ChatGPT

Choosing the Right Database for Your “Second Brain” Project

Overview: You are deciding between two database approaches for a home-based “second brain” – a system that stores all your personal, business, and project knowledge locally (off-grid) with the option to scale or sync to the cloud later. The two options we previously discussed are:

- **Option 1: A Traditional Relational Database** (e.g. using SQLite locally or PostgreSQL).
- **Option 2: A Graph/Multi-Model Database** (e.g. using a graph-oriented database like ArangoDB or Neo4j, which can store data as a network of nodes and relationships, often with flexible schemas).

Below we’ll compare these options in beginner-friendly terms, then recommend the one that best fits your needs (independence from “the grid”, flexibility, and future scalability).

Option 1 – Relational Database (Tables with Fixed Schema)

What it is: A relational database organizes data into tables (rows and columns) with predefined schemas. Each table represents an entity (for example, a **Person** table or a **Note** table), and tables can reference each other through keys (IDs). You use Structured Query Language (SQL) to store and retrieve data. Relational systems are known for strong consistency (ACID transactions) and a well-structured approach to data [1] .

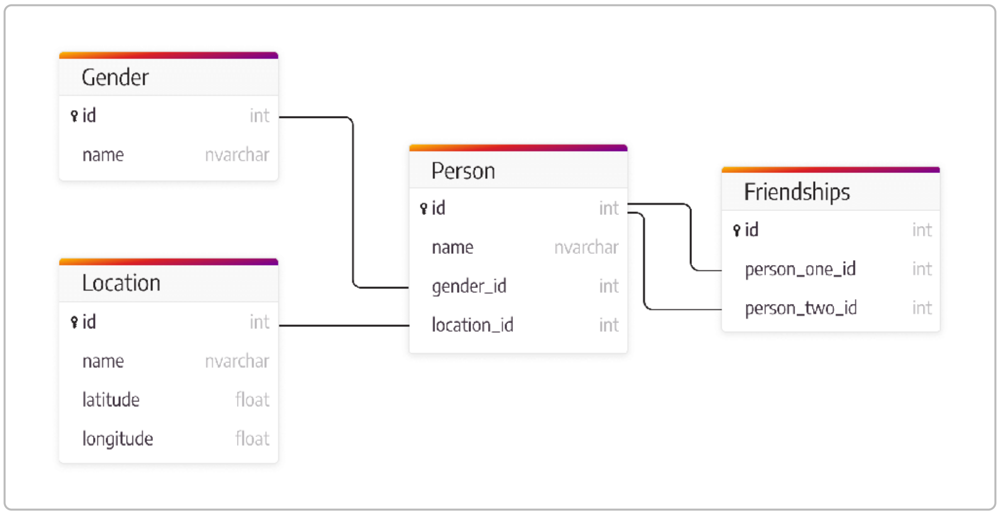

Illustration: A simple relational data model. In a relational design, you might have a **Person** table (with columns like name, gender, location), a **Location** table, and a **Friendship** table linking two people by their IDs. Each table is like a structured spreadsheet, and relationships are represented by storing reference IDs (for example, a Person’s record stores a location_id that points to an entry in the Location table, and the Friendship table stores pairs of person IDs).

Advantages of Relational DB for your use case:

- **Maturity & Reliability:** Relational databases (like PostgreSQL, MySQL, or SQLite) have been around for decades. They are very stable and **enforce data integrity** (e.g. ensuring references point to valid entries) [1]. This means you won't easily corrupt or lose links between pieces of data, which is great for a personal knowledge base you rely on. There's a huge community and ecosystem around SQL databases [2], so finding tools, documentation, and help is easier.

- **ACID Transactions:** They excel at preserving consistency – if you make multiple changes, an ACID-compliant relational DB ensures either **all or none** of the changes are saved, preventing partial updates. This is useful if, say, you update several related pieces of your "second brain" at once (you wouldn't end up with half-saved relationships).

- **Simple Setup (for single-user):** If you use an embedded relational database like **SQLite**, it's extremely lightweight and easy to start – the entire database is a single file on disk, no server required [3]. This fits well with a local/off-grid setup. You can copy or backup that one file to save your whole knowledge base (SQLite is **portable and backup-friendly** [3]). For example, many personal note-taking apps use SQLite under the hood for simplicity. (Using PostgreSQL locally is also possible but would involve running a server process; SQLite simplifies this.)

