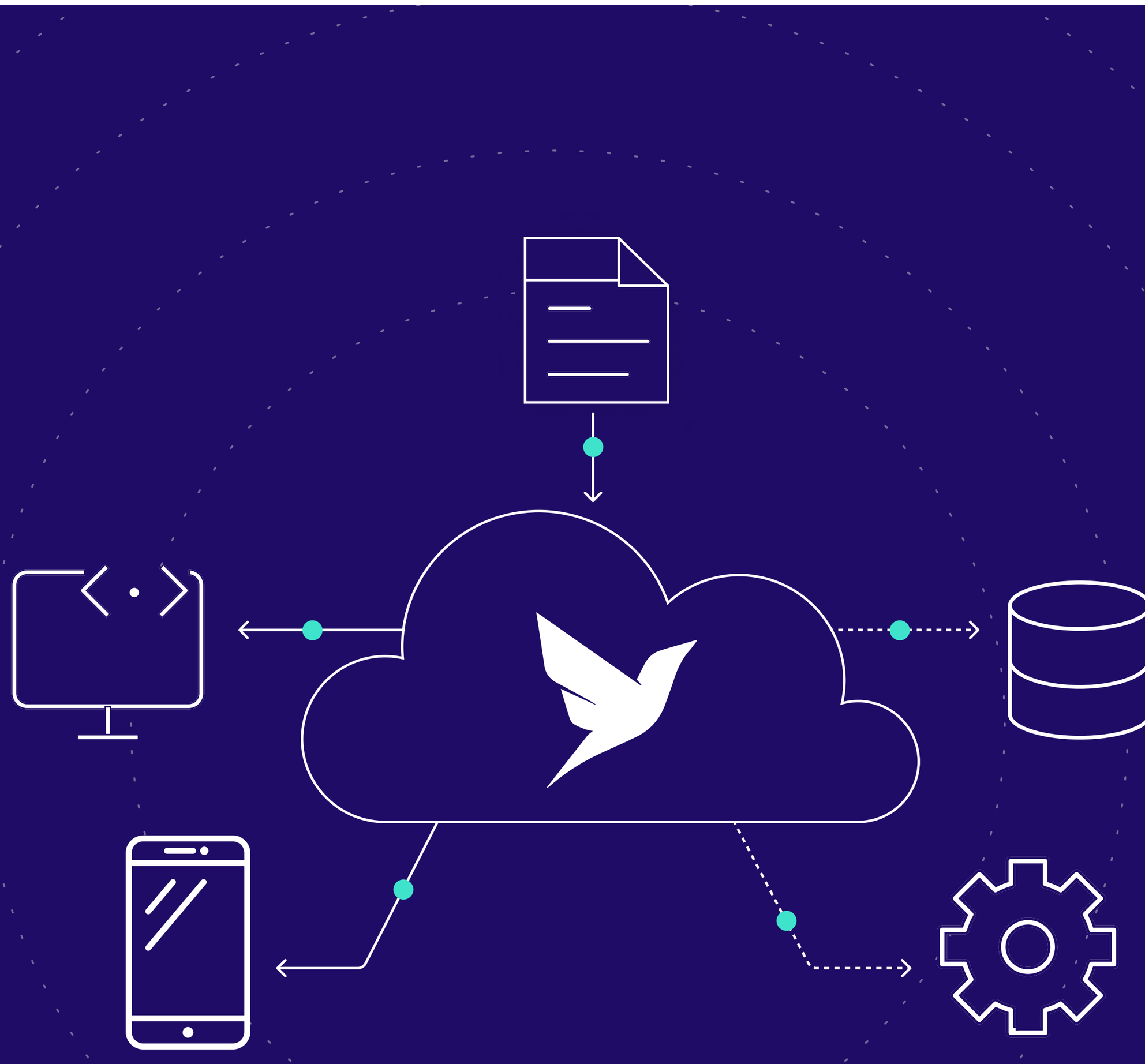
fauna

Architectural Overview



Background

The history of databases can be characterized as long periods of slow evolution punctuated by a few short bursts in which dramatically different databases were brought to market.

The business data era was arguably the first such period, marked by the birth of relational databases that stored data in tables, allowed complex joins across tables, and exposed data via the Structured Query Language (SQL). These relational database management systems are still widely used today, although they were built when the primary data consumer was an analyst sitting at a workstation. However, several issues with these systems have become obvious as end-users have supplanted data experts as the primary consumers of data. For example, SQL exposes data in structures that aren't compatible with modern software development practices, resulting in a proliferation of ORM libraries and an object-relational impedance mismatch. SQL itself is not type-safe, which leads to frequent security flaws, and its declarative programming model makes any specific query performance profile unknown and unpredictable. Finally, relational databases are typically deployed as self-managed or partially-managed instances or clusters that require operators to think about hosting, maintenance, and scaling.

