



The technical architecture of FaunaDB, an adaptive operational database

Table of Contents

Origins of the Adaptive Operational Database	3
Foundations	4
Data Model.	5
Query Language	6
Implementation.	6
Examples	7
Temporality	7
Security.	8
Identity.	8
Auditing and Logging	9
Scalability	9
Cluster Topology	10
Routing	10
Data Partitioning	10
Fault Tolerance and CAP	11
Performance	13
Durability.	14
Writes	14
Reads.	14
Performance	14
Consistency	15
Tradeoffs.	15
Implementation.	16
Resiliency	17
Quality of Service.	17
Background Tasks	18
Implementation.	19
Operations	19
Topological Changes	19
Other Maintenance	19
Success Patterns with FaunaDB.	20
Conclusion.	20
Contact	21

Origins of the Adaptive Operational Database

The history of database management systems is one of increasing abstraction. Initially, hierarchical and key-value databases abstracted query expression from storage format management, improving performance, reuse, and correctness.

Later, in the business data era, relational systems like SQL abstracted query expression from the underlying data structures and, to a degree, the query execution plan [1].

Now, in the cloud era, the typical database user is no longer an analyst sitting at a workstation. From an operational perspective, customers have supplanted data experts as the primary consumers of data. To meet customers' expectations the software enterprise is increasingly tasked with delivering a small number of query expressions to a very large number of end users and devices at high throughput, low latency, and global scale.

Historical data is invaluable for analytics and machine learning, but customers care primarily about what is happening now: which friends are online, or the location of their on-route car. While customers are served in real time, the underlying data, including historical data, must be made available for analytics and machine learning workloads which still rely on procedural interaction models.

... the emerging challenge is abstracting application architecture from operational infrastructure.

FaunaDB's team encountered this challenge at Twitter as we developed the distributed data systems for the core business objects: statuses, timelines, users, binary assets, the social graph, and the cache. When data systems do not both enable developer productivity and guarantee operational integrity, business agility is compromised.

The NoSQL movement attempted to address this problem by jettisoning complex database features in order to focus on either ease of initial application development, or ease of operations at scale, but never both. Most NoSQL systems achieved neither.

In this paper we introduce a platform approach that moves beyond both SQL and NoSQL. Our approach works to increase the degree of abstraction between operational and functional concerns. We refer to this approach as the adaptive operational database.

An adaptive operational database offers a single platform to support all common business data interaction and distribution patterns, mandating:

- A multi-model, transactional query language
- Temporality
- ACID consistency
- Global horizontal scalability
- Multi-tenancy
- Quality of service management
- Granular identity management, authentication, and access control

This paper presents the architecture and implementation strategy of FaunaDB, the first database management system to meet these requirements.

02 FOUNDATIONS

Foundations

Adaptive database systems are essentially distributed data operating systems. Like an operating system, FaunaDB is composed of several interrelated subsystems:

- **A functional query language** unifies relational, document, graph, temporal, analytical, and batch processing access patterns while restricting their execution contexts, increasing flexibility, ease-of-use, and safety.
- **A strongly-consistent transaction resolution engine** based on Calvin [1], backed by a distributed global log based on Raft [2] maximizes correctness and ease-of-use without limiting scalability.
- **An on-disk storage engine** based on log-structured merge trees (similar to Google Bigtable [3]) maximizes IO throughput.
- **A tenant-aware resource scheduler**, similar to an operating system kernel, fairly allocates compute, memory and IO resources to competing applications and workloads based on priorities and quotas.
- **Row-level security, identity, and isolation management**, similar to a filesystem, transparently protects data access.
- **A background task scheduler**, similar to Apache Hadoop YARN [4], enables asynchronous execution of long-running jobs.
- **A datacenter-aware routing layer**, similar to a software load balancer, minimizes effective latency and maximizes global availability.

In the same way that an operating system dynamically allocates machine resources to a prioritized and potentially conflicting set of users and processes, FaunaDB dynamically allocates globally distributed data resources to a prioritized and potentially conflicting set of applications, users, and workloads.

FaunaDB is based on technology from Twitter, Facebook, and Netflix, and influenced by the database research community. FaunaDB is implemented in Scala and Java, and runs on the JVM on all major operating systems.

FaunaDB is based on technology from Twitter, Facebook, Netflix and others, and influenced by the database research community.

03 DATA MODEL

Data Model

Modern applications no longer interact exclusively with relational data. Adaptability requires supporting multiple data structures within the same system. With this requirement in mind, FaunaDB implements an inheritance-free object-relational data model, a strict superset of the relational, document, object-oriented, and graph paradigms:

- Records are inserted as structured documents called **instances**, which can include recursively nested objects and arrays as well as scalar types.
- Instances are grouped into **classes**, which are similar to tables in a relational database. However, like a document database, full or partially shared schema within a class is optional, not mandatory.
- Classes are grouped into **databases**. Like filesystem directories, databases can recursively contain other databases.
- Database access is controlled by **keys**. Keys are credentials that identify the application requesting access and close over a specific database context. They can be assigned priorities, resource quotas, and access control roles.
- Derived relations are built with **indexes**. An index is a transformation of a set of input instances into one or more result sets composed of **terms** and **values**. Indexes are expressed as partially applied queries and can transform, cover, order their inputs, and enforce constraints (in FaunaDB, indexes and views are the same).
- Schemas, composed of structural or dependent types, can be optionally enforced by declaring validations. Validations apply partially applied query expressions to inserts, updates, and deletes.
- Queries can be parameterized as **functions**, similar to stored procedures in a SQL system, in order to share logic across applications, abstract logic from applications that are difficult to upgrade in place, or create custom security models.

