ChatGPT

Using Databases as Data Projections, Not Primary Data Stores

Introduction

In traditional systems, databases are treated as the **single source of truth** – the primary storage location for application and business data. However, this approach has drawbacks: databases become monolithic, stateful repositories that are costly to scale and maintain. A new paradigm is emerging where databases are **no longer the master stores of data, but are instead *projections or views*** of data kept elsewhere. In this model, raw data is stored in durable storage (like data lakes or event logs), and one or many databases are populated from that source for specific query needs. This shifts the role of databases from being the sole guardians of data to being *purpose-built query engines* on top of a more scalable storage foundation.

Traditional Approach: Databases as Source of Truth

For decades, organizations have kept their only copy of critical data inside relational databases or similar DBMSs. The database was the master, and all reads and writes went into it. Backups existed, but the live database held the authoritative state. This approach often led to:

- **Single Point of Failure & Complexity:** The database holds *all* the data, so any outage or corruption is catastrophic. Scaling or migrating such a database is complex, since it contains a superset of all data needs.
- **Unnecessary Data Bloat:** Databases end up storing far more data than any one use-case needs. Historical or infrequently used data still consumes expensive database storage and resources.
- **High Costs:** Large databases require significant compute, memory, and licensing costs to maintain performance. Much of this cost is incurred even when the database is idle (an always-on server running 24/7) [1] [2].
- **Lack of Ephemeral Environments:** Because the database is stateful and long-lived, it's difficult to tear down or spin up on demand. This hinders modern *serverless* or on-demand computing scenarios. You can't easily run the database "just when needed" because it holds persistent state by definition.

In summary, the traditional model ties data *and* query engine together in one always-on system that must serve all purposes. As data volumes and variety grow, this one-size-fits-all approach becomes a liability.

New Paradigm: Databases as Data Projections

In the projection-oriented approach, **data is stored in a separate, durable storage layer**, and databases are used as **flexible, disposable query layers** built on top of that source. The database no longer *owns* the data permanently – it is a *generated view*. Key characteristics of this paradigm include:

- **Single Source of Truth elsewhere:** The primary copy of the data resides in scalable storage (for example, an object store like S3, a distributed file system, or an immutable event log). This is the authoritative source of data, but not necessarily in a query-friendly form.
- **Databases as derived views:** The database is populated *programmatically* from that source of truth. In other words, the database's tables or graphs are a **transformation or subset of the master data**. They might be loaded via batch processes, streams, or on-the-fly queries, but crucially, if the database content is lost, it can be **recreated** from the source. The database becomes a *cached projection*.
- **Multiple specialized databases:** Since the data is not locked inside one monolithic DB, we can create **multiple databases, each tailored to a specific domain or use-case**. For example, a relational database for transactional queries, a graph database for relationship analytics, and a full-text index for search – all derived from the same underlying data. This polyglot persistence approach lets each database focus on what it does best [3] [4].
- **Ephemeral and on-demand usage:** Because the source of truth is elsewhere, a database instance can be spun up when needed and torn down after use without data loss. This is akin to running a query engine on-demand. In fact, it mirrors serverless analytics services (like AWS Athena or Google BigQuery) that load data from cloud storage and execute queries without persistent DB servers [2]. In practice, one might launch a fresh database container, load the needed subset of data, run queries, export results, and then shut it down – no always-on database required [5].

This paradigm is a different way of thinking about data architecture. Instead of the database being the center of the universe, **data storage is decoupled from data query**. The database is demoted to a *servant* of the data, not its master.

