Queries on Abstract Shards:
A Programming Model for Distributing Low-Latency Data-Parallel Queries

Paper #114

Abstract

We present the *queries on abstract shards (QAS)* programming model for distributing low-latency data-parallel queries. Such queries are characteristic of the emerging class of *query serving systems*, including online analytical processing (OLAP) systems like Druid, full-text search engines like ElasticSearch, and time series databases like InfluxDB. These systems achieve low latency by storing data in diverse specialized representations with custom data layouts, compression schemes, and indexes. Existing programming models for distributed systems (like Spark) cannot manage these specialized representations, so building a query serving system entails writing a custom distribution layer comprising tens of thousands of lines of complex, error-prone code.

Unlike existing programming models, QAS can encapsulate the custom data representations needed for low-latency queries. QAS provides a query model where operators act not on individual data items but on *shards*, which store data in a developer-controlled custom representation. A QAS developer needs only provide a shard data structure and query plans fitting this model. Our implementation of QAS, Uniserve, automatically distributes data and queries and provides features missing from many existing systems, like strong consistency, load balancing, and elasticity. To evaluate QAS, we build new systems on it, like a simplified data warehouse based on a single-node column store, and use it to replace the distribution layers of existing systems like Druid and Solr, adding features like elasticity. Each implementation requires <1K lines of code, but all match or outperform comparable systems.

1 Introduction

Specialized systems that perform data-parallel, low-latency computations are ubiquitous. These *query serving systems* include full-text search engines like ElasticSearch [13], online analytics (OLAP) systems like Druid [54], timeseries databases like InfluxDB [14], and many others [10, 11, 19, 41]. These systems are often critical for users; for example, when they visit Facebook, TAO [30] looks up their friends' posts; when they use a Walmart gift card, ElasticSearch checks for fraud [6]; when they browse the H-E-B marketplace, ElasticSearch serves information on products [5]. They are also critical for developers, for example, many engineers diagnose system performance issues using InfluxDB or Druid to store and analyze logs [15]. Overall, query serving systems have millions of users [12] querying up to hundreds of terabytes of data [44] at millisecond latency. They achieve low latency by storing data in diverse specialized representations with custom data layouts, compression schemes, and indexes.

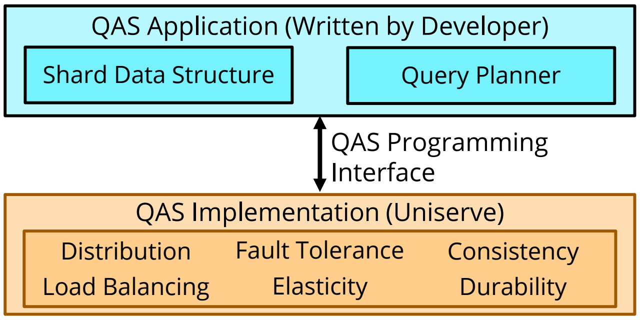

Figure 1: A QAS developer needs only provide code for data storage and query planning (blue) implementing the QAS programming interface. A QAS implementation like Uniserve (orange) distributes data and queries, providing distributed features.

Unfortunately, building query serving systems is difficult because their data-parallel low-latency computations are not supported by any high-level programming model for distributed computing, like Spark [56], Slicer [23], or Orleans [31]. Data-parallel abstractions like Spark cannot handle query serving workloads because they are optimized for large batch queries and their restricted in-memory data representations do not support the specialized data structures on which query serving systems depend. For example, naively searching a text file with Spark is orders of magnitude slower than using ElasticSearch because of ElasticSearch's inverted indexes. Other programming models are too low-level: Orleans offers only a messaging API while Slicer only manages data placement; neither provides a high-level query model.

Without a high-level programming model for distribution, all query serving systems must include a custom distribution layer that scales their computations and data across servers. These distribution layers comprise tens to hundreds of thousands of lines of complex code (for example the distribution layer of Druid is ~165K lines), written over many person-years. As a result, not only are new query serving systems difficult to build, but existing systems lack user-requested features that improve performance and reliability. For example Druid is fault-tolerant but not elastic and does not balance query load [7], while Solr recently added support for elasticity [4] but still lacks load balancing.

This paper proposes *queries on abstract shards (QAS)*, a novel programming model for distributing low-latency data-parallel queries. We sketch QAS in Figure 1. Unlike existing programming models, QAS can encapsulate the custom data representations needed for low-latency queries. QAS provides a query model where operators act not on individual data items, but on developer-defined data *shards*, which host data in a custom representation like an inverted index or Druid segment. Using this model, QAS implementations automatically manage the distribution of data and queries, including updates, consistency, fault tolerance, load balancing, and elasticity. Developers need only implement a shard data structure and query plans to automatically receive these features.

The principal challenge in designing QAS is that it must both enable distributed features like strong consistency, load balancing, and elasticity and be general enough to support the variety of query serving workloads and data representations, ranging from decision support on indexed tables to text search on inverted indexes to time series analysis on Druid segments. To address this challenge, QAS provides a minimal shard interface: shards need only expose serialize and deserialize methods and accept developer-defined query and update functions. This interface does not constrain the data representations shards can support; for example, it can be implemented by relational tables, inverted indexes, and Druid segments. However, it enables key functionality; for example, because shards are serializable, they can be backed up for durability and transferred for load balancing and elasticity.

To enable distributed data-parallel querying of data in shards without sacrificing performance, QAS provides a query model based on map and reduce operators. In conventional MapReduce implementations like Spark, map and reduce typically act on individual data items and the system uses its knowledge of data layout to shuffle data between stages. QAS instead defines operators on entire shards and gives developers control over how data is shuffled. QAS map acts on a shard, then partitions its output into chunks, one for each reducer, stored in a developer-controlled serialized format. QAS reduce takes in its chunks, one from each shard, and executes on them, returning a result. This lets developers implement high-performance query, partitioning, and serialization optimizations while abstracting away distribution concerns.

We implement QAS in a system called *Uniserve*, which distributes data and queries, guaranteeing durable, highly-available shard storage and scalable, low-latency query execution. Uniserve also provides critical features like fault tolerance, strong consistency, load balancing, and elasticity, which users demand but many existing systems lack. Because different workloads require different implementations of these features, Uniserve allows developers to control their ideal consistency guarantees and load balancing and auto-scaling behavior without modifying their core application code.

To evaluate QAS and Uniserve, we use them to distribute five systems. First, we use them to build two new systems: a social graph store similar to Tao [30] and a simplified data warehouse based on the single-node column store MonetDB [40]. Then, we use them to replace the distribution layers of three popular existing systems, Druid, MongoDB, and Solr, adding missing user-requested features like elasticity. Each system requires <1K lines of code to implement over QAS, though comparable custom distribution layers require tens or hundreds of thousands. However, all five QAS-distributed systems at least match the performance of similar natively-distributed systems. We additionally show how QAS can support many other popular systems, including Clickhouse [11], ElasticSearch [13], InfluxDB [14], and Pinot [41].