Modern applications no longer interact exclusively with relational data.

Query Language

SQL is a poor fit for modern operational workloads, as the proliferation of ORM libraries attempting to hide it from the developer has shown. It is not type-safe, leading to frequent security flaws. Its declarative programming model leaves the performance profile of any specific query unknown and unpredictable. It is not extensible in a portable way, and programming complex compute logic within a query is not practical.

FaunaDB implements a new relational query language that is functional, flexible, and type safe.

In order to conform to modern development practices, FaunaDB implements a new relational query language based on Lisp that is functional, flexible, and type safe. Interaction with the database is mediated by drivers that publish DSLs for popular application languages. The model is that of language-integrated query (the LINQ framework is a well-known example).

Developers using FaunaDB write application-native code in a functional style within a transaction context.

The FaunaDB driver reflects on the native expression and serializes it to the wire protocol. The transaction is then transmitted and executed in its entirety by the database. The experience is similar to working with a tuple space or other distributed computation architecture that moves computation to the data [7].

Implementation

FaunaDB's query language makes a number of tradeoffs designed to increase safety, predictability, and performance:

- **Queries are written in the host application language and inherit its safety mechanisms.** There is no additional string evaluation step that can lead to injection attacks.
- **Not all query functionality is permitted in all execution contexts.** For example, in the synchronous interface, table scans are disallowed, and all indexes must be explicitly referenced. This guarantees a consistent and scalable performance profile for customer-facing workloads.
- **All synchronous queries** (and all subqueries in asynchronous queries) **execute atomically and transactionally.** Session transactions are not supported. Because the database receives the entire transaction before constructing the execution plan, execution can proceed with maximum parallelization and data locality. Optimization opportunities like predicate pushdown apply universally and predictably.
- **Query semantics that are inherently unscalable are replaced with semantics that are.** For example, FaunaDB provides stable cursor-based paging instead of the offset/limit style of SQL.

- **Database sessions are stateless.** Every transaction is identified with an access token that closes over the transaction context. Connections are very low overhead and suitable for ephemeral use, making access from serverless applications or embedded devices practical.

These improvements are difficult or impossible in legacy query languages because they require restricting the language, not extending it, damaging standards compatibility.

At the same time, the extensibility of FaunaDB's query language allows the database to incorporate an effectively unlimited number of additional functions common to other data domains such as geographic indexing, full-text search, iterative machine learning, etc., without the burden of grafting custom syntax and extensions onto a closed standard model.

Examples

This transaction, written in Scala, inserts a blog post with case-insensitive tags:

```
Create(Class("posts"),
  Obj("data" -> Obj("title" -> "All Aboard",
    "tags" -> Map(Lambda { tag => Casefold(tag) },
      Arr("Ship", "Travel")))))
```

This read-only transaction looks up posts by the "travel" tag in an index:

```
Paginate(
  Match(Index("posts_by_tags"), "travel"))
```

This read-only transaction performs a relational join of blog posts and inbound references to them by primary key:

```
Paginate(
  Join(Match(Index("posts_by_tags"), "travel"),
    Index("linkbacks_by_post"))))
```

Temporality

Event sourcing, reactive programming and other audit and stream-oriented data architectures are prevalent in modern applications, but are not supported by the underlying datastores. Complex lambda architectures and inconsistent application-managed audit logs are common substitutes for native database support. Typically historical data cannot be accessed in realtime or with the same query capabilities as current data.

In order to support these fundamental interaction patterns with no additional application complexity, all records in FaunaDB (including schema records) are **temporal**. When instances are changed, their prior contents are not overwritten; instead, a new **instance version** at the current transaction timestamp is inserted into the **instance history**, either as a **create**, **update**, or **delete** event. FaunaDB supports configurable retention policies on a per-class and per-database basis.

All reads, including index reads, joins, or any other query expression in FaunaDB , can be executed consistently at any point in the past, or transformed into a **change feed** of events between any two points in time. This is useful for auditing, rollback, cache coherency, syncing to second systems, and forms a fundamental part of FaunaDB's isolation model.

Privileged actors can manipulate historical versions directly to fix data inconsistencies, scrub personally identifiable information, insert data into the future, or perform other maintenance tasks.

05 SECURITY

Security

Most legacy database implement schema-level user authentication only, since they were designed for small numbers of internal business users. But modern applications are exposed to millions of untrusted and potentially malicious actors and must implement identity management authentication, and row-level security at a minimum.

FaunaDB internalizes these concerns in order to deliver both administrative and application-level identity and security either through API servers or directly to untrusted clients like mobile, browser, and embedded applications.

Pushing security concerns to the database guarantees that all applications interacting with the same dataset implement the same access control, and dramatically reduces the attack surface, a critical business risk.