Benefits of the Projection Model

Adopting databases as projections rather than primary stores offers several compelling benefits:

- **Cost Efficiency:** Storing large volumes of raw data in cheap storage (like cloud object stores or distributed file systems) is far more economical than keeping it in a high-performance database. You pay for expensive database compute only when running queries. As Dinis Cruz notes in the context of graph databases, *"Pay for processing only when it runs. No costs incurred when the database is idle (since it isn't running at all)"* [2]. This approach eliminates the need for an oversized always-on database, much like serverless query engines that spin up workers per query and charge only for runtime.
- **Scalability and Elasticity:** The storage layer can scale independently (often nearly without limit), and databases can be added or expanded based on demand. Workloads that are bursty or variable are handled gracefully by launching more database instances or processes on the fly [6]. If you suddenly need to run 50 complex queries in parallel, you could spin up 50 independent database instances to handle them, each working on a portion of the data – a horizontal scaling approach not feasible with one big monolithic DB.
- **Polyglot Persistence (Right Tool for the Job):** Different databases excel at different types of queries. In a projections model, you can use the optimal data store for each access pattern without data fragmentation concerns. For example, Netflix's *Unified Data Architecture* adopts a

"model once, represent everywhere" philosophy where data is modeled centrally but served through multiple systems optimized for various needs [7] [4]. Similarly, event-sourced systems use **polyglot read models**: the event log is the truth, and multiple read-optimized databases (SQL, Redis, Elasticsearch, etc.) are fed from it [3]. This leads to faster queries and more flexible feature development, since each projection can evolve independently.

- **Improved Data Quality & Testing:** When databases are built from a master dataset via code, it opens the door to treat data transformation as a repeatable, testable process (infrastructure as code for data). Each time you build or refresh the database, it's an opportunity to enforce schema, run data quality checks, and apply transformations consistently. Some have proposed treating *data tests* just like unit tests for code – automatically verifying the integrity of each projection on creation [8] [9]. This automation catches anomalies early and ensures that the derived database meets expectations.
- **Resilience and Reproducibility:** Since the pipeline from source data to database can be automated, you gain **reproducibility**. If a database instance crashes or its data becomes suspect, you can rebuild it from scratch. As one architectural guide puts it, *if a read model is lost or corrupted, simply replay the source data or events to rebuild it – no special backup restore needed* [10]. This capability makes the overall system more fault-tolerant. You no longer fear "what if the DB is lost?" – because it's a cache, not the source. Disaster recovery becomes a matter of regenerating projections.
- **Domain Autonomy and Faster Changes:** In traditional enterprise data, a single schema change can become an ordeal, requiring agreement across many teams (the dreaded "one big database" bottleneck). With multiple domain-specific databases, teams have more autonomy to evolve their projection for their needs without impacting others – as long as the underlying source data and contracts are stable. This **decoupling of schemas** can actually speed up development while still maintaining a unified underlying data model [11] [12]. Each domain can add new derived fields or indexes in its own database to support features, without a central schema committee.

Real-World Patterns and Examples

This projections-oriented philosophy isn't just theoretical – it builds on established patterns in data architecture:

- **Event Sourcing and CQRS:** In event-sourced systems, all state changes are stored as an immutable sequence of events (often in a log or object storage). These events are the single source of truth. All queryable state is then *derived* by projecting those events into one or more read databases. This is a textbook implementation of "databases as projections" – the system intentionally does **not** hold authoritative data in the read databases. They are disposable and regenerable views for convenience [13] [14]. For instance, an e-commerce app might keep an event log of orders and payments, and generate a relational view for reporting and a separate Elasticsearch index for text search. If needed, any of these can be rebuilt from the event log. AWS's Prescriptive Guidance explicitly cites *"polyglot data projections from a single source of truth (SSOT)"* as a use-case for event sourcing [13]. In practice, companies like LinkedIn (with Apache Samza and Kafka) and others have used this approach to maintain multiple materialized views of data from a central log.
- **Data Lake and Lakehouse Architectures:** Modern data lakes store all enterprise data in raw form on scalable storage (e.g. Hadoop HDFS or cloud object storage). Analytics databases or warehouses then either **query the data in place** or create refined projections (like materialized views or data marts). A great example is how cloud data warehouses allow defining **external tables** that reference files in object storage. The database engine reads from the storage when executing queries, effectively acting as a **query layer on top of the data lake**. As one industry analysis noted, *"organizations store large volumes in object storage and then access this data*

through their preferred database with queries… they query and analyze data wherever it lives rather than moving it to a central database" [15] . This decouples storage from compute – multiple query engines (SQL, Spark, Presto, etc.) can all project from the same data repository. Even "lakehouse" systems like Databricks or Snowflake follow a similar principle: the durable storage (often in Parquet/Delta/Iceberg format on S3) is the base, and the warehouse compute clusters are effectively projecting that data for analysis.

