Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems

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

https://www.usenix.org/conference/nsdi22/presentation/kraft

This paper is included in the Proceedings of the
19th USENIX Symposium on Networked Systems
Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the
19th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Data-Parallel Actors:
A Programming Model for Scalable Query Serving Systems

Peter Kraft
Stanford University

Fiodar Kazhamiaka
Stanford University

Peter Bailis
Stanford University

Matei Zaharia
Stanford University

Abstract

We present *data-parallel actors (DPA)*, a programming model for building distributed query serving systems. Query serving systems are an important class of applications characterized by low-latency data-parallel queries and frequent bulk data updates; they include data analytics systems like Apache Druid, full-text search engines like ElasticSearch, and time series databases like InfluxDB. They are challenging to build because they run at scale and need complex distributed functionality like data replication, fault tolerance, and update consistency. DPA makes building these systems easier by allowing developers to construct them from purely single-node components while automatically providing these critical properties. In DPA, we view a query serving system as a collection of stateful actors, each encapsulating a partition of data. DPA provides parallel operators that enable consistent, atomic, and fault-tolerant parallel updates and queries over data stored in actors. We have used DPA to build a new query serving system, a simplified data warehouse based on the single-node database MonetDB, and enhance existing ones, such as Druid, Solr, and MongoDB, adding missing user-requested features such as load balancing and elasticity. We show that DPA can distribute a system in <1K lines of code (>10× less than typical implementations in current systems) while achieving state-of-the-art performance and adding rich functionality.

1 Introduction

Specialized systems that perform data-parallel, low-latency computations and frequent bulk data updates are becoming ubiquitous. These *query serving systems* include search engines like ElasticSearch and Solr [9, 13], online analytics (OLAP) systems like Druid and Clickhouse [11, 70], time-series databases like InfluxDB and OpenTSDB [14, 17], and many others [10, 15, 18, 21, 40, 51, 52]. These systems are critical to everyday applications: for example, Walmart uses ElasticSearch to check purchases for fraud in real time [6], Target and Capital One use Druid and InfluxDB for real-time monitoring in their production services [5, 7], and Facebook developed Unicorn [40] to provide graph-based search.

Developing query serving systems is challenging because their workloads typically run at large scale. Therefore, query serving system developers must implement complex distributed functionality, including data replication, update consistency, fault tolerance, and load balancing. These features vary little between query serving systems, but must be re-implemented in each of them, typically in custom distribution layers comprising tens of thousands of lines of complex code (e.g., ~70K lines in Druid) written over many person-years. As a result of this complexity, not only are new query serving systems hard to build, but existing ones are difficult to adapt to changing user demands. For example, most query serving systems were designed for fixed-size on-premise clusters, although users increasingly deploy them in the cloud. Therefore, they do not provide user-requested cloud features such as elastic cluster auto-scaling [3, 4]. Adding any new distributed feature to an existing, large codebase can take years, even when there is strong user demand [60, 74].

Ideally, developers would be able to write query serving systems using a high-level programming model that simplifies distributing their data and computations across a cluster. Unfortunately, current distributed programming models do not support the unique workloads of query serving systems, with their combination of data-parallel low-latency queries and frequent bulk data updates. Actor models like Erlang [29], Orleans [35] and Ray [61] can manage mutable state, but lack abstractions, such as consistency and atomicity, for data-parallel operations. Parallel processing frameworks like Spark [72] can execute data-parallel queries, but lack abstractions for managing data, assuming it to be immutable.

In this paper, we propose a new programming model called *data-parallel actors (DPA)* that extends the actor model to support the unique needs of query serving systems. DPA allows developers to construct a distributed query serving system from purely single-node components, as we show in Figure 1. The DPA runtime then automatically provides the system with complex distributed features such as fault tolerance, consistency, load balancing, and elasticity.

Designing a programming model for query serving systems is challenging because of their wildly different query and data models, from search engines to timeseries databases to document stores. DPA's insight is that the distributed functionality of a query serving system can be implemented largely independently of how the system stores and processes data on individual nodes. Therefore, DPA represents a query serving system as a collection of black-box data partitions, each encapsulated in a stateful actor. However, while conventional actor models focus on *concurrency*, where there are many actors but clients only communicate with one at a time, query serving systems also require *parallelism*: one operation can run over data in many actors, often with consistency and atomicity requirements. Thus, DPA provides parallel operators and updates over its stateful actors. Parallel operators let develop-

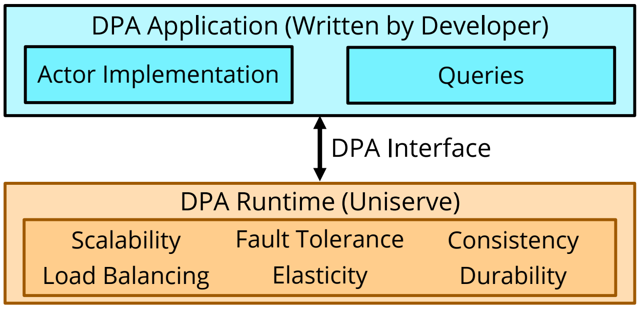


Figure 1: With DPA, a developer can construct a distributed query serving system from single-node components: code for actors and queries (blue) that implement per-node data structures and query processing logic. A DPA runtime like Uniserve (orange) manages actors and executes queries, automatically providing distributed features.

ers construct queries from generic operations such as map and broadcast, while parallel updates offer configurable consistency and atomicity guarantees. DPA defines these operations and enforces their guarantees, but is agnostic to how each node processes its part of the work. Thus, DPA *separates responsibilities* in building a query serving system, so that developers only implement single-node data structures and operations but receive a robust, performant distributed system.

We show that DPA can express the functionality of a wide range of current query serving systems, while adding powerful user-requested features that production systems lack. For example, we used DPA to wrap the existing single-node components in an OLAP system (Druid), search engine (Solr) and NoSQL database (MongoDB) into stateful actors in a few hundred lines of code. The DPA ports match the original systems on standard performance benchmarks, but also automatically receive user-demanded missing features like load balancing and elasticity, improving performance on skewed workloads by up to 3×. DPA's generality makes it a powerful abstraction for developing new query serving systems.

We implement the DPA programming model in a runtime called *Uniserve*. Uniserve manages stateful actors and executes queries, automatically providing distributed features like durability, fault tolerance, load balancing, and elasticity. Because workloads require different implementations of these features, Uniserve allows developers to configure systems' consistency guarantees and load balancing and auto-scaling behavior without modifying their core application code.