Pushing security concerns to the database guarantees that all applications interacting with the same dataset implement the same access control, and dramatically reduces the attack surface, a critical business risk.

Identity

Application actors in FaunaDB (such as users or customers) can be identified either with built-in password authentication, or via a trusted service that delegates authentication to some other provider. Once identified, application actors receive a **token** they can use to perform further requests that closes over their identity and access context, similar to an access token in OAuth2.

This allows untrusted mobile, web, or other fat clients to interact directly with the database and participate in the row-level access control system. Actors identified as instances never have access to administrative controls.

System actors are identified by keys; keys can have a variety of levels of privilege. Keys always close over a specific logical database scope and cannot access parent databases in the recursive hierarchy, although they optionally can access child databases.

Access Control

System actors have **roles** assigned to their keys which can only be changed by a superior actor. These roles limit activity to administrative access, read/write access to all instance data, or access to public instance data only.

- Row-level security for application access control is managed through assignment of identities and to read, update, create, and delete **access control lists** on instances, indexes, stored procedures, and classes.
- Data-driven rights decisions, such as access groups, can be implemented by assigning **access control query expressions** to ACLs. These query expressions are parameterized on the object that contains the role, and must return a set of identities allowed to fulfill the role.
- Finally, the identity of the actor performing a transaction can be accessed within the context of a stored procedure in order to implement completely custom access logic within the database.

Like a filesystem, FaunaDB transparently enforces row-level access control at all times; there is no way to circumvent it.

Auditing and Logging

All administrative and application transactions in FaunaDB can be optionally logged; additionally, the underlying temporal model preserves the previous contents of all records within the configured retention periods.

Although FaunaDB does not natively track data provenance, applications can tag every transaction with actor information and access that data historically as part of the instance versions.

06 SCALABILITY

Scalability

FaunaDB is designed to be horizontally and vertically scalable, self-coordinating, and have no single point of failure. Every node in a FaunaDB cluster performs three roles simultaneously:

- Serving as a **query coordinator**.
- Serving as a **data replica**.
- Serving as a **log replica**.

No operational tasks are required to configure the role of a node.

FaunaDB is designed to be horizontally and vertically scalable, self-coordinating, and have no single point of failure.

Cluster Topology

Every FaunaDB cluster is made up of three or more logical datacenters (a physical **datacenter** can contain more than one logical datacenter).

Abstracting management of physical hardware from application decisions about compliance, redundancy, and latency is a primary requirement of adaptability. To implement this abstraction in FaunaDB, replication is configured dynamically, at the logical database level. Each physical datacenter contains a copy of the global metadata, as well as copies of the contents of each logical database assigned to that datacenter.

For example, an enterprise can deploy a single FaunaDB cluster that spans multiple cloud infrastructure providers as well as on-premises hardware. Developers within the enterprise can choose on a database-by-database basis where they want to locate their application data, and change those decisions over time without operator intervention. The inverse is also true; operators can change the physical composition of the cluster without affecting the replication strategies of individual applications.

Routing

Any FaunaDB node in any datacenter can receive a request for any logical database in the cluster. If the node does not own the data for that particular logical database, it will forward to a node that does, potentially in another datacenter. This localizes load to the datacenters to which a logical database is assigned. Under some operational conditions, asymmetric routing can be used to partially localize bandwidth as well.

Once a transaction is routed to the correct datacenter, a local node will act as **query coordinator** and begin executing the transaction by pushing read predicates to **data replicas** that own the underlying data, waiting on the responses, and accumulating a write buffer if the transaction includes writes. Read predicates may be as simple as row-level lookups, or as complex as partial query subtrees. Multi-level predicate pushdown is supported. This dramatically reduces latency and increases throughput via increased parallelism and data locality.

... an enterprise can deploy a single FaunaDB cluster that spans multiple cloud infrastructure providers as well as on-premises hardware.

If the transaction is read-only a response will be returned to the client immediately; if the transaction includes writes, it will be forwarded to the appropriate **log replica** for transaction resolution. The log replica will forward the transaction to involved data replicas which will definitively resolve the transaction and return a response to the client-connected node, which will then return the response to the client.

Data Partitioning

Within each datacenter, the logical data layout is partitioned across all nodes within the datacenter. Instances, including their history, are partitioned by primary key. Indexes are partitioned by term. Both instances and indexes

scale linearly for reads and writes, regardless of their cardinality or the number of nodes in the physical cluster. Hotspots in indexes are possible if the read or write velocity of a specific index entry exceeds the median size by a substantial margin. In this case FaunaDB can partition the instance or index entry across multiple ranges and perform a partial scatter-gather query on read, similar to a search system.

Fault Tolerance and CAP

While it is impossible for any distributed system to guarantee both total consistency and total availability at the same time, FaunaDB aims to provide the ideal tradeoff between the two. For most applications, maintaining consistency is optimal. Hence, according to the CAP theorem [9], FaunaDB is a CP system.

Nevertheless, FaunaDB is resilient to many types of faults that would affect availability in a less sophisticated system. In particular, a FaunaDB cluster is not vulnerable to any single point of failure, even at the datacenter level.