Another period of rapid database innovation sprung up decades later with the rise of NoSQL databases, particularly the NoSQL subcategory of document databases. These offerings attempted to address the shortcomings of relational databases by storing data in formats familiar to developers and promising limited operations and infinite scale while sacrificing strong consistency guarantees in favor of eventual consistency. In practice, these offerings did not fulfill their full promise because they needed more support for complex access patterns (such as performing joins across different types of documents) that are frequently needed, and operations became more difficult at the scale of real-world workloads. Additionally, their less stringent consistency models pushed complex checks and edge-case handling into application code, creating a non-trivial burden for developers.

Developers that are building modern applications need the best of both of these worlds. They want the power and consistency of traditional relational systems along with the scalability, low management overhead, and ergonomics of document systems in a single package.

Overview

In this paper, we introduce Fauna, a distributed document-relational database delivered as an API. Fauna was built from the ground up to serve as the primary operational database for modern applications, bringing together the best from current document and relational databases along with many novel innovations in a single offering designed to meet the requirements of current applications. The following sections explore its unique properties, powerful data model, topology, architecture, and why it is well suited for today's applications.

World-class Developer Experience

Fundamentally, Fauna's architecture is entirely in service of providing a best-in-class experience for developers, giving them powerful and flexible ways to work with their data:

- The heart of Fauna is a document-relational data model that combines the flexibility and familiarity of JSON documents with the relationships and querying power of a traditional relational database.
- Data is exposed through the powerful Fauna Query Language (FQL), which is Turing complete, allowing any arbitrary business logic to run on top of the data.
- Transaction code can be stored as User Defined Functions (UDFs) for composability and reusability.

Distributed by Default

Compute workloads are increasingly moving towards the edge, making it important for applications to access data across wide geographies with low latency. Fauna makes this possible by replicating data across availability zones or regions:

- Fauna's architecture is inspired by the Calvin transaction protocol, which allows Fauna to deliver strictly serialized transactions across a designated geography or around the globe.
- Fauna can be deployed in a single region or a region group. Data is automatically replicated across zones or regions and can be accessed from the nearest region with the highest consistency guarantee.

Delivered as an API

Software is eating the world, and all companies are becoming software companies, making it more critical than ever for enterprises to move quickly to compete. Fauna empowers enterprises by removing undifferentiated heavy lifting and allowing them to focus on their application.

- Fauna exposes data through a single global API endpoint. Developers are not required to manage connections or other implementation details. Further, the API endpoint is natively supported by edge computing providers.
- An intelligent routing layer at Fauna's edge directs requests sent to the API to the fastest region where requests can be served without any client configuration.

Data Model

Fauna implements a document-relational data model that is a superset of the relational and document paradigms. The top-level container in Fauna is a database that can hold collections of documents or other nested databases. Arbitrarily complex transaction code can be passed to Fauna directly from the client or stored in a User Defined Function (UDF), which can be invoked from any transaction.

Document-relational

Fauna offers the benefits of both the relational and document models while eliminating many drawbacks. The fundamental data building block in Fauna is a semi-structured document, which can include scalar types, arrays, and references to other documents. Documents are schema-less by default, but partial or complete schemas can be specified and enforced at the collection level. Documents can contain references to other documents, allowing developers to execute joins across collections.

Unlike traditional relational databases, the document-relational model provides a high degree of flexibility in crafting query responses. A SQL query response is always a two-dimensional set of tuples that conform to a schema. Fauna query responses are effectively composable document structures; developers can craft results to correspond exactly to what the consuming application needs, building arbitrarily complex response structures by stitching together simpler subqueries. This level of FQL query and response expressiveness eliminates the issue of object-relational impedance mismatch and, most importantly, comes with no consistency or scalability tradeoffs.

Indexes

Fauna supports global secondary indexes, which are consistently maintained as part of transaction processing. An index is defined with zero or more lookup terms, which provide a way to partition the index into multiple physical components, including zero or more covered values that define its natural sort order.

Like indexes in a relational database, Fauna indexes enforce uniqueness constraints and accelerate queries by reducing the cost of selection and ordering where the query is aligned with the index structure. Indexes also allow queries to avoid fetching canonical document versions where read fields include the index's normative terms and values.

Unlike a relational database, the Fauna query execution engine does not implicitly use indexes based on query analysis. Instead, the developer must explicitly query an index by name. This provides the developer with a higher degree of control over query execution and a greater level of predictability appropriate to an operational database. The developer can avoid fighting the optimizer to choose the correct index. It is impossible for the query optimizer to unexpectedly adopt a different execution strategy with an exceptionally different performance profile based on subtle changes in dataset properties over time.

