ChatGPT

Why Cloud Storage is a Great Database (and Why I Use It)

Introduction

When we think about why we need a database, it boils down to storing and accessing data efficiently. Typically, we either **access data by a known identifier** or **query data to find what we need**. In my experience, modern cloud storage services (like AWS S3, Azure Blob Storage, Google Cloud Storage, etc.) can act as an excellent **key–value database** for many use cases. In fact, a file system itself is essentially a simple database: it's a hierarchical key-value store where the **key** is the file path and the **value** is the file's content. This means that cloud file storage can often fulfill the same needs as a NoSQL key-value database [1]. In this document, I will explain **why I consider cloud storage "a good database"**, describing its advantages in scalability, cost, maintenance, and even how it supports advanced patterns like graph structures, versioning, and provenance.

Filesystems as Key-Value Databases

Fundamentally, a traditional file system is already a kind of database – specifically a key-value store. The directory path and filename serve as a unique key to locate data, and the file's contents are the value. Cloud object storage services follow this model with a flat namespace: each object is stored with a unique key (often a path-like identifier) and associated data [1] [2]. For example, Amazon S3 is described as "a very large key–value store: the key is the filename, the value is the contents of the file" [1]. If your requirements are simply to **store a value and retrieve it by key**, cloud storage works brilliantly as a database. In fact, even Amazon's own e-commerce platform archives old order records in S3 for read-only access, leveraging S3's low cost storage for data that only needs simple key-based retrieval [3].

One huge benefit of using cloud storage as a datastore is that **it's universally available in any environment where you have compute**. Every major cloud provider offers robust distributed storage as a core service, typically with very high durability and availability. This means wherever your code runs (a VM, a container, a serverless function), it can read and write to cloud storage without you having to run or maintain any database server. The storage service itself handles durability by keeping multiple copies of your data behind the scenes [4]. For instance, object storage systems replicate data across multiple servers or zones, achieving **"enhanced durability"** so your data remains safe and highly available [4]. In short, cloud storage gives you a **serverless, fully-managed key-value database** out of the box.

Eliminating Server Maintenance (Pets vs. Cattle)

Using cloud storage as your database removes the burden of maintaining a dedicated database server. In DevOps parlance, running your own database is like taking care of a **"pet"** – a special server that requires constant care, feeding, and monitoring. By contrast, treating storage as a service makes your data layer more like **"cattle"** – easily replaceable and managed at scale by the cloud provider. You no longer have to worry about patching database software, monitoring uptime, or scaling the DB server. The cloud storage service is **stateless from your perspective** – nothing is running until you make a request, and there's no long-lived server process that can fail at 2 AM. This stateless nature is ideal for serverless architectures: if your application is idle, you pay nothing, yet the data is always there, ready to be accessed on-demand.

Another big pain point that evaporates is **replication and backup**. Traditional databases require careful setup for replication (master-slave, clusters, etc.) and routine backups to prevent data loss. In contrast, cloud object storage inherently replicates data across infrastructure. For example, S3 and similar services typically store multiple copies of each object (often across different availability zones), **ensuring both high availability and data integrity** without any effort on your part [4]. You also get features like point-in-time versioning out-of-the-box: you can enable versioning on a bucket so that older versions of objects are retained automatically, protecting against accidental deletions or overwrites [5]. All of this means **far less administrative overhead** compared to running a conventional database. The storage system itself is highly redundant and durable, so you don't need to manage failover servers or schedule backup jobs – it's handled by the platform.

Scalability and Cost Efficiency

A key advantage of cloud storage over traditional databases is **massive scalability at low cost**. Object storage can grow essentially without limit – you can store petabytes or more, and the flat object namespace can easily handle billions of keys [6]. There's no complex schema or indexing overhead that slows things down as the dataset grows; you simply add more objects. In fact, object stores are explicitly designed for **"unlimited growth to petabytes"** of data, whereas many databases run into scaling challenges beyond a certain size [6]. Scaling a relational or NoSQL database often means sharding data across nodes or investing in bigger hardware, which increases complexity and cost. By contrast, with cloud storage you **"pay for what you use"** – storage costs scale linearly with the volume of data, and you don't have to provision or pay for excess capacity in advance [7].