To evaluate DPA and Uniserve, we use them to distribute four systems: the three ports discussed above (Druid, MongoDB, and Solr) and a new simplified data warehouse we built based on the single-node database MonetDB [50]. We distribute each system with <1K lines of code. Nevertheless, on standard benchmarks, our ports match the originals' performance, while our data warehouse matches Amazon Redshift and outperforms Spark SQL. Each DPA-based system automatically receives powerful features, including fault tolerance, durability, consistency, load balancing, and elasticity. Some of these features, particularly load balancing and elasticity, are missing and frequently requested by users in Druid, MongoDB, and Solr. By adding these features, DPA improves these systems' performance by up to 3× on skewed workloads. In summary, our contributions are:

- We identify *query serving systems* as an important emerging class of distributed systems defined by low-latency data-parallel queries and frequent bulk updates. We show that their workloads are not supported by existing high-level distributed programming models.
- We propose *data-parallel actors (DPA)*, a novel programming model for building distributed query serving systems from purely single-node components. We build a DPA runtime, *Uniserve*, which automatically provides fault tolerance, consistency, durability, load balancing, and elasticity to query serving systems built with DPA.
- We demonstrate the power and practicality of DPA by using it to build a simplified data warehouse and porting the popular systems Solr, Druid, and MongoDB to it. Our implementations require <1K lines of code (replacing tens of thousands) but match or outperform current systems while providing rich missing functionality.

2 Background and Motivation

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

2.1 Case Studies

Apache Solr. 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 [34].

Apache Druid. Druid [70] 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 optimized with summarization, compression, and custom indexes.

MongoDB. MongoDB [15] is a NoSQL document database. Unlike Solr and Druid, it is not primarily an analytics system, but is often used for analytics [16]. 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 Motivating DPA

Solr, Druid, and MongoDB are popular [12] query serving systems that serve different workloads. However, while their physical data structures and query execution strategies are diverse, all use custom distribution layers to distribute data and

	Fault Tolerance	Load Balancing	Elasticity
Solr	✓	X	✓
Druid	✓	X	X
MongoDB	✓	X	X
Uniserve	✓	✓	✓

Table 1: Distributed features of query serving systems.

queries while handling failure and ensuring data consistency. These distribution layers are difficult to implement, requiring tens of thousands of lines of complex code (~90K LoC in Solr, ~70K LoC in Druid, and ~120K LoC in MongoDB).

The difficulty of distributing query serving systems complicates developing new systems, but also causes existing systems to lack user-demanded features. For example, Solr, Druid, and MongoDB struggle to provide load balancing and elasticity, as shown in Table 1. As a result, their users must over-provision clusters [45], go through the difficult and error-prone [3, 4] process of manually integrating external auto-scalers, or risk poor performance when load skews or spikes.

One reason popular systems are missing important features is that the requirements for distributed systems change over time. For example, elasticity is considered important today because most query serving systems run in the cloud, where scaling the size of a cluster is easy. However, many existing systems (including Druid, Solr, and MongoDB) were built when the cloud was less popular, so support for auto-scaling was less important and was not included. Unfortunately, the complexity of query serving systems' distribution layers makes it difficult to add new features when users demand them. For example, adding strongly consistent replication to MongoDB required designing a novel consensus protocol because design choices made early in MongoDB's lifetime precluded using any existing protocol [74]. Similarly, adding support for joins to Druid has been a slow, multi-year process because the system was originally built assuming queries would not require communication between data sources [60].

DPA helps solve these problems by *separating responsibilities* in a query serving system. A developer using DPA is only responsible for the core, unique functionality of their system: storing and querying data. DPA and its runtime Uniserve take responsibility for distribution and scalability, automatically providing distributed features like fault tolerance, consistency, load balancing, and elasticity. As user demands change, new features can be added to Uniserve with minimal modifications to underlying systems. This makes it easier to build new query serving systems and maintain existing ones, as they can obtain state-of-the-art distributed functionality by simply implementing the DPA interface in a ~1K LoC shim layer.

3 DPA Overview and Interface

DPA lets developers construct a distributed query serving system from purely single-node components. To use DPA, a developer must first implement an actor object that encapsulates a data partition like a Solr index or Druid segment. They must then implement a query planner that translates incoming user queries to the DPA parallel operators. We show the interface for actors and operators in Figure 2.

3.1 Actors and Data

In DPA, developers express a query serving system as a collection of stateful single-node actors, each encapsulating a partition of data and exposing methods for manipulating and querying it. Query serving systems use a wide variety of data representations, from Solr inverted indexes to Druid table segments, so DPA actors can encapsulate any data structure the developer chooses for storing a collection of records. We sketch the interface for an actor in Figure 2. DPA views an actor's implementation as a black box. Actors are only required to implement four core methods: create, destroy, serialize, and deserialize (an optional fifth method, *snapshot*, is discussed in §4.2), which the DPA runtime uses for data management. The DPA runtime also maps multiple actors to each physical machine and performs load balancing and auto-scaling. Actors typically implement other methods, e.g., custom methods for querying a search index, which are invoked by DPA operators or update functions when the runtime schedules those to run against an actor. Unlike in some general-purpose actor runtimes, actors in DPA can only communicate through DPA's APIs; they cannot pass arbitrary messages to each other.

DPA actors are organized into *tables*, logical collections of data comprising multiple actors. Tables enable systems to manage multiple datasets and address queries and updates. To partition data across actors in a table, DPA maps records inserted in the system to actors based on *partition keys*; all records with the same key are assigned to the same actor.

3.2 Data Updates

In a conventional actor model, clients communicate with one actor at a time, updating its state directly. In a query serving system, however, users often need to update data partitioned across several actors, typically with consistency or atomicity concerns. Therefore, DPA lets developers implement parallel *update functions*, which update multiple actors in a single table. To perform updates, users implement an UpdateFunction interface with several methods, as shown in Figure 2.

Users invoke update functions on DPA tables and supply them with sets of records to add or change. For example, if a user is maintaining a library catalog in Solr, they might supply an update function with records containing information on new books. The runtime maps the records to actors by partition key, then runs the user's update function on each actor with its corresponding records.