In summary, our contributions are:

- We identify *query serving systems* as an emerging class of distributed applications defined by low-latency data-parallel queries. We show that for query performance, these systems rely on custom data representations which existing programming models for distributed computation cannot manage.
- We propose and implement *queries on abstract shards (QAS)*, a novel programming model for low-latency data-parallel queries. A developer using QAS to build a query serving system is only responsible for query planning and physical data storage; QAS and its implementation Uniserve provide distribution, scalability, fault tolerance, load balancing, strong consistency, and elasticity.
- We demonstrate the power and practicality of QAS by building with it a new simplified data warehouse and graph store and porting to it the popular systems Solr, Druid, and MongoDB. Our implementations require <1K lines of code (replacing hundreds of thousands) but match or outperform natively distributed systems while providing new features like load balancing and elasticity.

2 Background and Motivation

In this section, we give three examples of widely used query serving systems, then make the case for QAS.

2.1 Case Studies

Apache Solr. Apache Solr [9] is a distributed full-text search system. It provides a rich query language for searching text documents and is optimized to serve thousands of queries per second at millisecond latencies. Solr stores documents in inverted indexes based on Apache Lucene [29].

Apache Druid. Apache Druid [54] is a high-performance analytics system. It provides fast ingestion and real-time search and aggregation of time-ordered tabular data, such as machine logs. Druid achieves its high performance through specialized *segment* data structures that store data in a tabular format, but use summarization, compression, and custom indexes to achieve query performance orders of magnitude better than conventional databases.

	Fault Tolerance	Load Balancing	Elasticity	Strongest Consistency
Solr	✓	X	✓	Eventual
Druid	✓	X	X	Eventual
MongoDB	✓	X	X	ACID
Uniserve	✓	✓	✓	ACID

Table 1: Distributed features of query serving systems.

MongoDB. MongoDB [16] is a NoSQL database. Unlike Solr and Druid, it is not primarily an analytics system, but is often used for analytics [17]. 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.

2.2 Discussion

Query serving systems such as Solr, Druid, and MongoDB are increasingly popular [12], with millions of users querying up to hundreds of terabytes of data [44]. Moreover, new query serving systems are constantly being developed. This is because "one size does not fit all" for data systems: innovations in physical data storage or query execution can often improve the performance of certain queries by orders of magnitude, inspiring a new system specializing in those queries [52].

Interestingly, while the physical data storage formats and query execution strategies of query serving systems are diverse, their distributed features are not. All must distribute custom data representations and low-latency data-parallel queries at scale while handling failure. Ideally, a unifying abstraction would provide these features for all systems. Unfortunately, query serving systems poorly fit existing programming models for distributed computing. Data-parallel frameworks like Spark [56] are only responsible for executing large batch computations on data represented in a restricted in-memory format. By contrast, query serving systems not only execute low-latency queries but also *manage* data stored in custom representations, updating it and ensuring its consistency and durability. Actor models like Orleans [31] provide a low-level messaging API between stateful actors, but query serving workloads require a high-level query model.

Because query serving system developers cannot easily use any high-level programming model, they must build custom distribution layers that comprise tens or hundreds of thousands of lines of code. These custom distribution layers all implement similar functionality, but are complex and difficult to build. As a result, query serving systems are frequently missing key user-requested features such as load balancing, elasticity, and strong consistency, summarized in Table 1. Therefore, users must over-provision clusters [38], go through the difficult and error-prone [1, 3] process of manually integrating external auto-scalers, and reason through unintuitive consistency models that can lose data [2, 8].

Shard Interface	
create() → Shard	Create a new (empty) shard.
destroy()	Destroy a shard.
serialize() →File	Serialize a shard to files on disk.
deserialize(File) →Shard	Deserialize a shard on disk into memory.
Row Interface	
getPartitionKey() →Int	Get a row's partition key.
Update Function Interface	
getUpdatedTable() →String	Name of the table to be updated.
prepare(Shard, List[Row]) → Bool	Implement changes, do not make them visible.
commit(Shard)	Atomically make prepared changes visible.
abort(Shard)	Roll back prepared changes.
Query Plan Metadata	
keysForQuery() →Map[String,List[Int]]	List queried tables and partition keys.
getAnchorTable() →Optional[String]	If there is an anchor table (§3.2), which is it?
outputTableName() → Optional[String]	If the query creates a table, what is its name?
getSubQueries() → List[QueryPlan]	List subqueries, if any exist.
getQueryCost(Shard) → Int	Cost of finished query operations on shard.
Query Plan Operators	
mapShard(Shard, Int) → Map[Int, C]	Map a shard, then repartition it into chunks serialized in a developer-controlled format.
reduce(List[C], Shard) → Shard or V	Process chunks and (optionally) an anchor table shard. Can return a shard or a value.
*combine(*List[V] → V')	If the query returns a value, combine the values produced by the reducers.

Figure 2: The QAS interfaces.

As a programming model for distributing low-latency data-parallel queries, QAS *separates responsibilities* in building a query serving system. A developer using QAS is only responsible for implementing the core, unique functionality of their system: query planning and physical data storage. Our implementation of QAS, Uniserve, is responsible for distribution and scalability, replacing systems' distribution layers. It executes each stage of each query plan on the appropriate data; guarantees the consistency, atomicity, and durability of updates; provides fault tolerance and failure recovery; manages data placement and load balancing; and provides elasticity. In existing systems, this functionality requires tens or hundreds of thousands of lines of code (~165K in Druid, for example), but implementations still lack key features such as those in Table 1. QAS can provide it for the cost of a small (<1K lines of code) shim layer, matching or exceeding native system performance while providing missing features.

3 QAS Overview and Interface

QAS is a programming model for distributing low-latency data-parallel queries. To use QAS, developers must provide single-node code for data storage and query plans implementing the QAS interfaces, outlined in Figure 2. Using these, an implementation of QAS, like Uniserve, can provide durable, high-availability data storage and scalable, fault-tolerant query execution. QAS is general and encapsulates the storage formats and query languages of many popular systems such as Druid, Pinot, MongoDB, ElasticSearch, and Solr.

3.1 Shards and Updates

In QAS, data is physically stored in *shards*, which host custom data structures. Data is added to shards in discrete units called *rows*. Shards are logically grouped into *tables*; a table is a set of shards comprising an entire data set. Each shard is a horizontal partition of its table: it contains some fraction of the table's rows, and each row is assigned to one shard.

One challenge in this data model is determining how to partition data into shards. Different query serving systems use different logical partitioning schemes; for example timeseries databases partition data by time range for temporal locality, while data warehouses often partition tables on particular columns to optimize queries. To give developers flexibility to control data partitioning, we require each row expose a partition key, not necessarily unique. These keys determine row-to-shard assignment; all rows with the same key are placed in the same shard. Therefore, a time series database can use time ranges as keys (a unique key per day, for example) while a data warehouse can use partition column values as keys.