Importantly, cloud object storage is **much cheaper per gigabyte** than high-performance database storage. Storing large volumes of data in a managed database service or on SSD-based database servers can be expensive. Cloud storage, built on commodity hardware with multi-tenancy, spreads out the cost. As a result, **raw storage costs have become so cheap** that in budgets they're almost negligible compared to compute costs [8]. For example, Amazon S3 storage costs are a fraction of the costs of running an equivalently sized database instance. One Stack Overflow discussion noted that while S3 is slower for lookups than Amazon's DynamoDB (a NoSQL DB), **S3 costs significantly less for storage** – making it very economical for large datasets or infrequently accessed data [3]. This cost efficiency means you can afford to retain **much more data** in its original form (as files or objects) without worrying about an exploding database bill. Many object storage systems also offer tiered storage classes (from hot, frequently accessed storage to cold archival storage) so you can optimize costs for your data's access patterns.

Performance-wise, although object storage doesn't match an in-memory database for rapid random access, it's surprisingly effective for a wide range of workloads. Modern cloud storage can deliver high throughput and, with recent enhancements, even low latency for certain operations. For instance, AWS introduced *S3 Express* storage class which is **purpose-built to deliver consistent single-digit millisecond latency** for access, a 10× improvement over standard S3 in some cases [9]. This shows that cloud storage performance continues to improve, narrowing the gap between object stores and traditional databases for many scenarios. In practice, if your access pattern is reading/writing whole objects (documents, JSON blobs, images, etc.) by key, the latency of standard cloud storage (tens of milliseconds) is often acceptable – and if you need faster, options like S3 Express One-Zone can get first-byte latency down to ~10 ms [9] [10]. Meanwhile, **throughput** for large data streaming from object storage can be extremely high (since you can parallelize reads of many objects). The bottom line is that for many large-scale or content-centric applications, cloud storage provides **sufficient performance at dramatically lower cost**, especially when combined with caching strategies for hot data.

Decoupling Compute from Storage

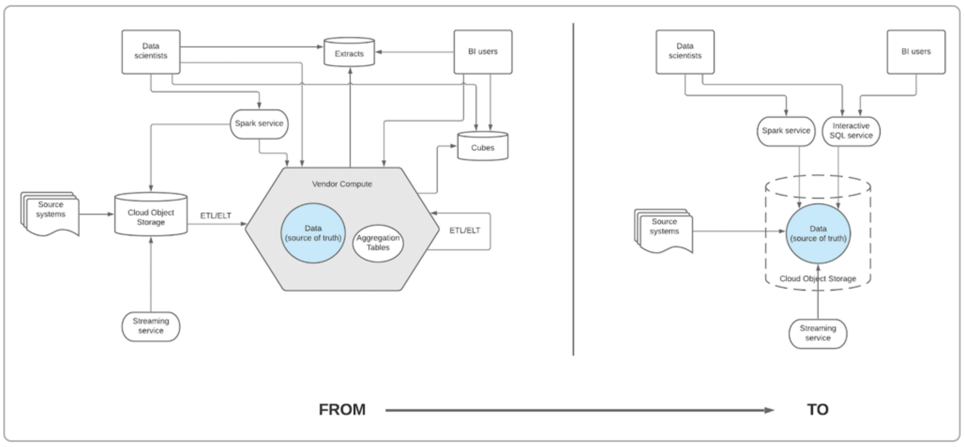

Figure: An illustration of decoupling storage from compute. Left: Legacy approach – data is tightly coupled inside a single database or data warehouse, requiring that system to manage storage, compute, and scaling. Right: Modern approach – data is stored in a cloud object storage "lake," accessible by various ephemeral compute engines (SQL query services, Spark, AI/ML tools) on demand. This separation allows each component to scale independently and reduces reliance on monolithic databases [8] [11] .