To support the diverse data models of query serving systems, DPA provides configurable consistency and atomicity guarantees for updates. These change how updates are implemented. If developers only require eventually consistent updates, they need only implement an "update" method ap-

Actor Interface	
create() → Actor	Create a new (empty) actor.
destroy()	Destroy an actor.
serialize() →File	Serialize an actor's data to files on disk.
deserialize(File) →Actor	Reconstruct an actor from files on disk.
snapshot() →Actor	Create a snapshot of an actor's data.
Record Interface	
getPartitionKey() →Int	Get a record's partition key.
Update Function Interface	
updatedTableName() →Table	Name of the table to be updated.
consistencyLevel() →Level	What consistency level to use? (§4.2)
update(Actor, List[Record])	Apply eventually consistent update to an actor.
prepare(Actor, List[Record]) → Bool	Prepare a serializable update.
commit(Actor)	Atomically make prepared changes visible.
abort(Actor)	Roll back prepared changes.
Parallel Operator Interface	
inputs() →List[Operator|Table]	What are the input operators and tables?
keysToQuery() →Map[Int, List[Int]]	For inputs, what partition keys are used?
operator() → OperatorFunction	The operator function. Signature depends on the operator (see below).
Parallel Operator Functions	
map(Actor) → Data	Apply a transformation to data.
scatter(Actor) → List[(K, C)]	Partition data into (attribute, chunk) pairs.
gather(K, List[C], Actor) → Data	Combine chunks with the same attribute, plus actors whose partition key matches that attribute; materialize the output.
query(Actor) → V	Query an actor to obtain a value.
combine(List[V]) → V'	Combine values into a query answer.

Figure 2: The DPA interface. It consists of callback functions implemented by the developer and invoked by Uniserve to manage data and execute queries.

plying an update to an actor. However, if they need stronger guarantees such as serializability, they must implement the participant protocol of two-phase commit (prepare, commit and abort). We discuss consistency in Section 4.2.

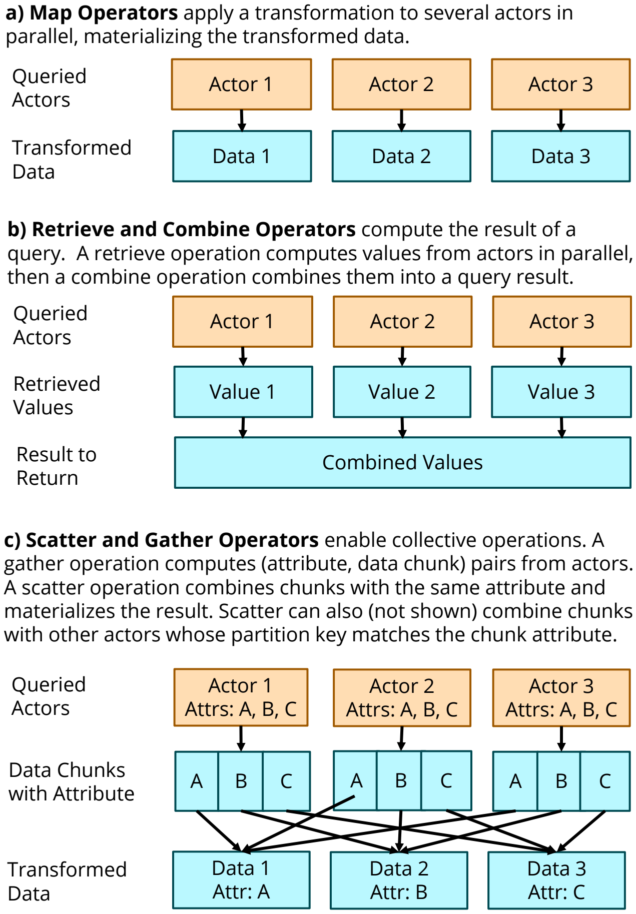
3.3 Queries

Unlike traditional actor models, query serving systems execute parallel queries over data stored in many actors. To enable these queries, DPA provides a small but general set of *parallel operators* which let developers construct queries from generic operations like map and broadcast. We list the parallel operators in Figure 2 and diagram them in Figure 3.

Users write queries by subclassing one of several parallel operator classes (e.g., MapOperator) and implementing appropriate callback functions. Queries may be composed of multiple operators. In practice, we expect developers to implement a query planner in their system's client library that translates queries to DPA operators for execution by the DPA runtime. Most existing query serving systems have similar planners. Both the query planning logic and operator execution callbacks can be single-node: the DPA runtime handles the work of distributing a plan's computation by executing each operator on each actor that contains relevant data.

Operators cannot modify actor state (only updates can), but instead materialize output data that later operators can read. The input to each operator is a list of tables and of data materialized by other operators. Operators can specify what partitions of their input data to query through their *keysToQuery* method, listing specific partition keys for each input.

DPA provides five generic parallel operators that we found sufficient to support the serving systems we considered (Section 5), though more operators could be added:


a) Map Operators apply a transformation to several actors in parallel, materializing the transformed data.

b) Retrieve and Combine Operators compute the result of a query. A retrieve operation computes values from actors in parallel, then a combine operation combines them into a query result.

c) Scatter and Gather Operators enable collective operations. A gather operation computes (attribute, data chunk) pairs from actors. A scatter operation combines chunks with the same attribute and materializes the result. Scatter can also (not shown) combine chunks with other actors whose partition key matches the chunk attribute.

Figure 3: The five DPA parallel operators.

Map. The *map* operator applies a function to actors in parallel and materializes the transformed data. For example, a map operator might search for documents in a collection based on a field, or in a subset of actors specified via *keysToQuery*.

Retrieve and Combine. The *retrieve* operator computes a value from an actor and returns it to the DPA client. It is used to retrieve the results of a query. If retrieve is executed on many actors in parallel, it must be followed by a *combine* operator, which aggregates retrieved values. Retrieve and combine must be the last two operators executed in a query. For example, if in Solr we have several actors storing indexed text data and wish to search it for the word "computer," we can execute a retrieve operation to find documents containing the word "computer" on each actor, then combine these results.

Scatter and Gather. The last two operators, *scatter* and *gather*, provide data communication between actors, enabling collective operations such as broadcast and shuffle. The scatter operator produces from an actor a set of (attribute, chunk) pairs, where the attribute can be any value and the chunk contains data stored in a developer-defined serialized format. A scatter operator must be followed by a gather operator. Gather executes one time for each attribute produced by the preceding scatter. Each execution of gather takes in all data chunks associated with that attribute, along with any actor containing data whose partition key matches the gather attribute, and materializes combined and transformed data.

To demonstrate scatter and gather, consider a shuffle join in a data warehouse setting. Say we have tables of customer data $C(c\_id, country)$ and order data $O(o\_id, c\_id, price)$, both partitioned across several actors. We wish to compute the total amount of money spent by each French customer: `SELECT c_id, SUM(price) FROM C, O WHERE C.country='France' GROUP BY c_id`. First, we execute a scatter operation on every actor containing data from C or O. This operator returns (attribute, chunk) pairs where every attribute corresponds to a set of customers (range of values of c_id) and every chunk contains data associated with those customers. We then execute a gather operator on the results of the scatter. Each gather execution takes in a unique attribute and all its associated chunks. In other words, it takes in all records from both tables corresponding to a set of customers. The operator executes the original query on this data, computing the amount of money spent by each French customer. Each execution materializes new data containing results for a different set of customers; this data collectively forms the result of the original query. Subsequent operators could then query this data; for example retrieve and combine operators could be used to find the ten top-spending French customers.