We sketch the row and shard interfaces in Figure 2. Rows need only expose their partition key; shards need only implement create, destroy, serialize, and deserialize functions. These interfaces enable data management; for example serialization can be used to transfer shards for load balancing or auto-scaling. However, because both interfaces are minimal, they are general and can encapsulate many specialized data representations. For example, in Solr, we implement QAS rows as Solr documents and shards as Solr collections (inverted indexes) and hash-partition rows. In Druid, we implement QAS rows as Druid rows and shards as Druid segments and partition rows on timestamp.

To add data items to shards (or modify existing items), developers must supply *update functions*. We show their interface in Figure 2. Update functions implement the participant protocol of two-phase commit. They consist of prepare, commit, and abort stages. The prepare stage adds a set of rows to a shard (or modifies existing rows), but does not make changes visible to queries. The commit stage atomically makes prepared changes visible. The abort stage rolls back prepared but uncommitted changes. Our implementation of QAS, Uniserve, guarantees the serializablility, linearizability, and durability of updates. We discuss updates in detail in Section 4.

3.2 Queries

The QAS query model enables low-latency scalable query execution. Developers using QAS must provide query plans fitting this model. Because query serving workloads are data-parallel, the QAS query model is inspired by MapReduce. However, unlike conventional MapReduce formulations, it is defined over shards instead of over individual data items. This requires generalizing the model to capture the diversity of shard implementations, but enables low-latency query execution over custom data structures, something not possible in existing MapReduce implementations like Spark. We show the query plan interface in Figure 2 and diagram it in Figure 3.

A QAS query executes on data stored in shards in one or more tables. It either creates a new table (which can then itself be queried) or returns a result to the user. The domain of a query is defined by its *keysForQuery* function, which lists the tables to be queried and the relevant partition keys in each table. QAS executes the query on the shards corresponding to these keys. For example, in a Druid query on a particular time range, these keys would define that range.

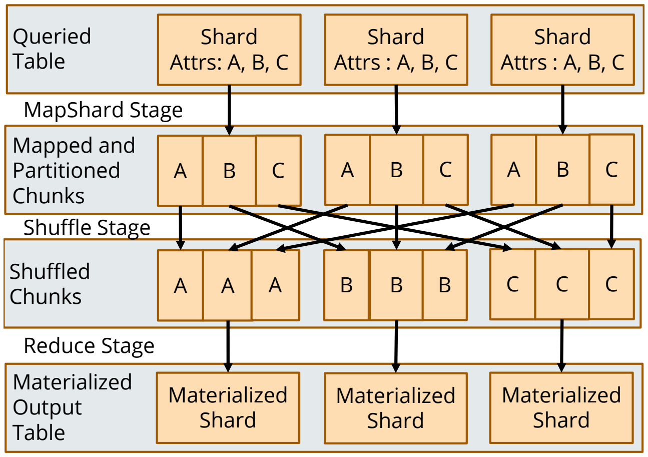

Figure 3: In QAS queries, MapShard executes a map operation on data in shards, then partitions its output by target attribute into chunks which are shuffled and reduced. Reduce output can either be materialized into a new table, as shown, or aggregated into a result.

QAS allows nested queries. Each query plan can have any number of sub-queries, which can themselves have sub-queries, forming a directed acyclic graph. Sub-queries are complete query plans that must execute before their parents. Typically, sub-queries materialize tables that their parents access. Nested queries must be executed in topological order, but logically independent subqueries can run in parallel.

Queries execute in two stages: map and reduce. Like conventional MapReduce, QAS shuffles (repartitions) data between these stages. The critical difference between QAS map (*MapShard*) and conventional MapReduce map is that in QAS, map is defined over a whole shard instead of a discrete data item. [1] This enables querying the specialized data structures encapsulated in shards. However, because QAS has no visibility into how a shard stores data, MapShard must explicitly partition data for shuffling. Each execution of MapShard operates on its shard, then partitions its output into several chunks, one for each reducer. Typically, this partitioning is done on some attribute. Each output chunk contains all the shard's data items whose values of the attribute are in a certain range. Output chunks are in a developer-defined binary format, giving the developer control over data serialization.

QAS reduce is similar to conventional reduce, but is defined over chunks of partitioned data instead of discrete data items. In between MapShard and reduce, QAS *shuffles* data so that each reducer receives a chunk of data from every mapper. These chunks correspond, so every data item with a particular value of the target attribute will be sent to the same reducer. Reduce processes its chunks, then returns an output. If the query is creating a table, this output is a shard; these shards together form the new table. If the query is returning a result, this output is a value; these are sent back to the client, com-

[1] Spark has mapPartitions, but partitions use a fixed representation as an iterable collection of records while shards are developer-defined.

bined there, and returned. After a query executes, it can report the cost of its operations on different shards; these are aggregated for use in load balancing (discussed in Section 4.4).

As an example, say we have $R(A,B)$ and $S(A,C)$ and wish to compute `SELECT A, SUM(B*C) as M FROM R, S GROUP BY A ORDER BY M LIMIT 10`, where A has high cardinality. Each MapShard partitions its shard of R or S into chunks by value of A. Each reduce receives all rows from all shards in both tables for a particular range of values of A. From these rows, it can create local, ephemeral tables R' and S', then run the original query on those tables, computing its results for a particular range of values of A. The client collects these results from all reducers and returns the top ten overall.

In some queries, one table does not require repartitioning. This may be because the table is already partitioned on the appropriate attribute or because the query does not depend on partitioning (e.g., a simple aggregation 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 its partitioning. Each reducer operates on one shard of the anchor table and corresponding chunks of data from other tables. Any logical map operation done on the anchor table must be pipelined into the reducer. For example, in a simple search query that counts how often a value appears in a table, there is no need for repartitioning, so the queried table would be an anchor. MapShards never executes and each reducer counts how often the value appears in its shard. The client returns the sum of these counts.

3.3 Case Study: Solr

We now describe how to create an QAS port of the distributed full-text search system Solr [9]. Natively, Solr stores data by sharding text documents across Lucene inverted indexes [29] on several machines. These indexes dramatically improve text search performance by (among other things) storing a precomputed mapping from search terms to relevant documents. 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 QAS, we implement shards as single-node Solr collections and rows as Solr documents. Just like Solr, our port hashes documents to obtain a partition key and assigns them to the corresponding shard. We implement shard and update functions using native equivalents; for example, shard creation using `createCollection`.

All Solr queries are searches: they take in a criterion, such as a query string, and return a list of 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 [20]. To port Solr's distributed query capabilities to QAS, we must translate Solr queries to QAS query plans. This is easy because Solr queries only access a single table and do not depend on its partitioning. Therefore, query plans are *anchored*: they need not perform a shuffle, but instead do all work in reducers which pipeline search and aggregation operations. Each reducer searches its target shard, then the client combines their results. 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 QAS port of Solr is implemented in ~500 lines of code (replacing ~300K lines of native Solr code), and can execute any query recognized by the standard Solr parser. As we show in Section 7, when distributed with Uniserve our port matches native Solr performance while providing features lacking in native Solr, including strong consistency and load balancing.

