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

Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, Matei Zaharia

## Abstract

Over the past few years, many organizations have developed *query serving systems* that enable real-time analytic queries over diverse data types. These include full-text search systems like ElasticSearch, time series databases like OpenTSDB, and OLAP serving layers like Druid. These systems must scale to support terabytes or petabytes of data. However, because their workloads are characterized by high-volume real-time queries dependent on specialized data structures, they are a poor fit for cluster computing frameworks such as Spark. Instead, developers scale them by building custom distribution layers requiring tens or hundreds of thousands of lines of code. To ease and improve the development of query serving systems, we propose Uniserve, a unifying distribution layer for query serving systems. The core of Uniserve is a small but expressive set of abstractions for data storage and queries. Uniserve can construct a distributed query serving system from single-node code that implements these abstractions, horizontally partitioning data into shards and distributing queries across shards. Uniserve can also provide key user-requested features such as elasticity and load balancing that are critical for fully leveraging cloud environments but are often missing from existing query serving systems. We evaluate Uniserve both by porting popular existing systems to use it, such as Druid, Solr, and MongoDB, and by building new systems on it, such as a simplified data warehouse based on MonetDB and a social graph store similar to Facebook TAO. Each of these implementations requires <1K lines of code, but all meet or exceed, sometimes by orders of magnitude, the performance of comparable natively distributed systems.

## 1 Introduction

The last few years have seen explosive growth in the need for systems that enable real-time analytic queries over diverse types of data [7]. Examples of these *query serving systems* include online analytical processing (OLAP) systems such as Druid [49], Pinot [34], and Clickhouse [6], which execute complex analytical queries over tabular data; time series databases such as OpenTSDB [12], which aggregate and analyze operational logs; and full-text search systems such as ElasticSearch [8] and Solr [4], which use sophisticated indexes to search text documents. Query serving systems have two defining characteristics. First, their workloads consist of high-volume heterogeneous parallel queries; they must often serve thousands of unique requests per second to power real-time applications. Second, they store and manage their own data, often using specialized data structures such as indexes. Organizations design new query serving systems because no single system can satisfy all needs: specialized systems can outperform general systems by orders of magnitude [46].

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. Accordingly, most query serving systems use custom-built distribution layers to scale their operations to hundreds or thousands of machines. These custom solutions often require tens or hundreds of thousands of lines of code (for example the core of Solr, a distribution layer for Apache Lucene [22], is over 300K lines of code) written over a correspondingly vast number of person-years. However, their actual functionality varies little between systems: they distribute data and queries across machines and reimplement solutions to standard distributed systems problems such as consistency and fault tolerance. Moreover, these custom distribution layers are often difficult to adapt to changing user demands. For example, most were designed for traditional cluster deployments, though users now increasingly deploy query serving systems in the cloud. As a result, most query serving systems do not leverage key cloud features such as elasticity: they do not resize their clusters as load changes and thus users must waste money keeping clusters overprovisioned or risk poor performance when load spikes [29, 48].

For many applications, especially those involving large batch computations, cluster computing frameworks such as MapReduce [28] and Spark [50] eliminated the need for custom distribution layers by providing a general model for distributing computation. However, these frameworks cannot distribute the workloads of query serving systems for two reasons. First, they assume workloads consist of a predictable number of large batch queries which take seconds to minutes to complete. However, query serving systems often serve thousands of concurrent queries per second to power real-time applications; this query load often varies both across different data items and over time [31, 49]. Therefore query serving systems have stricter latency requirements than cluster computing frameworks and benefit greatly from elasticity and load-balancing optimizations that cluster computing frameworks do not consider. Second, cluster computing frameworks give users little control over how their data is stored, using the same format for all data (e.g. a collection of objects in memory or a columnar-compressed file on disk). In contrast, most query serving systems store data in specialized data structures such as Solr and ElasticSearch inverted indexes (optimized for search) or Druid and Pinot segments (optimized for time series queries). These improve performance by orders of magnitude compared to a generic format.

In this paper, we investigate whether it is possible to create a more general framework for distributing query serving systems. We propose Uniserve, a unifying distribution layer that can be "bolted on" to query serving systems through a transparent interface. Uniserve makes query serving systems both more powerful and easier to develop: instead of building complex custom solutions their developers can leverage Uniserve's optimizations, features, and guarantees.

There are three core challenges inherent in designing a unifying distribution layer such as Uniserve. First, it must support and manage the diverse data structures of existing query serving systems, such as NoSQL stores, time series segments, and inverted indexes. Second, it must express the diverse query patterns of existing systems, which range from SQL dialects to specialized OLAP queries to full-text search. Third, it should maximize system performance through optimizations effective on diverse query serving workloads.

Uniserve addresses these challenges through a small but expressive set of abstractions for data storage and queries. Uniserve can construct a distributed query serving system from single-node code that implements these abstractions. We sketch the architecture of Uniserve in Figure 1.

The first challenge Uniserve addresses is managing the diverse storage types of query serving systems, including relational tables, time series segments, and inverted indexes. Uniserve unifies these data types with a *shard* abstraction. 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). Unlike the restricted data formats of conventional cluster computing frameworks, the Uniserve shard abstraction does not restrict how data is stored. Therefore, it can express many different data structures from different systems, including relational tables, Druid segments, and Lucene indexes.

The second challenge Uniserve addresses is expressing the diverse query patterns of query serving systems. 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 query workloads with another abstraction: a small but expressive programming interface for distributed queries based around executing map, aggregation, and join operations on data shards. This interface can express many operations including core relational algebra operators such as group-bys and joins as well as the query languages of Druid, MongoDB, Solr, and many other popular systems.

