Revolutionizing Data Management: A Comprehensive Study of FaunaDB's Impact on Modern Database Technologies

Luke Abbatessa

Yuting Zheng

April 8, 2024

Abstract

This report evaluates FaunaDB, a distributed, multi-model database tailored for modern application needs. It stands out as a document-relational database, offering a unique blend of flexibility and structure to accommodate diverse data models. We delve into its foundational principles, explore diverse use cases, and offer a detailed tutorial for practical understanding. Additionally, we meticulously assess its utility, ease of adoption, scalability potential, licensing intricacies, and the vibrancy of its community support. By meticulously scrutinizing these facets, we endeavor to furnish development teams with the requisite insights to make informed decisions regarding the integration of FaunaDB into their projects. Through this analysis, we aim to illuminate FaunaDB's suitability, capabilities, and potential challenges, thereby empowering development teams to navigate the complexities of database selection with perspicuity and confidence.

Introduction

The concept of a database (representative of a so-called "relational model") was established in the public sphere in 1970 by IBM Research Laboratory employee Edgar Frank Codd. Specifically, Codd described the relational model as "...a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other" [Codd, 2002]. Since then, relational databases have been used extensively throughout industry, and their ubiquity can be attributed to three main factors: the relational model, SQL, and ACID compliance. With the relational model, data is managed in tabular format, with columns representing individual attributes (each of which has its own data type), rows representing individual observations whose attributes are organized within a tuple, and the overall collection of columns and rows representing an entire relation. With SQL (which stands for Structured Query Language), it has been the relational database language paradigm for decades, allowing its users to ask a variety of questions about a particular dataset through the use of Create, Read (e.g. Select), Update, and Delete (CRUD) operations, among many others. With ACID compliance, when working with a relational database, all transactions are Atomic (all statements are applied or none are), Consistent (the transaction doesn't change the state of the database), Isolated (transactions are independent of one another), and Durable (transactions persist despite possible failures). In all, relational databases have dominated industry for over 50 years, thanks to their data persistence, concurrency support, transaction support, and application integration.

Nevertheless, at the dawn of the 21st century, non-relational databases emerged as a viable competitor to relational databases, with them coming the coining of the term "NoSQL". One of the earliest examples of a NoSQL database came with Google's BigTable, which was described as "...a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers" [Chang et al., 2008]. Among the reasons one may choose a NoSQL database over a relational database, the key ones include greater scalability (e.g. greater performance when handling possibly terabytes to petabytes of data), greater flexibility as a result of working with unstructured data, the opportunity to expand your knowledge and skillset in data management and data engineering, as well as the opportunity to increase your earning potential if working in industry. In terms of types of NoSQL databases, there are four main types that warrant consideration: key-value stores, document databases, graph databases, and columnar databases. Key-value stores are in-memory databases that query by key and are designed to be simple, fast, and scalable; a popular key-value store is Redis. Document databases store data as structured documents and are designed to be simple, flexible, and scalable; a popular document database is MongoDB. Graph databases are optimized for storing nodes, relationships, as well as node and/or relationship properties, and they are designed to be non-hierarchical and flexible; a popular graph database is Neo4J. Lastly, columnar databases store data as columns instead of rows, and they are designed for efficiency, reduced I/O, and better compression; a popular columnar database is Cassandra. In all, NoSQL databases have been able to flourish in recent years because many of these databases are open-source, designed for distributed storage, schema-less, and capable of working with so-called "big data". One NoSQL database in particular that we will choose to focus on is the document-relational database FaunaDB.

FaunaDB, per the database's website, "...is a distributed document-relational database that combines the flexibility of documents with the power of a relational, ACID compliant database that scales across regions, clouds or the globe" [FaunaDB]. FaunaDB is built upon the principles of writing user-defined functions (UDFs) using the Fauna Query Language (FQL), acting as a modern security model, assisting with event streaming, and allowing the import and export of data. Structurally, developers can write declarative queries using the power of the relational model, and transactions are ensured multi-region, ACID consistency that typical document databases do not have. Also, FaunaDB allows for the import of JSON documents and thus inhabits a flexible schema, and FQL combines the storage of data in documents akin to document databases with the modeling of data as separate entities akin to relational databases. As part of the Fauna Services Agreement, the licensing terms are specified in five subparts of the License to Services: License for Paid Use, Order Form, License for Free Tier, Support and Service Level Agreements, and Updates to Services. The License for Paid Use grants customers a limited license to access and use FaunaDB's services for internal business use. The Order Form allows customers to purchase a longer subscription than a month-to-month license via a form. The License for Free Tier allows customers a free trial of FaunaDB's services. The Support and Service Level Agreements are terms and conditions that only apply to subscriptions purchased through the Order Form. Lastly, the Updates to Service allows FaunaDB the right to update its services periodically as it sees fit. Globally, FaunaDB's users consist of 80,000+ development teams across 180 countries who have created 300,000+ databases total [FaunaDB]. In this report, we will provide examples of use-cases of FaunaDB; a comprehensive tutorial covering the installation, example FQL queries, and illustrative figures of the database; a description of the strengths and weaknesses of FaunaDB (particularly in comparison to MongoDB); as well as concluding remarks about

