that include the phrase "is also" and a sloppy query for the number of documents that include a phrase within edit distance four of the phrase "of the."

We benchmark Druid with two of the TPC-H queries used in the Druid paper [33, 46], running each against 6M rows of TPC-H data. The queries we use are `sum_all`, which sums four columns of data; and `parts_details`, which runs an top-K query on the result of a GROUP BY on a data column.

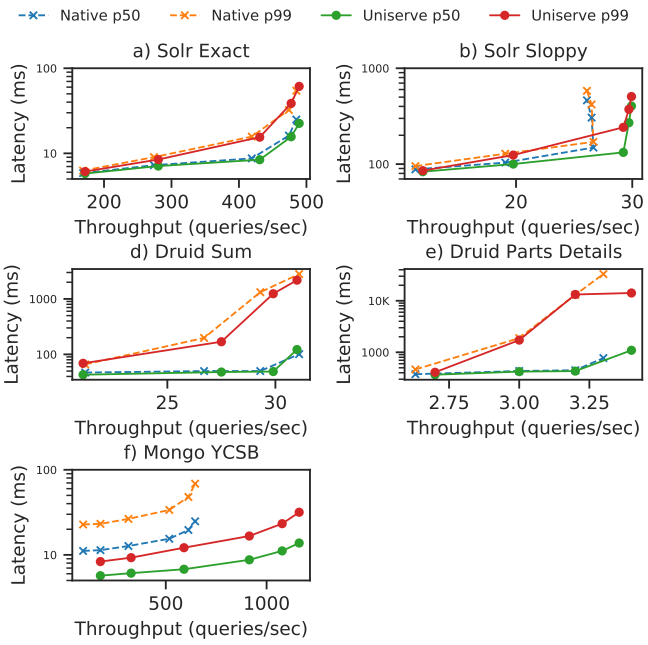

Figure 5: Throughput versus latency for natively-distributed and Uniserve-distributed queries on uniform and static workloads. Uniserve generally matches native performance.

We benchmark MongoDB using YCSB [27], simulating an analytics workload. Before running the workload, we insert 10000 sequential items into the database. We run a workload of 100% scans, where each scan retrieves one field from each of uniformly between 1000 and 2000 items. We base our YCSB client implementation on the MongoDB YCSB client from the YCSB GitHub repository [16].

8.3 End-to-End Benchmarks

Ideal Conditions. We first benchmark each workload under ideal conditions, distributing queries uniformly so each data item is equally likely to be queried. We run each benchmark with several client threads, each of which repeatedly makes the query and waits for it to complete, recording throughput and latency. We start with a single client thread and add more until throughput no longer increases, showing results in Figure 5. We find that, unsurprisingly, Uniserve performance is similar to native system performance on all benchmarks. The exception is the MongoDB benchmark where Uniserve performs significantly better; this is due to the high overhead of the `mongos` shard server.

We also evaluate the scalability of Uniserve, scaling the Solr benchmarks on 20M documents with one client thread from four to forty servers, showing results in Figure 6. Uniserve query performance scales near-linearly, with slightly better scaling on the more-expensive sloppy benchmark.

Hotspots. We next benchmark each workload under load skew. We look specifically at the Druid `parts_details` and MongoDB YCSB benchmarks (we do not know of a realistic

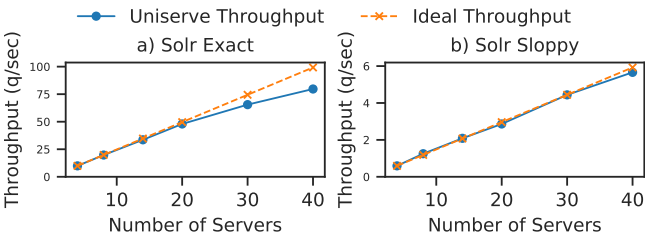

Figure 6: Uniserve scalability on the Solr benchmarks.

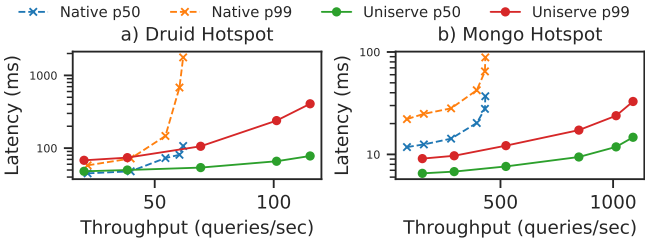

Figure 7: Throughput versus latency for natively-distributed and Uniserve-distributed queries with a hotspot, where one slice of data receives 7/8 of all queries. Because Uniserve replicates hot data, it outperforms native systems.

way to skew load in the Solr benchmarks). We send 7/8 of the queries to a single slice of data (a single month of data in Druid, the first 1/8 of the keys in MongoDB) and scatter the rest uniformly on the remainder of the data. We record throughput and latency while increasing the number of client threads as before, showing results in Figure 7. Because Uniserve balances query load but native systems do not, Uniserve dramatically outperforms native systems.