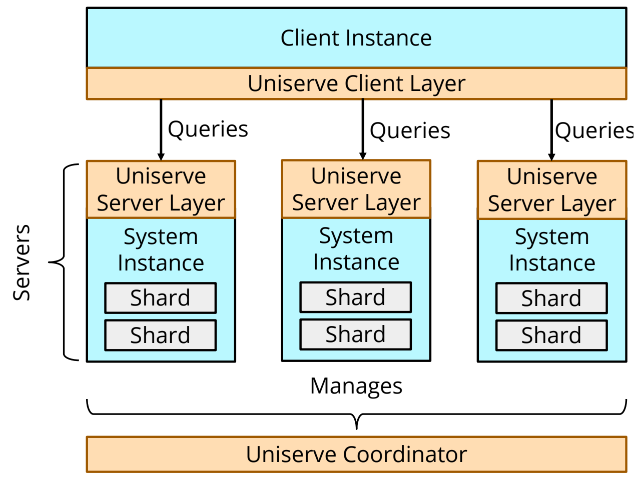

**Figure 1:** Uniserve constructs a distributed query serving system from single-node code. It runs as a distribution layer (in orange) bolted onto instances of the single-node code (in blue), partitioning data into shards (in gray) and distributing queries across shards.

The third challenge Uniserve addresses is providing features that maximize the performance of diverse query serving systems, such as auto-scaling and load balancing algorithms. Currently, all query serving systems must develop custom implementations of these features. Such implementations are often limited—for example, the Druid and MongoDB load balancers assume all data items receive the same amount of load (which is problematic in practice [31]) and few query serving systems are natively elastic, despite the importance of elasticity to the cloud [29]. Uniserve replaces these disparate custom implementations with general-purpose auto-scaling and load balancing algorithms that work with arbitrary query serving systems. Our auto-scaling algorithm uses consistent hashing (similar to Snowflake [48]) and replication-based shard prefetching to seamlessly scale cluster size without latency spikes or periods of unavailability induced by shard reshuffling. Our load balancing algorithm greedily reassigns shards so all servers receive the same amount of load (much like E-Store [47]), but uses a set of simple but effective rules to replicate extremely hot shards that the greedy algorithm cannot handle. We show that our general-purpose algorithms add new functionality to existing query serving systems while making it easier to build new ones.

To evaluate Uniserve, we use it to distribute five systems. First, we port three popular distributed query serving systems—Druid, Solr, and MongoDB—to use Uniserve as their distribution layer. Then, we build two new systems using Uniserve: a social graph store similar to Facebook TAO [23] and a simplified data warehouse based on the single-node column store MonetDB [33]. We can implement each of our five systems in fewer than 1000 lines of code, though they replace custom distribution layers that require tens or hundreds of thousands of lines. We show that our three ports match the performance of natively distributed systems on simple static workloads and perform better on dynamic or skewed workloads because of Uniserve features such as load balancing and elasticity. We also show that our social graph store matches the original performance of TAO, while our simplified data warehouse outperforms the popular distributed query engine Spark-SQL [17] and is competitive with Redshift [32]. We additionally show how Uniserve's API can support many other popular query serving systems, including Clickhouse [6], ElasticSearch [8], InfluxDB [9], and Pinot [34].

To summarize, our contributions are:

- We propose Uniserve, a bolt-on distribution layer for query serving systems that can construct a distributed system from single-node code using a small but expressive set of abstractions for data storage and queries.
- We develop general-purpose auto-scaling and load-balancing algorithms for Uniserve, improving performance on dynamic or skewed workloads.
- We port three popular systems to Uniserve and build two new ones on it and find they match or exceed the performance of comparable natively distributed systems.

## 2 Background and Motivation

To motivate Uniserve, we examine three popular distributed query serving systems in detail. We show how their workloads are a poor fit for existing cluster computing frameworks, but a good fit for Uniserve. We further show how each benefits from Uniserve features such as elasticity and load balancing that are missing from the original systems despite user demand.

### 2.1 Case Studies

**Apache Solr.** Apache Solr [4] is a distributed full-text search system. It provides a rich query language for searching text documents distributed over many machines and is optimized to serve thousands of queries per second at subsecond latencies. Solr stores documents in inverted indexes based on Apache Lucene [22] which improve search speeds by orders of magnitude. Solr does not balance server load, but few of its workloads exhibit load skew. It is not natively elastic, but does provide an auto-scaling API exposing useful information to an external auto-scaler. Solr resembles other distributed full-text search systems, such as Elasticsearch [8].

**Apache Druid.** Apache Druid [49] is a high-performance analytics system. Druid provides fast ingestion and real-time search and aggregation of time-ordered tabular data, such as machine logs; it can serve thousands of queries per second at millisecond latencies. It achieves its high performance through specialized *segment* data structures that store data in a tabular format, but use summarization, compression, and multiple levels of indexes to achieve query performance orders of magnitude better than conventional databases [49]. The Druid load balancer only ensures all servers host the same number of segments [3]; it assumes all segments receive the same amount of load [31, 36]. Druid is not natively elastic and has no native support for auto-scaling. Druid resembles OLAP systems such as Pinot [34] and Clickhouse [6], as well as time series databases such as OpenTSDB [12].

**MongoDB.** MongoDB [10] is a NoSQL database. Unlike Solr and Druid, it is not primarily an analytics system, but is often used for analytics [11]. MongoDB performs search and aggregation queries over semi-structured data. It uses a schemaless document-oriented data format, backed up by indexes, to give users flexibility in how their data is stored and queried without sacrificing performance relative to a traditional tabular RDBMS. The MongoDB load balancer only ensures all servers host the same number of shards; it assumes all shards receive the same amount of load. MongoDB is not natively elastic and has no native support for auto-scaling [2].

### 2.2 Discussion

Solr, Druid, and MongoDB each provide valuable functionality that makes them some of the most popular analytics systems in the world [7]. Users desire this functionality at massive scale–searching terabytes of documents, for example, or analyzing months of logs from millions of machines. However, query serving workloads are a poor fit for existing distributed computing frameworks, such as MapReduce or actor models. MapReduce frameworks like Spark [50] are designed for a low volume of large batch queries on data stored in restricted formats, but query serving workloads are characterized by high-volume heterogeneous queries that depend on specialized data structures. Actor models like Orleans [24] provide a low-level messaging API between isolated stateful actors, while query serving workloads require a high-level query model and abstractions such as replication.

Because query serving systems cannot use these frameworks, they depend on custom distribution layers that comprise tens or hundreds of thousands of lines of code but are frequently missing key user-requested [1–3, 31, 36] features such as load balancing and elasticity. In the remainder of this paper, we show how systems can use Uniserve to replace these custom distribution layers with only a few hundred lines of code. We further show how Uniserve can improve query serving system performance through features such as load balancing and elasticity.