One of the most powerful aspects of using cloud storage as a database is the **clean separation between data and compute**. In traditional architecture, your database server is both a storage engine and a compute engine (handling queries, indexes, transactions, etc.). This means scaling up your database for more data also often involves scaling compute, and vice versa. Cloud storage flips that model: storage lives in one layer, and compute (processing power) lives in another, and they communicate over a network. This **decoupling of compute and storage** brings several benefits:

- **Independent Scaling:** You can scale your storage capacity independently of your computing power. If you suddenly need to store 10× more data, you simply put it in object storage (which can expand essentially infinitely [6]) without having to scale any servers. Conversely, if you need more compute for intensive processing, you can spin up temporary processing clusters (or serverless functions) without moving data around – they all access the same centralized storage. This independent, elastic scaling was a *"critical step forward for efficiency"* in cloud architecture [8] . It means you pay for heavy compute only when needed, and you're not forced to keep an oversized database server running 24/7 just in case.

- **Cost Savings:** As noted earlier, separating storage and compute often lowers costs. **Raw storage became so cheap** in the cloud era that it's practically "free" relative to other IT costs [8] . Meanwhile, compute can be treated as a pay-per-use utility. By isolating compute costs, you only pay for actual query or processing time, which **further lowers overall costs** compared to monolithic databases that bill you for both storage and CPU together whether you use them or not [8] . Many cloud analytics services (like AWS Athena, Google BigQuery, Snowflake, etc.) embrace this model by charging separately (or primarily) for the amount of data scanned/ processed, while the data can reside cheaply in cloud storage.

- **Flexibility and Choice:** With data in a universal storage layer (often called a **data lake** when used this way), you aren't locked into a particular vendor's database engine for all uses of the data. You can freely bring different tools to analyze or process the same dataset. For example, you might use a SQL query service for reporting, a Spark cluster for machine learning, and a custom Python script for ad-hoc analysis, all reading from the same files in storage. If the data were inside a traditional database, you'd be stuck using that database's query interface (and paying for it). Decoupling thus **"allows for universal data access from unlimited services and applications"** because the data is in open formats on the storage layer [12]. This freedom to choose best-of-breed processing tools is a big win for agility.

- **Higher Availability and Simpler Recovery:** When compute and storage are separate, a failure in your compute layer doesn't threaten your data integrity – you can always spin up a new compute instance and point it at the same stored data. There's no need to restore databases from backups in many scenarios, since the data persistently lives in storage. This also makes it trivial to clone an environment: for example, starting a test analytics cluster against production data stored in S3 is straightforward, whereas doing that with a production database might require complex cloning or replication.

The industry trend toward **data lakes and lakehouse architectures** is essentially built on these principles. Store all your data (structured *and* unstructured) in the cloud storage layer, and then **load data on demand** into whichever processing engine needs it. This is exactly how I treat my file storage – as a **serverless database** where data is loaded into memory only when needed for computation, then released when done. It's analogous to how a data lake query engine works: you might have terabytes of log files or JSON in S3, and a service like AWS Athena or Presto will scan only the necessary files when you run a query, with no always-on database in between. By designing my applications to load files from cloud storage into memory on demand, I achieve a very similar effect: *nothing is running until a query or operation is initiated*, at which point the needed data is fetched and processed, and then compute resources can scale back down.

On-Demand Data Loading and Processing

Using cloud storage as a database encourages a pattern of **on-demand data loading**. Rather than having a single, long-lived database service that holds all your data in memory or on disk, you perform *"compute in bursts"* against the data in storage. In my own implementations, I created a caching layer and a "serverless" graph query engine that pulls data from cloud storage (or even from compressed archives or SQLite files stored in cloud storage) into memory only when needed. This design is powerful because it leverages the fact that **cloud storage throughput is high** and **startup time for loading data is low** enough that you can do it dynamically. For example, reading a few hundred MB or even a few GB from S3 into an in-memory structure is typically a matter of seconds. Once in memory, you can run complex queries or graph algorithms very fast. After processing, you can discard the data from memory, knowing it persists safely in the storage layer.