4 Implementing QAS in Uniserve

To evaluate the use of QAS, we implement it in a system called Uniserve. Uniserve can distribute and scale a query serving system built with QAS, handling concerns such as query routing, update consistency, fault tolerance, load balancing, and elasticity. Uniserve sits on top of developer-provided code responsible for query planning and physical data storage, managing it via the QAS interface.

Our design of Uniserve is based on two principles. First, it must minimize query execution overhead. Queries are high-volume, low-latency and do work in opaque QAS functions. Uniserve can thus do little to improve query performance, but must not add overhead. We therefore design the query execution critical path to be as lightweight as possible. For example, instead of doing fine-grained scheduling of individual queries, we do coarse-grained load balancing of whole shards.

Second, Uniserve assumes most updates will occur in large batches. This is typical of most of its target systems, which mostly ingest log data that is loaded periodically [41, 54]. Uniserve therefore prioritizes maximizing write throughput and providing strong, easy-to-understand update consistency and durability guarantees over minimizing write latency.

4.1 Architecture

A Uniserve cluster consists of many data servers. Each runs a thin Uniserve layer over developer-provided single-node code responsible for physical data storage. Clients send queries and updates to servers. Each client runs a thin Uniserve layer above a developer-provided query planner. A central coordinator manages cluster state with the help of ZooKeeper. We diagram the Uniserve cluster architecture in Figure 4.

Servers store data and execute queries. In each server, a thin Uniserve layer runs above developer-provided single-node code responsible for physical data storage and query execution. For example, if we were to distribute Solr using QAS and Uniserve, each server would run a Uniserve layer above a single-node Solr instance. The Uniserve layer facilitates query execution. It receives query plans and updates from clients and executes them in the underlying system using the QAS interface (sometimes in coordination with other servers, such as for shuffles or update replication). Additionally, it adds or

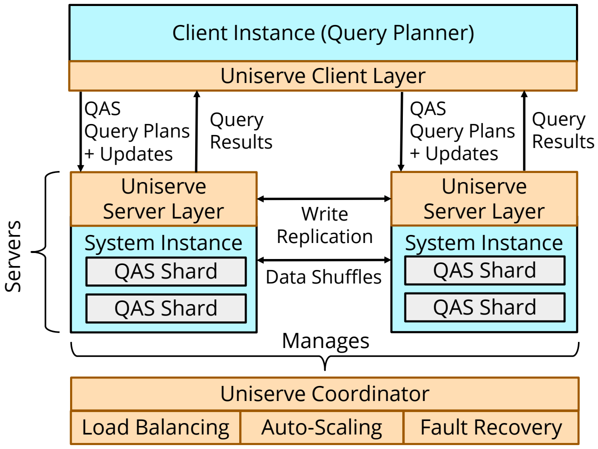

Figure 4: The Uniserve architecture. Clients and servers run a thin Uniserve layer (in orange) above instances of the underlying system (in blue). Data is partitioned across shards (in gray) logically managed by Uniserve but physically stored in the underlying system. A coordinator manages cluster state and provides distributed features.

removes shards in response to coordinator commands.

Clients plan queries and submit them to servers. In each client, a thin Uniserve layer runs alongside a developer-provided query planner. The query planner receives user queries (or update requests) in some query language and translates them to QAS query plans (or update functions). The client then submits these to the appropriate servers, eventually receiving and returning a result. Clients learn shard locations from the coordinator and ZooKeeper so they know to which servers to send queries or updates.

A Uniserve cluster contains a single coordinator that manages cluster state. It is responsible for many distributed capabilities including load balancing, failure recovery, and elasticity, which we discuss in more detail later. It backs up cluster state to ZooKeeper. To minimize query latency at scale, the coordinator is entirely off the query critical path.

4.2 Shard Updates

Query serving systems often provide weak consistency guarantees; for example replicas are typically eventually consistent and may not reflect completed updates. This frequently leads to confusion or even data loss [2, 8]. To mitigate this issue, Uniserve provides a set of strong and clear consistency guarantees: all updates to shards are linearizable, serializable, and durable. Queries made after an update completes always reflect the update, updates always execute in serial order, and updates once complete are never lost. To provide these guarantees, we must sacrifice some update latency, but as discussed above, this is acceptable for most target workloads.

To guarantee linearizability, Uniserve follows a two-phase commit protocol for all updates. Update functions must implement separate prepare, commit, and abort stages. Uniserve only commits an update if its has successfully prepared on all shards and their replicas, aborting if any failures occur. To ensure the cluster remains in a consistent state in case of a client crash, the client writes ahead any commit or abort decision to ZooKeeper; servers can reference this if the client fails (or abort if the client fails before making a decision).

To guarantee serializability, Uniserve does not support concurrent writes to a table. Only a single update per table may be active at a time, though it may concurrently write any number of rows to any number of shards in the table. This is enforced through distributed locks in ZooKeeper. It follows from the above model of large batched writes, trading off latency for stronger consistency without sacrificing throughput.

To guarantee durability, Uniserve synchronously uploads updated shards to durable remote storage such as S3. This upload can optionally be made asynchronous, relaxing the guarantee in exchange for better write performance.

Uniserve can optionally provide snapshot isolation for reads (and read-write atomicity) through multi-version concurrency control. If this is enabled, Uniserve creates a versioned copy of each shard upon write and ensures that read queries see consistent shard versions, guaranteeing snapshot isolation at a cost to write performance.

On modern file systems with support for sharing data blocks between files (for example, reflink in XFS [42]), Uniserve can use shadow paging to optimize its snapshot isolation guarantee and almost entirely eliminate its overhead. Instead of creating a physical copy of a shard, Uniserve cheaply creates a reference copy that shares the original shard's disk blocks. It then applies the update using copy-on-write, only copying and modifying disk blocks that are affected by the write (typically, few of them). This works best if each shard stores its data on disk in a single directory, otherwise serialization or marshaling is required.

4.3 Fault Tolerance

Uniserve mitigates server failures, even mid-query, using common fault tolerance techniques. Like many distributed systems [48], 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 simplify failure recovery.

Uniserve servers ping each other in a decentralized heartbeat system inspired by RAMCloud [48]. 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.

Uniserve guarantees query fault tolerance using a retry protocol. If a read to a shard fails, the client reloads its replica list from ZooKeeper and tries again with a different 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 must be restored from durable storage.

4.4 Load Balancing and Data Placement

Query serving systems often have unpredictable workloads skewed towards a small number of data items or partitions, so load balancing is necessary for consistent performance [38]. Conventional MapReduce systems balance load through fine-grained query scheduling, but this is impractical for Uniserve because of its stricter latency requirements and because all queries must run on specific shards. Instead, Uniserve balances load through data placement, managing the shard-to-server assignment to ensure no server is overloaded.