3.4 Case Study: Solr

We now describe how to create a DPA port of the distributed full-text search system Solr [9]. Natively, Solr stores data by sharding text documents across Lucene inverted indexes (custom data structures enabling ultra-fast search [34]) 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 DPA, we encapsulate inverted indexes in actors. We add data to actors in units of Solr documents, which act as DPA records. Just like Solr, we hash documents to obtain a partition key, then use it to assign them to actors.

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 DPA, we must translate Solr queries to DPA queries. Because all Solr queries are searches, we can implement them using DPA retrieve and combine operators. Each retrieve operator searches its target actor for a set of results, then the results are combined and returned. For example, in a query that searches for books whose title contains the word "goblin," retrieve operators run in parallel on every queried actor, searching their data for "goblin." A combine operator then combines the results. The DPA port of Solr is implemented in <1K lines of code (replacing ~90K lines of native Solr code), and can execute any query recognized by the standard Solr parser. As we show in Section 7, our port matches native Solr performance while providing features lacking in native Solr, such as load balancing.

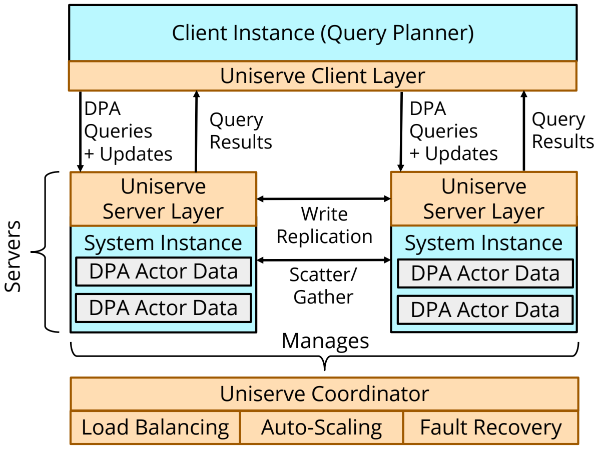


Figure 4: The Uniserve architecture. Servers run a thin Uniserve layer (orange) above actors encapsulating partitions of data (gray) stored in instances of the underlying system (blue). A coordinator manages cluster state and provides distributed features.

4 Uniserve: A Runtime for DPA

We implement DPA in a runtime called Uniserve. In the DPA programming model, developers take responsibility for implementing actors and queries on a single node, but Uniserve takes responsibility for distributing them, managing actors and executing queries at scale. Uniserve automatically provides critical distributed features such as fault tolerance, durability, consistency, load balancing, and elasticity.

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 [49]. Uniserve additionally requires an external durable storage system (e.g. S3 or HDFS) to back up data. 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 DPA and Uniserve, each server would run a Uniserve layer above a single-node Solr instance. DPA actors encapsulate physical partitions of data stored in this system, so for example each actor might encapsulate a Solr inverted index. The Uniserve layer facilitates query execution. It receives query operators and updates from clients and executes them in the underlying system using the DPA interface. Additionally, it handles update replication, maintains a log of the most recent updates, periodically backs up data to durable storage, and transfers actors between servers 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 DPA parallel operators (or update functions). The client then submits these to the appropriate servers, eventually receiving and returning a result. Clients learn actor 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 Update Consistency and Atomicity

Conventional actor runtimes do not provide cross-actor data consistency guarantees, assuming operations occur on a single actor at a time. Query serving systems, however, perform parallel updates on partitioned and replicated data, so Uniserve provides cross-actor consistency and atomicity guarantees.

Query serving systems typically ingest bulk data for analytics; for example time series in Druid or logs in Solr. Updates are usually append-only, but modification of existing data is possible. Most updates are batched, and systems provide high update throughput but not necessarily low update latency. Many systems, like Druid, do *not* support transactional semantics. However, they still provide update consistency and atomicity guarantees of varying strength.

Uniserve automatically provides primary-backup actor replication and data consistency guarantees to query serving systems. Because query serving system data models vary, we make these guarantees *configurable*: when implementing an update function (§3.2), developers can choose a level of consistency appropriate to their data model. Uniserve provides the consistency levels most common in the query serving systems we surveyed. In the remainder of this section, we describe these guarantees and what developers must implement to obtain them. Then, in Section 4.3, we explain how Uniserve upholds its guarantees in case of failures.

Eventual Consistency. By default, Uniserve provides eventual consistency, guaranteeing only that all replicas of an actor eventually converge to the same state. Many systems, like Solr and Druid, use eventual consistency [19]. To write an eventually consistent update function, developers need only implement an "update" method. To execute an eventually consistent update, Uniserve applies it to the primary of an actor synchronously, then replicates it asynchronously. All replicas of an actor apply the same updates in the same order.

Serializable Updates. Uniserve can guarantee serializability for updates, so the outcome of a sequence of updates is equivalent to the outcome of the updates executed serially. As implemented, this also guarantees linearizability, so read queries made after an update completes always reflect the update. This functionality was recently added to MongoDB [74] and is common in data warehouses. To write a serializable update function, developers must implement the participant protocol of two-phase commit, with separate prepare, commit, and abort stages. Uniserve only commits an update if it has successfully prepared on all actors and their replicas, aborting if failures occur. We currently do not allow multiple serializable updates to run concurrently on the same table, but plan to add concurrency control in the future.

Full Serializability. Uniserve can make updates serializable (and therefore atomic) with respect to read queries, so a parallel read query either sees an update applied to all actors or to none of them, as in SQL databases. To obtain this guarantee, developers must both provide serializable update functions and implement the optional *snapshot* actor method (Figure 2). Using this method, Uniserve creates a versioned copy of each actor's data upon update and ensures that read queries see consistent data versions across actors. We expect developers to implement snapshot using optimizations such as shadow paging and copy-on-write to minimize its cost.

4.3 Fault Tolerance and Failure Recovery

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 the coordinator and ZooKeeper are always available; if either fails the cluster will be unavailable until they are restarted, with the coordinator restoring its state from ZooKeeper.

Durability. Uniserve provides update durability through replication and through asynchronous backup to durable storage such as S3. If all replicas of an actor fail, the coordinator orders a random surviving server to load the actor from durable storage. Thus, Uniserve can only lose data if all replicas of an actor fail, and will only lose data committed since the last backup. Additionally, eventually consistent updates can be lost if the primary fails before the updates are replicated.