User-defined Functions (UDFs)

Transaction code meant to be reused across applications can be parameterized and stored as a UDF (similar to a SQL stored procedure) to abstract logic from applications that are difficult to upgrade in place. UDFs can be configured to execute with a role different from the general request context, ensuring its code always executes with a specific set of permissions. UDFs are versioned and can be rolled back if needed.

Schema

Fauna introduces a sophisticated type system designed to support gradual typing, offering flexibility in the earliest phases of development and strict enforcement as applications mature and scale. The type system extends to UDFs and computed fields, and a rich migration system makes schema changes transactional and ensures that all types adhere to the schema at every point in time. Like traditional Relational Databases, Fauna's type system removes the need to reason about edge cases associated with mismatched types, but unlike Relational Databases, it does not come with the baggage of complex schema migrations that are error prone, require downtime, and are difficult to reason about.

Fauna supports schema evolution with zero downtime, allowing changes such as adding new fields, changing data types, or enforcing new rules without interrupting application performance or availability. By supporting versioned migrations and dynamic schema validation, Fauna ensures that your application can evolve alongside your data model without compromising data consistency or scalability.

Interfaces

General-purpose programming languages lack domain-specific functionality to query and manipulate data out of the box, so every database must implement its own domain-specific language (DSL) for data access. Some databases expose the data model to developers using an embedded DSL shipped as a library, borrowing syntax from a host language. Other databases use a general DSL, such as the Structured Query Language (SQL), which stands independently. To meet the needs of modern application developers, Fauna implements a new query language as a general DSL that is intuitive, flexible, and type-safe.

Fauna Query Language (FQL)

FQL is an elegant query language that features TypeScript-inspired syntax, but specifically tailored towards issuing concise relational queries. It brings together the combined power of relational querying and the flexibility of documents in a single, unified model. The language natively supports relational features such as foreign keys, views, and joins and combines the ability to express declarative queries and functional business logic in transactions that are strongly consistent across geographic regions. It is statically typed by default, and it features an indexing system that allows developers to take an iterative approach to query optimization. All considered, FQL brings together a unique set of characteristics that make it the ideal language for modern operational applications.

Developers can execute FQL queries against their data manually via the Fauna dashboard or CLI, or

programmatically by using one of Fauna’s lightweight, open source driver implementations in TypeScript, Go, and Python. The drivers bring FQL into each language in a way that is natural and idiomatic, providing out-of-the-box capabilities such as pagination, templating, retries, throttling, and error handling. In addition, the underlying HTTP API is simple enough to call directly within resource-constrained application environments such as IoT or via no-code platforms. This new wire protocol and HTTP API have been extensively documented, making it possible for us along with Fauna's community to bring support for FQL to more host languages over time.

Topology

Cloud providers typically offer services in regions that reside in a single city and contain multiple availability zones. Regions provide a blast radius containment mechanism should service availability issues occur. They also allow customers to understand the physical location of data so they can reason about the latency characteristics of accessing data from other physical locations. For some use cases, a regional database is sufficient. Still, the rise of edge computing has increased the criticality of pushing data out of a single location and making it available for fast access from a broad physical footprint for specific types of workloads.

Regions and Region Groups

To meet the needs of those modern applications that run at the edge, Fauna replicates data across availability zones in a region or regions in a region group. Each Fauna deployment corresponds to a geographic area that may be a single region such as Virginia or Portland, a region group that corresponds to a broader geography such as the United States or the European Union, or even the entire globe. Replication across zones or regions is fast, providing several benefits, including fast local reads and the ability to serve requests in the face of a complete zonal or regional failure.