- **Serverless Analytics and On-Demand Databases:** Cloud services have embraced ephemeral use of databases. Google BigQuery, for example, manages storage under the hood and spins up compute resources to run SQL queries on demand – the user doesn't maintain a persistent server. Amazon Athena similarly lets you run SQL against data in S3 without having to load it into a persistent database beforehand. These services prove that **ephemeral, projection-based querying** can handle large-scale workloads. There are also emerging patterns of **ephemeral databases in containerized workflows**: e.g. launching a Neo4j graph database in a container, loading data, computing results, then destroying it when done [16] [17] . In one case study, Neo4j was run on-demand to perform graph analytics, using S3 for storing both the input data and the results – between runs, no database was kept running [18] [19] . This resulted in significant cost savings and complete reproducibility of analyses, since each run started from a clean slate.

- **Domain-Oriented Data Mesh:** A related concept in data architecture is the *Data Mesh*, which advocates for domain teams owning their data pipelines and treating datasets as products. In a data mesh, you might have separate databases or warehouses per domain, but federated via a governance layer. The projection model aligns well with this – each domain could create its own optimized databases from common raw data. The **Database Mesh** concept extends this idea: *"a database mesh distributes data management across multiple specialized databases, each serving specific business domains or use cases"* [4] . The underlying data might reside in a central lake, but each team spins up their own projections and keeps them updated. The mesh ensures interoperability (through metadata and APIs) so that these multiple databases can still be queried or joined when needed. Essentially, it operationalizes "multiple projections, one truth."

Implementation Strategies

Shifting to this model requires changes in how we ingest, store, and serve data. Key strategies for implementation include:

- **Use Immutable, Central Storage for Raw Data:** Identify a scalable storage solution for your system of record. This could be a cloud object storage (with open formats like JSON, Parquet, Avro, etc.), a distributed filesystem, or an event streaming platform (e.g. Apache Kafka or Pulsar storing events that can be replayed). Ensure this layer is **durable, versioned, and ideally schema-managed** (so you know how to interpret the data). It will hold the ground truth that all projections derive from. This approach might involve adjusting how applications write data: for example, instead of writing straight to a DB, an app could write an event to a log, or save a JSON to storage, which then triggers updates to the databases.

- **Automate Projection Pipelines:** Create processes that can *build or refresh databases on demand* from the source data. This can be done via batch ETL jobs, streaming consumers, or a combination. For example, you might have a nightly job that scans the day's events and updates a relational database summary. Or you might use change data capture to continuously reflect new object-storage files into an index. Containerization and orchestration tools (Docker, Kubernetes) can help by spinning up database instances, running a load script, then saving results. The goal is to treat each database like an *output of a data transformation pipeline*. When

the pipeline runs, the DB is there; when it's not, the DB can be absent or safely stale. Modern data workflow engines and Infrastructure-as-Code can be leveraged to manage these pipelines in a reproducible way. This is analogous to how CI/CD pipelines build software artifacts; here we are building data artifacts (the databases) from source.

- **Leverage External Table and Virtual Query Technologies:** In cases where you don't even need a persistent transformed copy, you can use query engines that read directly from the raw data. Many SQL engines support external tables or connectors that fetch data from files or other stores at query time [20]. Technologies like Presto/Trino, Apache Drill, or Spark SQL can query heterogeneous data sources without a centralized DB. This provides the benefit of using SQL or graph queries on the data *without* loading it fully into a new store. It effectively turns the query engine itself into a transient projection. The trade-off is usually performance – for frequent queries, it might be better to materialize a projection in an optimized database. A hybrid approach is to use caching or intermediate materialized views for hot data while still sourcing cold data on the fly.