## 3 Uniserve Overview and Interfaces

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

### 3.1 Using Uniserve

To construct a distributed query serving system using Uniserve, a developer must provide a single-node 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

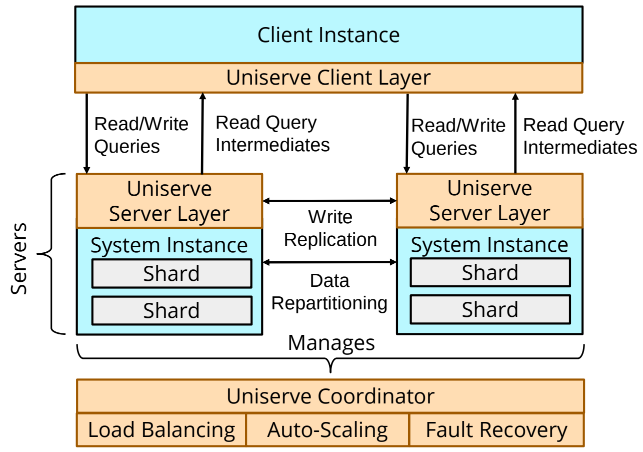

**Figure 2:** The Uniserve architecture. Uniserve runs as a thin layer (in orange) on client and server instances of the underlying system (in blue). Uniserve partitions data into shards (in gray) and distributes queries across shards. A coordinator manages cluster state and provides distributed features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

how to create, destroy, serialize, and deserialize shards. Systems implement shards using data structures that can store horizontal partitions of data. Commonly, these are database tables or the local equivalent, such as MongoDB collections or Druid datasources.

The query interface is a distributed query planner that constructs Uniserve read and write query plans from queries in the underlying system's query language. Read query plans execute queries on shards; write query plans add data to shards.

## 3.2 Uniserve Architecture

A Uniserve cluster is made up of many servers, each running an instance of the underlying single-node system. We diagram a Uniserve cluster in Figure 2. A thin Uniserve layer runs on each server and client. Each server layer receives Uniserve query plans from clients and executes them using its instance of the underlying system. By storing data and executing queries in the underlying system, Uniserve automatically leverages the system's storage and query optimizations. Each client layer constructs Uniserve query plans from native queries and sends them to servers for execution. A coordinator manages cluster state: it assigns shards to servers, handles failures, and adds and removes servers as load changes. Uniserve also assumes the existence of durable storage (such as S3 or HDFS) on which it backs 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 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.

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

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

interface Row:
  Int getPartitionKey()

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

interface<E, I, T> ReadQueryPlan:
  Map<String, List<Int>> keysForQuery()
  String getAnchorTable()
  List<Int> getAnchorPartitionKeys(Shard s)
  Map<Int, E> mapper(Shard s, int numReducers, Map
      <Int, List<Int>> anchorPartitionKeys)
  I reducer(Shard anchorShard, Map<String, List<E
      >> repartitionedData)
  T aggregateResults(List<I> reducerResults)

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

**Figure 3:** The Uniserve shard and query interfaces. Uniserve can construct a distributed query serving system from a single-node system implementing these interfaces.

Uniserve divides shards into *tables*. A table is a group of shards containing a single set of data, much like a database table. Uniserve write queries are always made to a specific table; read queries may be made to one table or to multiple.

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.

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 Solr document. Each row must expose an integer partition key accessed through `getPartitionKey`. Uniserve assigns rows to shards using the hash of their partition key; it guarantees that all rows with the same partition key are assigned to the same shard. Critically, while Uniserve writes data using rows, it imposes no restrictions on how that data is stored; shards can use any data structure imaginable.

## 3.4 Clients and the Query Interface

The Uniserve client layer constructs query plans from queries and sends them to servers for execution. It requires the underlying system to provide a `QueryPlanner`, a distributed query planner that constructs Uniserve read and write query plans from native system queries.

A Uniserve `ReadQueryPlan` repartitions data across the cluster, then aggregates it into a query result. Each query runs on a set of tables and shards defined by its the `keysForQuery` function, which returns a map from table name to a list of partition keys in that table relevant to the query. Queries execute in three steps. First, mappers repartition shards, returning a mapping from partition number to a chunk of repartitioned data. Second, reducers aggregate the repartitioned data. Each reducer processes one partition of data, taking in all chunks of data created by mappers for that partition and returning an intermediate result. Third, the client aggregates all intermediates into a final query answer using `aggregateResults`.

As a simple example, let us say a query counts the number of rows in the natural join of two tables, neither of which is partitioned on the join column. In this query, mappers repartition shards on the join column, returning a map from partition number to the set of rows whose join column values are in that partition. Reducers receive all rows in a partition from both tables, use the underlying system to compute their natural join, and return its size. The client receives the size of the natural join of each partition and returns their sum.

Sometimes, one table in a query does not need repartitioning. This may be because the table is already partitioned on the query's join column or because the query is a simple single-table query such as a row count or search. In such a case, that table may be declared an *anchor table*. Mappers do not execute on the anchor table, but instead repartition other tables to match the anchor table's partitioning. They do this using a map of anchor table shard partition keys, created with `getAnchorTablePartitionKeys`. Each reducer combines one shard of the anchor table with its partitions of data from other tables. For example, in a query on a large data table and small fact table, the large data table would be the anchor. The mapper broadcasts the entire fact table to every shard of the data table. Each reducer executes the original query on the fact table and one shard of the data table. The client aggregates the reducers' results into a final query answer.