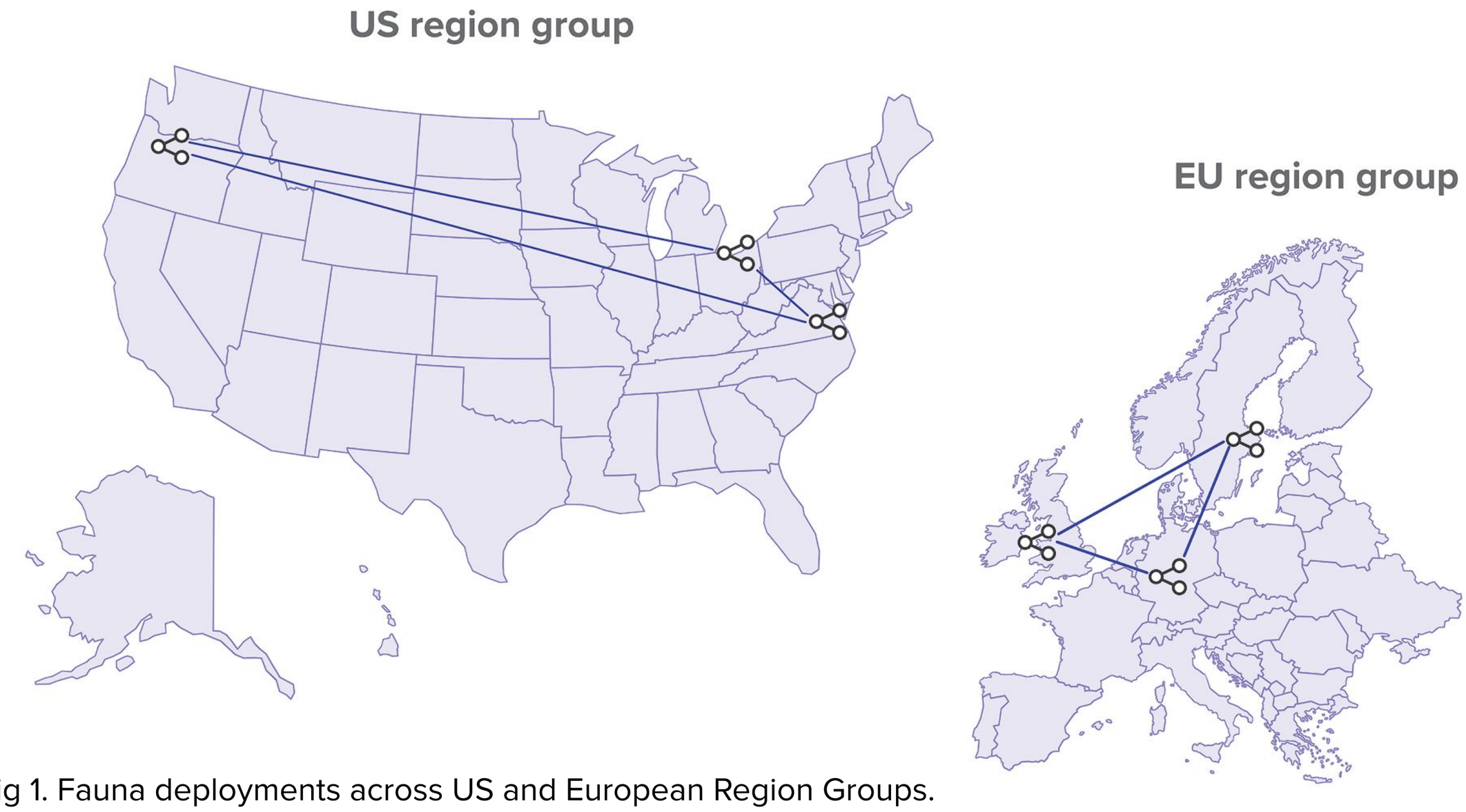


Fig 1. Fauna deployments across US and European Region Groups.

Overall, Fauna deployments are optimized for both external latencies and internal latencies. External latencies are reduced by allowing requests to quickly be routed from edge servers directly to the highly efficient, internal node network into which Fauna is deployed. Internal latencies are optimized by several factors, including distributing writes in transaction log batches which retain all ACID guarantees. Read requests can be served regardless of whether other zones or regions are available. Write requests require communication between a majority of zones or regions in the deployment to succeed, which can be scaled with added regions in a region group.

Public regions and region groups are multi-tenant, which means that many customers store and access data on shared hardware, with multiple layers of protections in place to ensure that tenants cannot access data that belongs to another tenant. Current public region groups span geographic areas that typically conform to local compliance regulations and data residency requirements.

Private Fauna deployments can be created within a single cloud provider region or a region group. They are single-tenant, providing VM-level isolation to data for a specific customer, allowing the customer to specify the regional footprint and set of cloud providers where they want their data to reside. To give a sense of scale, a single tenant has successfully leveraged a private Fauna region group spanning five regions and consisting of more than 50 hosts to field over 30k requests per second to power their customer metadata store – numbers that are relatively large but far below the theoretical scaling limits of a single deployment.

Environments

Fauna is a managed service; all resources associated with production regions or region groups run in Fauna-managed cloud provider accounts. Resources in both public and private deployments are modeled using an Infrastructure as Code (IaC) service that offers providers for AWS, GCP, and Azure. This means infrastructure for new regions or region groups can be provisioned with a simple source code change. Cloud provider-specific dependencies are limited to compute, blob storage, and VPC/routing resources, which means that Fauna can be deployed into additional environments in the future, such as Cloudflare and Fastly Points of Presence (PoPs).

Core database services are deployed to IaC-provisioned Virtual Machines (VMs) in each cloud environment. Each region or region group is homed in its own cloud provider account, minimizing the impact of an issue in any specific account. Ancillary containerized services are deployed to environment-specific clusters using an off-the-shelf container orchestration service.

