ChatGPT

# Why I Built My Own Serverless Graph Database

I set out to create a **serverless graph database** because no existing graph database met all of my requirements. Popular graph databases usually forced trade-offs or lacked certain features I considered essential. Below I outline the key requirements that drove me to build a custom solution, and why mainstream options didn't fit these needs.

## Serverless, On-Demand Operation

My first requirement was a **pure serverless architecture**. This means the database should spin up **only when in use** and scale down to zero when idle. I wanted to **pay only for actual usage**, with no constantly running servers: - **No idle compute:** There should be **no running compute instances when the data isn't being accessed.** If the database isn't actively handling a query or update, it uses **zero computing resources** (and incurs zero cost). - **Automatic scaling:** When a query does occur, the system should automatically scale the necessary compute resources to handle it, then turn off again. This on-demand model ensures efficiency and cost-effectiveness.

Most existing graph databases are not truly serverless – they typically require a persistent server or cluster running 24/7 (incurring cost even when idle). My solution needed to be fundamentally different by being *"off"* when not in use and eliminating any always-on infrastructure.

## Stateless Architecture (No Persistent Servers)

Related to serverless operation, I required a **stateless architecture** with **no long-lived server state**: - **Ephemeral compute:** There should be no dedicated server process or daemon that must be maintained. **Each request should stand up just enough compute to do the job and then terminate.** - **No maintenance overhead:** With nothing running in the background, there's **no server to patch, upgrade, or monitor** continuously. This reduces administrative burden and points of failure.

In practice, this means the graph database behaves more like a set of **on-demand functions**. Traditional graph databases didn't meet this need since they often rely on server processes or clusters that maintain state in memory. I wanted a system where **the only state is the data storage itself**, and all query processing is stateless and transient.

## Hyperlinked Data Across Multiple Formats

I needed the database to handle **highly hyperlinked data** across various formats in a very flexible way. Every piece of data stored should be linkable via unique IDs, enabling a true graph of heterogeneous items: - **Universal linking:** Any item in the database – whether it's a text record, a JSON document, a binary blob, etc. – should have an identifier and be able to link to any other item. **Everything is a node** (or can be treated as one) that can have connections. - **Multiple data formats:** The system must accommodate storing **files, JSON data, CSV datasets, binary files, nodes and edges lists**, or even entire databases as entries. In other words, it should accept a variety of content types as nodes in the graph. - **Flexible relationships:** Because of this hyperlinking, I can connect disparate data types. For example, a node representing a user could link to a JSON file of profile data, which could link to an

image file, which could link to another sub-database of activity logs. The graph database should not be limited to a single type of node or edge.

Most graph databases expect relatively uniform node types (e.g. all nodes are records in the graph). **My use case demanded a more liberal, web-like approach** where **anything can link to anything**, effectively blurring the line between database records and external data resources.

## Graph of Graphs (Nested Databases)

Extending the idea of hyperlinked data, my solution supports a **"graph of graphs"** – the ability to have **databases within databases**: - **Nested graphs:** I wanted to **link one graph database to another as if it were a node**. This creates a hierarchy or network of graphs. For example, one graph could reference another complete graph as part of its dataset. - **Adjustable scope:** This allows dynamically adjusting the scope of queries. I can navigate from one sub-graph to a related sub-graph seamlessly. It's possible to start in a small focused graph, then jump to a larger related graph, and so on, **traversing across databases** when needed. - **Modularity and isolation:** Each sub-graph (or sub-database) can remain relatively small and focused. This modular approach means the overall system can scale by adding more interconnected graphs rather than growing one monolithic graph to an enormous size.

This "graph of graphs" capability is crucial for **preventing performance degradation** (each graph stays lean) and for logically organizing data. Conventional graph databases don't easily allow one database to reference another; they usually assume one global graph or require manual federation, which is cumbersome. My design treats graphs themselves as first-class entities that can interconnect.