We next repeat the experiment, fixing the number of client threads at twelve but varying the fraction of queries sent to the hotspot. We show results in Figure 8. We find that changing skew does not affect Uniserve performance because it keeps load balanced under any load distribution. However, native system performance worsens with increasing skew.

Dynamic Load. We next investigate the performance of the Uniserve auto-scaler with changing load levels. We run the Solr sloppy benchmark for six hours, gradually scaling the number of client threads from one to twelve and then back down to one. Uniserve starts with one server and adds or removes more as load changes. We show results in Figure 9. We find the auto-scaler accomplishes its goal of keeping average CPU usage below 70% and above 30%, except for brief periods when new servers are being added or removed. This, in turn, keeps query latencies in a steady range. Moreover, we do not observe oscillation or overutilization; Uniserve never removes a server shortly after adding one and eventually removes all new servers as load is reduced.

Failures. We next investigate how Uniserve deals with server failures. We use the Druid `sum_all` benchmark, computing the sum of four data columns across the entire Druid TPC-H dataset. We run this benchmark for ten minutes with

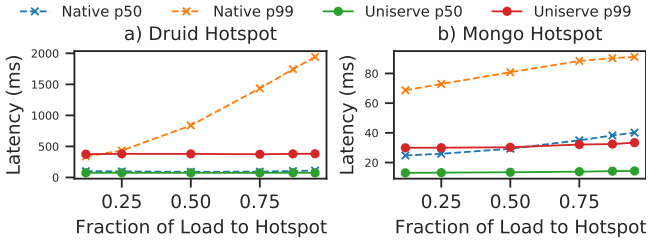

Figure 8: Latencies of natively-distributed and Uniserve-distributed queries on benchmarks with a hotspot, where one slice of data receives a varying fraction of all queries. The Uniserve load balancer keeps performance constant as skew increases; native systems see performance drop.

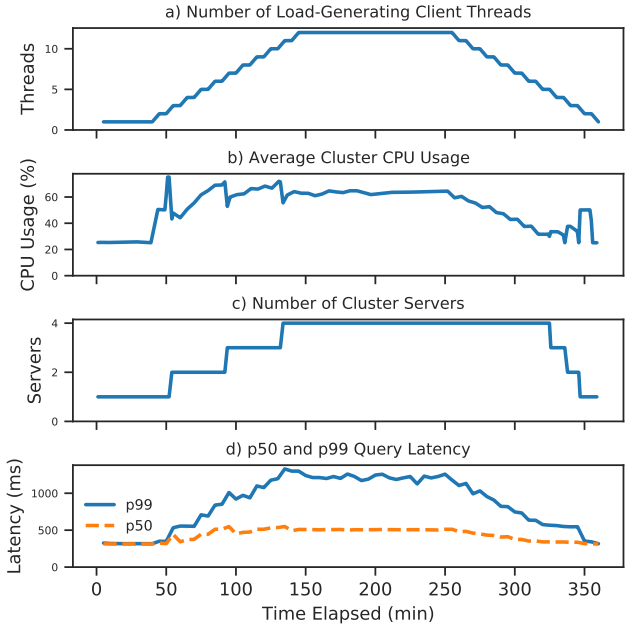

Figure 9: On the Solr sloppy benchmark with Uniserve auto-scaling, varying the number of load-generating client threads and observing effects on average cluster CPU usage, the number of cluster servers, and query latencies.

a client sending 500 asynchronous queries per minute. Three minutes into the benchmark, we `kill -9` a data server. We record how many queries succeed during each minute of the benchmark. We run the benchmark twice, once starting with four replicas of each shard (one on each server), and once with just a single replica. We show results in Figure 10.