Update Fault Tolerance. When providing eventual consistency, Uniserve only guarantees that all replicas of an actor will eventually converge to the same state. Therefore, it is possible for an update to partially succeed—to succeed on some actors but fail on others. If the primary of an actor fails, the coordinator chooses the replica with the most advanced update as the new primary, relying on the guarantee that all replicas apply the same updates in the same order. All other replicas then sync with the new primary, applying missing updates from its log to converge to its state.

When providing serializability, Uniserve guarantees that all updates either totally succeed or abort. Uniserve only commits an update if it has successfully prepared on all actors 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).

Query Fault Tolerance. If a failure occurs during the execution of a parallel operator on an actor, the client retries with a different replica. It keeps retrying until it has exhausted all replicas; this occurs only if all are lost, in which case the actor must be restored from durable storage and the query fails.

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 [45]. The obvious way to balance load is through fine-grained query scheduling, but this is impractical for query serving systems because of their strict latency requirements and because all queries must run on specific data partitions. Instead, Uniserve balances load through data placement, managing the actor-to-server assignment to ensure no server is overloaded.

By default, Uniserve provides a greedy load balancing algorithm, similar to that of E-Store [67], which repeatedly moves the most-loaded actors from the most-loaded servers to the least-loaded servers while also replicating actors 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 actors to servers. If a policy is provided, Uniserve takes responsibility for collecting its input data and implementing its output assignment, moving actors to new locations.

A data placement policy must be expressed as a function that takes in the total query load (self-reported by the underlying system) and memory and disk usage of each actor, as well as the current assignment of actors to servers. It returns an updated assignment of actors to servers, expressed as a map from actor number to a list of server IDs. Assignments may replicate actors across multiple servers, either for redundancy or to spread out their load.

To physically move actors during load balancing, Uniserve first prefetches, from durable storage, replicas of reassigned actors on their target servers. These then sync with the actor primary, applying updates from its log. Only after replicas are ready does Uniserve notify clients of the actor movement. Then, after notifying clients, it deletes the original copies of the actors if necessary. If some of the deleted actors were primaries, Uniserve designates randomly selected replicas as new primaries. This procedure ensures high query availability during shard transfer, but if a primary is removed updates may briefly block while a new primary is designated.

	System Type	Data Type	Query Operations
Druid [70]	OLAP	Indexed Tables	Aggregations, joins
Pinot [51]	OLAP	Indexed Tables	Aggregations
ClickHouse [11]	OLAP	Indexed Tables	Aggregations, joins
Atlas [10]	Timeseries DB	Time series	Aggregations
InfluxDB [14]	Timeseries DB	Time series	Aggregations
Solr [9]	Full-Text Search	Indexed text	Text search
ElasticSearch [13]	Full-Text Search	Indexed text	Text search
Unicorn [40]	Graph Database	Social Graphs	Graph Search
FAISS [52]	Vector Database	Vectors	Vector Search
Pinecone [18]	Vector Database	Vectors	Vector Search
Vespa [21]	Vector Database	Vectors	Vector Search
MongoDB [15]	NoSQL	Documents	Aggregations, search
MonetDB [50]	Data Warehouse	Relational Tables	SQL

Table 2: Systems we believe can be distributed with or ported to DPA and their properties. Systems we have implemented are in bold.

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 [32]. 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 actors as necessary.

An auto-scaling policy must be defined as a function that takes in the CPU utilization, memory and disk usage, and total query load 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 (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 actors; if servers are removed this reassignment is done preemptively so availability is not affected.

5 Generality of DPA

In this section, we demonstrate the generality of DPA 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 rapidly answer multidimensional analytics queries over tables. They are closely related to time series databases, which query time-ordered data. Both typically store data in a compressed and indexed columnar format. Their workloads usually filter, group, and aggregate this data. This naturally fits DPA: we partition data by key columns across actors (e.g., by time range) to support partition filtering, and implement most aggregations with retrieve and combine operators, using scatter and gather to shuffle or broadcast data if necessary. We implement a port of Druid [70], which is both an OLAP system and timeseries database, on DPA; its design patterns generalize to others from both categories such as Pinot [51], Clickhouse [11], 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 [34]. Because all their queries are searches, they are easy to fit to DPA, as we showed in Section 3.4. We implement a port of one full-text search system, Solr [9], and can generalize to others like ElasticSearch [13].

Vector Databases. Vector databases store data using vector indexes to perform fast nearest neighbor search, often for machine learning workloads. Recent examples are Pinecone [18], Vespa [21], and FAISS [52]. Like full-text search systems, they easily fit DPA as their queries are searches.

Graph Databases. Graph databases represent data using a graph data model. Some graph database queries are data-parallel, including whole-graph algorithms like PageRank and queries like finding all checkins at a certain location in a social network graph [40]. 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 data-parallel operators such as iterative self-joins. Data-parallel graph databases such as Facebook's Unicorn [40] search engine fit the DPA programming model.

Other Systems. DPA can distribute other systems with data-parallel queries. For example, we implement a DPA port of the NoSQL document store MongoDB. We also implement a simplified OLAP data warehouse based on the single-node columnar database MonetDB.

Limitations of DPA. DPA has two major limitations. First, its query model works best for data-parallel queries. As we have shown, this is sufficient for many popular query serving systems, but not some specialized query types like graph traversal queries. Nonetheless, we believe DPA would have made many of today's query serving systems easier to develop, and can augment them with missing functionality.

Second, DPA is not designed to provide low latency for small point updates, especially with transactional guarantees. Small transactional updates are rare in query serving systems because these are often updated in bulk (e.g., using data collected in a message queue like Kafka). However, they are common in other contexts such as online transactional processing (OLTP) workloads, which DPA does not target.

6 Distributing Systems with DPA

To demonstrate the practicality of DPA, we use it to distribute four systems. First, we port Druid, Solr, and MongoDB to DPA, replacing their native distribution layers. Then, we build a new system using DPA: a simplified data warehouse based on the single-node column store MonetDB.

We implement each of our four systems in <1K lines of code (LoC). This number includes all code needed to implement the DPA interfaces with each system's already-existing single-node implementation, but not any code in Uniserve. This demonstrates that DPA simplifies building distributed query serving systems, as it replaces custom distribution layers totaling ~90K LoC in Solr, ~120K LoC in MongoDB, and ~70K LoC in Druid. For comparison, Uniserve itself is ~10K LoC. This smaller size is because Uniserve makes use of tools like ZooKeeper and gRPC for basic functionality that other systems implemented themselves.

Solr. We described the port of Solr in Section 3.4.