A Uniserve `WriteQueryPlan` writes rows of data to shards. Uniserve writes each row to the shard corresponding to its partition key. Writes are replicated and are consistent, durable, and linearizable. We 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 [4]. Natively, Solr stores data by sharding 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. To port Solr's distributed data storage capabilities to Uniserve, we run a single-node Solr instance on each cluster server. We implement Uniserve shards as single-shard Solr collections 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. We implement all shard and write query function using direct native equivalents; for example, we implement `createNewShard` using `createCollection`.

| | System Type | Data Type | Query Operations |
|---|---|---|---|
| **Druid** [49] | OLAP | Tables | Aggregate, group, filter, broadcast join |
| Pinot [34] | OLAP | Tables | Aggregate, group, filter |
| ClickHouse [6] | OLAP | Tables | Aggregate, group, filter, join, |
| OpenTSDB [12] | Time series DB | Time series | Aggregate, group, filter |
| Atlas [5] | Time series DB | Time series | Aggregate, group, filter |
| InfluxDB [9] | Time series DB | Time series | Aggregate, group, filter |
| **MonetDB** [33] | Data Warehouse | Tables | Aggregate, group, filter, join |
| **Solr** [4] | Full-Text Search | Indexed text | Group (including faceting), search |
| ElasticSearch [8] | Full-Text Search | Indexed text | Group (including faceting), search |
| **TAO** [23] | Graph Database | Graphs | Search graph edges and nodes |
| **MongoDB** [10] | NoSQL | Documents | Aggregate, group, filter, map, broadcast join |

**Table 1:** Some systems Uniserve can distribute and their properties. Systems we have implemented are in bold.

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. Natively, Solr distributes queries by searching each shard separately, then combining results on a single node [13]. To port Solr's distributed query capabilities to Uniserve, we implement a `QueryPlanner` that constructs an anchored single-table `ReadQueryPlan` from any Solr query. In these plans, reducers search shards separately, then the client combines their results using `aggregateResults`. For example, in the plan for a query that searches for books whose title contains the word "goblin," grouped by year, reducers search shards separately for "goblin" and group their results by year. Then, the client combines the per-year lists. The Uniserve port of Solr is implemented in ~500 lines of code, can execute any query recognized by the standard Solr parser, and, as we show in Section 8, matches or exceeds native Solr performance.

## 4 Generality of Uniserve

In this section, we demonstrate Uniserve's generality by cataloging some of the diverse systems it can distribute, summarized in Table 1. 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 and rankings of processed (grouped, filtered, etc.) tabular data. These queries are easy to implement in Uniserve by sharding data, querying each shard separately, and aggregating results. Some OLAP systems allow joins of multiple tables. To implement these queries, we repartition tables on the join key before executing the query as before. We have implemented a Uniserve port of one OLAP system, Druid [49]. Other OLAP systems that Uniserve can distribute include Pinot [34] and Clickhouse [6].

**Time series Databases.** Time series 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 time series. All these operations are easy to implement in Uniserve, so Uniserve can distribute most time series databases, such as OpenTSDB [12], Atlas [5] and InfluxDB [9]. However, Uniserve is optimized for read-mostly workloads while time series 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 text search queries. Because all their queries are searches, they are easy to implement in Uniserve, as we showed in Section 3.5. We have implemented a Uniserve port of one full-text search system, Solr [4]; it can also distribute ElasticSearch [8].

**Graph Databases.** Graph databases query graph data structures. Their queries vary from whole-graph algorithms such as PageRank to graph traversal queries such as finding all nodes within *N* hops of a target to tabular queries such as finding all graph nodes that meet some criteria. Uniserve can distribute some graph databases, such as TAO [23], whose query languages can be expressed as aggregations of queries on shards. However, it cannot distribute others, such as Neo4j, whose graph traversal queries Uniserve cannot efficiently express.

**Other Systems.** Uniserve can distribute systems whose queries can be expressed as aggregations of potentially repartitioned data even if they do not fall into the above categories. Two such systems we implemented are the NoSQL store MongoDB and a simplified data warehouse based on MonetDB.

**Limitations.** Uniserve is optimized for read-mostly analytics workloads. It has two major limitations. First, it can only execute queries that can be expressed by its query model. As we have shown, this includes the query languages of many popular query serving systems, but not some specialized query types such as graph traversal queries.

Second, Uniserve does not support workloads with many small point updates. It only allows a single writer at a time, which works well for large coarse-grained batch writes but not for small point updates. Small point updates are rare in analytics systems, but common in other systems such as relational databases which Uniserve cannot effectively distribute.

## 5 Writes 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. The coordinator keeps every replica of a shard aware of every other replica. To execute a write, the Uniserve client layer sends each row to a replica of the shard corresponding to the row's partition key. These shards `prepareCommit` the write on the rows they receive and pass them on to all other 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. After a write commits but before it returns, a replica synchronously uploads the updated shard to durable remote storage such as S3.

Uniserve does not support concurrent writes to a table—only a single write transaction per table may be active at any point in time. This is enforced through distributed locks in ZooKeeper. However, a single write transaction may concurrently write any number of rows to any number of shards in a single table. We assume most writes are large batch updates, which are typical in most of our target systems.

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. Because writes cannot be concurrent, they must occur in serial order. Because shards are synchronously backed up to durable storage after each write, writes are durable and cannot be lost.

### 5.2 Fault Tolerance

Uniserve mitigates server failures, even mid-query, using common fault tolerance techniques. Like many distributed systems [40], 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 ping each other in a decentralized heartbeat system inspired by RAMCloud [40]. 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 only copy of a shard, the coordinator orders a server to load the shard from durable storage.

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.

## 6 Elasticity and Load Balancing

In this section, we discuss how Uniserve maximizes the performance and efficiency of query serving systems by providing two critical features: elasticity and load balancing.

### 6.1 Elasticity

When deployed in an elastic cloud environment such as Amazon EC2, Uniserve automatically scales cluster size in response to load changes. The greatest challenge in scaling cluster size is that of reshuffling shards efficiently: changing shard-to-server assignments when servers are added or removed without hurting performance as expensive shard transfers compete with queries for resources. Uniserve solves this problem using consistent hashing and shard prefetching.

Uniserve assigns shards to servers using ring-based consistent hashing similar to Chord [45] or Snowflake [48]. This guarantees that when a server is added or removed, only on the order of $1/N$ shards (where $N$ is the number of servers) must be moved. This also guarantees that when a server is added, all shard transfers are to the new server and that when a server is removed, all shard transfers are from the removed server. These guarantees minimize the amount of reshuffling needed and make reshuffling easier.

Uniserve scales cluster size using a simple utilization-based algorithm similar to the algorithms used in cloud autoscalers [21]. Uniserve periodically collects average cluster utilization 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.