- **Familiar Query Language:** SQL is a standard query language widely taught and used [4]. Even as a beginner, you can find countless tutorials and examples. This standardization means you're not locked into one vendor – you could start with SQLite, and later move to a cloud PostgreSQL with minimal query changes, since both use SQL.

Drawbacks/Considerations for Relational DB:

- **Fixed Schema (Less Flexible):** In a relational model, you need to define tables and their columns upfront. Changing the structure later (adding a new field or a new type of entity) means altering the schema. For example, if you initially set up a table for Notes with certain fields and later want to add a new attribute to each note, you must add a column and possibly update existing records. If you want to start tracking a completely new kind of item (say, a "Project" with its own properties), you'd typically create a new table for it and define relationships (foreign keys) to link it to other tables. This isn't terribly hard, but it's a **rigid approach** compared to a more flexible schema-less system. A source comparing the two notes that in relational databases, a new column must be added for each additional attribute, whereas in a flexible schema system you can add properties on the fly [5]. In short, relational databases are excellent when your data structure is well-understood and doesn't change often [6], but they can require maintenance when your data model "evolves" with new ideas.

- **Handling Relationships and Queries:** Relational databases *can* represent relationships (using foreign keys and join tables, as in the **Friendship** table example above), but highly interconnected data can get complex. Each link between entities is retrieved by performing a **JOIN** (matching an ID in one table to a row in another table). For a few relationships this is fine, but if you have data that is *very interlinked* (like a concept map of ideas, or tasks connected to many projects and people), queries might involve joining many tables or recursive logic. This can become slow or complicated as those connections deepen [7]. For example, finding "all notes related to my project X through any chain of references" might require multiple joins or recursive common table expressions in SQL.

Relational databases can do it (SQL has features for recursive queries), but it's not their strongest suit and performance may suffer if the query touches a lot of tables or levels [7]. In essence, relational systems shine with clearly structured, tabular data, but **struggle as the web of relationships becomes more complex** [7].

- **Local but Not Graph-Optimized:** Your desire is to have a "second brain" that likely mimics how concepts connect to each other. While a relational DB can represent connections, it doesn't *natively* think in graphs – it thinks in tables. If your primary operations will be things like "find everything connected to this item" or "explore the network of how idea A is linked to B," a relational approach can be a bit unwieldy. It isn't **optimized for traversal** of networks (those operations require joins that can be resource-intensive when you hop through many links) [8].

- **Scaling Up:** Since you mentioned possibly offloading to the cloud for scaling later, note that traditional relational databases typically scale **vertically** (handle more data by using a bigger machine). PostgreSQL, for example, can manage a **very large** dataset on one server and can do read-replication, but it doesn't automatically shard data across multiple servers out-of-the-box [9]. There are extensions and cloud services that partition or distribute Postgres, but it's not as straightforward as with some newer databases. This means if one day you wanted a distributed cloud setup serving many users at once, a relational DB would likely require additional tooling or a move to a managed service that handles clustering. That said, for personal or even small-team usage, a single good server might be plenty – Postgres is quite robust and can handle many millions of records on one instance. Just keep in mind that horizontal scaling (spreading one database across nodes) isn't the forte of classic SQL systems without extra layers [9].

Option 2 – Graph/Multi-Model Database (Nodes & Relationships)

What it is: A graph database stores information as **nodes** (entities) and **edges** (relationships) between those nodes. In essence, it treats relationships as first-class data, not just implied links via IDs [7]. A **multi-model** database like ArangoDB is similar but also allows other formats (for example, storing documents or key-value pairs) alongside graph data. This means you could, say, store a note or contact as a JSON document and also create graph edges to represent connections (like "Note A references Project X" or "Person Y is friend of Person Z"). The query languages for graph databases (e.g. Cypher for Neo4j, or AQL for ArangoDB) are designed to easily navigate these relationships. They often read somewhat like natural language patterns (e.g. "find people who **are friends with** someone **who lives in** London"). The key idea is that if your data forms a network, a graph DB can retrieve connected information very efficiently by following links, rather than doing multiple table joins.