the efficacy of the database overall.

Use-Cases

Software as a Service (SaaS) Company CloudConnect CRM is a SaaS company that offers a comprehensive Customer Relationship Management (CRM) platform to businesses of all sizes. The platform helps companies manage their customer interactions, streamline sales processes, and improve customer service. CloudConnect CRM leverages Fauna's capabilities to address various challenges and provide a robust solution for its customers.

Use Case 1 Multi-tenancy Management:
CloudConnect CRM serves a diverse customer base ranging from small startups to large enterprises. With Fauna's support for multi-tenancy out of the box, CloudConnect CRM can efficiently host data from multiple tenants within a single database instance. Each tenant's data is securely isolated, ensuring data privacy and compliance with regulatory requirements. Fauna's hierarchical parent-child database construct simplifies management and eliminates the need for complex engineering overhead associated with traditional databases [FaunaDB].

Use Case 2 User Data + Metadata Management:
Managing metadata is critical for CloudConnect CRM to adapt to evolving business needs and customer requirements. Fauna's flexible schema and document-relational model allow CloudConnect CRM to easily manage metadata without schema rigidity constraints. This flexibility enables the platform to introduce new data types or relationships seamlessly, without downtime for migrations. Additionally, Fauna's attribute-based access control ensures granular control over user permissions, enabling real-time authorization and access management [FaunaDB].

Use Case 3 Global Distribution + Data Compliance:
As CloudConnect CRM expands its user base globally, it faces the challenge of ensuring low latency access and compliance with local data regulations. Fauna's distributed transaction engine offers auto-replication across distributed nodes and regions, ensuring high availability and low latency access for users worldwide. Moreover, Fauna's adherence to data protection and privacy regulations ensures compliance with local data regulations, safeguarding sensitive customer information [FaunaDB].

Use Case 4 Microservice Support:
CloudConnect CRM adopts a microservices architecture to modularize its CRM platform into specialized services, including customer management, lead tracking, sales analytics, and marketing automation. Each microservice operates independently, running in its own process and communicating with other services via lightweight HTTPS APIs. Fauna's API delivery model seamlessly integrates with CloudConnect CRM's microservices architecture, allowing each service to interact with the database via secure HTTP requests. This decoupled approach enables CloudConnect CRM to scale individual microservices independently based on demand, ensuring optimal performance and responsiveness for its users [FaunaDB].

Retail and E-Commerce Company E-Shop Emporium is a retail and e-commerce company aiming to provide a seamless shopping experience for customers worldwide. Leveraging Fauna's innovative solutions, E-Shop Emporium tackles various challenges inherent to the retail and e-commerce industry, ensuring efficient operations, accurate inventory management, and personalized customer experiences.

Use Case 1 Storing Comprehensive Product and User Metadata:
E-Shop Emporium faces the challenge of storing and managing detailed product information, user profiles, and transactional data at scale. Fauna's document-relational model empowers E-Shop Emporium to flexibly store and relate evolving e-commerce data, accommodating descriptions, specifications, customer information, and pricing details. Fauna's schema flexibility enables seamless adjustments to changing data structures and access patterns, ensuring adaptability to evolving business needs [FaunaDB].

Use Case 2 Efficient Order Processing:
With a global customer base and high-scale transaction demands, E-Shop Emporium requires efficient and reliable order processing capabilities. Fauna's distributed transaction engine ensures strong consistency and low-latency transactions across the globe without the need for extensive engineering operations. This enables E-Shop Emporium to process orders accurately and reliably, even during peak demand periods, enhancing customer satisfaction and trust [FaunaDB].