To add a server, Uniserve asks the cloud environment to create a new server using a specified image and launch script. When the new server comes online, Uniserve does not immediately notify clients or other servers. Instead, it creates on the new server replicas of all shards that will be assigned to it when the consistent hash is updated to add that server. These replicas receive no queries, but replicate all writes that the originals receive. Only once all replicas are loaded does Uniserve notify all clients and servers of the new server. Clients then begin routing queries to the new server. Since the original copies of the transferred shards are no longer handling any queries, they can then be safely deleted. This process maximizes query performance by ensuring no query runs concurrently with a shard download.

Removing a server follows a similar process in reverse. Uniserve first selects at random a server for removal. It then creates on other servers replicas of the shards that will be assigned to them when the consistent hash is updated with the selected server removed. Once all the replicas are created, Uniserve notifies all clients and servers about the server removal. Clients then stop querying the selected server; they instead query the replicated shards on other servers. Since the selected server is no longer handling any queries, Uniserve can safely ask the cloud environment to terminate it. This process guarantees that servers can be removed without any period of shard unavailability. It requires queries and shard transfers to run concurrently, but this is not a problem because servers are only removed when cluster load is low and servers have excess compute capacity.

**Algorithm 1:** Load Balancer

```
input  : $N$: number of servers in cluster
         ($L_m$): load on server $m$
         ($l_s$): load on shard $s$
         ($r_s$): number of replicas of shard $s$
         $L^{\delta}$: max difference from avg. server load
         $l^{dr}$: shard load de-replication threshold
/* replication                                        */
for every shard $s$ do
    while $l_s/r_s > \sum_m L_m/N$ do
        host new replica of $s$ most under-loaded server
          $m$ that does not already host $s$
        $r_s \leftarrow r_s + 1$
        update ($L_m$) with new shard load distribution
    end
    while $r_s > 1$ and $l_s/r_s < l^{dr}$ do
        remove $s$ replica from most loaded host server
        $r_s \leftarrow r_s - 1$
        update ($L_m$) with new shard load distribution
    end
end
/* shard transfer                                     */
for every server $m$ do
    while $L_m > (\sum_m L_m/N) + L^{\delta}$ do
        transfer largest shard $s$ from $m$ to
          under-utilized server $m'$ that is not hosting a
          replica of $s$ and satisfies $L_{m'} + l_s/r_s < L^{\delta}$
        $L_{m'} \leftarrow L_{m'} + l_s/r_s$,  $L_m \leftarrow L_m - l_s/r_s$
    end
end
```

### 6.2 Load Balancing

Uniserve must balance query load, defined as the number of queries sent to shards on a server, across servers to effectively utilize resources and provide consistent performance. Past work on load balancing for similar problems falls into two categories. First, several algorithms have been proposed for calculating optimal shard assignments using integer programming or cubic solvers [31, 43], but these algorithms are too expensive to run over millions of shards in a large query serving system. Second, researchers have proposed greedy load-balancing for shard placement in relational databases [47], but this class of work assumes there will only be one replica per shard because it targets write-heavy transactional workloads where replicating shards would be too expensive. This assumption is problematic in Uniserve because it makes load balancing difficult when the load on a single shard is large.

In Uniserve, we propose a simple but effective greedy algorithm that extends prior work [47] to consider replication of hot shards without the high cost of optimization-based methods. Algorithm 1 describes this algorithm at a high level. Uniserve adds a replica to a shard when the load on each existing replica exceeds the average server load; the new replica is spawned on the least loaded server that does not already host a replica. Uniserve removes a replica from a shard when the load on each existing replica drops below a user-specified threshold $l^{dr}$; the replica is removed from the most-loaded server that hosts one. The user also specifies the threshold for the maximum deviation of server load from average server load ($L^{\delta}$). If the load on a server exceeds the sum of average load and $L^{\delta}$, Uniserve attempts to transfer its most-loaded shards to the least-loaded servers that can host them without increasing their own load beyond that threshold.

To track shard movement and replication, Uniserve maintains a *reassignment map* as part of its consistent hash. The reassignment map maps from shards to (potentially multiple) servers. Processes computing the consistent hash of a shard first check the reassignment map and use the reassignment map value, if present, instead of the consistent hash value.

When Uniserve moves or replicates a shard, it creates an entry in the reassignment map. It then prefetches a replica of the shard on the target server. To avoid any period of unavailability, Uniserve does not notify clients of the transfer until the replica finishes loading. After notifying clients, if the shard was only being moved (instead of being replicated), Uniserve deletes the original copy of the shard.

## 7 Distributing Systems with Uniserve

We evaluate Uniserve using five systems. First, we port three popular distributed query serving systems—Druid, Solr, and MongoDB—to use Uniserve as their distribution layer. Then, we build two new systems using Uniserve: a social graph store similar to Facebook TAO [23] and a simplified data warehouse based on the single-node column store MonetDB [33]. We can implement each of our five systems in fewer than 1000 lines of code, though they replace custom distribution layers that require tens or hundreds of thousands of lines. In this section, we describe how we built our ports and new systems.

**Solr.** We described the Uniserve port of Solr in Section 3.5.

**Druid.** In our Uniserve port of Druid [49], 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 time series 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 grouped data from datasources. The Uniserve port of Druid currently only supports simple Druid queries: sums, counts, or rankings of filtered and grouped data. We implement all Druid queries as single-table anchored `ReadQueryPlans` which execute the query on all shards separately then aggregate the results. Druid uses a similar execution model natively. Our port could easily be extended to support any other Druid query by adding support for more aggregation operators.

**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 tasks such as filtering, grouping, and accumulating documents. Uniserve can support any MongoDB operator, 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 shard separately, then combining results.

**TAO.** We have built using Uniserve a social graph store that implements the interface of Facebook TAO [23]. It stores data in SQLite. TAO data items are either objects (graph nodes, such as people or places) or associations (directed graph edges, such as friendships or check-ins). Because associations are directed edges, each has a primary object and secondary object. We implement shards as a pair of SQLite tables: one for objects and one for associations. We partition data just as TAO does: objects are hash-partitioned and associations are stored on the same shard as their primary object. We implement all `Shard` interface functions using direct SQLite equivalents.

The TAO query language, as described in the TAO paper, consists of four query types, each of which requests some information about a particular object and its associations. For example, the `assoc_count` query returns the number of associations of a certain type that have a certain primary object, such as the number of Facebook friendships a particular person has. We implement all four of these query types in SQLite and distribute them with Uniserve.