Some specific faults that FaunaDB can tolerate are:

- A node is temporarily unavailable (process crash; hardware reboot).
- A node is permanently unavailable (physical hardware failure).
- A node becomes slow (local resource contention or degraded hardware).
- A network partition isolates a datacenter—in this case, the isolated datacenter can continue to serve reads, but cannot accept writes.

A FaunaDB cluster maintains availability in the face of faults due to the redundancy inherent in maintaining multiple replicas of the dataset. For example, in a cluster configured with five datacenters, as long as three datacenters remain available, the cluster can respond to all requests.

While it is impossible for any distributed system to guarantee both total consistency and total availability at the same time, FaunaDB aims to provide the ideal tradeoff between the two.

Although a FaunaDB cluster is capable of responding to transactions despite a partial or total failure in multiple datacenters, it is still in a degraded state. An additional concurrent failure in another datacenter may impact availability.

FaunaDB does not automatically decommission failed nodes or datacenters; this decision is left to the operator to avoid triggering cascading failures.

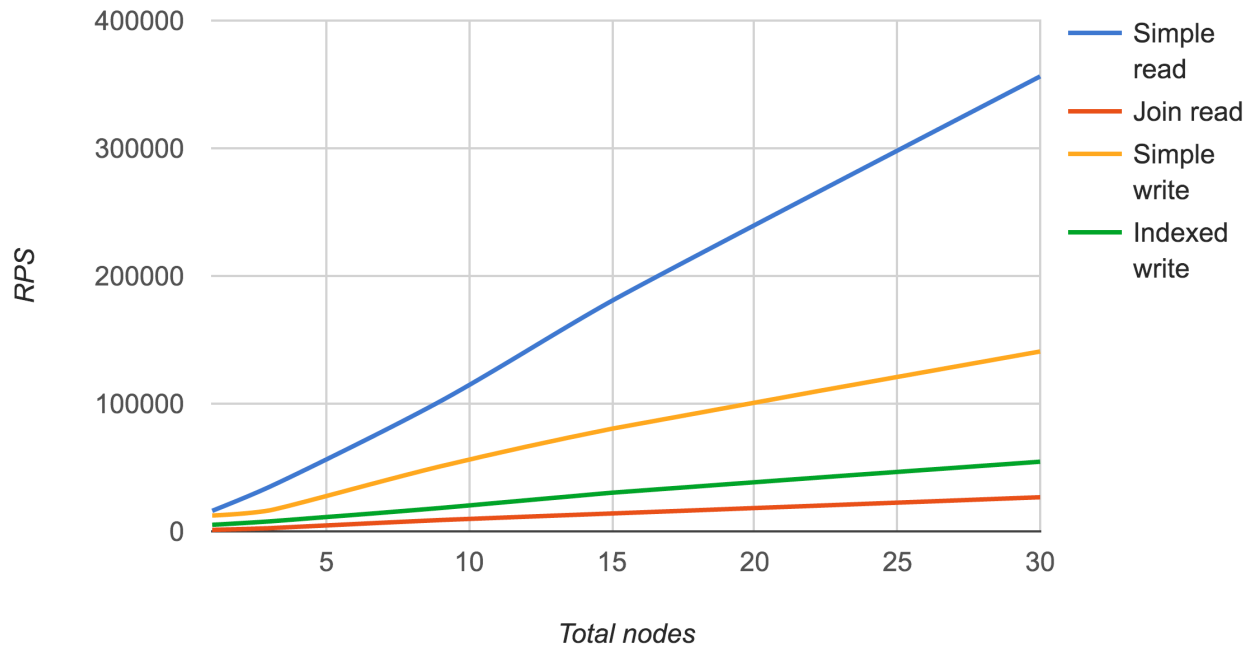


Figure 1: Throughput benchmark showing linear scalability for typical query patterns