This approach has several implications. First, it means **you only pay for compute when you actually use it** – a big advantage of the serverless philosophy. If no queries are running, you're not paying for an idle database server. Second, it enforces a **clean separation of state**: the authoritative state is in the file storage, and your compute instances are effectively stateless workers that can scale out as needed. This makes things like horizontal scaling or multi-region deployment much easier, since workers don't carry local state that needs synchronization; any worker can fetch the latest data from the central storage. It also simplifies *deployments and updates*: you can destroy and recreate your compute environment (cattle, not pets) at any time without data loss, since data lives in the external store.

Another benefit is that **upgrading or changing your data processing technology becomes easier**. Suppose you have your data in files (say Parquet, JSON, CSV, etc.) on cloud storage, and you've been using one framework to process it. Tomorrow, a new faster library or a whole new paradigm (like a specialized ML engine) comes along – you can point it at the same data files and start using it, without complex migrations. Your data is not locked inside a proprietary database format. This future-proofs your architecture and lets you take advantage of innovations in data processing while keeping the data lake as the consistent source of truth.

Of course, the **trade-off** with on-demand loading is that queries might have a bit more latency for cold data, since you pay the cost to load from storage. However, this can be mitigated with caching (keeping recently or frequently accessed data in memory or local disk) and by using the aforementioned high-performance storage classes if low latency is critical. In practice, I've found that many queries do not suffer noticeably, especially when the working set of data is not enormous. And for truly latency-sensitive use cases, one could always keep a cache or use an in-memory database as a frontend while still leveraging cloud storage as the system of record.

Building Graphs and Relationships on Files

One might wonder: can you represent complex relationships or do ad-hoc queries without a traditional database? The answer is yes – you can build **graph structures and indices on top of files**. A file-based approach doesn't mean you are limited to simple key lookups; it just means you implement relationships at a higher level. In my use, I often create **hyperlinks between files** to indicate relationships, very much like the World Wide Web uses hyperlinks between documents. For example, one file can contain references (paths or IDs) pointing to other files. This effectively creates a graph or network of data items. By traversing these references, you can query relationships. In fact, this is conceptually similar to how some graph databases work (storing pointers to related nodes), except here the "nodes" and "edges" are represented as files and paths.

The **file-as-graph** idea becomes really powerful for building knowledge bases or interconnected datasets. You can maintain a folder of files where each file is a node (with content) and it lists links to other files (its neighbors in the graph). This could be in any domain – for instance, a file could represent a user profile linking to files representing that user's transactions, which link to files representing products, and so on. With a bit of code or an in-memory index, you can traverse and query this graph without needing a specialized graph database engine. The overhead of following a link is just reading another file or looking up another key in storage.

Crucially, using files for this allows **much easier versioning and provenance tracking** than a typical database. Each file is an immutable record (if you choose to treat it that way) of some piece of information, and you can store multiple versions of that file over time. Because files are discrete units, it's straightforward to keep an *audit trail* of changes – either by using cloud storage's object versioning or by creating new files with timestamped names. In a graph of files, you could even link a node to its previous version (forming a chain of historical states). This kind of **fine-grained version control** is invaluable in scenarios like machine learning or generative AI systems where you need to explain why the system did something. You have a permanent record of the data (and even the intermediate results) that led to an outcome, each stored as files that can be examined. In an AI agent context, files can serve as the "memory" of the agent – the agent can append new facts or observations as new files, link them to related knowledge, and thus maintain a transparent chain of reasoning.

Speaking of provenance and explainability: because data in a file-based system is not hidden behind a monolithic binary blob (as is often the case with database files), it's inherently **more transparent**. You can open any file and inspect it. You can attach rich metadata to each file (object storage allows custom metadata tags, or you can embed metadata in file content) to record where it came from, when it was last updated, etc. This aligns well with the needs of AI systems that must be able to justify their outputs by pointing to source data. It’s much easier to say “here is the document (file) that provided the information for this answer” when your data store is literally documents, rather than saying “I have it somewhere in my database”.