Druid. In our port of Druid [70], actors encapsulate single-node 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 actor manipulation and update functionality using the Druid datasource API. Serializing and deserializing data is easy because Druid segments live in portable directories on disk.

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 queries use retrieve and combine operators to separately query actors then aggregate the results. Druid uses a similar model natively. We can also use scatter and gather operators to support Druid's recently-added [60] broadcast joins.

MongoDB. In our port of MongoDB [15], actors encapsulate single-node MongoDB collections, analogous to database tables. We implement most actor manipulation and update functionality using the MongoDB API for manipulating collections. We implement actor data 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 implementations are similar to those in our Druid port and those in native MongoDB: querying actors separately, then combining the results.

MonetDB. We have built using DPA a simplified data warehouse based on the single-node column store MonetDB [50]. It stores data in MonetDBLite [65], the embedded implementation of MonetDB. Each server runs MonetDBLite embedded in the same JVM as the Uniserve layer. Actors encapsulate MonetDB tables and implement interface methods using equivalents in the MonetDBLite API.

Our simplified data warehouse supports a large subset of

SQL, including selection, projection, equijoins, grouping, and aggregation. We implement simple aggregation queries with retrieve and combine operators, as in other systems. To execute more complex queries, such as joins, we use scatter and gather operators to shuffle or broadcast data, then use retrieve and combine operators to produce a query result.

7 Experimental Evaluation

We evaluate DPA and Uniserve using the four systems discussed in Section 6. As we have shown, DPA makes distributing these systems considerably simpler; each requires <1K lines of code to distribute as compared to the tens of thousands of lines in custom distribution layers (~90K in Solr, ~120K in MongoDB, and ~70K in Druid). Our evaluation shows that:

1. Distributed systems built using DPA and a specialized single-node system, such as our MonetDB-based simplified data warehouse, can match or outperform comparable distributed systems such as Spark-SQL and Redshift.

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

3. DPA 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 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 0.20.1, MongoDB 4.2.3, and MonetDBLite-Java 2.39. We use four data servers for smaller-scale benchmarks and forty for large-scale benchmarks. In both cases, an additional node is set aside for the coordinator.

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 Experiment Workloads

We evaluate each system with a representative workload taken when possible from the system's own benchmarks. All of our comparison systems achieve state-of-the-art performance on their benchmarks, so DPA also achieves state-of-the-art performance by matching them. We benchmark Solr with queries from the Lucene nightly benchmarks [59]. We run each query on a dataset of 1M Wikipedia documents (more for large-scale benchmarks) taken from the nightly benchmarks. We use two representative 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 benchmark queries from the Druid paper [54, 70]. These are TPC-H queries modified by the Druid developers to reflect the strengths of Druid; we run 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 [39], simulating an analytics workload. Before running the workload, we insert 10M sequential items (10GB of data) into the database. We run a workload of 100% scans, where each scan retrieves 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 [22].

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.

7.3 Benchmarks

Ideal Conditions. We first benchmark our Solr, Druid, and MongoDB ports on a uniform workload where 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, as expected, our ports' performance is similar to native system performance on all benchmarks.

Scalability. We next evaluate Uniserve scalability, scaling the Solr benchmarks with one client worker from four to forty servers. We scale the amount of data to maintain a constant 5 GB of data per server. We show results in Figure 6. Because all queries access all data, we expect performance to be near-constant as the number of servers (and amount of data) increases, and indeed it is.

Data Warehouse Benchmarks. We next benchmark our simplified data warehouse based on MonetDB, comparing its performance with native MonetDB, Spark-SQL [28], and Redshift [47]. We use three TPC-H queries: Q1, an aggregation query; Q3, a three-way join; and Q10, a four-way join. We implement Q3 and Q10 using scatter and gather operators to perform both broadcast and shuffle 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

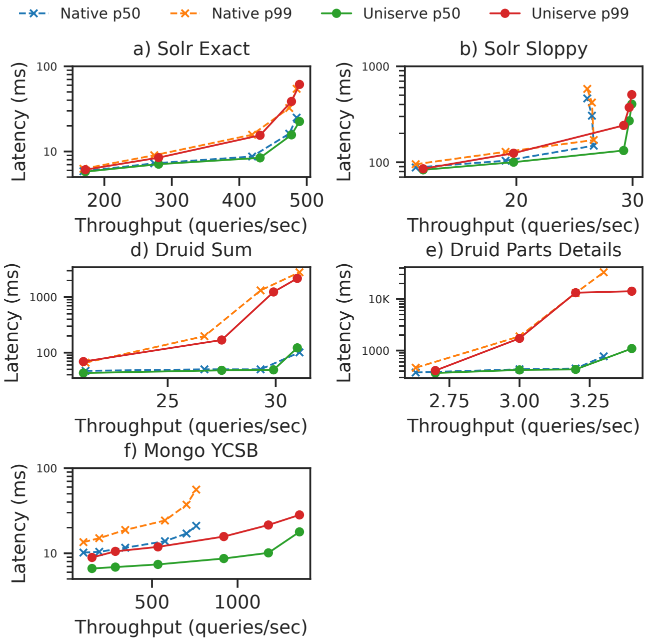


Figure 5: Throughput versus latency for native systems and DPA ports on uniform and static query workloads. Our ports match native system performance.

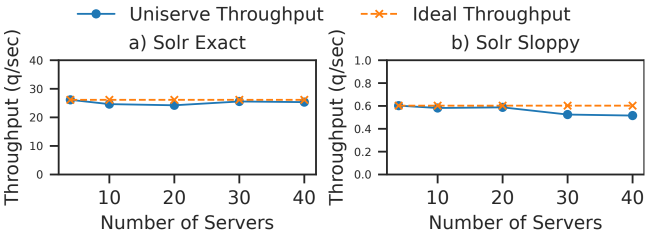


Figure 6: Uniserve scalability on the Solr benchmarks.