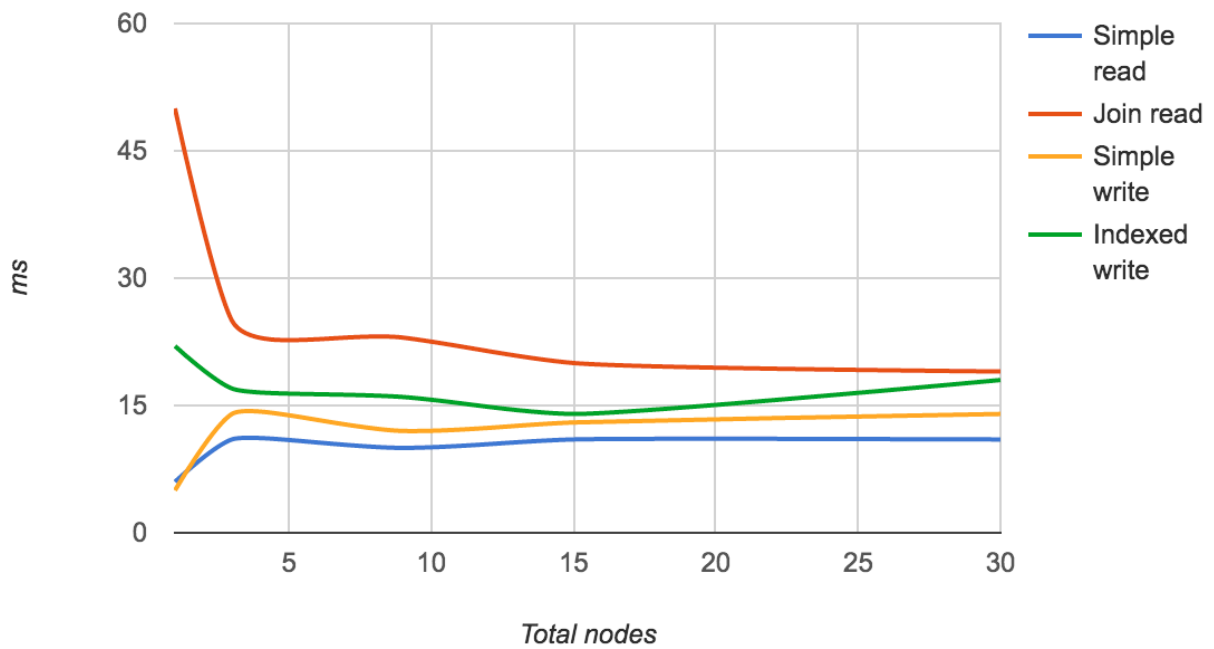


Figure 2: 50th-percentile latency benchmark showing the benefits of distributed computation within a single datacenter

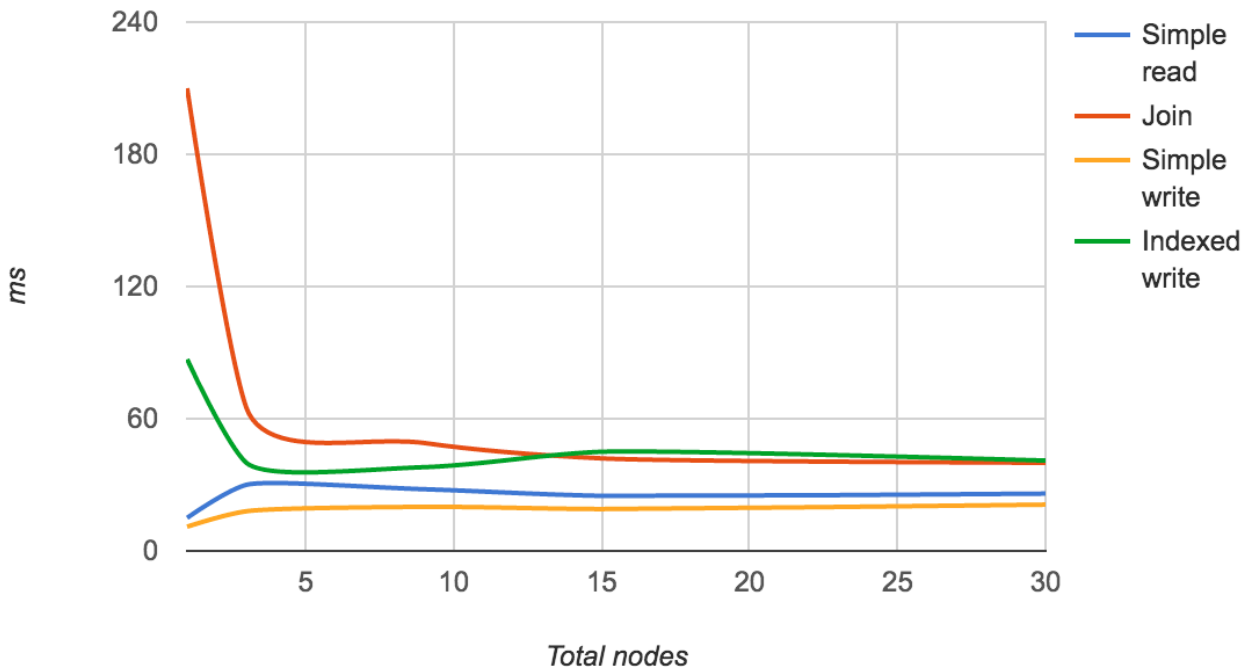


Figure 3: 95th-percentile latency benchmark showing the benefits of distributed computation within a single datacenter

Performance

A scalable system is only useful if it is able to take advantage of the underlying physical hardware adequately. Like scalability, performance is an end-to-end problem, and not easily retrofit onto legacy systems, or even recent systems designed without performance in mind.

Benchmarks on commodity hardware demonstrate that FaunaDB’s performance compares very favorably to other distributed databases. These graphs are based on internal data; in order to focus on relative scalability rather than absolute numbers, we will not specify the underlying hardware configuration here.

... writes respond well under contention and never interfere with reads or non-overlapping writes.

Throughput for all operations scales linearly; unlike legacy SQL systems, there are no thresholds in overall dataset size that trigger planning heuristic changes and lead to unexpected performance faults. Additionally, writes respond well under contention and never interfere with reads or non-overlapping writes.

Predicate pushdown is extremely effective at parallelizing complex queries, and per-query latency improves with larger cluster sizes (both for committing writes, for observing write effects, and for performing compute on result sets). For historical reasons, these capabilities are rarely found in other distributed database systems.

Durability

FaunaDB's local storage engine is implemented as a compressed **log-structured merge tree** [10], similar to the primary storage engine in Google Bigtable [3]. LSM storage engines are well-suited to both magnetic drives and SSDs.