From a **data modeling perspective**, using a filesystem as your database encourages a schema-on-read approach. You can mix structured and unstructured data, and evolve schemas incrementally. A commenter on OSNews aptly noted that in databases your data must often **conform to the structure of the database**, whereas with files the *structure adapts to your data* [13]. This means you can start with loosely structured data in files and gradually add structure or new relationships as needed, without costly migrations. The filesystem doesn’t enforce a fixed schema upfront. This flexibility often leads to data models that are more aligned with the real-world business domain, rather than contorting the domain to fit a rigid table structure. If a certain type of record no longer fits your old schema, you can just change what’s in the file or how you organize files, without needing to rebuild a big database schema. **Refactoring your data layout** can be as simple as moving files to a new folder or splitting a file into smaller ones – operations that are trivial in a filesystem, but could be very complex in a huge database.

Direct Access and Integration Benefits

Another advantage of using cloud storage as the data hub is how easily it integrates with other systems and allows direct access patterns. Because cloud object stores are typically accessible via HTTP(S) and REST APIs, you can often serve data directly to users or client applications from storage, **bypassing your server** altogether when appropriate. For example, with AWS S3 you can generate a *presigned URL* that gives a client temporary read access to a specific object. The client can then download that file straight from S3, rather than your application server fetching it and relaying it [14]. This is hugely efficient for things like serving images, videos, or documents in a web app – essentially using the storage service as a global content server (often backed by a CDN for even faster delivery). Many cloud providers also allow mounting object storage or providing direct links, turning your storage bucket into a quasi web server for static content. The benefit is reduced load on your application and often faster delivery, since the cloud storage endpoints are optimized for throughput and can handle many concurrent requests.

Additionally, treating storage as the database means **language and platform independence**. Any environment or programming language that can call a REST API or use an SDK can interact with your data. You’re not tied to a specific database client library or driver. This makes polyglot development easier – one microservice could be in Python, another in Node.js, a third in Rust, and all simply read/write to the same storage bucket using HTTP calls. The learning curve is minimal (CRUD operations on files) compared to learning the intricacies of a particular database system.

It’s also worth noting that many cloud services implicitly use this approach under the hood. There are “serverless database” or query services that, behind the scenes, are **storing data in files and using ephemeral compute** to provide query functionality. For example, some managed services will store JSON records in an object store and on query will spin up a transient engine (maybe even loading data into a SQLite or an in-memory database) to execute the query, then shut it down. Instead of letting those services hide the pattern, you can embrace it directly: design your system such that you **“do it by design, not behind the scenes.”** By explicitly using files as your source of truth, you remain in control and avoid black-box services that might abstract things but at the cost of flexibility or cost efficiency.

Considerations and When to Use a Database

All that said, using cloud storage as a database isn't a silver bullet for every scenario. It's important to understand the **trade-offs and limitations**. Cloud object storage is eventually consistent and does not support multi-object transactions or complex queries by itself. If your application truly needs **ACID transactions** spanning multiple pieces of data, or needs to perform **complex queries (like JOINs, aggregations across many records, etc.) in real-time**, a traditional database or a specialized query engine might be necessary. For example, **relational databases excel at exploiting relationships** among data with optimized join algorithms – whereas if you store everything as files, you might have to write custom code to perform equivalent operations, which could be slower for real-time queries. One commenter put it succinctly: if you don't have complex relational data needs, a filesystem can suffice, but *"when you do need transactions or a query DSL, then you don't really have a choice"* but to use a database [15]. In other words, **know your requirements**: for simple key-value lookups, object storage is great; for analytics on huge data, object storage + external query engine is great; but for high-frequency transactional updates or arbitrary queries, a database may still be the appropriate tool.

Performance is another consideration. As discussed, object storage has higher latency per operation than an in-memory or on-disk database on a local server. If you have a workload that does **tons of small reads/writes (especially under 4KB each)** or needs sub-millisecond latency, you might find a traditional or NoSQL database (or even an in-memory cache like Redis) more suitable. In fact, for very small objects, the cost model of cloud storage can be less efficient because you pay per request; one AWS expert noted that for payloads around 4KB, DynamoDB (a NoSQL DB) could actually be cheaper and faster than S3 when request costs are factored in [16]. Also, while cloud storage can scale to high throughput, it has limits on request rates per prefix, etc., that you need to be mindful of if you're approaching thousands of operations per second. Many of these limits can be designed around (e.g., using key naming best practices to avoid hot spots), but a high-throughput transactional system might be better served by a purpose-built database.