When all servers have replicas of all shards (10b), Uniserve recovers instantly, routing queries to replicas. Druid, however, takes approximately thirty seconds to begin routing queries to replicas, resulting in hundreds of query failures. When there is only one replica of each shard (10a), both systems fail hundreds of queries but recover in approximately thirty seconds by restoring replicas from durable cloud storage. However, while every single query sent to Uniserve either fails or successfully completes, some "successful" Druid queries

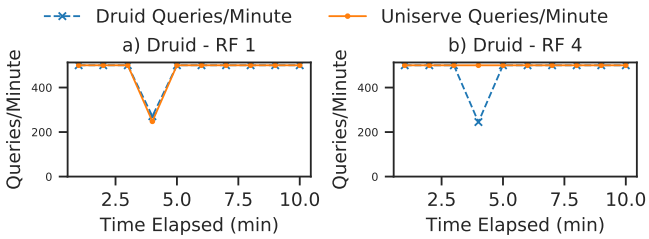

Figure 10: Query throughput (with a target of 500 queries/minute) of Druid-distributed and Uniserve-distributed TPC-H `sum_all` queries when one data server is killed after three minutes. The left graph shows performance starting with a single replica of each shard; the right graph with four.

return incorrect results. Druid's query fault tolerance is known to be problematic in large-scale deployments [34]; Uniserve addresses its issues.

8.4 Microbenchmarks

Load Balancer Factor Analysis. We now perform a factor analysis on the Uniserve load balancer to determine the relative contributions of different components of its design. We run this analysis on the MongoDB YCSB workload with 16 client threads, initially co-locating the shards for adjacent ranges. First, we run without a load balancer. Then, we use a load balancer that solves the simple bin-packing problem described in E-Store [45]: finding an assignment of shards to servers where all servers have the same load but there is only a single copy of each shard. Next, we use a more powerful load balancer which can replicate hot shards, but with the assumption that all queries are equally costly; this provides similar guarantees to the algorithm described in Getafix [31]. Then, we estimate the cost of a query as the number of documents it touches to more accurately balance load. Finally, we use the parallelism-maximizing load balancing formulation described in Section 6. We show results in Figure 11.

First, we analyze the YCSB workload with a hotspot (11a). Load is concentrated on just two shards, so without load balancing, performance is poor. Both load balancing and replication increase performance dramatically (>50%), but the cost model and parallelism optimization have smaller effects (<15%). We hypothesized that because most of the query load went to the two hotspot shards, any load balancer would place them on separate machines, naturally maximizing parallelism. We analyze the same workload without a hotspot (11b) and see smaller effects from balancing and replication and a larger effect from parallelism maximization (~25%). We then modify the query to increase parallelism by doubling the number of shards, increasing scan size, and fixing scan start points. We first run this more-parallel query with a hotspot (11c) and find large gains from load balancing and replication, but small gains from parallelism maximization. Once again, this is because the hotspot shards were all naturally placed on separate machines by the load balancer. We then run this more-parallel query without a hotspot (11d) and see large gains from parallelism maximization, improving tail latency by over 2×.

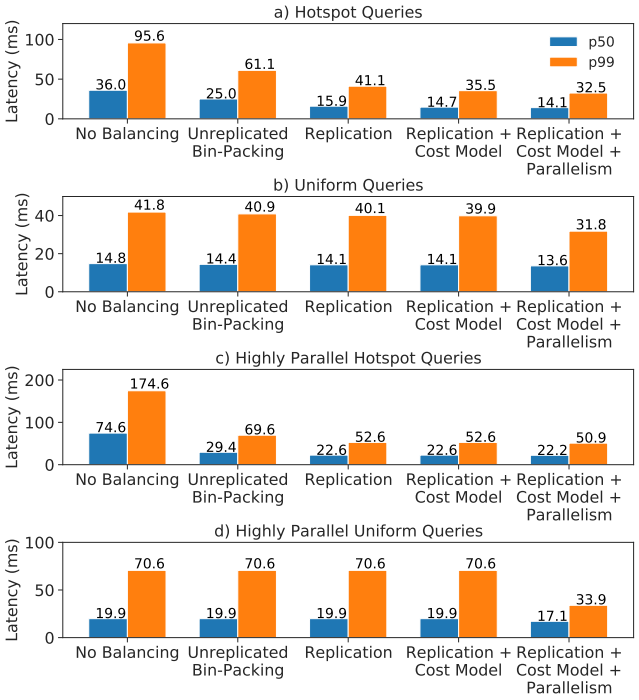

Figure 11: A factor analysis of the Uniserve load balancer conducted on MongoDB YCSB workloads with a hotspot, with a uniform query distribution, and with highly parallel queries with and without a hotspot.