- **Design for Eventually Consistent Updates:** When using multiple projections, especially in an event-driven setup, it's common to have a slight delay between data arriving in the source and it appearing in all the projections. Embrace **eventual consistency** where appropriate. The source of truth is always correct; projections will catch up as their update processes run. This means applications and users need to understand that recent changes might not reflect immediately in all read models. In many analytical or content-aggregation scenarios this is acceptable. Where strong consistency is needed, you might still use transactions in a primary DB for that subset of data (e.g. a user account record) but could supplement it with projection databases for other querying needs. Setting expectations and designing idempotent, retryable projection updates will make the system robust even with async updates [21].

- **Monitor and Version Your Data Transformations:** Treat the creation of each database as a software process to monitor. Log when projections were last updated, how many records were loaded, and validate that the numbers match expectations (for instance, if the source has 1,000 new events, ensure 1,000 new rows were added to the projection). By tracking this, you can detect staleness or lag in a projection. Additionally, consider versioning your projection schemas and keeping track of the code (scripts, queries) used to generate them. This way, if something changes (say you improve the projection logic), you can rebuild consistently or even maintain multiple versions of a projection for backward compatibility. Essentially, **data ops** practices become important in this model – you are now managing pipelines and derived data products as first-class entities.

Challenges and Considerations

While the benefits are significant, using databases as projections instead of primary stores introduces some challenges to plan for:

- **Initial Complexity:** This approach requires a shift in mindset and additional infrastructure. Teams must build and maintain data pipelines or event streams, whereas a traditional app might simply perform CRUD on a single database. The **complexity of rebuilding state** from events or files can be non-trivial without good tooling [22]. It's important to gradually build trust in the pipelines and possibly start with a hybrid system (e.g. keep the current DB as backup while introducing projections) until the kinks are worked out.

- **Latency of Updates:** Depending on how projections are updated, there can be lag. In a tightly coupled system, one might worry that having to, say, write to storage and then update a database could slow things down. Mitigating this may involve using fast streaming and putting careful thought into what needs to be real-time vs. what can be eventual. In many read-heavy scenarios, the slight delay is a worthwhile trade-off for the increased read performance and decoupling. But for truly real-time requirements, you might design certain projections to update synchronously or use triggers.
- **Data Duplication (Storage Costs):** This model will intentionally duplicate data across multiple databases (since each projection might contain overlapping subsets of the source). While storage is cheap, it's still a factor – you need to manage the lifecycle of projections (e.g. drop ones that are no longer useful) to avoid uncontrolled sprawl. The key is that this duplication is **strategic and controlled**, not ad-hoc copies. Each projection exists for a reason and can be regenerated, so at least the duplication is transparent and not a mystery. Also, note that the *primary* copy still exists only once (in the central store); the projections are like cached indexes. This is similar to how search engines make an index of the web – yes, it duplicates content, but it's what makes fast search possible.
- **Consistency and Reconciliation:** Bugs in projection code or pipeline failures could lead to a projection that doesn't accurately reflect the source (e.g. missed updates). It's crucial to have monitoring and, ideally, automated reconciliation. For instance, periodically compare counts or checksums between the source data and the projection to ensure they match. If a discrepancy is found, one can trigger a full rebuild of the projection. In an event sourcing world, this is straightforward by replaying from the beginning. In a data lake world, one might re-scan all files. These processes can be time-consuming, but since they're automated, they can run in the background. The ability to rebuild from scratch is a powerful safety net.
- **Security and Access Control:** When data moves through multiple systems, ensuring consistent security controls is important. The source of truth and each projection DB might have their own access rules. It's wise to propagate restrictions from the source to projections (so you don't, for example, accidentally make a sensitive data field accessible in a projection that certain roles can query). Implement data governance such that any new projection is registered and assessed for compliance (similar to creating a new data mart in traditional warehousing – you'd ensure it adheres to policies). Automation can help here too, tagging data and using those tags to drive what gets projected.

Despite these considerations, many organizations find that the **benefits outweigh the complexities**. The approach aligns well with cloud-native architectures: it embraces *disaggregation* (separating storage and compute), *automation*, and *specialization* of components. Over time, tooling is improving to support this model (for example, there are frameworks for event sourcing, and modern data orchestration platforms for pipelines).