**MonetDB.** We have built using Uniserve a simplified data warehouse based on the single-node column store MonetDB [33]. It stores data in MonetDBLite [42], the embedded implementation of MonetDB. Each server runs MonetDBLite embedded in the same JVM as the Uniserve server layer. We implement shards as MonetDB tables and `Shard` interface functions, such as creation, destruction, serialization, and deserialization, using equivalents in the MonetDBLite API.

Our simplified data warehouse supports queries containing aggregations, filters, group-bys, and shuffle and broadcast joins, but not subqueries or views. It is not meant to be a full-fledged data warehouse, but rather a demonstration of Uniserve's ability to efficiently execute complex queries. It does not yet have distributed query planner; we instead plan queries manually. We implement simple aggregation queries using a single-table anchored query plan, as in our other systems. To execute queries that require repartitioning, such as shuffle joins, we have mappers query each data shard (pushing down filters and projections), retrieve data in binary format, and partition it by reducer. Reducers insert all data they receive into their local MonetDBLite instances and execute the original query on their data partition. The client then combines per-partition responses into a final result.

## 8 Evaluation

We evaluate Uniserve using the five systems discussed in Section 7. We demonstrate that:

1. Uniserve ports of distributed systems match the performance of natively distributed systems under ideal conditions, such as static workloads without load skew.

2. Uniserve ports of distributed systems outperform natively distributed systems under less ideal conditions – workloads that change, have load skew, or have server failures – because of Uniserve features such as elasticity and load balancing.

3. Distributed systems built using Uniserve and a specialized single-node system, such as our simplified data warehouse built using MonetDB, can match or outperform popular distributed systems, such as Spark-SQL and Redshift, on their core workloads.

### 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 MonetDBLite-Java 2.39.

When benchmarking Solr, Druid, and MongoDB 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. We also disable query caching and set the minimum replication factor to 1.

When benchmarking systems with Uniserve, we use the implementations described in Section 7. 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 [39]. We run each query on 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 [35, 49], 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.

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 [14].

We benchmark our data warehouse based on MonetDB using several TPC-H queries (Q1, Q3, and Q10) at scale factors of 5 and 25, requiring 5GB and 25GB of data respectively.

We benchmark TAO using LinkBench [19], a suite of benchmarks created by Facebook engineers and designed to mimic real TAO workloads. We run LinkBench with default settings but only read queries, producing 5M rows of data.

### 8.3 Benchmarks

**Ideal Conditions.** We first benchmark our Solr, Druid, MongoDB, and TAO workloads 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 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 4. 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.

For TAO, we cannot compare to the native system directly because it is proprietary. However, we note that the ~1 ms median latency achieved by our implementation of the TAO API at low load is similar to the ~1 ms median latency on read queries reported by Facebook in the original TAO paper [23].

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 5. Uniserve query performance scales near-linearly, with slightly better scaling on the more-expensive sloppy benchmark.

**Data Warehouse Benchmarks.** We next benchmark our simplified data warehouse based on MonetDB, comparing its performance with native single-node MonetDB, the distributed SQL engine Spark-SQL [17], and the data warehouse

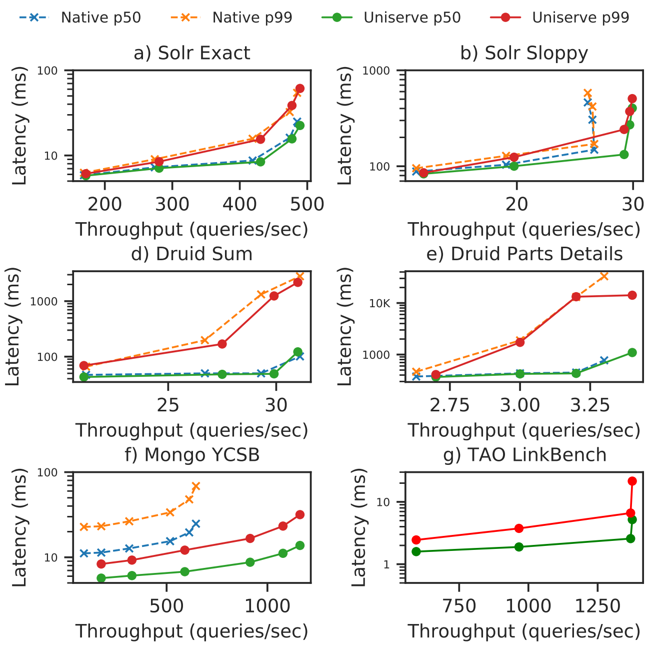

**Figure 4:** Throughput versus latency for natively-distributed and Uniserve-distributed queries on uniform and static workloads. Uniserve generally matches native performance.

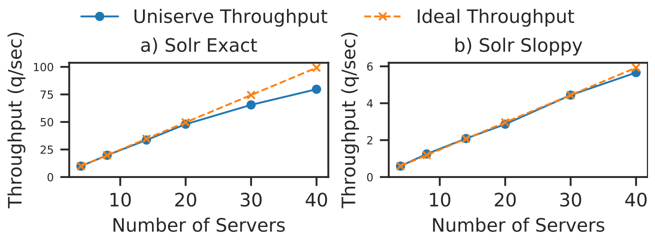

**Figure 5:** Uniserve scalability on the Solr benchmarks.

Redshift [32]. We use three TPC-H queries: Q1, a simple aggregation query; Q3, a three-way join; and Q10, a four-way join. We execute Q3 and Q10 using a mixture of broadcast and shuffle joins. We show results in Figure 6. We run multiple trials of each benchmark, reporting the average of results after performance stabilizes. This ensures Spark-SQL and Redshift have the opportunity to cache data in memory. To investigate the communication overhead of Uniserve, we first compare our system running on a single node to native single-node MonetDB. We find that Uniserve performs the same as native MonetDB on the simple aggregation query Q1 and performs significantly but not unreasonably worse on Q3 and Q10 due to the overhead of shuffling data. We then compare our system to Spark-SQL and Redshift on 160 cores (forty nodes for Uniserve and Spark-SQL, five dc2.8xlarge Redshift nodes). We find that Uniserve significantly outperforms Spark-SQL and is competitive with Redshift. This shows that by distributing a specialized system like MonetDB, Uniserve can in <1K lines of code match or outperform popular distributed systems such as Redshift and Spark-SQL on their core workloads.