## Open Source and No Lock-In

It was very important to me that the core of the database be **open source** and free of proprietary lock-in: - **Transparent codebase:** I want to be able to **inspect, modify, and extend the database code**. An open-source foundation ensures the community (and I) can improve it or adapt it to specific needs. - **No feature paywalls:** Many "open-source" products today hold back advanced features for paid enterprise editions. I believe the essential capabilities should be in the open source version, not hidden behind a paywall. **Users shouldn't discover that critical features are only available in a closed-source, paid tier.** - **Community and collaboration:** Open source encourages a community of users and developers to collaborate, audit security, and contribute improvements. This aligns with the principle of transparency in how data is managed.

By building my system on open-source principles, I ensure that anyone can use it freely or run it themselves, and there's no dependency on a single vendor. In contrast, many graph databases (especially cloud-hosted ones) are proprietary or have closed extensions, which didn't satisfy my requirements.

## Strong Security and Access Control

Another core requirement was support for **state-of-the-art security practices and fine-grained access control**: - **Role-Based Access Control (RBAC):** The system should allow definition of roles and permissions down to very specific levels. For example, I might restrict certain subgraphs or data types to particular user roles. Complex, real-world access rules need to be enforceable. - **Data protection:** It should integrate well with encryption and other data protection mechanisms. Even though it's open source and serverless, it must not compromise on security features like authentication, authorization, auditing, and encryption at rest and in transit. - **Flexible security policies:** Just as the data model is

flexible, the security model should also be flexible. This means supporting custom security rules or plugins that can handle unusual scenarios or enterprise-grade requirements.

Most existing solutions have basic authentication and maybe simple role separation, but I wanted a platform where **sophisticated security models** (hierarchical roles, attribute-based access control, etc.) could be implemented. My design doesn't force a single security scheme but allows plugging in the necessary controls.

## Fast Reads, Tolerant of Slower Writes

For my use case, **read performance** is far more critical than write speed. I designed the database with an **asymmetry favoring reads**: - **Super fast reads:** Queries that fetch or traverse data should be **as fast as possible**, ideally returning results in real-time even as the dataset grows. The system is optimized for quick graph traversals and lookups. - **Slower writes acceptable:** It's okay if writes (adding or updating data) are slower or if the data takes a moment to become consistent across the system. I prefer to handle writes in the background or with a slight delay, rather than compromising read speed. - **Eventual consistency:** Embracing eventual (or slightly delayed) consistency is acceptable. The priority is that once data is written and settled, **subsequent reads are very quick** and the system can handle a high read throughput.

Most graph databases try to balance read and write performance, or they focus on transactional consistency which can slow down reads. My requirement was different – it's fine for the system to **take a bit longer ingesting data** as long as once the data is in, **retrieving and traversing it is lightning fast**. This trade-off is well-suited to scenarios where the database is read far more often than it's written.

## Horizontal Scale Without Slowing Down

I wanted to avoid the common scenario where **the larger the graph becomes, the slower queries get**. The design therefore emphasizes horizontal scalability and keeping query scopes narrow: - **No monolithic slowdown:** In many systems, as the dataset grows huge, operations slow down because there's more to search or maintain. I wanted to ensure that growth in data volume **does not linearly degrade performance**. - **Horizontal scaling:** The database should **scale out** by distributing data across multiple stores or nodes (especially since it's serverless, this could mean spreading data in storage across services or shards). If one graph becomes too large, it can be split or linked as subgraphs (as described in the "graph of graphs" approach) so that queries deal with manageable chunks. - **Targeted queries:** The architecture encourages querying only the relevant subset of the data rather than the entire dataset. By linking graphs and isolating data, a query can target just the portion needed. This means **each query scans a smaller effective dataset**, keeping performance high even if the total data is massive.

In essence, the system should **scale like a network of small graphs rather than one giant graph**. This prevents the "bigger it gets, the slower it becomes" problem. Traditional graph databases can struggle at very large scale or require big, always-on clusters to handle growth. My approach uses the combination of serverless scaling and graph-of-graphs partitioning to maintain speed at scale.