Writes

Transactions are committed in batches to the **global transaction log** (the equivalent of a distributed write-ahead log, discussed below). Replicas tail the log and apply relevant write effects atomically in bulk. This model maintains very high throughput with log-structured merge trees and avoids the need to accumulate and sort incoming write effects in a memory table. Because FaunaDB's temporal data model is composed of immutable versions, there are no synchronous overwrites.

Reads

The on-disk levels act as a series of overlays that logically combine into the local dataset. This means that reads may need to check multiple levels to find the complete result set they are interested in. This degrades read performance compared to a data structure that is modified in place, like a B+ tree.

Performance

A variety of other optimizations such as local index structures are kept in memory to minimize the need to seek through each level file itself to find if a data item is present.

The level files themselves are compressed on disk with the LZ4 algorithm to reduce disk and IO usage. This also improves the performance of the filesystem cache. Since level files are immutable, compression only occurs once per level file, minimizing the performance impact.

In order to mitigate the latency impact of multi-level reads, a local background process called **compaction** is triggered when the number of levels exceeds a fixed size. Compaction performs an incremental merge-sort of the contents of a batch of level files and emits a new combined file. In the process, expired data is evicted, shrinking the on-disk storage usage.

FaunaDB's consistency model is designed to deliver strict serializability across multi-key transactions

Compaction can be performed asynchronously, but progress must be guaranteed over time or read performance will degrade. The compaction tasks are managed via the process scheduler in order to balance their resource requirements with the need to prioritize synchronous transactions.

Consistency

Strict serializability is widely recognized as the ideal data consistency model. It is easy for developers to reason about, minimizes application complexity, and reduces the total amount of data that needs to be stored, but often comes at a substantial performance cost, especially in distributed environments.

FaunaDB’s consistency model is designed to deliver **strict serializability** across multi-key transactions in a globally-distributed cluster without compromising availability, scalability, throughput, or read latency.

All read-write transactions are strictly serializable based on their position in a **global transaction log**. Read-only transactions are serializable. Database drivers maintain a high watermark of the log position of their last request—equivalent to a causal token—guaranteeing a monotonically advancing view of the global transaction order. Additionally, each datacenter uses a synchronization scheme to share the most recently applied log position among all **query coordinators**, in order to provide a consistent view in the presence of hidden coordination across client.

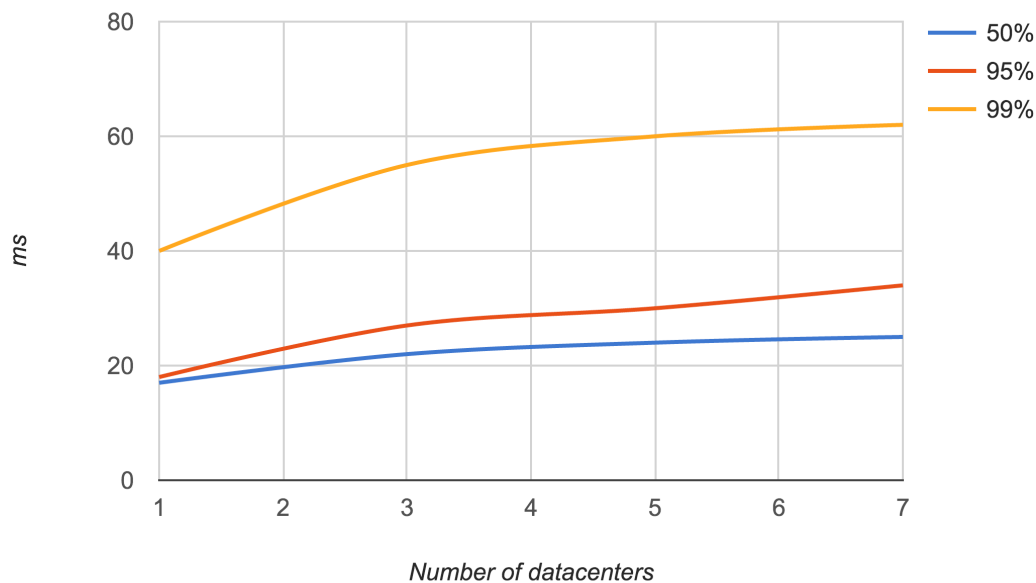


Figure 4: Latency benchmark showing asymptotic scalability of writes as additional datacenters are added

Finally, write transactions are restricted to a single logical database, but read-only transactions that recursively span multiple logical databases maintain the same **serializability guarantee** as single database read-only transactions if appropriate permissions have been assigned.

Tradeoffs

FaunaDB’s primary tradeoff is that write latency has a lower bound of the round-trip latency between the originating datacenter and a subset of the other datacenters. The scalability function is logarithmic; as additional datacenters

are added to improve availability, the marginal impact of each one on latency decreases.

This network latency is strictly worse than the latency exposed by eventually-consistent, distributed key/value stores, which do not attempt to guarantee global durability before acknowledging a commit, or SQL systems which rely on asynchronous replication and cannot serve consistent reads from replicas. Increased application parallelism can compensate for the latency increase on individual writes with no impact on overall throughput.