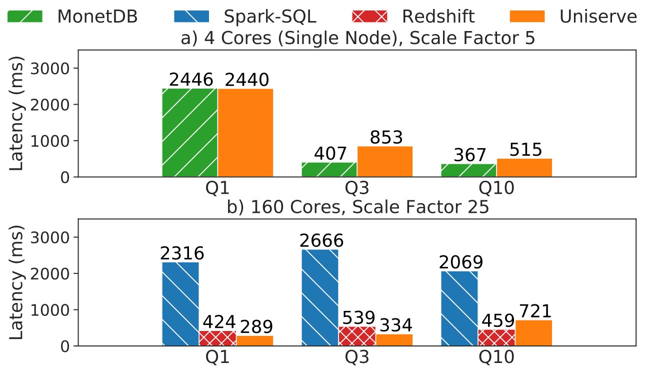

**Figure 6:** Comparisons between our simplified data warehouse, native single-node MonetDB, Spark-SQL, and Redshift on TPC-H queries Q1, Q3, and Q10 on 4 cores (single-node) and on 160 cores with TPC-H scale factors of 5 and 25.

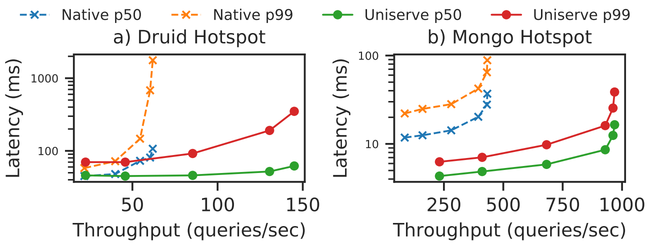

**Figure 7:** Throughput versus latency for natively-distributed and Uniserve-distributed queries where one slice of data receives 7/8 of queries. Uniserve balances load and so outperforms native systems.

**Hotspots.** To investigate the performance of the Uniserve load balancer, we next benchmark workloads under load skew. We look specifically at the Druid `parts_details` and MongoDB YCSB benchmarks. We send 7/8 of the queries to a single slice of data (four months 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 partition data so this slice is initially placed on one machine. We record throughput and latency while increasing the number of client threads, showing results in Figure 7. Because Uniserve balances load and moves items in the hotspot but Druid and MongoDB do not, Uniserve dramatically outperforms both.

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, Druid and MongoDB performance worsens with increasing skew.

**Dynamic Load.** We next investigate the performance of the Uniserve auto-scaler. We run the Solr sloppy benchmark for six hours sending queries at a target throughput, which varies from 240 to 1300 queries per minute. Uniserve starts with one server and adds or removes more as load changes. We show results in Figure 9. We see that Uniserve is always able to scale to meet the target throughput. As load increases, it adds

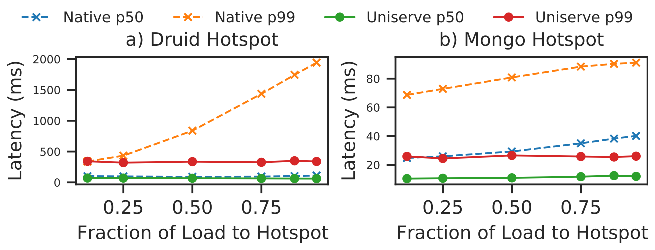

**Figure 8:** Latencies of natively-distributed and Uniserve-distributed queries on benchmarks with a hotspot, where one slice of data receives a varying fraction of all queries. Uniserve keeps performance constant as skew increases; native systems do not.

servers so there are always enough to process each query in time. As load decreases, it removes unnecessary servers but keeps enough to process incoming queries. Because the target query runs in parallel on all shards, adding servers decreases latency (as the query can run in parallel on more cores on more servers) and removing servers increases latency.

Importantly, Uniserve can resize clusters without losing performance. By using consistent hashing for shard assignment, Uniserve guarantees that only $1/N$ shards are moved when a server is added, and that all those shards are moved from current servers to the new server. Moreover, by prefetching replicas of those shards onto the new server before serving any queries, Uniserve guarantees that queries need not contend with shard transfers for resources. As a result, Uniserve can add or remove servers without affecting throughput or median latency. Tail latency does spike briefly when a server is added, but this represents only the handful of queries sent between when Uniserve notifies servers of the new server and when it notifies clients.

**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 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 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 all queries sent to Uniserve either fail or successfully complete, some "successful" Druid queries return incorrect results. Druid's query fault tolerance is known to be problematic in large-scale deployments [36]; Uniserve addresses its issues.

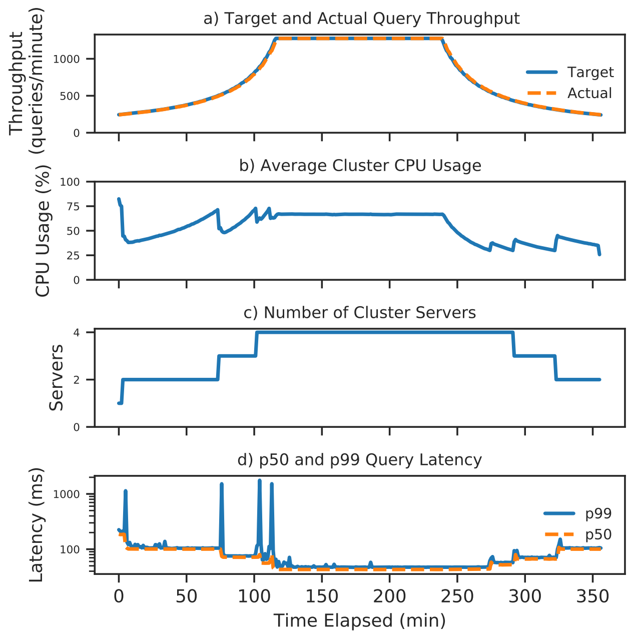

**Figure 9:** On the Solr sloppy benchmark with Uniserve auto-scaling, varying target throughput and observing effects on actual throughput, average cluster CPU usage, the number of cluster servers, and query latencies. We observe that actual throughput always matches target throughput and that resizing minimally affects performance.