By default, Uniserve provides a greedy load balancing algorithm, similar to that of E-Store [53], which repeatedly moves the most-loaded shards from the most-loaded servers to the least-loaded servers while also replicating shards whose load exceeds average server load. However, some applications may want to instead use a custom algorithm. Therefore, we allow developers to define a *data placement policy*, which uses information on cluster utilization to compute an assignment of shards to servers. If a policy is provided, Uniserve takes responsibility for collecting the policy's input data and physically implementing its output assignment, moving shards to their new locations.

A data placement policy must be expressed as a function which takes in the total query load (as defined in the QAS query interface) and memory and disk usage of each shard, as well as the current assignment of shards to servers. It returns an updated assignment of shards to servers, expressed as a map from shard number to a list of server IDs. Assignments may replicate shards across multiple servers, either for redundancy or to spread out their load.

Uniserve periodically (using a configurable interval) executes the data placement policy and implements its assignment. To guarantee high availability when load balancing, Uniserve prefetches replicas of reassigned shards on their target servers. Only after replicas are ready does Uniserve notify clients of the shard movement. Then, after notifying clients, it deletes the original copies of the shards if necessary.

4.5 Elasticity and Auto-Scaling

Query serving system load often varies over time, so they benefit from *elasticity*, the ability to dynamically adjust cluster size. As a result, when deployed in an elastic cloud environment such as EC2, Uniserve automatically scales cluster size in response to load changes.

By default, Uniserve provides a utilization-based auto-scaling algorithm similar to the algorithms used in cloud auto-scalers [28]. It adds servers if CPU utilization exceeds an upper threshold and removes them if it is below a lower threshold. However, like in load balancing, Uniserve also gives developers the option of defining their own *auto-scaling policy*, which uses information on cluster utilization to decide whether to add or remove nodes. Uniserve provides the policy with its input and physically executes its commands, adding or removing nodes and transferring shards as necessary.

	System Type	Data Type	Query Operations
Druid [54]	OLAP	Indexed Tables	Aggregate, group, filter, broadcast join
Pinot [41]	OLAP	Indexed Tables	Aggregate, group, filter
ClickHouse [11]	OLAP	Indexed Tables	Aggregate, group, filter, join
OpenTSDB [19]	Timeseries DB	Time series	Aggregate, group, filter
Atlas [10]	Timeseries DB	Time series	Aggregate, group, filter
InfluxDB [14]	Timeseries DB	Time series	Aggregate, group, filter
MonetDB [40]	Data Warehouse	Relational Tables	SQL
Solr [9]	Full-Text Search	Indexed text	Text search
ElasticSearch [13]	Full-Text Search	Indexed text	Text search
TAO [30]	Graph Database	Graphs	Search graph relationships
MongoDB [16]	NoSQL	Documents	Aggregate, group, filter, map, broadcast join

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

An auto-scaling policy must be defined as a function which takes in the CPU utilization, memory and disk usage, and total query load (as defined in the QAS query interface) of each server. It returns the number of servers to be added or removed, as well as the IDs of the servers to be removed, if any (which can be chosen randomly if there is no preference).

Uniserve periodically executes the policy (using a configurable interval) and adjusts cluster size. After adding or removing a server, Uniserve uses the load balancer to reassign shards; if servers are removed this reassignment is done preemptively so availability is not affected.

5 Generality of QAS

In this section, we demonstrate the generality of QAS by describing some of the diverse systems it can distribute, summarized in Table 2. We also discuss its limitations.

OLAP Systems and Time Series Databases. OLAP systems are designed to rapidly answer multidimensional analytics queries. They are closely related to time series databases, which ingest and query time-ordered data. Both typically store data in a columnar tabular format optimized through encoding, bit packing, and pre-aggregation (rollup) and augmented by indexes, especially bitmap indexes. Their workloads usually filter, group, and aggregate this data. These data representations and query models map naturally onto QAS: we can partition data by time range into shards to maintain temporal locality and implement most aggregations in the reduce stage of an QAS query plan (with a shuffle beforehand, if needed). However, our implementation of QAS is optimized for read-mostly workloads while time series workloads sometimes include many small writes; these must be batched into a few coarse-grained writes for QAS to work effectively. We implement a port of Druid [54], which is both an OLAP system and timeseries database; its design patterns generalize to others from both categories such as Pinot [41], Clickhouse [11], OpenTSDB [19], Atlas [10] and InfluxDB [14].

Full-Text Search. Full-text search systems execute search queries over text data stored in specialized data structures such as inverted indexes [29]. Because all their queries are searches, they are easy to fit to QAS, as we showed in Section 3.3. We implement a port of one full-text search system, Solr [9], and can generalize to others like ElasticSearch [13].

Graph Databases. Graph databases represent data using a graph data model. Their implementations and queries vary widely. Some graph database queries are data-parallel, including whole-graph algorithms like PageRank and queries on graph contents like finding all checkins at a certain location in a social network graph. Others are not; for example, a graph traversal query, like finding all nodes within *N* hops of a target, is most efficiently implemented using breadth-first search, not a data-parallel technique such as iterative self-joins. QAS can distribute graph databases whose queries are data-parallel, such as TAO [30], but not others, such as Neo4j [18].

Other Systems. QAS can distribute other systems with data-parallel read-mostly queries. For example, we implement a QAS port of the NoSQL store MongoDB. We also implement a simplified data warehouse based on the single-node column store MonetDB and show its performance is competitive with popular data warehouses Spark-SQL [24] and Redshift [39].

Limitations. QAS has two major limitations. First, its query model works best for data-parallel queries. As we have shown, this encompasses the query languages of many popular query serving systems, but not some specialized query types such as graph traversal queries.

Second, our implementation, Uniserve, is optimized for read-mostly workloads and not workloads with many small point updates. Small updates are rare in analytics systems, but common in other contexts such as online transactional processing (OLTP) workloads which QAS cannot efficiently distribute.

6 Distributing Systems with QAS

To demonstrate the practicality of QAS, we use it to distribute five systems. First, we port to QAS popular query serving systems Druid, Solr, and MongoDB, replacing their native distribution layers. Then, we build with QAS two new systems: a social graph store similar to TAO and a simplified data warehouse based on the single-node column store MonetDB.

We implement each of our five systems in <1K lines of code (LoC). This number includes all code needed to implement the QAS interfaces with each system's already-existing single-node implementation, but not any Uniserve code or native code. This demonstrates that QAS simplifies building distributed query serving systems, as it replaces custom distribution layers totaling ~300K LoC in Solr, ~98K LoC in MongoDB, and ~165K LoC in Druid.

Solr. We described the port of Solr in Section 3.3.