Use Case 3 Inventory Management Across Regions:
Synchronizing inventory information across multiple regions and time zones is critical for E-Shop Emporium's op-

erations. Fauna's strong consistency across regions simplifies inventory management, ensuring accurate stock levels and minimizing the risk of over-selling or stockouts. This eliminates the need for extensive engineering operations associated with legacy databases, allowing E-Shop Emporium to focus on delivering exceptional customer service and optimizing inventory turnover [FaunaDB].

Use Case 4 Personalized Shopping Experience:

E-Shop Emporium aims to offer personalized shopping experiences tailored to each customer's preferences and behavior. Fauna's API delivery model, combined with its flexible querying and indexing capabilities, enables E-Shop Emporium to build powerful personalization engines. By leveraging edge compute functions, E-Shop Emporium can deliver context-aware content, product recommendations, and targeted marketing campaigns with ultra-fast response times, enhancing customer engagement and driving sales [FaunaDB].

Gaming Company Epic Quest Adventures is a gaming company that specializes in developing immersive, multiplayer online role-playing games (MMORPGs). Leveraging Fauna's modern distributed database solutions, Epic Quest Adventures addresses various challenges inherent to game development, ensuring seamless gameplay experiences, robust social features, and efficient session management for players worldwide.

Use Case 1 Multi-player Dynamic Game State:

In Epic Quest Adventures' open-world MMORPGs, the game world dynamically changes based on player interactions and events, requiring frequent updates to the game state. Fauna's distributed transaction engine ensures low-latency updates and strong consistency across all regions, enabling Epic Quest Adventures to handle high write throughput and synchronize inventory states seamlessly. Whether tracking leaderboards, managing digital inventories, or processing in-game transactions, Fauna's scalability and reliability support the dynamic nature of the game world [FaunaDB].

Use Case 2 Social Features:

Epic Quest Adventures prioritizes social interaction within its gaming experiences, offering features such as friends lists and in-game messaging. Fauna's relational capabilities seamlessly accommodate many-to-many relationships, facilitating efficient management of social connections and interactions. Additionally, Fauna's native event streaming enables instant notifications and updates across player networks, enhancing social engagement and fostering a sense of community within the game [FaunaDB].

Use Case 3 Session Management:

Managing user sessions is crucial for Epic Quest Adventures to ensure uninterrupted gameplay experiences across different devices and platforms. Fauna's automatic global distribution and fast, consistent data access enable seamless session management across multiple servers and regions. Players can start, pause, and resume their gameplay seamlessly, with Fauna's built-in temporality allowing for automatic tracking of session changes over time. This ensures that players experience no interruption or lag when accessing or updating their session data, enhancing the overall gaming experience [FaunaDB].

Tutorial

Download/Install FaunaDB

1) Visit the following URL: `https://fauna.com` (you'll know you're on the right website if you see the silhouette of a purple bird followed by the word "fauna" in the top left corner of the screen).

2) Click either "START FREE TRIAL" in the middle of the screen or "START FOR FREE" in the top right corner of the screen. This will bring you to a page that has you sign up with a Company Email and Password, sign up via GitHub, or sign up via Netlify; regardless of which option you choose, once you sign up, you will begin a 30-day free trial with FaunaDB and receive some sort of confirmation of your sign-up.

3) Assuming you sign up with a Company Email and Password, it will bring you to a page that first has a popup with three sets of questions that FaunaDB asks you to get a sense of your database background and project intentions.

4) Once you answer the questions, the page you will see will have a "Databases" panel with a "Create Database" button, which you should click.

5) After clicking the "Create Database" button, in the resulting "Create database" popup, give your database a name under "Name", select the region group you geographically associate yourself with under "Region Group",

select ”CREATE”, and you’re done; you’ve successfully created a database! The resulting screen will show available resources including collections, indexes, and functions on the left side, and the Fauna web shell to execute Fauna queries on the right side.