Distributed Transaction Engine (DTE) Architecture

The heart of Fauna is a Distributed Transaction Engine based on the Calvin transaction protocol. Calvin is a transaction scheduling and data replication layer that uses a deterministic ordering guarantee to significantly reduce contention costs associated with distributed transactions. Fauna uses Calvin to guarantee that every

replica sees the same log of transactions and guarantees not only a final state equivalent to executing the transactions in this log one-by-one, but also a final state equivalent to every other replica.

DTE Logical Layers

Fauna's architecture separates core database services into logical layers:

- Routing
- Query Coordination
- Transaction Logging
- Data Storage

Within each zone in a regional deployment or region in a region group deployment, routing, query coordinator, transaction log, and data storage nodes understand the full topology of the deployment and can forward requests to nodes in other availability zones or regions if a local node is not responding.

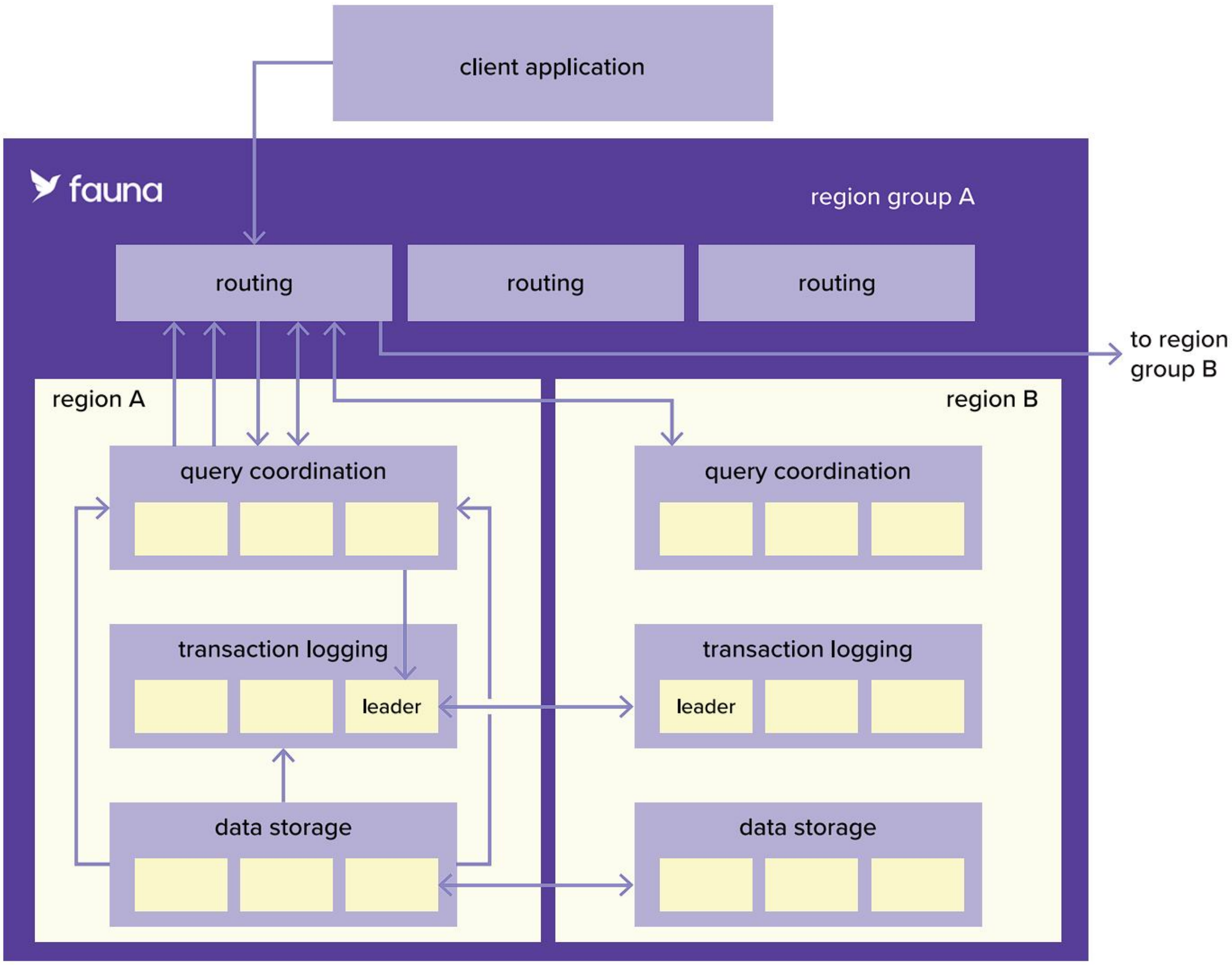


Fig. 2. Fauna's core architectural service layers.