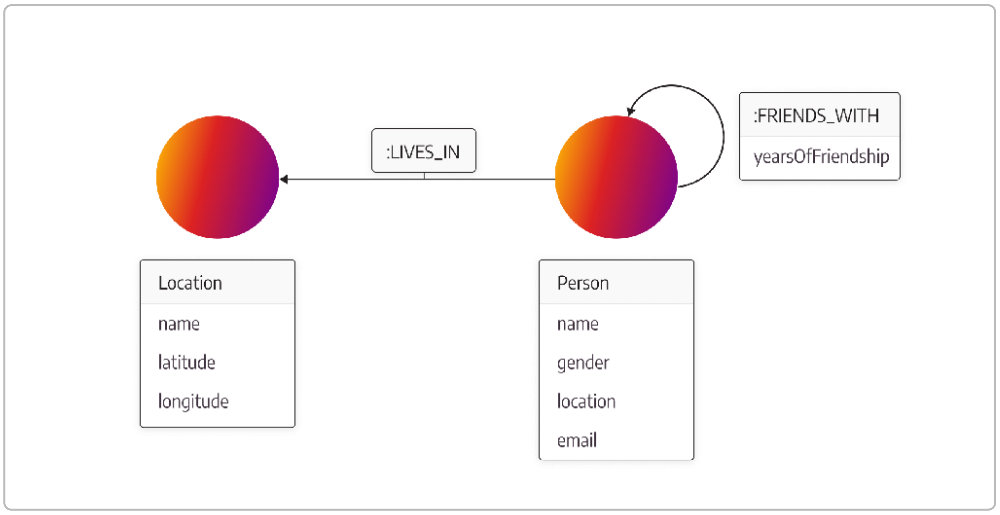

Illustration: A graph data model example. Here we see **Person** nodes and **Location** nodes connected by relationships. Each Person node stores properties (name, gender, etc.), and we have an edge **FRIENDS_WITH** linking two Person nodes (with a property for yearsOfFriendship on that relationship), as well as a **LIVES_IN** relationship from a Person to a Location. In a graph database, these connections are stored directly as part of the data. This is analogous to a mind-map or network diagram of your information – every piece of data (node) can be linked to others without needing a join table, and you can add new types of links easily.

Advantages of Graph/Multi-Model DB for your use case:

- **Naturally Models a "Second Brain":** A second brain often isn't just rows in isolation – it's a web of interrelated ideas, notes, tasks, people, etc. Graph databases **excel at highly connected data** where relationships are central [10]. If you have a note that links to another note, which is part of a project that involves a person – in a graph DB these are just direct connections that you can traverse. Queries like "find all notes related to Project X" or "who are the people connected to my current task and how?" become straightforward because the database is built to **follow hops** through the graph efficiently [7]. Traditional SQL could answer these, but with more complex queries (multiple joins or recursive logic); a graph DB can often do it in a single, simpler query because relationships are first-class citizens [7]. In short, if you imagine your knowledge base as an interconnected network (which a "second brain" essentially is), a graph-oriented store fits that mental model directly.

- **Flexible, Evolving Schema:** Graph and document-based databases are schema-flexible. You don't need to predefine all your entity types and properties at the start. You can add a new type of node or a new property to some nodes without affecting unrelated parts. For example, you might start storing basic notes and links. Later, you decide to add a "priority" field to some notes – in a document/graph DB you can just start adding that field to those entries; other entries without it are fine. Or you introduce a new node type "Idea" or "Goal" and start linking it – you don't have to restructure a bunch of tables. This flexibility is noted as a key benefit: **"with the flexible graph database schema, new properties can be added on the fly"** [5]. If your data model changes often or if you anticipate discovering new categories/relationships as you use the system, this is a huge plus. Your foundational "universal database" can grow organically with your thought processes, rather than you hitting a wall and needing to do a big migration each time.

- **Unified Storage of Various Data Types:** A multi-model database like **ArangoDB** is capable of storing documents (like JSON objects), key-value pairs, and graph edges all in one engine [11]. For your second brain, this means you could store structured records (say, a contact with name, email, phone), unstructured content (maybe a JSON representing a free-form note), and the relationships between them, all in one place. You don't have to maintain separate systems for structured versus unstructured data – everything can live in the same database and be queried together [11]. This "all your logic in one place" is essentially what you described as a *foundational universal database*, and ArangoDB's multi-model nature is built for that use case [12].