Import JSON files, CSV files, or directories into a Collection

1) Install nvm; this can be done by running the following command line prompt:

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
   ```

   To prove nvm is successfully installed, run the following command line prompt:

   ```
   nvm -v
   ```

   If the command line outputs something like ”0.39.7”, nvm is successfully installed.

2) Using nvm, install Node.js using the following command line prompt:

   ```
   nvm install node
   ```

   To prove node is successfully installed, run the following command line prompt:

   ```
   node -v
   ```

   If the command line outputs something like ”v21.7.2”, node is successfully installed.

3) Using nvm, install the latest version of npm using the following command line prompt:

   ```
   nvm install-latest-npm
   ```

   To prove npm is successfully installed, run the following command line prompt:

   ```
   npm -v
   ```

   If the command line outputs something like ”10.5.1”, node is successfully installed [Karrys, Bhaskar, & Thomson, 2023]; [Harband, 2014].

4) Install fauna-shell by running the following command line prompt:

   ```
   npm install -g fauna-shell
   ```

5) To configure your connection to the Fauna account you created in the beginning, run the following command line prompt:

   ```
   fauna cloud-login
   ```

6) Enter your email and password from your Fauna account, and choose an endpoint to set as default (exit-us or exit-eu).

7) To import data to Fauna, run the following command line prompt [Videla et al., 2018]:

   ```
   fauna import --path <value>
   ```

Queries to Model CRUD Operations

You’ll want to create collection(s) to apply FQL queries to; to do this, select the ”+” sign on the Collections option to the left of the screen, provide a name under ”Name” for the ”Create collection” popup, and click ”CREATE” to create the collection.

Create

Let’s say you wanted to create a user with the name ”William T” and the email ”ryker@enterprise.com”; assuming you created a User collection, the FQL code snippet for this that uses the create() function would look like this:

```
User.create(name: "William T", email: "ryker@enterprise.com")
```

and the resulting success message would look something like this:

```
>> SUCCESS id:  "360940140394184781", coll:
User, ts:  Time("2023-04-02T23:29:51.680Z"),
name:  "William T", email:  "ryker@enterprise.com",
```

We can see that the user is automatically assigned an id, and the success message also shows the timestamp of the transaction.

Read

Let's say you first create a to-do list with a Todo collection using Create; the list could look something like this:

```
Todo.create(title:  "Get Milk", status:  "Pending")
Todo.create(title:  "Do Laundry", status:  "Pending")
```

and you'd like to simply read your to-do list; the FQL code snippet for this that uses the .all() function would look like this:

```
Todo.all()
```

and the code that would return would look something like this:

```
data:  [id:  "360941936862822477", coll:  Todo, ts:
Time("2023-04-02T23:58:24.920Z"), title:  "Get Milk", status:  "Pending",,
id:  "360942191116288077", coll:  Todo, ts:
Time("2023-04-03T00:02:27.385Z"), title:  "Do Laundry", status:
"Pending",,],
```

Update

Let's say you wanted to update the status of the "Get Milk" task from "Pending" to "Done"; the FQL code snippet for this that uses the update() function would look like this:

```
let todo = Todo.byId("360941936862822477") todo.update(status:  "Done")
```

and the code that would return would look something like this:

```
id:  "360941936862822477", coll:  Todo, ts:
Time("2023-04-03T03:43:10.010Z"), title:  "Get Milk", status:  "Done",
```

Delete

Let's say you wanted to delete the "Get Milk" task since it's now done; the FQL code snippet for this that uses the delete() function would look like this [FaunaDB]:

```
let todo = Todo.byId("360941936862822477") todo.delete()
```

FaunaDB In Action

Set the Stage

To demonstrate the capabilities of FaunaDB, I will connect a CSV file to FaunaDB, perform data transformations in Python, generate visualizations from the resultant data, and interpret the visualizations. The dataset I am using consists of 1,000 people, where each person has an index, user ID, first name, last name, sex, email, phone, date of birth, and job title [Datablist]. Below are the steps to follow along:

1) In the Fauna Dashboard, create a new database titled "FaunaCSVPython" and a new collection titled "Person".
2) Open a .py file within a Python IDE and import the necessary libraries; these include the following imports:

   ```
   from faunadb import query as q
   from faunadb.objects import Ref
   from faunadb.client import FaunaClient
   ```

3) Initialize your connection to Fauna with the following line of Python code:

```
client = FaunaClient(secret=config.secret)
```

The ”secret” parameter represents a secret account key you have to create in the Fauna Dashboard to authenticate to the Fauna Logs API, and the account key itself is put in a config.py file to ensure security.

4) Read in the csv file using the reader() function and make sure to skip the first line of headers.
5) Obtain the age of each person by extracting the year from each person’s date of birth and subtracting it from today’s year.
6) Sort the ages of the people into four age groups: ”Child” (e.g. ages 0-12), ”Adolescent” (e.g. ages 13-17), ”Adult” (e.g. ages 18-60), and ”Senior” (e.g. ages 60-above).
7) Add the data to the FaunaDB database using the following FQL snippet [Kattimani, 2020]:

```
client.query(q.create(q.collection("Person"), "data":"Index": row[0],
"User Id": row[1], "First Name": row[2], "Last Name": row[3],
"Sex": row[4], "Email": row[5], "Phone": row[6], "Date of birth":
row[7], "Job Title": row[8], "Age": age, "AgeGroup": AgeGroup))
```

8) Begin the process of developing visualizations in a separate .py file by importing the libraries and initializing the connection to Fauna the same way you did when connecting the CSV file to Fauna.
9) Start a pagination loop (e.g. the process of iterating through sets after sets of documents to capture all documents from the ”Person” collection) by initializing an empty list to store the documents and, within a while loop, querying the ”Person” collection with this FQL snippet:

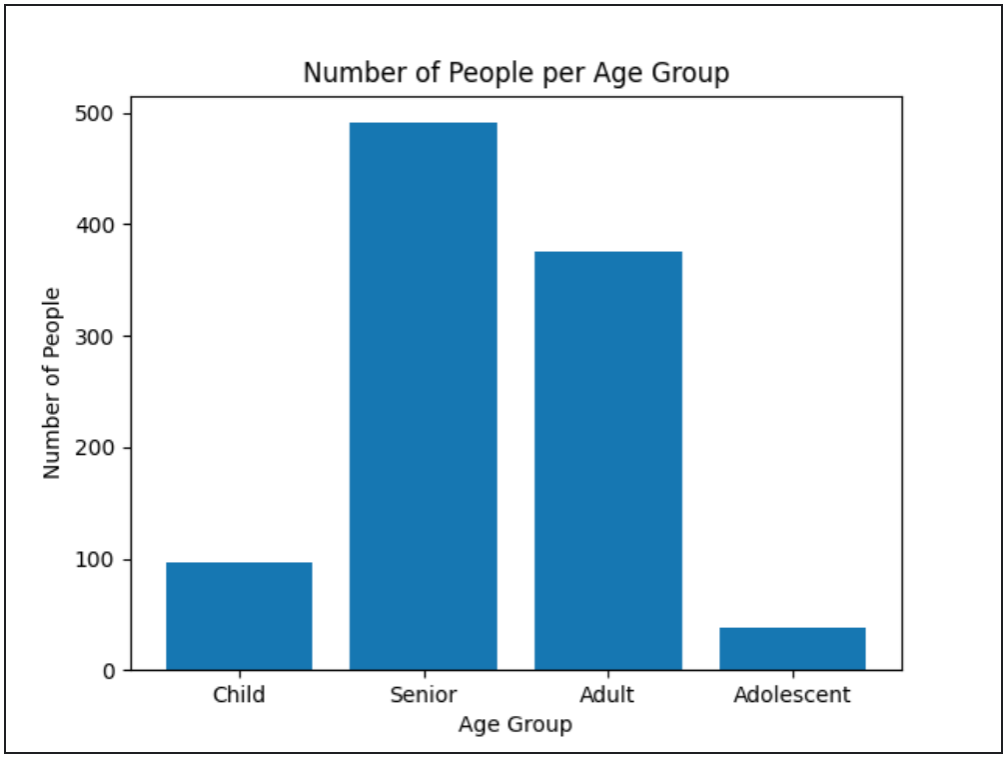
```
result = client.query(q.paginate(q.documents(q.collection("Person")),
size=100, after=nextpage))
```

Also within the while loop, add the data to the list of documents, and get the next page cursor if available.

10) Extract the AgeGroup from each document by coding this FQL snippet:

```
for ref in alldocuments: document = client.query(q.get(ref))
agegroup = document["data"]["AgeGroup"]
```

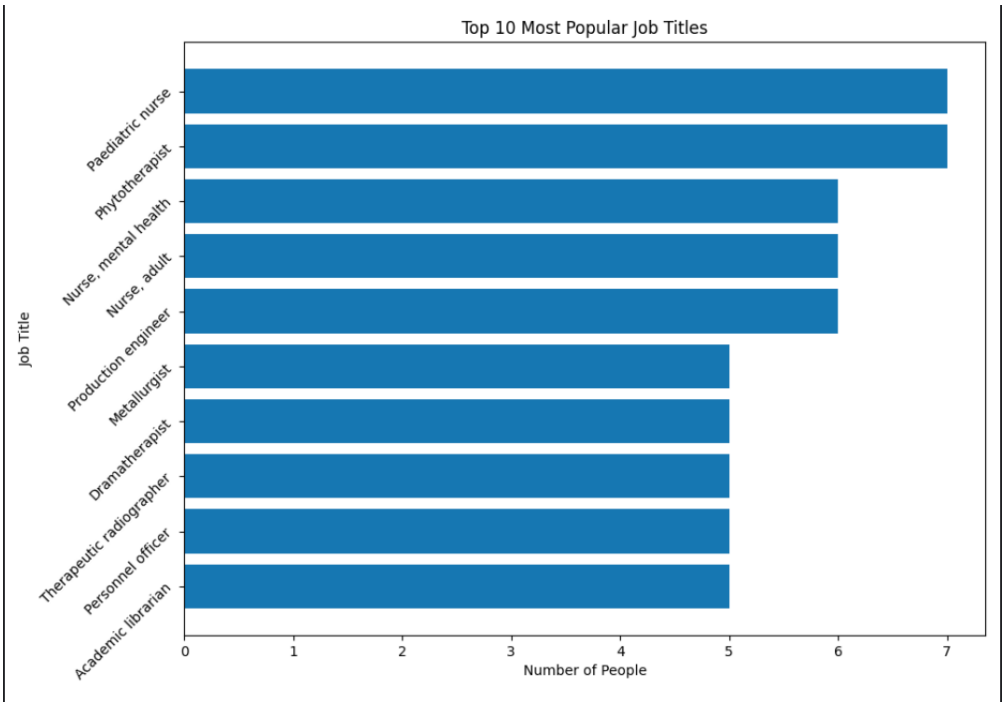
11) Count the occurrences of each age group, extract AgeGroup labels and counts for plotting, and plot age group versus number of people via matplotlib.



Looking at the figure above, one can see that the majority of the people contained in the data are Seniors, followed closely by Adults, and then Children and Adolescents are both few in number. The original source of the data was unclear about the context of the dataset, but given that each person's name, sex, email, phone number, DOB, and job title were collected, this could've been some sort of survey. If that is the case, it makes logical sense that seniors occupied the largest demographic since they likely have the most free time.

Building off of this, to visualize, say, the top 10 most popular job titles:

1) Extract job titles from each document
2) Count the occurrences of each job title, get the top 10 most popular job titles, extract job titles and counts for plotting, and plot number of people versus job title via matplotlib.



Looking at the figure above, at first glance, it's very surprising that job titles aren't more popular, especially since there are 1,000 people in the dataset. However, after looking more closely at which job titles are the most popular, you begin to realize that many of the people from the dataset work in health care, thus it makes sense that certain job titles aren't more popular because of how many departments there are in a hospital and how many specializations people in the medical field can choose from and pursue. Overall, it appears that Paediatric nurses and Phytotherapists are most popular, however I will note that I took a closer look at the dataset and certain jobs are represented multiple ways in the dataset (i.e. "Therapeutic radiographer" and "Radiographer, therapeutic"), thus certain jobs may be underrepresented in popularity.

Strengths and Weaknesses

In order to gauge the strengths and weaknesses of FaunaDB, I will directly compare FaunaDB to MongoDB, a highly touted document database. Beginning with FaunaDB's strengths, FaunaDB has characteristics of both document and relational databases, unlike MongoDB which is strictly representative of document databases. Also, unlike MongoDB, FaunaDB ensures strict serializability, which means that transactions you submit and read model a system already shaped by committed, independent transactions in the past. In addition, FaunaDB is considered a global API with zero-ops with regards to scaling, and it is more compatible with modern applications and serverless frontends compared to

MongoDB. Architecturally, FaunaDB also surpasses MongoDB, in that it is naturally distributed and has less custom building required [FaunaDB].

At the same time, FaunaDB isn't without its weaknesses. For one, MongoDb is free to install and thus open-source, while FaunaDB is not open-source and provides no more than a 30-day free trial before outlining different pricing options. Also, while MongoDB is self-hostable, FaunaDB is not, which is something that many developers look for in a database [MongoDB, 2024b]. In addition, FaunaDB has some latency issues; certain "get" and "match" operations may take hundreds of milliseconds, while for MongoDB they may take only a few milliseconds [MongoDB, 2024a]. Also, unlike FaunaDB, MongoDB comes equipped with real-time data analytics on MongoDB Atlas [MongoDB, 2024c]; [Fireship, 2020].

Conclusions

FaunaDB emerges as a compelling option in the realm of modern databases, offering a unique blend of document and relational database features. Its ability to combine the flexibility of documents with the power of a relational, ACID-compliant database makes it suitable for a wide range of applications, from SaaS platforms to e-commerce solutions and online gaming.