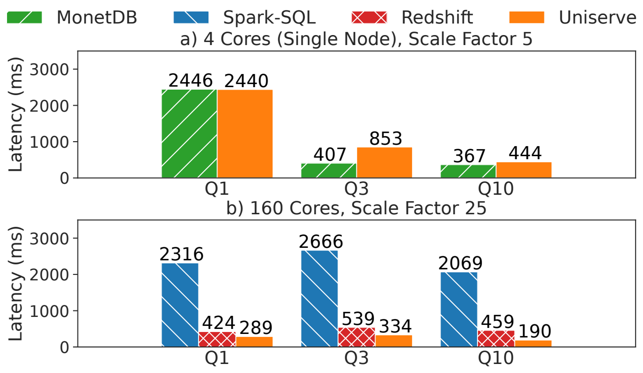


Figure 7: Comparison between our simplified data warehouse, single-node MonetDB, Spark-SQL, and Redshift on TPC-H queries Q1, Q3, and Q10 on 4 cores (single-node) and on 160 cores with TPC-H scale factors of 5 and 25. Uniserve is competitive on a single node and outperforms Spark-SQL and matches Redshift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

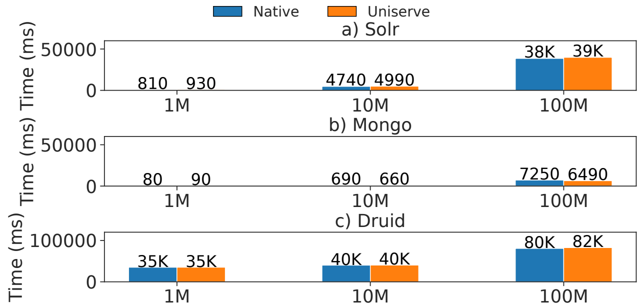


Figure 8: Execution time of 1 MB, 10 MB, and 100 MB updates with native systems and with Uniserve. Uniserve matches native system performance.

Q1 but 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 servers for Uniserve and Spark-SQL, five dc2.8xlarge Redshift servers). We find that our data warehouse outperforms Spark-SQL and matches Redshift. This shows that by distributing a single-node system like MonetDB, DPA can in <1K lines of code match or outperform popular distributed systems like Redshift and Spark-SQL on their core workloads.

Update Performance. We next investigate Uniserve update performance. We benchmark 1 MB, 10 MB, and 100 MB updates on Solr, Druid, and MongoDB, using each system's benchmark dataset. We use these bulk writes because they are typical of query serving system workloads. We compare native system performance to Uniserve performance, showing results in Figure 8. For Solr and Druid, we provide eventual consistency, matching those systems' semantics; for MongoDB we enable update serializability (through two-phase commit in Uniserve) and perform the update on four partitions in parallel. We find that across the board, Uniserve matches native system update performance.

Hotspots. To demonstrate the importance of load balancing, we next investigate the performance of the default Uniserve load balancer on benchmarks with load skew. We compare against Druid, whose load balancer ensures each server hosts the same amount of data but does not balance query load. First, we execute a workload where 7/8 of the queries are sent to a single slice of data (four months) and scatter the rest uniformly on the remainder of the data, showing results in Figure 9a. Because Uniserve balances load in the hotspot, it outperforms Druid by up to 3×.

We next repeat the experiment, fixing the number of clients at twelve but varying the fraction of queries sent to the hotspot. We show results in Figure 9b. We find that changing skew does not affect Uniserve performance because Uniserve keeps load balanced under any load distribution. However, Druid 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 workload. We run the Solr sloppy benchmark for six hours sending

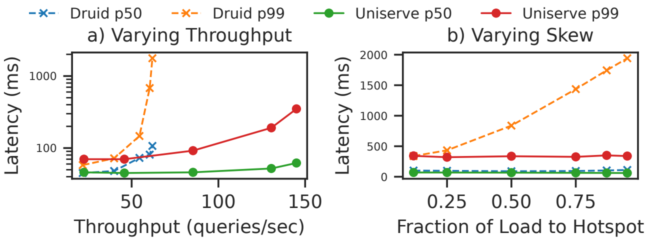


Figure 9: Effect of query skew and load balancing for Druid- and Uniserve-distributed queries. On the left, we vary throughput in a workload where one slice of data receives 7/8 of queries; Uniserve balances load and so outperforms Druid. On the right, we vary the fraction of queries received by the hot slice; Uniserve keeps performance constant as skew increases but Druid does not.

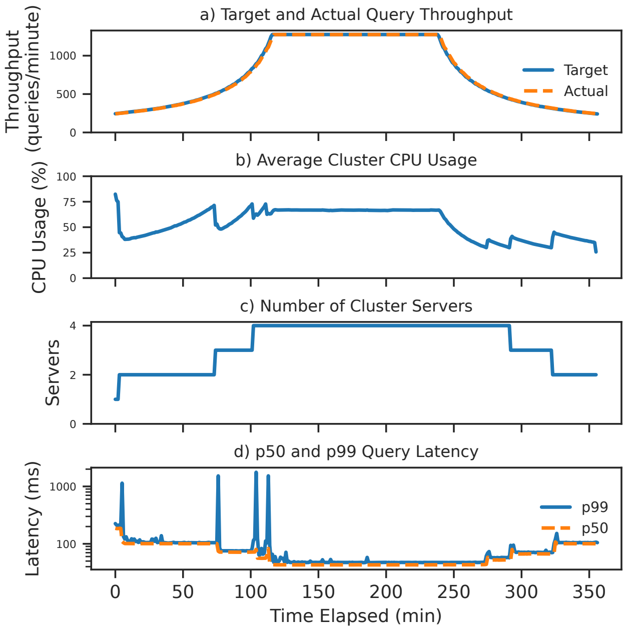


Figure 10: On the Solr sloppy benchmark with Uniserve autoscaling, varying target throughput and observing effects on actual throughput, average cluster CPU usage, the number of cluster servers, and query latencies. Uniserve scales the cluster so that actual throughput always matches target throughput; 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 servers.

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 actors, 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 actors onto new servers before serving any queries, Uniserve guarantees that queries need not contend with actor 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, using the Druid `sum_all` benchmark. We run this benchmark for ten minutes with a client sending 500 asynchronous queries uniformly 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 data partition or actor (one on each server), and once with just a single replica. We show results in Figure 11.

When all servers have replicas of all partitions (11b), Uniserve recovers instantly, routing queries to replicas. Druid, however, takes thirty seconds to route queries to replicas, resulting in hundreds of query failures. When there is only one replica of each partition (11a), both systems fail hundreds of queries but recover within 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. This experiment confirms previously-reported issues Druid faces in large-scale deployments [56] and shows Uniserve can address them.

8 Related Work

Actors Actor models are abstractions for concurrent computation built around stateful agents called actors [26]. Prior surveys [53] identified five characteristics of an actor model: actors encapsulate their own state, exhibit location transparency, are mobile, are scheduled fairly, and communicate through message passing. DPA actors exhibit four of these properties: they encapsulate shards of data, are addressable through partition keys, can be moved between servers, and share resources on each machine. However, unlike prior actor models, DPA actors do not communicate via message passing but are instead acted on by parallel operators and updates. Other systems based on actors include Erlang [30], a programming langauge with built-in actor support; Akka [8], which supports actors on the JVM, including persistent actors with durable state; Orleans [33,35], which supports virtual actors that are only instantiated on-demand when required; and Ray [61,69], where developers can call remote procedures on stateful actors.