- **Efficiency in Relationship Queries:** Because graph DBs store connections explicitly, querying paths and networks tends to be faster and more direct for large, deeply connected datasets. They avoid the performance issues that SQL databases face with many-table joins or recursive queries [8]. For example, finding all indirect connections (like a friend-of-a-friend-of-a-friend in a social graph) can be done by traversing the graph, which is what these systems are optimized for [8]. If your second brain grows to thousands of nodes with complex links, a graph database can handle those "find connections" queries more gracefully. (It's worth noting for a *small* dataset, performance isn't a huge concern either way – but as it grows or if your queries get complex, the graph DB scales those particular patterns better.)

- **Scalability and Future Cloud Offloading:** Modern graph/multi-model databases often have built-in support for clustering and horizontal scaling. For instance, ArangoDB supports sharding (spreading collections of documents across servers) and replication for scaling out [9]. This means if someday your project becomes a multi-user cloud app, you could deploy ArangoDB in a cluster and distribute the load. In contrast to a single-file SQLite or a lone Postgres instance, a clustered ArangoDB could handle many concurrent users or a massive knowledge graph by adding servers [9]. In summary, **Option 2 aligns well with the "start local, then possibly scale out" plan** – you can run it on your home machine now, and later use the same tech on cloud servers to serve more people, without completely changing database paradigms. Your data model (nodes & edges) would remain; only the deployment would change (single-node to cluster).

- **Graph Analytics Possibilities:** This might be a bit beyond your immediate needs, but it's worth noting: once your data is in a graph form, you can leverage graph algorithms for insights (e.g. finding clusters of related ideas, important central nodes, etc.). For example, you could use a *PageRank* algorithm on your knowledge graph to see which idea or note is most linked/influential, or use path-finding to assist in recommendations ("this task is similar to that project you did before"). Relational databases don't natively offer these algorithms, whereas graph ecosystems do have this capability if you ever need it [13]. It's not something to worry about now, but it speaks to the richness of what a graph representation enables in the long run.

Drawbacks/Considerations for Graph/Multi-Model DB:

- **Learning Curve and Tooling:** Graph databases introduce new query languages or patterns that you might not be familiar with. Instead of SQL, you might use **Cypher** (Neo4j) or **AQL** (ArangoDB's query language). These are powerful and designed for graph operations, but as a beginner you'd have to learn their syntax and way of thinking [14]. SQL, by comparison, is more ubiquitous. That said, languages like Cypher are often described as quite intuitive (almost sentence-like for relationships) [15] [16]. It's a learning curve, but not an insurmountable one – just a point that using a less-common

DB means fewer out-of-the-box examples at your fingertips. The community for something like Neo4j is growing, but it's still smaller than the SQL community; ArangoDB's community is smaller than PostgreSQL's [2]. This means you might find fewer beginner tutorials specifically for Arango, but their documentation is pretty good and there are active forums. Overall, be prepared for a bit of initial overhead learning the database's tools and query style.

- **Setup and Maintenance Overhead:** Unlike SQLite which is zero-config, a graph or multi-model database will run as a server process. For example, you'd install ArangoDB and start its service, which then remains running to serve queries. On a personal computer, this is usually straightforward (ArangoDB provides a web interface for administration which is convenient), but it is a heavier footprint than an embedded library. It will consume some memory/CPU while running in the background. If you're truly going off-grid on minimal hardware (say a Raspberry Pi or a battery-operated setup), note that something like ArangoDB will use more resources than SQLite. In practice, on a normal modern PC, this shouldn't be an issue – just something to consider if **"free of the grid"** for you also implies minimal power usage or very lightweight environment. On the plus side, since it's a service, multiple applications or scripts can connect to it concurrently via network/API if you wanted to (enabling, for example, both a local app and a mobile app to talk to the same DB on your home server). But you'd need to ensure the service is always running when you need your data. In contrast, with an embedded DB, the data is accessed by the app directly and doesn't require a separate process.