Additionally, unlike strongly-consistent systems like Google Spanner [6], FaunaDB has no window of ambiguity for which all reads may potentially be delayed—a property that becomes untenable on public networks at global scale.

Finally, session transactions are disallowed in FaunaDB. While session transactions are a convenient feature for some applications, they are also a frequent source of performance problems, and can be simulated via snapshot reads and compare-and-swap functionality if necessary.

FaunaDB's primary trade-off is that write latency has a lower bound of the round-trip latency between the originating datacenter and a subset of the other datacenters.

Implementation

Transaction resolution in FaunaDB is inspired by the Calvin protocol [1], backed by a highly optimized version of Raft [2]. Raft serves to replicate a distributed transaction log, while Calvin manages transaction resolution across multiple data partitions.

The globally replicated transaction log maintains an order of all transactions within a logical database. The log is processed as an ordered series of batches called **epochs**. The typical epoch window in FaunaDB is 10 milliseconds, which serves to allow the cluster to parallelize transaction application with minimal impact on observed latency.

When a transaction is submitted to a **query coordinator**, the coordinator speculatively executes the transaction at the latest known log timestamp to discover read and write intents. If the transaction includes writes, it then is forwarded to the nearest **log replica**, which records it as a part of the next epoch, as agreed upon by consensus with the other replicas.

At this point, all required cross-datacenter communication has occurred. The order of transactions within the epoch and with respect to the transaction log is resolved, the transaction is stamped with a logical timestamp reflecting its position within the log, and each datacenter proceeds to independently and deterministically resolve transaction effects.

The transaction is then forwarded to each local **data replica**, as determined by its read and write intents. Each data replica receives only the subset of transactions in the epoch that involve reads or writes of data in its **partitions**,

and processes them in the pre-determined order. Each data replica will block on reads for values it does not own, and forwards reads to all other involved partitions for those it does. Once it receives all read values for the transaction, it will resolve the transaction and apply any local writes. If any preconditions of the original speculative execution fail (e.g. a read dependent on a value that has changed is no longer covered by the set of read intents), the transaction will be aborted.

Because the transaction log maintains a global order, and data nodes are aware of their own position in the log, reads can be consistently served from the local datacenter at all times, and the causal order of two transactions can always be determined by the ordering of their respective log positions.

Although transaction throughput in Calvin-based systems is theoretically constrained by the degree of contention among nodes within each epoch, in FaunaDB resolution context is partitioned by logical database, so aggregate transaction throughput is unbounded. The theoretical upper throughput bound per logical database is approximately one million transactions per second.

Resiliency

The transaction processing pipeline is tolerant of node failure and latency at each step.

- If the coordinating node cannot communicate with the local log replica, it may safely forward its transaction to another log replica.
- If a data replica does not receive an epoch batch from the local log replica in a timely manner, it may retrieve the epoch batch from another log replica.
- If, during transaction application a data replica does not receive part of the transaction's reads from other partitions, it may safely read the missing values at the specific log position from other replicas of the failed partition.

09 QUALITY OF SERVICE

Quality of Service

Adaptability requires both responsiveness and isolation. Noisy neighbor problems and static provisioning constraints have doomed legacy databases to inefficient and operationally-intensive hardware isolation only. Even if virtualized, these systems are cumbersome and aggregate resource utilization is typically low.

In order to effectively respond to rapidly changing workloads, FaunaDB implements a **process scheduler** that dynamically allocates resources and enforces quality of service. The scheduler is implemented as a recursive series of work queues that mirrors the logical database hierarchy in the FaunaDB cluster. Individual transactions are

slotted into queues first by their **execution context** (synchronous or asynchronous), and secondarily by their **priority context**, which is either the priority of their logical database, or the priority of their access key if it has one.

Execution proceeds via cooperative multitasking. Transactions are selected for execution according to a recursive, weighted fair queuing algorithm [8] and scheduled onto native threads. FaunaDB's query planner evaluates transactions as a series of interleaved and potentially parallelizable compute and IO stages, and guarantees that execution will always yield at predictable and granular barriers (for example, loop iteration). This restricts the complexity of continuations and lets the executor context switch and re-enter scheduling each time a predictable quantity of resources is consumed from each IO or compute execution thread, without requiring a complex and non-portable pre-emptive multitasking scheme.

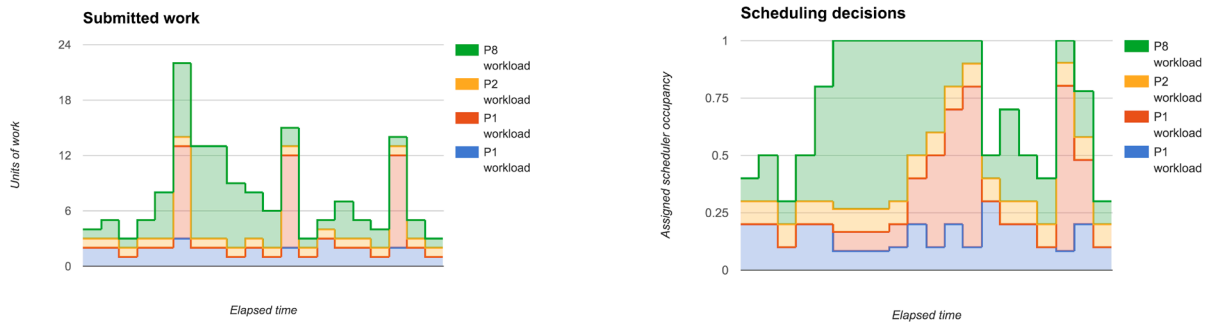


Figure 5: Idealized process scheduler example showing priority-based preemption

The overall impact is that workloads can be recursively ordered by business priority, and low priority tasks can burst into whatever idle capacity remains in the cluster, dramatically improving aggregate utilization. The more diverse applications, datasets, and workloads are hosted in a single FaunaDB cluster, the better the price/performance becomes compared to a traditional, statically provisioned siloed data architecture.