Critically, most existing actor models focus on *concurrent* computations, not the *parallel* ones performed by query serving systems and DPA. Most actor models do not support cross-actor transactions, requiring users to manually implement protocols such as two-phase commit. Even in systems

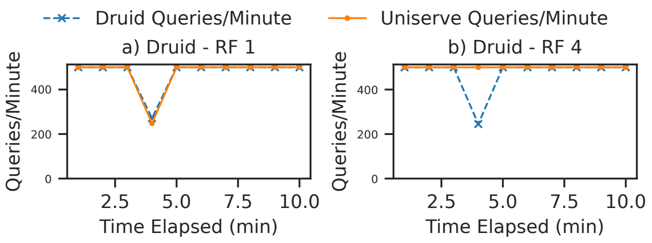


Figure 11: Query throughput (targeting 500 queries/min) of Druid- and Uniserve-distributed `sum_all` queries when one data server is killed after three minutes. The left graph shows performance starting with a single replica of each partition; the right graph with four.

like Ray which allow many actors to be accessed in parallel, there is only limited support for collective operators like gather or scatter [23] and no support for consistency or atomicity guarantees across actors. DPA instead reasons about parallel operators directly, taking advantage of the fact that query serving systems mostly need to support bulk updates as opposed to many concurrent write transactions, and offers multiple consistency levels to support different system designs. Eldeeb and Bernstein extended Orleans with a transactional actor concept [43], but that work focused on allowing clients to make multiple calls to the same actor as a single transaction and tracking these calls' effects on downstream actors through message passing, which would be expensive for the large data-parallel operations that DPA targets. Early versions of Akka also supported transactional actors on the same server [2], but this was removed because the mechanism was hard to extend to multiple servers [1].

Other Distributed Programming Models One class of programming model often used for parallel queries are batch frameworks like MapReduce [42], Hadoop [66], Percolator [64], Dryad [71], and Spark [72]. Unlike query serving systems, these only execute computations and do not provide abstractions for managing data, typically assuming its immutability. Moreover, they are not designed for low latency and typically do not implement many of the optimizations used in query serving systems, such as augmenting data with secondary indexes. Researchers have attempted to build updatable data structures over Spark RDDs, such as PART [41], but these are greatly limited by the immutability of RDDs.

Streaming and dataflow systems like Spark Streaming [27, 73], Naiad [62], and Flink [36] execute queries in real time on streaming data. However, unlike query serving systems, they focus primarily on continuous computation (incrementally updating the result of a query as data comes in) and do not perform data management or low-latency query serving. They are often used to write data into a query serving system.

Cluster management systems like Helix [46], Mesos [48], and YARN [68] are designed to deploy distributed systems at scale. Mesos and YARN are primarily concerned with assigning resources to each application. Helix, like Uniserve, automatically manages the applications running on it, providing features such as elasticity and fault tolerance. However, it is not designed for query serving workloads and lacks a query model and abstractions for consistency and atomicity.

Auto-sharding systems like Slicer [25], Centrifuge [24], and Shard Manager [55] assign data and queries to shards based on partition keys, like DPA. Slicer and Centrifuge *only* manage key affinity, telling applications what keys are assigned to what servers. Shard Manager goes further and manages data placement, moving data shards between servers. However, unlike Uniserve, these systems do not provide high-level abstractions on top of shards, such as parallel operators and updates with consistency and atomicity guarantees.

Thor [58] stores data in persistent distributed objects for heterogeneous applications to access. These objects resemble DPA actors, but Thor must run object operations on client machines and does not provide high-level abstractions such as a query model or configurable consistency guarantees.

Middleware systems for databases automatically distribute data and queries across existing database installations and provide features like fault tolerance [57, 63] and load balancing [31]. However, these solutions are typically specialized to particular database types, like relational databases [37, 38] or NoSQL stores [44], and do not provide general abstractions to support a wide range of data and query models like DPA.

9 Conclusion

Query serving systems are an important emerging class of distributed systems that power many Internet applications. Traditionally, they are implemented from scratch, requiring substantial effort to add distributed query processing and data management functionality. We presented *data-parallel actors (DPA)*, a high-level programming model that allows developers to build reliable and performant distributed query serving systems from single-node data structures and logic. We showed that DPA can express the functionality of a wide range of query serving systems by building a simplified data warehouse and porting Druid, Solr, and MongoDB to DPA in <1K lines of code, matching performance and adding rich missing functionality such as automatic load balancing and auto-scaling. We believe DPA is a valuable tool to help organizations more easily develop these important systems.

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

References

[1] Akka 2.4 migration guide. https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html.

[2] Akka transactors documentation. https://doc.akka.io/docs/akka/2.2/scala/transactors.html, 2015.

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

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

[5] Why Architecting for Disaster Recovery is Important for Your Time Series Data. https://www.influxdata.com/customer/capital-one/, 2018.

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

[7] Enterprise Scale Analytics Platform Powered by Druid at Target. https://imply.io/virtual-druid-summit, 2020.

[8] Akka. https://akka.io/, 2021.

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

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

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

[18] Pinecone. https://www.pinecone.io/, 2021.

[19] Shards and Indexing Data in SolrCloud, Aug 2021.

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

[21] Vespa. https://vespa.ai/, 2021.

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

[23] Ray collective communication. https://docs.ray.io/en/latest/ray-collective.html, 2022.

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

[25] 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.

[26] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[27] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[28] 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.

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

[30] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, September 2010.

[31] 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.

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

[33] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

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

[35] 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.

[36] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[37] 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.

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

[39] 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.

[40] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

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

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

[43] Tamer Eldeeb and Phil Bernstein. Transactions for distributed actors in the cloud. Technical Report MSR-TR-2016-1001, October 2016.

[44] 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.

[45] 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.

[46] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, Adam Silberstein, Kapil Surlaker, Ramesh Subramonian, and Bob Schulman. Untangling cluster management with helix. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[47] 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.

[48] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[49] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[50] 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.

[51] 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.

[52] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[53] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference*

on Principles and Practice of Programming in Java, pages 11–20, 2009.

[54] Xavier Léauté. Benchmarking Druid. 2014.

[55] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, New York, NY, USA, 2021. Association for Computing Machinery.

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

[57] 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.

[58] 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.

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

[60] Gian Merlino. Druid Initial Join Support, Oct 2019.

[61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[62] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[63] 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.

[64] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

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

[66] 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.

[67] 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.

[68] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[69] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021.

[70] 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.

[71] 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.

[72] 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.

[73] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[74] Siyuan Zhou and Shuai Mu. Fault-tolerant replication with pull-based consensus in mongodb. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021.