One of FaunaDB's notable strengths lies in its support for multi-tenancy management, enabling organizations to securely host data from multiple tenants within a single database instance. Its flexible schema and document-relational model facilitate efficient metadata management and adaptability to evolving business needs. Additionally, FaunaDB's distributed transaction engine ensures high availability, low latency access, and compliance with global data regulations, making it suitable for global distribution and data compliance requirements.

Moreover, FaunaDB seamlessly integrates with microservices architectures, enabling modularization and scalability of applications. Its support for event streaming, user-defined functions, and edge compute functions empowers developers to build sophisticated applications with ease.

Despite its strengths, FaunaDB does face some limitations, such as its proprietary nature and lack of self-hosting options compared to open-source alternatives like MongoDB. Latency issues may also arise in certain operations, and real-time data analytics are not as readily available as with some other platforms.

In essence, FaunaDB offers a compelling solution for organizations seeking a flexible, scalable, and globally distributed database for their modern applications. Its innovative features and robust performance make it a strong contender in the ever-evolving landscape of database technologies. By carefully evaluating its strengths and weaknesses, development teams can make informed decisions regarding its adoption, ensuring that FaunaDB aligns with their project requirements and objectives.

Bibliography

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), 1–26. https://doi.org/10.1145/1365815.1365816

[Codd, 2002] Codd, E. F. (2002). A relational model of data for large shared data banks. In M. Broy & E. Denert (Eds.), *Software Pioneers*, 263–294. https://doi.org/10.1007/978-3-642-59412-0_16

[Datablist] Datablist - Made in Nantes (France). (n.d.). Download Sample CSV Files for free. *Datablist*. https://www.datablist.com/learn/csv/download-sample-csv-files#people-dataset

[FaunaDB] FaunaDB *Fauna*. https://fauna.com/

[Fireship, 2020] Fireship. (2020, October 15). FaunaDB Basics - The Database of your Dreams. *YouTube*. https://www.youtube.com/watch?v=2CipVwISumA&t=183s

[Harband, 2014] Harband, J. (2014, December 21). nvm. *GitHub*. https://github.com/nvm-sh/nvm

[Karrys, Bhaskar, & Thomson, 2023] Karrys, L., Bhaskar, & Thomson, E. (2023, October 22). Downloading and installing Node.js and npm. *npm Docs*. https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

[Kattimani, 2020] Kattimani, R. (2020, July 27). Python Data Transformations: Connecting CSV to FaunaDB. *YouTube*. https://www.youtube.com/watch?v=psMeb7U6H5k&list=RDCMUC6OrQk8WsnCOR1OezlUU9zQ&start_radio=1&rv=psMeb7U6H5k&t=1

[MongoDB, 2024a] MongoDB, Inc. (2024a). Best Practices Guide for MongoDB Performance. *MongoDB*. https://www.mongodb.com/basics/best-practices#:~:text=How%20fast%20are%20MongoDB%20queries,size%20and%20machine%20specs%2C%20etc.

[MongoDB, 2024b] MongoDB, Inc. (2024b). MongoDB Community Edition. *MongoDB*. https://www.mongodb.com/products/self-managed/community-edition#:~:text=MongoDB%20is%20a%20general%2Dpurpose,well%20as%20in%20the%20cloud.

[MongoDB, 2024c] MongoDB, Inc. (2024c). Real-Time Analytics. *MongoDB*. https://www.mongodb.com/use-cases/analytics/real-time-analytics#:~:text=a%20competitive%20advantage.-,MongoDB%3A,via%20the%20MongoDB%20Query%20API

[Videla et al., 2018] Videla, A., Zinkevych, S., Macneale, N., Chase, Stuart, C., Berkeley, W., Edwards, E., Parkhomenko, S., Henry, Quaresma, B., Neris, M., Maia, R., Harris, R., Paterson, P., Duarte, V., Cryans, J., Wusatosi, Schrader, S., Caudill, N., … Pintor, E. (2018, April 29). fauna-shell. *GitHub*. https://github.com/fauna/fauna-shell?tab=readme-ov-file