## Flexible, Multi-Layered Schema Capability

Another requirement was the support for **highly flexible schemas** – even schemas of schemas: - **Schema optionality:** The database should not force a single, rigid schema for all data. It should allow

each dataset or even each file to have its own schema definition if needed, or to be schema-free. **You can mix structured and unstructured data in the graph.** - **Multiple schema layers:** There might be a schema at the database level (general rules for that graph), but within it, different files or subgraphs could have their own schemas. This “schemas of schemas” approach means, for example, one subgraph might enforce an ontology or data model, while another subgraph linked to it might have a different structure. - **Ontology and taxonomy support:** The system should let you build **ontologies (formal definitions of types and relationships)** or taxonomies on top of the data, but using them is optional. In other words, you could have well-defined types and relations (for scenarios where consistency is needed), or remain completely flexible for other parts of the data.

This flexibility is in contrast to many graph databases that either enforce a single global schema or are entirely schema-less. I wanted the best of both: you can define **consistent schemas where useful** but also link in data that doesn’t conform, effectively handling **both structured and unstructured information together**. As the data evolves, the schema can evolve too – it’s not a fixed contract you must set in stone at the start.

## Tiered Storage and Cost Efficiency

To keep costs low and handle different data value, the system supports **multiple levels of storage with different cost/performance profiles**: - **Hot vs cold storage:** Frequently accessed data can live in faster (but perhaps more expensive) storage, while infrequently used data can reside in cheaper, slower storage. The database should integrate this concept so that data can move between tiers transparently. - **Pay-as-you-go storage:** You only pay for the storage you actually use. If you have a huge amount of data but much of it is archived in a low-cost tier, your costs stay low. Similarly, if data is deleted or moved, costs drop accordingly – there’s no pre-provisioned disk that you pay for regardless of usage. - **Bandwidth-cost awareness:** In a serverless model, especially using cloud storage, **data egress (transfer)** can cost money. The system should minimize unnecessary data transfer and make you **pay only for the queries or operations you actually perform**.

By having tiered storage, the graph database can manage billions of nodes and edges without breaking the bank, because old or seldom-used parts of the graph don’t sit in pricey high-performance storage. Traditional graph databases often assume one type of storage (all data in memory or on fast disk attached to the server). My approach treats storage as a continuum – with integration to things like object stores for cold data – to optimize cost.

## Built-in Version Control for Data

I also envisioned the database with **version control capabilities** for the data: - **Data versioning:** Any file or node in the graph should be versionable. If something changes, you can trace **what changed and when**, similar to how source code version control (like Git) works but applied to database entries. - **Historical tracking:** Being able to query or retrieve an older version of a node/edge or file is extremely useful for auditing and understanding evolution of the data. The system ideally would let you **navigate the graph not only in space (links) but also in time (versions)**. - **Integration or native support:** In my implementation, I have the ability to build version control on top of the graph store. Ideally the database itself could support it natively (like keeping diff logs or snapshots). If not built-in at core, it should at least be designed such that adding a version control layer is straightforward.

Most graph databases do not have built-in version control for data; at best, you’d have to manually copy data or use an external system to track changes. Given my requirements (especially with a file-based

storage approach), I wanted to ensure that changes to data could be tracked over time easily, increasing trust and debuggability of the graph content.

## Rich and Strongly-Typed Data Fields

Unlike many NoSQL or graph systems that treat values as generic strings or numbers, I wanted support for **rich, strongly-typed data**: - **Beyond primitives:** The database should allow defining custom data types or constraints. For example, an "Age" field could be defined as a positive integer, a "Username" field as a string matching a certain pattern, or a "GitHubRepoName" as a string that follows GitHub naming rules. - **Validation and consistency:** By having strong types, the system can **validate data on write**, ensuring that invalid data doesn't enter the graph (e.g., a negative age, or an improperly formatted ID). This adds consistency and quality to the data. - **Self-describing data:** Each type can carry meaning (semantic information). Instead of every value just being an untyped blob, knowing that a field is of type "EmailAddress" or "GeoCoordinate" provides context. This can help tools and queries to treat data appropriately (for example, a visualization might recognize a GeoCoordinate type and plot it on a map).