Transactions in the system start with a request sent by the client, which is handled by one or more logical layers in sequential order.

Routing

Fauna's managed service includes a sophisticated routing layer that handles requests at ingress and optimally routes them to the correct region. The first step in routing relies on a highly available DNS service offered by a public cloud provider, with latency-based forwarding to routing nodes in the nearest zone or region.

When a customer wants to read or write data,

1. The customer sends an HTTP request to the IP address associated with one of the local routing nodes in DNS. At ingress, the request is handled by an open-source ingress controller that runs the request through a series of plug-ins that can block a request based on properties such as the IP address it is coming from. Plug-ins also have access to information about recent traffic patterns and can throttle requests in the event of a large request burst from an IP address or if read, write, or compute operations for a specific database or key exceed provisioned capacity or defensive rate limits.
2. After passing through plug-ins, requests are routed using a service mesh. Requests to the database are handled by a component of the service mesh that keeps a mapping between database keys and the region or region group where the associated database lives. Suppose the component has yet to see the key associated with the request before. In that case, it sends a multicast message to compute coordinators in each region or region group looking for a successful response.
3. When a successful response is received, the service caches the location of the database for the given key. It forwards the request to a query coordinator in the closest region in the appropriate region or region group.

This machinery allows customers to send all database requests to a single API endpoint regardless of where their data resides without updating endpoint configuration or sacrificing performance – a much simpler model than other managed databases currently offer.

Query Coordination

The Calvin transaction protocol removes the need for per-transaction locks by leveraging a pre-computation step that computes the inputs and effects of each transaction ahead of time. Fauna handles this pre-computation on stateless query coordinators that scale horizontally.

When a query coordinator receives a request,

4. The coordinator selects a snapshot time for the request based on either the time specified in the request or the current time if a time was not specified.
5. The coordinator inspects the transaction associated with the request and optimistically executes the

transaction without committing writes, communicating with data storage nodes to fetch the values for any necessary reads at the transaction time. The output of the execution is a flattened set of reads made as part of the transaction and a set of writes to be committed if there is no contention. The reads must be checked for contention.

6. If the flattened transaction contains no writes, the coordinator returns the read results to the client.
7. If the flattened transaction contains writes, the coordinator passes the transaction to a transaction log node to be sequenced and committed. If the transaction log or data storage nodes are unavailable in the local zone or region, the coordinator communicates with the appropriate nodes in other zones or regions.

Transaction Log

The transaction log is the only system layer where cross-replica coordination is necessary. The log functions as a global write-ahead log, split into multiple segments that span replicas. The number of segments can be increased to scale log throughput. Each segment runs an optimized version of the Raft consensus algorithm to elect a leader for the segment. A leader for each log segment is elected from all healthy nodes, and non-leader nodes forward transactions to the nearest leader. A new leader election is triggered if the current leader becomes unavailable.

When a transaction log node receives a flattened transaction,

8. The node passes the transaction to the log leader in its segment. The segment leader periodically assembles received transactions into batches to commit based on a configurable interval called the epoch interval.
9. The leader communicates with other leaders in the Raft ring to agree on the full set of transactions in the epoch.

After the batch has been replicated using Raft, its transactions are considered optimistically committed, although their write effects have not yet been applied. When all log segments have committed their respective batch transactions for a given epoch, the epoch's transactions are available for downstream processing by data storage nodes.

It should be noted that real-time global clock synchronization in Fauna is not required to guarantee correctness. Log nodes, which are the only ones generating epochs, are the only ones where clock synchronization occurs, and epochs are thus generated at about the same time. Based on epoch other, a timestamp is applied to every transaction that reflects its real commit time, within milliseconds of real-time, and its logical, strictly serializable order with respect to other transactions.

Based on epoch order, the system can assign a timestamp to each transaction, which both reflects its real commit time, within milliseconds of real-time, and its logical, strictly serializable order with respect to other transactions. For example, a developer might determine that the last transaction that wrote to document A

logically happened before the last update to document B based on the fact that the update time of A is the lesser of the two times.

Data Storage

Data storage nodes in each replica are assigned ranges of keys for which they are responsible, with the full key space represented. All data is stored in each zone or region, and every document is redundantly stored on at least three nodes.

For each new transaction,

10. Each storage node maintains a persistent connection to each local log node on which it listens for newly committed transactions for the ranges of keys it covers.
11. The storage node validates that none of the values read during transaction execution have changed between the execution snapshot time and the final transaction commit time, communicating with its peers if needed to get the state of values it does not cover.
12. If there is no conflict among the read values, the storage node updates the values it covers and informs the query coordinator of the transaction's success. If there is a conflict, the storage node drops the transaction's writes and informs the coordinator of the failure. The set of checks on the transaction read set at the transaction timestamp is deterministic, so all storage nodes either apply or fail every transaction in the log.
13. Upon receiving the transaction commit result from at least one storage node, the coordinator notifies the client of the result.