- **Data Consistency and Complexity:** While graph DBs can be ACID compliant on a single node, some trade-offs might appear in distributed mode or with certain operations. Relational databases are extremely strict about consistency by design; some NoSQL or multi-model systems prioritize flexibility and speed, potentially easing some consistency guarantees in certain scenarios. For a single-user local setup, this is rarely a problem – ArangoDB, for example, supports transactions on a single instance (and even multi-collection transactions) to ensure consistency. But it's worth noting that the **relational model's simplicity can be conceptually easier to ensure consistent updates**. With a graph, you have to think about maintaining relationships (though the DB won't let you have an edge to a missing node thanks to how they store relationships). This is a minor point, but for completeness: if absolute robustness with complex transactions is needed, relational still has an edge in proven track record. That said, your use case doesn't seem to involve extremely tricky transactions, so a graph DB should be fine on consistency for normal use.

- **Portability and Lock-In:** If you start with a graph database and later decide to switch, it's not as straightforward as moving from one SQL database to another. SQL databases share a common language standard (with minor differences), so migrating from, say, SQLite to PostgreSQL is doable via export/import. Graph databases each have their own query languages and data formats. There's no universal standard (though efforts like GQL are in the works). If in the future you wanted to move your data out of ArangoDB to another system, you'd likely have to write a custom migration or at least export all data into a neutral format (JSON, CSV, etc.) and reconstruct relationships in the new system. This isn't a reason to avoid them, but it means you are somewhat **committing to the technology**. Choosing a popular open-source graph DB mitigates risk (your data is still yours and can be extracted), but it will take more work to migrate than with standard SQL. Given that you want a long-term "second brain," it makes sense to pick the solution you intend to stick with for a long time, so this is just a consideration in making that initial choice.

Comparing the Options Head-to-Head

To summarize the key differences with your goals in mind, let's compare Option 1 (Relational) and Option 2 (Graph/Multi-model) on important points:

- **Data Model & Flexibility:** A relational database is like a collection of structured spreadsheets; you design the structure (tables/columns) beforehand and it's very organized. This is great for consistency but less adaptable to change. A graph or multi-model database is like a flexible mind-map; you can add new nodes or connections without upfront schema changes [5]. For a "universal" knowledge base that might evolve, the graph approach offers more freedom to accommodate new ideas and link types.

- **Handling Relationships:** In the relational approach, relationships are indirect – you link data via keys and join tables. The more links you have, the more complex the joins can become, which can slow down queries on deeply connected data [8]. In a graph approach, relationships are stored directly and traversing many connections (hops) is what it's optimized for [7]. If your second brain relies on exploring connections (which it likely will), Option 2 is purpose-built for that pattern.

- **Ease of Use & Learning:** SQL (Option 1) is widely used and you'll find plenty of resources. As a beginner, you might find comfort in the sheer number of guides available. Graph query languages (Option 2) are newer and different. However, many find languages like Cypher quite intuitive for expressing relationships, sometimes even more so than SQL for those specific queries [16]. There will be a learning phase regardless of option (since database tech is new to you), but consider whether you prefer to learn the "classic" method first or jump into a graph mindset. Neither is "wrong" – it's about which aligns with how you think of your data. Given you envision your data as one interconnected system, learning the graph way might actually feel natural once you get past the initial unfamiliarity.

- **Off-Grid Implementation:** Both options can run completely locally without internet – satisfying the "free of the grid" requirement. Option 1 (relational) can be as simple as a file (SQLite), which is extremely portable and even **battery-friendly** for embedded use [3]. Option 2 (graph) will involve running a database server program on your machine. It's still local (no cloud needed), but it's a background process to manage. If you are comfortable with a one-time setup (installing the DB and starting it), this is not a big hurdle. Many graph databases also come with user-friendly desktop versions (for example, Neo4j Desktop gives you a GUI to manage the DB locally). So **both** can be self-sufficient on your hardware; just note that the relational (SQLite) route is the lightest-weight possible, whereas the graph route gives more capabilities at the cost of a bit more resource usage.