Druid. In our port of Druid [54], we implement shards as Druid datasources. These are analogous to database tables and are backed by Druid segments, which are optimized tabular stores for timeseries data. We implement most shard manipulation and update functionality using the Druid datasource API. To serialize shards, we copy the on-disk segments to a target directory, then export Druid's segment metadata table. To deserialize shards, we copy the segments into the Druid data directory, then load the exported segment metadata.

All Druid queries aggregate filtered and grouped data from datasources. Our port supports most common Druid queries: simple aggregations (sums, counts, or averages) of filtered and grouped data. It could easily be extended to support any other query by adding support for more aggregation operators. Our Druid query plans separately query all shards then aggregate the results. Druid uses a similar model natively.

MongoDB. In our port of MongoDB [16], we implement shards as MongoDB collections, analogous to database tables. We implement most shard manipulation and update functionality using the MongoDB API for manipulating collections. We implement shard serialization and deserialization using the `mongodump` and `mongorestore` tools.

MongoDB queries apply an "aggregation pipeline" of operators to a collection. These operators perform tasks such as filtering, grouping, and accumulating documents. We can support any MongoDB operator, but so far have only implemented operations for filtering, projecting, summing, counting, and grouping data. Our query plan implementations are similar to those in our Druid port and those in native MongoDB: querying shards separately, then combining the results.

TAO. We have built using QAS a social graph store that implements the interface of Facebook TAO [30]. 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 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 query plans for all four of these query types using SQLite.

MonetDB. We have built using QAS a simplified data warehouse based on the single-node column store MonetDB [40]. It stores data in MonetDBLite [50], the embedded implementation of MonetDB. Each server runs MonetDBLite embedded in the same JVM as the Uniserve layer. We implement shards as MonetDB tables and implement shard and update functions using equivalents in the MonetDBLite API.

Our simplified data warehouse supports a large subset of SQL, including selection, projection, equijoins, grouping, and aggregation. It is not meant to be a full-fledged data warehouse, but rather a demonstration of Uniserve's ability to efficiently execute complex queries. We have not yet implemented a query planner, but instead plan queries manually. We implement simple aggregation queries by querying shards separately then aggregating, as in other systems. To execute queries that require repartitioning, such as shuffle joins, MapShard queries each shard, executes pushed-down predicates, retrieves data in binary format, and partitions it by reducer. Reduce inserts all data it receives into its local MonetDBLite instance and queries the resulting partition. The client then combines per-partition responses into a final result.

7 Experimental Evaluation

We evaluate QAS and Uniserve using the five systems discussed in Section 6. As we have shown, QAS makes distributing these systems simple; each requires <1K lines of code to distribute as compared to the hundreds of thousands of lines of code in custom distribution layers (~300K LoC in Solr, ~98K LoC in MongoDB, and ~165K LoC in Druid). In this section, we demonstrate that QAS-distributed systems are performant, specifically that:

1. Distributed systems built using QAS and a specialized single-node system, such as our MonetDB-based simplified data warehouse, can match or outperform popular distributed systems such as Spark-SQL and Redshift.
2. QAS ports of distributed systems match the performance of natively distributed systems under ideal conditions, such as static workloads without load skew.
3. QAS ports of distributed systems provide new features such as elasticity and load balancing and so outperform natively distributed systems under less ideal conditions – workloads that change, have load skew, or have failures.

7.1 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 8.6.1, 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 6. We place the Uniserve coordinator and a ZooKeeper server on a machine by themselves and data servers on the other nodes.

7.2 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 [47]. 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 representative modified TPC-H queries from the Druid paper [43, 54], 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 performs a group-and-aggregate.

We benchmark MongoDB using YCSB [34], 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 [21].

We benchmark our data warehouse using representative 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 [26], 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.

7.3 Benchmarks

Ideal Conditions. We first benchmark our Solr, Druid, MongoDB, and TAO ports under ideal conditions, distributing queries uniformly so each data item is equally likely to be queried. We run each benchmark with several client workers; each repeatedly makes the query and waits for it to complete, recording throughput and latency. We start with a single worker and add more until throughput no longer increases, showing results in Figure 5. We find that, unsurprisingly, our ports' performance is similar to native system performance on all benchmarks. The exception is the MongoDB benchmark where our port 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 [30].

We also evaluate the scalability of Uniserve, scaling the Solr benchmarks on 20M documents with one client worker 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.

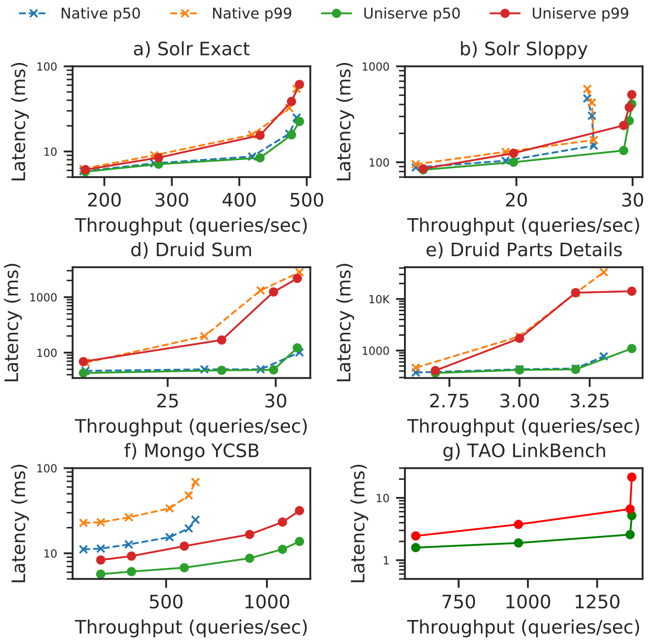

Figure 5: Throughput versus latency for native systems and QAS ports on uniform and static query workloads. QAS ports generally match native performance.

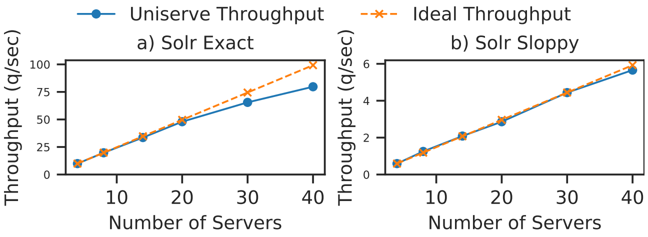

Figure 6: Uniserve scalability on the Solr benchmarks.

Data Warehouse Benchmarks. We next benchmark our simplified data warehouse based on MonetDB, comparing its performance with native single-node MonetDB, Spark-SQL [24], and Redshift [39]. We use three TPC-H queries: Q1, an aggregation query; Q3, a three-way join; and Q10, a four-way join. We implement Q10 as a two-stage query that repartitions data between joins. We show results in Figure 7. We run multiple trials of each benchmark, reporting the average of results after performance stabilizes. This ensures Spark-SQL and Redshift can cache data in memory.