Written documents in applied transactions are not overwritten. Instead, a new document version at the current transaction timestamp is inserted into the document history, either as a create, update, or delete event. Retention policies for versions are configurable for each database.

Other optimizations, such as local index structures, are kept in memory to minimize the need to seek through each level file to determine if a data item is present. Files on disk are sorted string tables, which are compressed using the LZ4 algorithm to reduce disk and I/O usage. This also improves the performance of the file system cache. Because files are immutable, compression only occurs once per level file, which minimizes the performance impact.

Files are stored in logical levels, and a leveled compaction strategy ensures that files are compacted when the number of levels exceeds a fixed size. Compaction performs an incremental merge-sort of the contents of a level file batch and emits a new combined file. In the process, expired and deleted data is evicted, shrinking on-disk storage usage and ensuring high-performance reads.

Fauna’s storage system supports temporal queries, which means that all transactions can be executed consistently at any point in the past. This is useful for auditing, rollback, cache coherency, and synchronization to other systems, and forms a fundamental part of the Fauna 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.

The storage system also facilitates event streaming, which allows customers to subscribe to notifications when a document or a collection is updated. Streaming requires that customers keep a connection to Fauna open and allows them to take action in the client as their data changes.

Task Scheduler

Background tasks such as index builds are a frequent source of availability problems in legacy database systems. To mitigate this, Fauna background work is managed internally by a journaled, topology-aware task scheduler, similar to Hadoop YARN.

The Fauna task system implementation is general-purpose and can be extended to orchestrate new types of tasks that may need to operate on data nodes. Tasks can be limited to a database, zone, or region, or can be assigned a specific data range. The execution state of each task is persisted in a consistent metadata store, meaning that scheduled tasks can run node-agnostic if a node fails. Failed node tasks are automatically reassigned to other valid nodes and restarted or resumed.

Ancillary Services

Fauna deployments include additional services that run at the regional, region group, environment, or global level. For example, the Fauna dashboard is a React web application that operates at the environment level. It is deployed to Vercel and lets customers visually interact with their accounts and data. Separate containerized services at the region or region group level are used for user authentication/authorization, metrics and billing data aggregation, database backup/restore capabilities, and other functionality.

System Properties

Fauna is designed to be horizontally and vertically scalable, self-coordinating, and without a single point of failure. Any query coordinator in any region can receive any request, and coordinator nodes can communicate with log and data nodes in any other region.

Scalability

The Fauna compute coordination layer is stateless, so nodes can be scaled both vertically and horizontally at any time. In the future, compute nodes will leverage reactive autoscaling or run on serverless compute fabrics.

The transaction log layer is split into segments, and throughput can be increased by increasing the number of log segments. However, coordination across segments is still required because each data storage node must receive a batch from each log segment before it can start to apply transactions in the epoch. The theoretical upper bound of log throughput is higher than any real-world transactional workload today.

Finally, the logical data layout in the data storage layer is partitioned across all nodes within the replica. Documents, including their history, are partitioned by primary key, while indexes are partitioned by lookup term.

Both documents and indexes scale linearly for reads and writes, regardless of their cardinality or the number of nodes in the deployment. This makes it possible for nodes to be scaled vertically or horizontally at any time. 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, the index can be configured to partition individual terms across multiple ranges and perform a partial scatter-gather query-on-read, similar to a search system.

Availability

While no distributed system can guarantee total consistency and total availability at the same time, Fauna seeks to provide the optimal tradeoff between the two. Fauna is technically a CP system according to the criteria put forth in the CAP Theorem, guaranteeing consistency across the system at the cost of availability in the event of a network partition. But in today's world, network partitions are rare enough in practice that the system can provide many 9s of availability.

Fauna is resilient to many types of faults that affect availability in other systems. In particular, Fauna is not vulnerable to a single point of failure, including at the zone or region level. For example, Fauna can tolerate temporary or permanent node unavailability, increased node latency, or a network partition that isolates a zone or region.

Durability

The Fauna local storage engine is implemented as a compressed log-structured merge (LSM) tree, similar to the primary storage engine in Bigtable. LSM storage engines are well-suited to both magnetic drives and SSDs.

Transactions are committed in batches to the global transaction log. Replicas tail the log and apply relevant write effects atomically in bulk. This model maintains a very high throughput with log-structured merge trees and avoids the need to accumulate and sort incoming write effects in a memory table. Atomicity is maintained, and data is preserved, notwithstanding the loss of nodes and even entire replicas. Because the Fauna temporal data model is composed of immutable versions, there are no problematic synchronous overwrites.