In many graph databases, schema (if present) might allow simple type constraints like integer vs string, but they don't typically support *domain-specific types* with custom rules. My ideal system treats types as first-class citizens, making the data more robust and easier to work with. It's like having a schema with real data types rather than just saying "property X is a string" – instead, "property X is a **Password** or an **Email**" and enforcing the rules that come with that.

## Fluent and Expressive Querying

Querying the data should be **powerful but also intuitive**. While graph databases often come with their own query languages (like Cypher, Gremlin, etc.), I want a system that **promotes a fluent design** for queries: - **High-level graph traversals:** It should be easy to express complex traversals (e.g., "find all files linked to user X within two hops in the graph") in a concise way. Ideally, the query interface feels natural to the developer or user, whether that's through a domain-specific language or an API in a familiar programming language. - **Composability:** Small query components should be easy to compose into bigger operations. This might mean supporting functional patterns (map/reduce on graph paths) or method chaining in an API. The goal is to avoid writing long, unintelligible queries – instead, break them into understandable pieces. - **Similar to coding with the graph:** The querying could feel like writing code that navigates objects. For example, following edges could be as straightforward as accessing object properties or calling methods, making the graph feel like an in-memory object graph, even though it's a database.

Many graph databases do have powerful query languages, but they often have steep learning curves or require a lot of boilerplate. I aimed for a query experience that is **developer-friendly** and encourages exploring the graph data without excessive ceremony. This goes hand-in-hand with having strongly-typed data – queries can leverage the type information to be more precise and safer.

## Integrated Visualization and Self-Management

Lastly, a distinguishing requirement for me was having **built-in visualization and using the graph to manage itself**: - **Visualization as a feature:** Instead of treating visualization as an afterthought (export your data to some tool to visualize), the database should provide ways to **visualize the graph structure on the fly**. This could mean an interactive UI that lets you expand nodes and see connections, or generating diagrams as part of queries. Visualization helps understand and trust the

data relationships. - **Using the graph to visualize the graph:** This sounds meta, but the idea is to **store the visualization metadata in the graph itself**. For example, you could have nodes that represent visual layouts or user-defined diagrams that link to the actual data nodes. The system can then read those and render a picture. In essence, you're using graph features (like linking and attributes) to describe how to visualize portions of the graph. - **Self-referential management:** Similarly, the database can manage its own schema or configuration through the graph. Instead of config files or separate systems, configuration could be done by storing special nodes/edges that the system interprets as settings. This means **administration of the database (security rules, schemas, etc.) could be done via the graph database itself** – eating its own dog food, so to speak.

The goal is an environment where you don't leave the graph ecosystem to manage or understand the graph. Many current databases rely on external tools for visualization or require writing separate config code for schema and security. I wanted a unified approach where **the graph database is both the storage and the interface for interacting with the structure and metadata**.

## Conclusion: A Custom Solution Using a File System and Serverless Functions

After listing all these requirements, I realized no off-the-shelf graph database met every single one of them. The popular choices each fell short in at least one area – for example, some are not serverless, some aren't open source, some can't handle arbitrary file data or lack advanced security, etc. This gap is why I decided to build my own solution.

The approach I took was to use a **file system as the data store** (which inherently gives a flexible, hierarchical storage for various file types and easy horizontal scaling) and add a layer of **serverless compute components** on top. In practice, this means the data lives in something like object storage (with files representing nodes, edges, and other content), and all database operations are handled by on-demand functions that spin up, do the work (traversal, query, update), and then terminate. This design naturally achieves the serverless and stateless criteria, while leveraging the durability and scalability of modern distributed file storage.

By combining these elements, I achieved a graph database that: - **Meets all my requirements** outlined above, without sacrificing one feature for another. - Remains cost-efficient and scalable, since it only uses resources as needed and can store immense amounts of data in cheap storage. - Can evolve with my needs, because it's open source and flexible in schema, types, and integration.

In summary, building my own serverless graph database was driven by the vision of a system that is **truly usage-based, highly flexible in data and schema, secure, and capable of linking anything to anything** – something I couldn't get from existing solutions. The result is a unique graph-of-graphs platform that grows with my data, stays fast as it scales, and gives me full control over its behavior.

---