Shift Towards Embedded Databases for Serverless
DuckDB, LanceDB and KuzuDB are usurping the need to host 24/7 database-engines.

✉ colum.mccoole@analect.com
in colum-mccoole-746b946a
○ analect/technical-docs-hierarchy

20

Version: 0.1 Dated: 2024-10-15 Authored by: CM

Docs: ▷ Level 2 ▷ AI/ML Value-chain

Topic Navigation

How does this topic relate to other relevant content in this space.

What Problem are We Solving For?

The client/server architecture for databases has been around for a very long time, and has been proven to be a successful commercial model, which is why they are the norm in large-scale production use cases. However, they aren't well adapted to serverless use-cases, since they require to be on all the time, incurring server costs, even if they aren't being actively used. It turns out that new embedded database

What are Embedded Databases?

There are a number of very informative blog-posts from The Data Quarry on embedded-databases that are recommended reading.

An embedded database is an in-process database management system that's tightly integrated with the application layer. The term "in-process" is important because the database compute runs within the same underlying process as the application. A key characteristic of embedded databases is how close the storage layer is to the application layer. Additionally, data that's larger than memory can be stored and queried on-disk, allowing them to scale to pretty huge amounts of data (TB) with relatively low query latencies and response times.

The evolution of modular data-tooling has been an important driver of the new database models, with Arrow as a de-facto standard for efficiency gains with in-memory and on-disk capabilities that are core features of this new class of embedded database. The embedded architecture is still relatively new, at least for OLAP databases, nevertheless vendors are delivering rich feature-sets as open-source solutions, even as they figure out their monetization strategies.

Viability as a Serverless Solution

Being able to dispense with traditional databases in favour of cheaper object-stores, such as S3 is a game-changer. It means you can work with ever-scaling amounts of data, that working with AI-based solutions requires, benefit from the low-cost object-store model and get to keep the query-processing power that embedded-databases bring. You also don't have to compromise on query latency, with the advent of more performant variants such as S3 Express.

We illustrate the usage of LanceDB as part of a Serverless RAG implementation on page 27. Unlike most other RAG-based solutions, requiring you to store raw data in one place and embeddings elsewhere, both get housed in the same place with LanceDB, reducing considerably the overhead of working with these technologies.

Greater Alignment to ML / AI Workloads

Per this Data Council talk, Chang She, CEO of LanceDB, contends that the current need to duplicate data depending on the end use-case, in the context of experimentation with AI, is inconvenient and expensive. You have a requirement to hold raw data, vector data, often tensor-based data in a different form again, with each potentially duplicated further with experimentation around adjusting a relevant feature-set.

LanceDB is able to have a single data-store and therefore a single source of truth that is able to hold multi-modal raw data and the calculated vectors from that data as well as various forms of indexing that sits above that data, making it potentially seamlessly available to different end use-cases in the requisite formats. This makes it much more amenable to being a composable solution for data in an AI training and exploration context.

DuckDB

Among it's many technical features, perhaps most importantly, DuckDB has fast become a universal **data connector**, due to the sheer number of data formats it's able to natively read data from: - A lot of ETL processes involve expensive transformations that reshape data from one form to another - DuckDB natively reads from formats like CSV, JSON (including nested JSON), Parquet, and has scanners to directly read from Postgres and SQLite databases - Thanks to the *Arrow* format, data can very easily move from a DuckDB table to a Pandas or Polars DataFrame, and vice-versa - It also offers connectors that allow users to directly read Parquet data from S3, GCS and Azure storage - Because DuckDB natively supports a lot of these formats, it's able to perform efficient scans on-disk, without having to materialize them in-memory all the time (unlike Pandas).
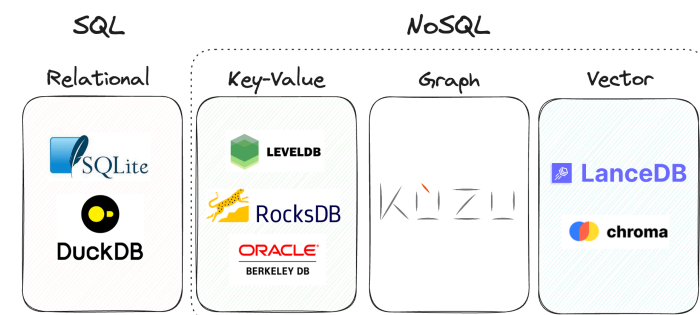

Figure 1. Embedded databases organized by data model paradigm. source

KuzuDB

KùzuDB is an open-source graph database management system (GDBMS) that also happens to be embedded. It has:

- first-class support for the labelled property graph (LPG) model
- use openCypher as its query language
- built for vectorized execution of OLAP-type queries
- designed to be the storage layer of choice for graph machine learning applications

LanceDB

LanceDB is an open-source embedded database for vector search built with persistent storage, which greatly simplifies retrieval, filtering and management of embeddings. On the surface, it is a vector database written in Rust, but underneath, it's a collection of specialized modular components which are themselves independent components of the Rust tooling ecosystem.

Key features include:

- Incredibly lightweight (no DB servers to manage), because it is entirely in-process
- Extremely scalable from development to production
- Ability to perform full-text search (FTS), SQL search and vector search
- Multi-modal data support (images, text, video, audio, point-clouds, etc.)
- Zero-copy (via Arrow) with automatic versioning of data

LanceDB implements its own vector index on top of the underlying Lance data format, which is an IVF-PQ disk-based index. Tantivy is an open source full-text search engine incorporated to allow keyword-based search via BM25. DataFusion, an embeddable SQL query engine, is used to power the full-text/vector search queries via a SQL interface. The Apache arrow format is used to allow a smooth transition between in-memory and on-disk data storage, and also for seamless interoperability with other data formats from systems like DuckDB, Pandas or Polars.