However, for the majority of applications I work with, these constraints are not a deal-breaker. The pattern of *load-process-unload* works efficiently for analytics, batch processing, content management systems, AI knowledge stores, and more. And cloud providers continue to innovate to narrow the gap – features like S3 Express One Zone (for latency) and improvements in request parallelism mean object storage is becoming faster and more interactive [9] [10]. There are also hybrid approaches: for instance, using a lightweight indexing service or a cache in front of object storage. One could store data in S3 but keep an **index in memory** (perhaps using a tool like Redis or an ElasticSearch for search queries) to quickly find which object to retrieve. This still retains the benefit of cheap durable storage for the bulk of data while adding just enough database-like functionality for the specific query patterns needed.

In summary, the decision isn't an all-or-nothing one. You can combine cloud storage with selective use of databases or caches as needed. But leaning on cloud storage as the backbone offers simplicity and cost savings that are very attractive. As one technologist humorously remarked, *"Unless there are data relations to exploit, a relational database is just a tool looking for a problem"* [17]. That captures my sentiment: if you don't truly need the heavy machinery of a database, why incur its cost and maintenance? A well-organized set of files can often do the job.

Conclusion

Cloud file storage, in my view, makes for a **much better database** for many scenarios due to its simplicity, scalability, and low cost. By using the file system paradigm as a database, I gain a very clean separation between data and the application logic. The data lives in a durable, infinitely scalable store, and the compute can scale independently and run only when needed. This architecture is **cheaper to run**, easier to manage, and even encourages better data organization. Instead of cramming everything into one monolithic data store, I can structure my files in intuitive ways (by topic, by date, by user, etc.), which is often more aligned with how the business or project thinks about the data. It's trivial to move or refactor data in a file-based system – you're often just copying or renaming files – whereas refactoring a large database can be a nightmare of migrations and downtime.

Using cloud storage as a database also aligns perfectly with modern cloud-native and serverless trends. It treats persistence as an **infrastructure commodity** (provided by the cloud platform) and treats processing as ephemeral. This means *no state to maintain in the app layer*, which in turn means high reliability and resilience. If a server dies, no problem – spin up another and point it at the same data. If you need to globally distribute data, just replicate the storage (or use the provider's multi-region features), without having to set up complex database clustering.

Finally, the approach offers a path toward **data transparency and longevity**. Files are a universal format; they can live for decades and still be read, whereas proprietary databases may come and go. Files can be easily versioned, archived, shared, inspected – giving you confidence in the provenance of your data. The **flexibility** of a filesystem database means your data architecture can evolve organically, accommodating structured, semi-structured, and unstructured data side by side. This is increasingly important in an era of big data and AI, where you want to mix all kinds of data sources and still keep track of it all. With cloud storage as the foundation, I can build the indices, graphs, and queries I need on top, rather than being constrained by the capabilities (or costs) of a single database system.

In conclusion, cloud storage shines as a database when used thoughtfully. It's **not appropriate for every last use case**, but by recognizing it as *"an organized collection of data, stored electronically"* (which is the very definition of a database [18]), we unlock a powerful paradigm. We let the cloud handle the heavy lifting of persistence and scaling, and we as developers focus on extracting value from the data with whatever compute tools make sense. For many applications, this results in a simpler, more scalable, and more cost-effective solution than the traditional approach of spinning up a big dedicated database. It's the reason I reach for file storage first as my data store – more often than not, it just makes life easier while delivering the performance and capabilities that I need.