- **Community and Ecosystem:** Relational databases have an enormous ecosystem – countless tools for backup, visualization (you can use any number of SQL clients or admin tools), ORM libraries in every programming language, etc. If you foresee needing integrations or lots of community support, that's a plus for Option 1 [2]. The graph database community, while smaller, is quite enthusiastic and growing. You'll find active communities especially around popular ones like Neo4j. ArangoDB, being multi-model, has a smaller niche community but it's fairly robust and the company behind it provides good documentation. Since your use is a bit unique (personal second brain), either way you might end up building some custom stuff. There are not as many off-the-shelf "second brain" solutions using graph databases yet (most personal knowledge base apps use simpler storage), but that also means you could be on the cutting edge of something powerful. You might not rely heavily on community plugins/etc. beyond the database itself, so as long as the core tech is solid (which it is), community size is perhaps a secondary factor.

- **Scalability Path:** If "many people" end up using this (each with their own instance initially), consider how expansion might look. With relational/SQLite, each person could have their own DB file – that's fine but if you later want a centralized cloud service, you'd have to merge or move those into a bigger SQL server. It's doable, but requires data migration and a new architecture. With a graph DB like ArangoDB, you could design from day one such that each person's data is a subset of a larger graph or in separate collections, and then on the cloud you might run a multi-tenant Arango cluster. The same queries and structure would work, just on a larger scale. In other words, Option 2 might make it easier to **scale the unified knowledge model** to a multi-user system, since it's already designed for distributed scenarios [9]. Option 1 (especially if using SQLite) would need a re-architecture to scale – though if using PostgreSQL from the start, you could also simply host a Postgres instance per user or a big one for all users (with proper user separation). Both paths can scale, but the graph/multi-model option is architecturally ready for clustering if needed [9], whereas the relational option might require more work to scale out (relying on vertical scaling or complex partitioning).

Recommendation: Option 2 (Graph/Multi-Model Database)

Considering your goals – keeping everything in one **unified** database, being able to work offline now but potentially expand later, and capturing the richness of interconnections in your data – a graph or multi-model database is the better fit. **I recommend choosing a graph-oriented database (such as ArangoDB)** for your second brain project.

Rationale (in beginner-friendly terms):

- **Aligns with How You Think:** Your second brain isn't a bunch of separate spreadsheets; it's more like a big mind-map where everything connects. A graph database will let you store data *the way you naturally think about it*. You can literally have an entry for "Project Alpha" and link it to "Idea note from July" and to "Bob (person)" without jumping through hoops. Later, if you add "Location" or "Event" information, those too can be linked in easily. This **reflects real-world relationships directly**. You won't need to constantly redesign tables – the database will accommodate new link types or properties as you go [5]. For a dynamic personal knowledge base, that flexibility is gold.

- **One Database to Rule Them All:** You expressed a desire for a *foundational universal database* that holds personal, business, and other logic in one place. A multi-model DB like ArangoDB is literally built for that scenario [12]. You can keep all kinds of data together – structured entries, free-form notes, files metadata, etc. – and interconnect them. There's no need to split things into siloed databases or systems. This will make your life easier when you want to query or update across different domains of data. For example, a single query or view could pull a project, all notes and tasks under it, the people involved, etc., because they're all in one system.

- **Future-Proofing for Scale:** While the focus now is on local use, you mentioned possibly offloading to cloud for scaling. By choosing a solution that can scale out (ArangoDB can shard and replicate data across servers [9]), you're investing in a path that won't hit a dead-end if your project grows. If your second brain remains personal, you'll still enjoy snappy performance for complex queries because graph databases handle interconnected queries efficiently. If it evolves into something more (for example, a collaborative knowledge base for a team or a product that others use), you can deploy the same database design on a bigger setup. In contrast, starting with something like SQLite would give simplicity now but would require a significant change later to support multiple users or large data. **Option 2 gives you a smoother growth path** – basically "start small, think big."

- **Maintaining Independence:** Both options keep you off the grid initially, but Option 2 does so while also enabling richer functionality. You won't be forced onto a proprietary cloud knowledge graph service or anything – ArangoDB and Neo4j Community are open-source, so you remain in control of your data. It's true that running a graph DB is a bit more involved than a file-based DB, but since you're aiming for a somewhat ambitious personal system, the trade-off is worth it. You'll be able to do more with your data *on your own machine*. Also, because the data is structured as a graph, if you later integrate an AI or advanced analytics, having those explicit relationships can improve results (for instance, an AI agent could traverse your knowledge graph to find relevant info). This synergy aligns with the idea of a "second brain" that can actually understand relationships, not just store facts.