We first investigate the overhead Uniserve adds to single-node MonetDB. On a single node, our data warehouse performs the same as native MonetDB on the aggregation query Q1 and significantly but not unreasonably worse on Q3 and Q10 due to the communication cost of shuffling. 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 our data warehouse outperforms Spark-SQL and matches Redshift. This shows that by distributing a single-node system like MonetDB, QAS can in <1K lines of code match or outperform popular distributed systems like Redshift and Spark-SQL on their core workloads.

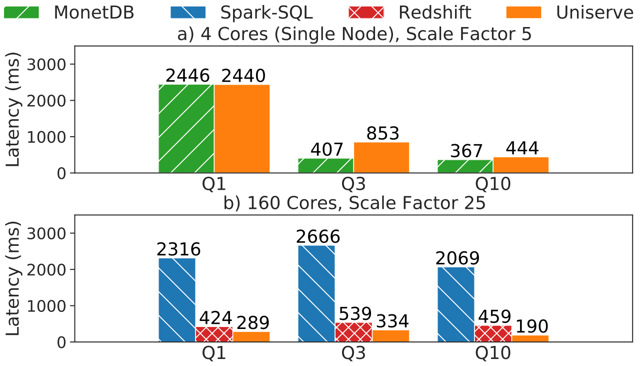

Figure 7: Comparison between our simplified data warehouse, single-node MonetDB, Spark-SQL, and Redshift on TPC-H queries Q1, Q3, and Q10 on 4 cores (single-node) and on 160 cores with TPC-H scale factors of 5 and 25. Uniserve is competitive on a single node and outperforms Spark-SQL and matches Redshift at scale.

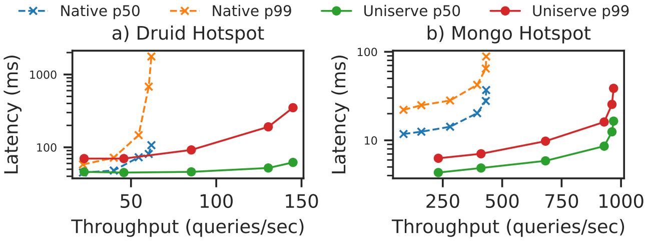

Figure 8: Throughput versus latency for natively-distributed and Uniserve-distributed queries where one slice of data receives 7/8 of queries. Uniserve balances load and so outperforms native systems.

Hotspots. To demonstrate the importance of load balancing to query serving systems, we next investigate the performance of the default Uniserve load balancer on benchmarks with 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 workers, showing results in Figure 8. Because Uniserve balances load and moves or replicates items in the hotspot but Druid and MongoDB do not, Uniserve dramatically outperforms both.

We next repeat the experiment, fixing the number of workers at twelve but varying the fraction of queries sent to the hotspot. We show results in Figure 9. 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. To demonstrate the importance of elasticity in query serving systems, we next investigate the performance of the default Uniserve auto-scaler on a dynamic work-

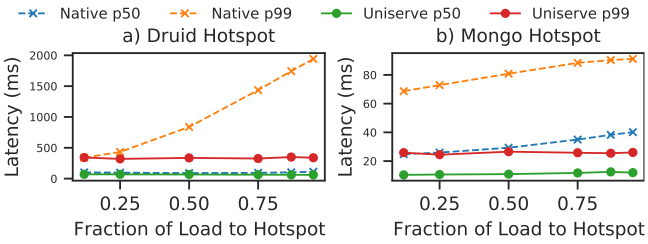

Figure 9: Latencies of natively-distributed and Uniserve-distributed queries on benchmarks with a hotspot, where one slice of data receives a varying fraction of all queries. Uniserve keeps performance constant as skew increases; native systems do not.

load. We run the Solr sloppy benchmark for six hours sending queries at a target throughput, which varies from 240 to 1300 uniformly distributed queries per minute. Uniserve starts with one server and adds or removes more as load changes. We show results in Figure 10. We see that Uniserve is always able to scale to meet the target throughput. As load increases, it adds 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 prefetching replicas of moved shards onto new servers 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 11.

When all servers have replicas of all shards (11b), 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 (11a), 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 [44]; Uniserve addresses its issues.

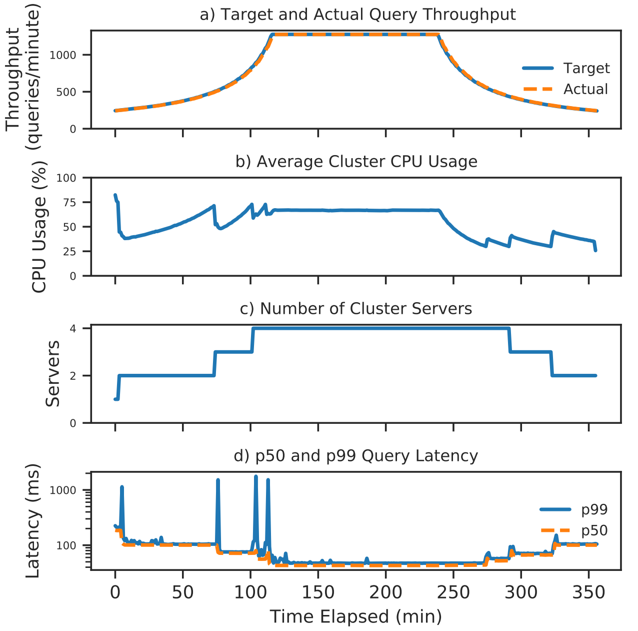

Figure 10: On the Solr sloppy benchmark with Uniserve autoscaling, varying target throughput and observing effects on actual throughput, average cluster CPU usage, the number of cluster servers, and query latencies. Uniserve scales the cluster so that actual throughput always matches target throughput; resizing causes only brief (<1 sec) spikes in query latency. Latency decreases as cluster size increases because all queries run on all data and their parallelism increases as the data is spread over more nodes.

Write Performance. We next investigate Uniserve write performance. We benchmark 1 MB, 10 MB, and 100 MB writes to a 100 MB shard on Solr, Druid, MongoDB, and Tao, using each system's benchmark dataset (Lucene nightly data for Solr, TPC-H data for Druid, YCSB data for MongoDB, Linkbench data for Tao). We compare system native performance (except for Tao, where the native system is proprietary) to Uniserve performance with and without Uniserve's optional snapshot isolation guarantee. We show results in Figure 12.

Uniserve adds some overhead to each write. This comes entirely from the synchronous upload to durable storage that Uniserve makes before completing a write; it is shown in yellow in Figure 12. This overhead depends on the size of the existing shard–not that of the write–so it is more significant for smaller writes. We note that for workloads that require smaller writes, this upload can optionally be made asynchronous, relaxing the durability guarantee in exchange for eliminating almost all Uniserve write overhead. The overhead is especially large for MongoDB because the `mongodump` utility that serializes MongoDB data to disk for upload is slow.