In order to effectively respond to rapidly changing workloads, FaunaDB implements a process scheduler that dynamically allocates resources and enforces quality of service.

10 BACKGROUND TASKS

Background Tasks

Background tasks such as schema changes, anti-entropy checks, or even user-submitted long-running queries, are a frequent source of availability problems in legacy database systems.

In order to mitigate this, background work in FaunaDB is managed internally by a journaled, topology-aware **task scheduler**, similar to Apache Hadoop YARN [5].

Implementation

FaunaDB's implementation is general purpose. Tasks may be limited to one instance across a cluster or per database, or be assigned a specific data range (in which case, the executing node will be one that is a data replica for that range). The execution state of each task is persisted in a consistent metadata store, meaning that scheduled tasks may run in a node agnostic manner: if a node fails or leaves the cluster, its tasks are automatically reassigned to other valid nodes and restarted or resumed.

Task execution throughput is controlled by FaunaDB's resource scheduler. General background work not associated with any tenant is run at low priority, allowing the task to proceed as idle resources allow and eliminate the impact of background tasks on synchronous requests.

11 OPERATIONS

Operations

FaunaDB is designed to be straightforward to operate. The cluster management infrastructure reuses the consistency mechanism and process scheduler to guarantee that the cluster is always in a coherent state, and that work generated by operational changes does not adversely affect production workloads.

Topological Changes

Cluster topology changes include:

- Adding, removing, or replacing a physical node in the cluster.
- Adding or removing a datacenter.
- Changing the replication configuration of a logical database.

During cluster transition states, node failures and other interruptions do not affect cluster availability. If the node running the supervisor process fails, then its lease on the supervisor role will expire and another node will assume the role. All incremental steps within each transition process are idempotent and may be safely restarted or reverted.

Other Maintenance

Other maintenance tasks include:

- Taking logical and storage-format backups.
- Performing anti-entropy checks.
- Upgrading the on-disk storage format.

These do not require a state machine transition, but still rely on the process scheduler to avoid impacting production traffic.

12 SUCCESS PATTERNS WITH FAUNADB

Success Patterns with FaunaDB

A variety of success patterns with FaunaDB have emerged from customer usage:

- Product teams building new applications for mobile, web, or other fat clients can use FaunaDB as a distributed application backend.
- Developers of software-as-a-service products can use FaunaDB's quality-of-service and multi-tenancy capabilities to isolate and improve their customers' experience.
- Enterprises with data in legacy silos can use FaunaDB as a scalable data fabric to integrate them.
- Multi-product enterprises can realize significant utilization improvements by consolidating applications into a shared FaunaDB cluster.
- Companies with data hosted in a single geographic location can use FaunaDB to distribute their data globally.
- Enterprises in transition to the cloud can use FaunaDB to unify on-premises and cloud data.
- Operators struggling to manage cross-team/cross-workload access to shared data can use FaunaDB to efficiently allocate resources and improve availability.

As FaunaDB matures we anticipate customers will experience compounding success on their investment. Unlike multi-system database deployments, each additional FaunaDB feature or client application incurs no additional cost in operational complexity, infrastructure footprint, availability, or scalability.

13 CONCLUSION

Conclusion

This paper gives an overview of the design and architecture of FaunaDB, a quality-of-service managed, multi-tenant, globally distributed database with a transactional, object-relational query model.

FaunaDB is our effort to build an adaptive operational database for modern enterprises. Our work is based on the state-of-the-art research and industry progress in databases and operating systems, as well as a wealth of hands-on operational experience.

Although we have attempted to be comprehensive, we have left out many details, and many details remain to be finalized still. We look forward to working with the developer community, academia, and current and future customers to realize a shared vision of what modern operational database management should be.

14 CONTACT

Contact

We're working with developers, customers, and partners at all levels of growth. We'd love to help you get started with FaunaDB as well.

For more information and to discuss working together, please email our team at priority@fauna.com, or visit us on the web at fauna.com/request-info.

References

- [1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97-137.
- [2] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 1-12.
- [3] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14)*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, Berkeley, CA, USA, 305-320.
- [4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.
- [5] 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. 2013. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages.
- [6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (August 2013), 22 pages.
- [7] David Gelernter. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (January 1985), 80-112.
- [8] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and simulation of a fair queueing algorithm. In *Symposium proceedings on Communications architectures & protocols (SIGCOMM '89)*. ACM, New York, NY, USA, 1-12.
- [9] E. Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* 45, 2, 23-29.
- [10] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351-385.