Load Balancer Scaling. Finally, we evaluate the ability of the load balancer to scale to large numbers of shards and servers. We test the execution time of the load balancer using an IBM CPLEX 12.1 solver with access to 25 Intel Xeon 2.6 GHz cores and 20 GB of memory, reporting results in Figure 12. We average execution time over ten rounds of load balancing running a dynamic synthetic workload where shard loads are assigned from a Zipfian distribution. We set the number of servers to one tenth the number of shards and assign all shards the same memory usage such that each server can host at most sixteen shards. We evaluate with and without the parallelism optimization. With the optimization, execution time is similar to that reported by prior MILP-based load balancers such as Accordion [41]–152 seconds at 1024 shards. Without the optimization, performance is better–21 seconds at 1024 shards. We also tested the the sensitivity of parallelism load balancer performance to the number of parallel queries considered and saw no significant change in execution time between 100 and 2000 unique queries considered.

9 Related Work

Abstractions for Distributed Systems. The inherent difficulty of working in a distributed setting has encouraged the

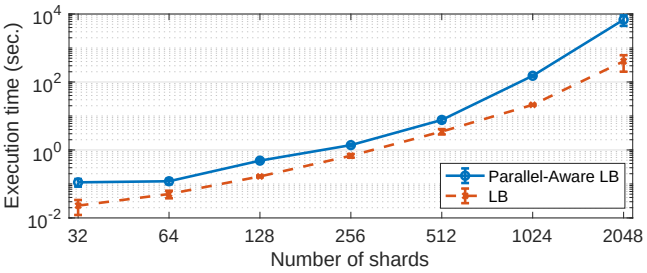

Figure 12: Log-log execution time of the load balancer.

development of many abstractions for distributed computing. Perhaps the most successful has been MapReduce [28], on which the Uniserve query model is based, though Uniserve extends MapReduce with its shard abstraction to more easily query dynamically changing data structures such as indexes. Uniserve shards are similar to persistent objects in Thor [36]; both systems protect objects from server failures and provide a transactional interface to modify them. However Thor, predating both MapReduce and the modern cloud, does not provide a distributed query interface, load balancing, or auto-scaling. Uniserve shards are also analogous to actors in systems such as Erlang [19] and Orleans [24]. However, actors are units of computation while Uniserve shards are partitions of data so implementations differ greatly; for example, neither system provides a distributed query model over actors.

The auto-sharding system Slicer [18], like its predecessor the lease manager Centrifuge [17], assigns data and queries to shards based on partition keys, much like Uniserve. However, Slicer is less integrated than Uniserve: it does not move Shard X to Server Y, but instead tells Server Y that it is about to start receiving queries for Shard X. As a result, Slicer require extensive infrastructure and struggle with consistency; for example it cannot replicate writable shards.

Many middleware systems, like Uniserve, have been developed to ease building distributed databases. Middleware systems distribute data and queries across existing database installations; like Uniserve they provide useful features such as fault tolerance [35, 40] and load balancing [20]. However, middleware solutions are typically specialized to particular database types, such as relational databases [25, 26] or NoSQL [30] stores. To the extent of our knowledge, Uniserve is the first system to effectively distribute many diverse data types and query models, including those of popular query serving systems such as Druid, MongoDB, Solr, and the many systems we discussed in Section 4.

Load Balancing and Elasticity. The mixed-integer linear programming (MILP) formulation underlying the Uniserve load balancer is inspired by similar formulations described in E-Store [45] and Accordion [41]. Both E-Store and Accordion find the placement of data on servers that minimizes data transfer costs while balancing query load and and ensuring no server's memory capacity is exceeded. This works well for write-heavy transactional workloads, but as Uniserve targets read-mostly analytics workloads, we extended the formulation to maximize query parallelism and replicate hot shards, dramatically improving performance as we showed in Section 8.

Other systems have developed load balancers for analytics workloads. The most similar to Uniserve is Getafix [31], a proposed load balancer for Druid that treats balancing as a bin-packing problem. This is effective at balancing query load and replicating hot shards, but unlike Uniserve it is designed exclusively for Druid, assumes all queries are equally expensive, and is not parallelism-maximizing or elastic. NashDB [37] balances read-only OLAP workloads using a greedy algorithm inspired by economic models; unlike Uniserve it requires users to assign a monetary "value" to each query, assumes keys can freely move between shards, is implemented only for SQL queries, and does not consider query parallelism. While we do not know of any system implementing a parallelism-maximizing optimization similar to Uniserve, transactional systems such as Accordion [41] and Clay [42] do the opposite, placing data frequently queried together on the same machine to avoid distributed transactions.