Conclusion

Using databases as projections rather than primary data stores represents a significant evolution in data architecture. It flips the old script: instead of pouring all data into one big persistent database, we **keep data in a durable, flexible store and spin up targeted databases to view or manipulate that data as needed**. This approach leads to more scalable systems, lower costs (by not running heavy databases 24/7 and by leveraging cheap storage), and greater agility in serving diverse workloads.

Crucially, this is not just a theoretical idea – it builds on proven patterns from event-sourced microservices (with their polyglot read models) to big data platforms (with their separation of storage and compute). Companies like Netflix unify data models at the storage layer while enabling many representations, and cloud providers are offering services that implicitly follow this pattern [15] [13].

Even in everyday architecture, we see hints of this approach whenever caches, read replicas, or full-text search engines are added on top of a core database – the projection model takes it to its logical conclusion by making *all* databases into such replaceable, purpose-driven views.

In adopting this model, organizations should prepare for a cultural shift: **treat your data as a first-class asset independent of any particular database technology**. Once that mindset sets in, databases become what they should have always been: powerful tools to *access* and *project* data, rather than vaults to imprison it. The result is a data infrastructure that is more modular, resilient, and aligned with the fast-changing needs of modern applications. Companies that successfully leverage this paradigm can achieve the holy grail of data management: *model data once, and use it everywhere* – without being bottlenecked by a single, unwieldy database.

Sources:

- AWS Prescriptive Guidance – *Event Sourcing Pattern* (on using a single source of truth with polyglot projections) [13] [21]
- Dinis Cruz, *Ephemeral Neo4j Instances for On-Demand Graph Analytics* – illustrating cost and reproducibility benefits of on-demand databases [2] [19]
- MinIO Blog – *Object Storage Optimized Databases* (discussing querying data in object storage via databases) [15]
- *CQRS + Event Sourcing with Polyglot Persistence* (Medium article by Sooraj V.) – describes using multiple read-model databases from a single event store [3] [10]
- Navicat Blog – *Database Mesh Architecture* (on domain-specific databases in a decentralized architecture) [4]
- Otter.AI Transcript – User's notes on using databases as projections (insights on cost, state management, and serverless considerations).

---

[1] [5] [8] [9] [16] [17] Ephemeral Neo4j Instances, Data Tests, and Self-Evolving Knowledge Graphs
https://www.linkedin.com/pulse/ephemeral-neo4j-instances-data-tests-self-evolving-knowledge-cruz-ixxde?trk=public_post

[2] [6] [18] [19] Ephemeral Neo4j Instances for On-Demand Graph Analytics - Dinis Cruz - Research Hub
https://docs.diniscruz.ai/2025/06/25/ephemeral-neo4j-instances-for-on-demand-graph-analytics.html?trk=public_post_comment-text

[3] [10] [14] CQRS + Event Sourcing with Polyglot Persistence — End-to-End Flow (with Mermaid Diagrams) | by SOORAJ. V | Nov, 2025 | Medium
https://medium.com/@v4sooraj/cqrs-event-sourcing-with-polyglot-persistence-end-to-end-flow-with-mermaid-diagrams-e897a8ae94a4

[4] Building Modern Distributed Data Systems Using a Database Mesh Architecture
https://www.navicat.com/en/company/aboutus/blog/3181-building-modern-distributed-data-systems-using-a-database-mesh-architecture.html

[7] Model Once, Represent Everywhere with Alex Bertails - YouTube
https://www.youtube.com/watch?v=BAE_4tCr5cs

[11] [12] Model Once, Represent Everywhere: UDA (Unified Data Architecture) at Netflix | Hacker News
https://news.ycombinator.com/item?id=44275575

[13] [21] [22] Event sourcing pattern - AWS Prescriptive Guidance
https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/event-sourcing.html

[15] [20] Object Storage Optimized Databases: Trends & Industry Leaders
https://blog.min.io/databases-for-object-storage/