- **Beginner-Friendly (once set up):** While relational databases are traditional, a newcomer might actually find the graph approach more intuitive for this specific use case. You don't have to normalize everything into many tables or think in terms of rows and IDs. You can start by thinking: "What are my important entities? What links to what?" – which is a fairly natural thought process. The queries you write (in AQL or Cypher) will directly use those relationships. For example, a query might be "find all notes that **referenced** 'Project Alpha' in the last month" – in a graph query language, that's one clear query following the `REFERENCES` edges from the Project node. In SQL, you'd have to JOIN the Notes table with a References table, filter by project ID, etc. Neither is magic, but many find the graph style more readable for such tasks [17]. And if you're ever visualizing your data, graph databases often come with visualization tools that show nodes and connections, which can be a great UI for a second brain (imagine seeing your knowledge as a constellation of nodes). This could provide a more **user-friendly experience** for exploring your own data compared to rows in a table.

Closing Thought: By choosing a graph/multi-model database, you're essentially setting up a system that mirrors the way knowledge connects in real life. It gives you flexibility now and power later, all while keeping you in full control of your data off-grid. Relational databases are fantastic for many applications (they're like the sturdy, formal file cabinets of data), but for a personal second brain that needs to be both free-form and deeply interconnected, a graph-based approach is the more **innovative and suitable choice**. It will require learning some new tools, but the long-term payoff in capability and adaptability will serve your needs better.

Lastly, remember that no choice is absolutely final – but in this case, starting with the right foundation will save you a lot of restructuring down the road. Given all the above, **Option 2 is the singular choice that best aligns with your vision** [12] [11]. You'll be building your second brain on technology designed for connectivity and growth, which is exactly what your project calls for.

Sources:

- Achala Sharma. *"Different Types of Databases…how to choose right database."* (Highlights use-cases for relational vs graph vs multi-model) [18] [1] .
- *Memgraph Blog – Graph Database vs Relational Database.* (Explains how graph model handles relationships and schema flexibility compared to relational) [5] [19] .
- Volodymyr Pavlyshyn. *"Do We Need Graph Databases for Personal AI…?"* (Discusses graph DB advantages for highly connected data and complexities to consider) [7] [14] .
- **ArangoDB vs PostgreSQL** – StackShare Comparison. (Outlines differences: data model flexibility, scalability, community, etc.) [11] [9] .
- **Personal AI/Second Brain Discussion** – *Medium article on Personal Knowledge Graphs with SQLite (libSQL)*. (Emphasizes the importance of embeddable, single-file DB for local apps) [20] .
- *FreeCodeCamp – Graph DB vs Relational DB.* (Beginner-friendly walkthrough of modeling and querying differences, confirming graph DB strengths in changing schema and connected queries) [10] [6] .

---

[1] [12] [18] Different Types of Databases Model and how to choose right database for your requirement | by Achala Sharma | Medium
https://medium.com/@nimma0701/how-vector-db-helps-in-rag-llm-4c26d0f6fdd6

[2] [9] [11] ArangoDB vs PostgreSQL | What are the differences?
https://stackshare.io/stackups/arangodb-vs-postgresql

[3] [20] Personal Knowledge Graphs in AI RAG-powered Applications with libSQL | by Volodymyr Pavlyshyn | Artificial Intelligence in Plain English
https://ai.plainenglish.io/personal-knowledge-graphs-in-ai-rag-powered-applications-with-libsql-50b0e7aa10c4?gi=7e900e8c55da&source=post_page-----574e98b9eea1---------------------------------------

[4] [5] [6] [8] [10] [13] [15] [16] [17] [19] Graph Database vs Relational Database
https://memgraph.com/blog/graph-database-vs-relational-database

[7] [14] Do We Need Graph Databases for Personal AI and Agents? | by Volodymyr Pavlyshyn | Artificial Intelligence in Plain English
https://ai.plainenglish.io/do-we-need-graph-databases-for-personal-ai-and-agents-574e98b9eea1?gi=b4d95d23af39