**Load Balancer Scaling.** Finally, we investigate the performance scaling of the Uniserve load balancer. We run a synthetic workload where load is assigned to shards according to a Zipfian distribution and change the skew of the distribution over time to cause load imbalance. We run each trial for 40 iterations, use the first 10 to stabilize the shard-to-server assignment and report average execution time over the last 30. We set $L^{\delta}$ to one-fifth average load. We show the results in Figure 11. Load balancer execution time scales linearly with the number of shards in the cluster; Uniserve can balance load on clusters containing over 1M shards at 64 shards/server in approximately 10 seconds with an average of 64 transfers in each iteration, and 115 seconds in the extreme case of 4 shards/server with an average of 900 transfers per iteration.

## 9 Related Work

**Abstractions for Distributed Systems.** The inherent difficulty of working in a distributed setting has encouraged the development of many abstractions for distributed computing. Perhaps the most successful have been cluster computing frameworks based on MapReduce [28], such as Hadoop [44] and Spark [50]. Unlike Uniserve, these systems cannot effectively distribute query serving system workloads because they give users no control over how their data is stored (while query serving systems depend on specialized data structures) and are optimized for workloads containing a few large batch queries (while query serving workloads are comprised of high-volume real-time heterogeneous queries). Uniserve shards are similar

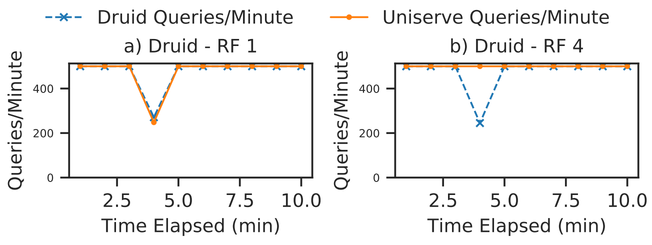

**Figure 10:** Query throughput (with a target of 500 queries/minute) of Druid-distributed and Uniserve-distributed TPC-H `sum_all` queries when one data server is killed after three minutes. The left graph shows performance starting with a single replica of each shard; the right graph with four.

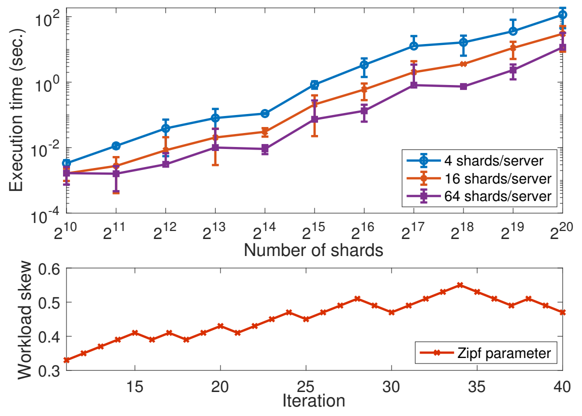

**Figure 11:** Load balancer average execution time (top) and changes in workload skew between load balancer iterations (bottom).

to persistent objects in Thor [38]; 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 [18] and Orleans [24]. However, actors are units of computation while Uniserve shards are partitions of data, so their implementations differ greatly; actors offer only a low-level messaging API while Uniserve offers a high-level query model.

The auto-sharding system Slicer [16], like its predecessor the lease manager Centrifuge [15], 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 requires extensive infrastructure and struggles with consistency; for example it cannot replicate writable shards.

Many middleware systems have been developed to ease building distributed databases. Database middleware systems distribute data and queries across existing database installations; like Uniserve, they provide useful features such as fault tolerance [37, 41] and load balancing [20], though not elasticity. However, middleware solutions are typically specialized to particular database types, such as relational databases [25, 26] or NoSQL stores [30]. To the extent of our knowledge, Uniserve is the first system to distribute many diverse data types and query models, as shown in Section 4.

**Elasticity and Load Balancing.** 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 [47]. 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. Unlike these systems, Uniserve uses consistent hashing and replication-based shard prefetching to reduce the performance impact of cluster resizing. Using consistent hashing to scale stateful computation has a long history, going back to the peer-to-peer lookup service Chord [45] More recently, the elastic distributed data warehouse Snowflake [48] uses consistent hashing to assign shards to servers, although unlike Uniserve it supports only a fixed data type (tables) and query model and does not support replication or shard prefetching.

The Uniserve load balancer uses a greedy approach to move shards from overloaded servers to underloaded ones. Many transactional systems have used similar load balancing algorithms, such as E-Store [47]. However, unlike these systems, Uniserve automatically replicates hot shards to spread out read load, which would be unnecessary and counterproductive in write-heavy transactional workloads. Several systems have used less approximate methods for load balancing, such as a linear programming in Accordion [43] or a cubic solver in Getafix [31]. However, these do not scale past a few hundred servers or a thousand shards, while the Uniserve load balancer scales to the much larger sizes of modern deployments, which may host millions of shards on thousands of servers.

## 10 Conclusion

Query serving systems, such as OLAP systems, full-text search systems, and time series databases, are increasingly popular and important. Unfortunately, building a new query serving system is unnecessarily difficult because scaling it to modern data workloads requires writing a custom distribution layer comprising tens or hundreds of thousands of lines of code. In this paper, we describe Uniserve, a unifying distribution layer for query serving systems. The core of Uniserve is a small but expressive set of abstractions for data storage and queries. Uniserve can construct a distributed query serving system from single-node code that implements these abstractions, while providing user-requested features such as elasticity and load balancing that are often missing from existing systems. We evaluate Uniserve by porting existing systems to it and building new systems on it. Our implementations require <1K lines of code, but match or exceed the performance of natively distributed systems.

## 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] New Coordinator Segment Balancing/Loading Algorithm. https://github.com/apache/druid/issues/7458, 2019.

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

[17] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[18] Joe Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 6–1, 2007.

[19] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.

[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] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1917–1923, 2015.

[33] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[34] 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.

[35] Xavier Léauté. Benchmarking Druid. 2014.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] Mark Raasveldt. MonetDBLite: An Embedded Analytical Database. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1837–1838, 2018.

[43] 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.

[44] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.

[45] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

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

[47] 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.

[48] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*, pages 449–462, 2020.

[49] 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.

[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.