The optional Uniserve snapshot isolation guarantee adds further overhead when enabled, shown in purple in Figure 12, which comes from synchronously copying a shard before writing to it so that queries made while the write executes

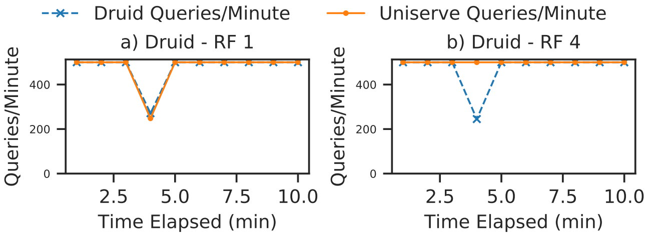

Figure 11: Query throughput (with a target of 500 queries/minute) of Druid- and Uniserve-distributed `sum_all` queries when one data server is killed after three minutes. The left graph shows performance starting with a single replica of each shard; the right graph with four.

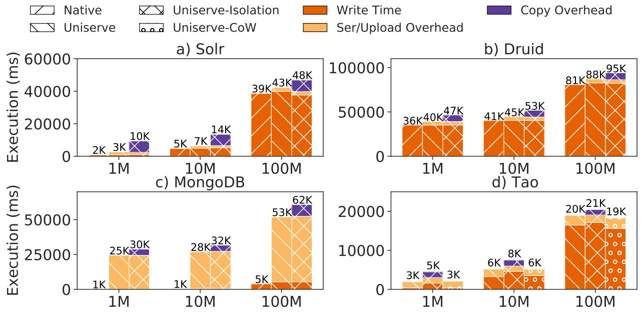

Figure 12: Execution time of 1 MB, 10 MB, and 100 MB writes to a 100 MB shard on native systems (except for Tao, where the native system is proprietary), with Uniserve, and with Uniserve snapshot isolation enabled. For Tao, we also examine the performance of our copy-on-write (CoW) optimization. Uniserve matches the write performance of native systems (orange), but adds overhead from the synchronous upload it makes for durability (yellow) and the synchronous copy it makes for isolation if enabled (purple).

can see the original, globally consistent version of the shard. However, this overhead can be almost entirely eliminated by our copy-on-write optimization, discussed in Section 4.2. We apply this optimization to our Tao implementation, which is a prime target because it stores each shard's data on disk in its own directory. We find that it eliminates the guarantee's overhead by only copying the modified parts of a shard.

8 Related Work

The difficulty of working in a distributed setting has encouraged development of many programming models for distributed computing. Perhaps the most successful have been cluster computing frameworks based on MapReduce [36], like Hadoop [51], Dryad [55], and Spark [56]. Unlike QAS, these systems only execute computations, while query serving systems also manage data and provide guarantees like consistency and durability. Moreover, they use fixed tabular in-memory data representations (while query serving systems depend on specialized data structures) and are optimized for workloads containing a few large batch queries (while query serving workloads contain high-volume low-latency queries). Researchers have attempted to build updatable data structures over Spark RDDs, such as PART [35], but their data models are greatly limited by the immutability of RDDs.

QAS shards are similar to persistent objects in Thor [46]; both systems protect objects from failures and provide a transactional interface to modify them. However, Thor, predating both MapReduce and the cloud, does not provide a distributed query interface, load balancing, or auto-scaling. Shards are also analogous to actors in systems such as Erlang [25] and Orleans [31]. However, actors are units of computation while shards are partitions of data, so actors offer only a low-level messaging API while QAS offers a high-level query model.

The auto-sharding systems Slicer [23] and Centrifuge [22] assign data and queries to shards based on partition keys, much like QAS. However, they *only* manage key affinity, telling applications what keys are assigned to what servers. QAS and Uniserve actually manage data and execute queries.

Middleware systems simplify building distributed databases by distributing data and queries across existing database installations; like QAS and Uniserve, they provide features like fault tolerance [45, 49] and load balancing [27]. However, middleware solutions are typically specialized to particular database types, like relational databases [32, 33] or NoSQL stores [37]. To the extent of our knowledge, QAS is the first programming model to distribute many diverse data representations and query models, as shown in Section 5.

9 Conclusion

This paper proposes *queries on abstract shards (QAS)*, a novel programming model for distributing low-latency data-parallel queries. Unlike existing programming models for distributed computing like Spark, QAS can encapsulate the specialized data representations (including custom data layouts, compression schemes, and indexes) that low-latency queries rely on. A developer using QAS needs only provide a data representation and query plans fitting the QAS query model; our implementation of QAS, Uniserve, distributes data and queries and provides features like strong consistency, load balancing, and elasticity. We evaluate QAS by building new systems with it and porting existing systems to it. Our implementations require <1K lines of code, replacing tens of thousands, but match or outperform natively distributed systems.

References

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

[2] Jepsen Analysis of Elasticsearch 1.5.0. https://aphyr.com/posts/323-call-me-maybe-elasticsearch-1-5-0, 2015.

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

[4] Major Changes in Solr 7. https://solr.apache.org/guide/7_0/major-changes-in-solr-7.html#autoscaling, 2017.

[5] Elastic @ H-E-B: Providing a better shopping experience at Central Market. https://www.elastic.co/elasticon/tour/2019/austin/, 2019.

[6] How Walmart is Combating Fraud and Saving Consumers Millions. https://www.elastic.co/elasticon/tour/2019/dallas/, 2019.

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

[8] Jepsen Analysis of MongoDB 4.2.6. http://jepsen.io/analyses/mongodb-4.2.6, 2020.

[9] Apache Solr. https://lucene.apache.org/solr/, 2021.

[10] Atlas. https://github.com/Netflix/atlas, 2021.

[11] ClickHouse. https://clickhouse.tech/, 2021.

[12] DB-Engines Ranking. https://db-engines.com/en/ranking, 2021.

[13] Elasticsearch. www.elastic.co, 2021.

[14] InfluxDB. https://www.influxdata.com/, 2021.

[15] InfluxDB Infrastructure and Application Monitoring. https://www.influxdata.com/customers/infrastructure-and-application-monitoring/, 2021.

[16] MongoDB. https://www.mongodb.com/, 2021.

[17] MongoDB for Analytics. https://www.mongodb.com/analytics, 2021.

[18] Neo4j. https://neo4j.com/, 2021.

[19] OpenTSDB. http://opentsdb.net/, 2021.

[20] Solr Distributed Requests. https://solr.apache.org/guide/8_8/distributed-requests.html, 2021.

[21] YCSB GitHub. https://github.com/brianfrankcooper/YCSB, 2021.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Persistent adaptive radix trees: Efficient fine-grained updates to immutable data.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Matt Keenan. XFS - Data Block Sharing (Reflink), Jan 2020.

[43] Xavier Léauté. Benchmarking Druid. 2014.

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

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

[51] 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.

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

[53] 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.

[54] 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.

[55] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed

Data-Parallel Computing Using a High-Level Language. *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2009.

[56] 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.