Performance

Fauna was designed to handle the performance requirements of demanding, highly responsive applications. This is achieved by replicating data within or across regions to bring it closer to the end user and by optimally routing requests from ingress to the data. Requests are routed to the closest zone or region where the data lives by default, even in the case of complete zonal or regional failure. Read requests can be served out of the closest zone or region, which often shaves tens or even hundreds of milliseconds of round trip latency off of a request, particularly in applications built using the client-serverless architectural

paradigm. Write requests must be replicated to a majority of log leaders in the log segment before a response can be sent. In practice, Fauna's public region groups typically exhibit single-digit millisecond latency for reads and double-digit millisecond latency for basic writes in addition to the cost of the round trip to get to the regions in the region group.

While Fauna hasn't published benchmark results, public region groups typically handle hundreds of thousands of requests per minute and burst to millions of requests per minute on current hardware. The query coordinator layer can be scaled rapidly to handle orders of magnitude more traffic based on demand. Public region groups also store many TB of data and storage nodes can be scaled to massively increase storage.

Consistency

Unlike most distributed databases, Fauna provides strict serializability, widely recognized as the ideal consistency model. Strict serializability is easy for developers to reason about, minimizes application complexity, and reduces the total amount of data that needs to be stored through normalization. The Fauna consistency model is designed to deliver strict serializability across multi-key transactions in a globally-distributed cluster without compromising scalability, throughput, or read latency.

All read-write transactions are strictly serializable based on their position in a global transaction log because the order reflects the real-time processing order. Read-only transactions are serializable with an additional consistent-prefix Read Your Own Writes (RYOW) guarantee, which is facilitated by the driver maintaining a high watermark of the most recent logical transaction time observed from any prior transaction. This most recent transaction time is passed to the service with subsequent requests, which acts as a minimum bound on the snapshot time used for the read.

Security

Fauna was designed to handle the performance requirements of demanding, highly responsive applications. Fauna provides several tools for customers to secure their data. Developers can restrict data access based on the user's identity or the combination of identity and accessed data attributes. Keys used for access always close over a specific logical database scope and cannot access parent databases in the recursive hierarchy, although they can optionally access child databases.

Identity

Fauna users and their customers can be identified with either built-in password authentication or by using a trusted service that delegates authentication to a third-party identity provider, such as Okta or Auth0. On identity confirmation, the actor receives a token, which can be used to perform additional requests that close over their identity and access context, similar to an access token in OAuth2. This permits untrusted mobile, web, or other fat clients to interact directly with the database and participate in the row-level access control system. Actors are identified by keys that can have various privilege levels configured via Role-Based Access Control (RBAC).

Attribute-based access control (ABAC)

Fauna extends RBAC to allow privileges to be dynamically determined based on any attribute of the actor attempting to access or modify data, any attribute of the data to be accessed or modified, or contextual information available during a transaction. Access rules can be delegated to a predicate function written in FQL, which supports arbitrarily complex logic.

Native Multi-Tenancy

Each Fauna database acts as a permission boundary, so calls with a key mapped to a database cannot access data or perform actions outside that database. This model can be used to secure customer data in modern client-serverless architectures in ways that other databases cannot. For example, developers can create per-customer databases with unique keys that can be passed to authenticated users and stored on end devices for subsequent database calls. This eliminates concerns about customers crossing tenant boundaries and accessing other customers' data.

Auditing and logging

All administrative and application transactions can be optionally logged. Additionally, the underlying temporal model document versions preserve the previous contents of all records within the configured retention periods. Although Fauna does not natively track data provenance, applications can tag every transaction with actor information and access that data historically as part of the document versions.

Conclusion

In this paper, we've provided an overview of Fauna, a distributed document-relational database delivered as an API to meet the requirements of modern application developers. Our work building and operating Fauna is based on state-of-the-art research and industry progress in databases and operating systems, in addition to a wealth of hands-on operational experience. The database itself has been battle tested – tens of thousands of customers have created hundreds of thousands of databases, stored hundreds of terabytes of data, and sent billions of requests to the service in production.

References

Object-relational impedance mismatch. https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

The Raft consensus algorithm. https://raft.github.io/

The LZ4 compression algorithm. https://github.com/lz4/lz4

Hadoop YARN. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

The CAP theorem. https://en.wikipedia.org/wiki/CAP_theorem

The log-structured merge (LSM) tree. https://en.wikipedia.org/wiki/CAP_theorem

Bigtable overview. https://cloud.google.com/bigtable/docs/overview

Role-based access control (RBAC). https://en.wikipedia.org/wiki/Role-based_access_control

Attribute-based access control (ABAC). https://en.wikipedia.org/wiki/Attribute-based_access_control