References (Sources)

- John Rotenstein, **Stack Overflow** – *"Amazon S3 effectively is a NoSQL database... a very large Key-Value store. The Key is the filename, the Value is the contents of the file."* [1]
- John Rotenstein, **Stack Overflow** – S3 is slower than a specialized NoSQL DB but *"certainly costs significantly less for storage!"* [3]
- Kushal Khandelwal, **DBSync Blog** – Object storage vs databases: Object stores offer *"unlimited scalability"* and *"pay-per-use, economical solutions for large datasets"*, whereas databases often have *"higher costs for large-scale storage"* [6] [7].
- Kushal Khandelwal, **DBSync Blog** – Benefits of object storage: *"Unlimited Scalability... Enhanced Durability: multiple copies of data across different locations ensure it stays available"* [4]. Also notes object storage's flat architecture easily scales beyond what databases can handle [19].
- Billy Bosworth, **Dremio Blog** – On separating compute and storage: after separation, *"raw storage costs became so cheap they were practically 'free'... Compute costs were isolated (pay only for what you use)... Independent scaling of storage and compute allowed on-demand elasticity"* [8]. Emphasizes the efficiency and flexibility gains of decoupling data from compute.
- Billy Bosworth, **Dremio Blog** – Further benefits of decoupling data: *"Extreme reduction in complex and costly data copies... open data formats allow universal access from unlimited services... future*

cloud services can access data directly instead of through a warehouse's proprietary format." [20]. This highlights how storing data in cloud storage (open format) avoids vendor lock-in and data silos.
- Jeffrey Lee & Brent Everman, **AWS Storage Blog** – *"S3 Express One Zone is a high-performance, single-AZ storage class purpose-built to deliver consistent single-digit millisecond first byte latency for latency-sensitive applications."* [9]. Demonstrates new cloud storage options achieving sub-10ms latencies (91% latency reduction vs standard S3 in one case [10]), closing the gap with traditional databases for read performance.
- AWS Documentation – **Presigned URLs** allow direct, secure access to S3 objects: *"grant time-limited access to objects in S3 without updating bucket policy… [the URL] can be used by a program to download an object"* [14]. This enables efficient direct data access patterns, offloading work from application servers.
- OSNews Discussion – *"Keep in mind that a file system is a database too. It isn't an SQL database and isn't relational, but there's no reason we can't unify file systems and databases under similar abstractions."* [21]. Commenter notes that filesystems already have database-like qualities and could adopt DB features like transactions.
- OSNews Discussion – *"Regular databases use a level of 'conformity' not found in regular file systems… in [databases] your data is made to adhere to the structure of your DB, and in [file systems] the structures were made to adapt to your data."* [13]. This emphasizes the flexibility advantage of file-based storage: the data's natural structure can dictate storage form, rather than forcing data into rigid tables.
- Hackaday (comment) – *"Unless there are data relations to exploit, a relational database is just a tool looking for a problem, not a solution."* [17]. An opinion that resonates with the idea of using simpler storage when complex relations/transactions aren't needed. This underlines the central thesis: use the right level of abstraction for the job, and often the filesystem is the perfectly adequate database for many needs.

---

[1] [3] [15] [16] amazon web services - Using S3 as a database vs. database (e.g. MongoDB) - Stack Overflow
https://stackoverflow.com/questions/56108144/using-s3-as-a-database-vs-database-e-g-mongodb

[2] [4] [6] [7] [19] Object Storage vs. Databases for Data replication
https://www.mydbsync.com/blogs/object-storage-vs-databases

[5] Retaining multiple versions of objects with S3 Versioning
https://docs.aws.amazon.com/AmazonS3/latest/userguide/Versioning.html

[8] [11] [12] [20] Separation of Compute and Data: Shift in Data Architecture
https://www.dremio.com/blog/separation-of-compute-and-data-a-profound-shift-in-data-architecture/

[9] [10] How Fetch reduced latency on image uploads using Amazon S3 Express One Zone | AWS Storage Blog
https://aws.amazon.com/blogs/storage/how-fetch-reduces-latency-on-image-uploads-using-amazon-s3-express-one-zone/

[13] [21] Computer files are going extinct – OSnews
https://www.osnews.com/story/130834/computer-files-are-going-extinct/

[14] Download and upload objects with presigned URLs - Amazon Simple Storage Service
https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-presigned-url.html

[17] [18] Linux Fu: Databases Are Next-Level File Systems | Hackaday
https://hackaday.com/2021/06/08/linux-fu-databases-are-next-level-file-systems/