The Uniserve auto-scaler uses CPU utilization thresholds to scale cluster size, adding nodes when load exceeds an upper threshold and removing nodes when load is below a lower threshold. This is similar to the auto-scaling algorithms of existing database systems such as E-Store [45]. It is also similar to popular cloud auto-scaling services such as AWS Auto-Scaling [21], which use proprietary algorithms but share the objective of keeping resource utilization levels near a target. NashDB [37] takes a related approach, asking users to assign a monetary "value" to each query and a "cost" to each server, then adding servers until the marginal value a server provides is less than its cost. Alternatively, Accordion [41] uses a MILP formulation to model server capacity for distributed transactions and uses the smallest number of servers that have capacity for all transactions occurring on the cluster; however this is specialized for distributed transactional workloads.

10 Conclusion

Query serving systems are increasingly important, but developing them is unnecessarily difficult because they all must reimplement the same distributed concerns. In this paper, we described Uniserve, which architecturally separates these concerns into a distribution layer that can be bolted on to query serving systems to distribute them while providing key capabilities such as fault tolerance, load balancing, and elasticity. Uniserve distributes systems using simple abstractions for queries and data shards that most query serving systems can express, despite their diversity. We have used Uniserve to distribute three popular query serving systems: Druid, Solr, and MongoDB, showing equal or superior performance, and have shown it is general enough to distribute many others.

References

[1] How to Setup ElasticSearch Cluster with Auto-Scaling on Amazon EC2? https://stackoverflow.com/questions/18010752/, 2015.

[2] MongoDB Cluster with AWS Cloud Formation and Auto-Scaling. https://stackoverflow.com/questions/30790038/, 2016.

[3] Apache Solr. https://lucene.apache.org/solr/, 2020.

[4] Atlas. https://github.com/Netflix/atlas, 2020.

[5] ClickHouse. https://clickhouse.tech/, 2020.

[6] DB-Engines Ranking. https://db-engines.com/en/ranking, 2020.

[7] Elasticsearch. www.elastic.co, 2020.

[8] InfluxDB. https://www.influxdata.com/, 2020.

[9] Janus Graph. https://janusgraph.org/, 2020.

[10] MongoDB. https://www.mongodb.com/, 2020.

[11] MongoDB for Analytics. https://www.mongodb.com/analytics, 2020.

[12] Neo4j. https://neo4j.com/, 2020.

[13] OpenTSDB. http://opentsdb.net/, 2020.

[14] Solr Distributed Requests. https://lucene.apache.org/solr/guide/8_5/distributed-requests.html, 2020.

[15] TimescaleDB. https://www.timescale.com/, 2020.

[16] YCSB GitHub. https://github.com/brianfrankcooper/YCSB, 2020.

[17] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, volume 10, pages 1–16, 2010.

[18] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[19] Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1, 2007.

[20] Jaiganesh Balasubramanian, Douglas C Schmidt, Lawrence Dowdy, and Ossama Othman. Evaluating the performance of middleware load balancing strategies. In *Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004.*, pages 135–146. IEEE, 2004.

[21] Jeff Barr. New AWS Auto Scaling – Unified Scaling For Your Cloud Applications. 2018.

[22] Andrzej Białecki, Robert Muir, and Grant Ingersoll. Apache Lucene 4. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, page 17, 2012.

[23] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[24] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[25] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-Based Database Replication: the Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 739–752, 2008.

[26] Emmanuel Cecchet, Marguerite Julie, and Willy Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*, number CONF, 2004.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[28] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2004.

[29] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, D Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[30] Felix Gessert, Florian Bücklers, and Norbert Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *2014 IEEE 30th international conference on data engineering workshops*, pages 215–222. IEEE, 2014.

[31] Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta, and Himanshu Gupta. Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[32] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, et al. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data*, pages 583–594, 2018.

[33] Xavier Léauté. Benchmarking Druid. 2014.

[34] Roman Leventov. The Challenges of Running Druid at Large Scale, Nov 2017.

[35] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2005.

[36] Barbara Liskov, Atul Adya, Miguel Castro, Sanjay Ghemawat, R Gruber, U Maheshwari, Andrew C Myers, Mark Day, and Liuba Shrira. Safe and efficient sharing of persistent objects in thor. *ACM SIGMOD Record*, 25(2):318–329, 1996.

[37] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 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*, pages 1253–1267, 2018.

[38] Michael McCandless. Lucene nightly benchmarks. 2020.

[39] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[40] Marta Patiño-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375–423, 2005.

[41] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[42] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[43] Jihoon Son. Apache Druid 0.18.0 Released. 2020.

[44] Michael Stonebraker. One Size Fits All: An Idea Whose Time has Come and Gone. *Communications of the ACM*, 51(12):76–76, 2008.

[45] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

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

[47] William Zola. On Selecting a Shard Key for MongoDB. 2019.