Vision document – logical database

James Deikun

February 23, 2026

1 Introduction

The purpose of this document is to present a high-level description of how the logical database will function and be used in the context of a Safeguarded AI deployment, in order to aid in refining requirements for individual system components.

2 Time and concurrency

2.1 Time and clocks

Time in the logical database takes the form of a directed acyclic graph of *moments*. In addition to the relation of one moment immediately following a *predecessor*, there is a further relation of a moment being a *rebase* of a *referent* from an *old base* to a *new base*. Both of these relations are transitively acyclic, and further there cannot be any mixed cycles through predecessor and referent.

There are multiple clocks reflecting different views of the time axis. The *logical clock* is the primary clock which reflects the time when logical data was added to the database. The *system clock* reflects non-logical changes to the database schema, such as the addition of indices or (de)materialization of views. The *session clock* reflects the ephemeral changes to the session namespace (see section 4.1).

Question 1. Do we really need the session clock?

Question 2 (Naming of moments)**.** How should we name moments so that we can refer to them across related backends? Most users of DAG-style time are version control systems, and many use content hashes to name their equivalent of moments. However, given the state of cryptological knowledge around one-way and pseudorandom functions, we should not assume that even a slightly superhuman AI cannot break a cryptographic hash function thoroughly.

2.2 Merging and reconciliation

Concurrent writers will maintain their own *branch* time streams for individual writes. In order for writers to create a common product, *merges* will take place, where moments will follow from multiple other moments. In some cases this will work well as is, simply combining the events that have happened in all the moments to be merged. However, as this logical database is insert-only, there will be cases where a merged moment will violate semantic constraints that cannot be reconciled by further inserts. This case requires *reconciliation*, a process where some branches are rebased to achieve the same logical end while respecting the progress of another branch. This process requires active participation from the application.

2.3 Backends and sessions

An ordinary backend will support multiple concurrent readers but only one concurrent writer, and will be optimized for small writes. A *coordinator* is a backend supporting multiple concurrent writers and large writes, specialized for merging and for acting as "systems of record" that keep the many agile backends in a large cluster in sync by receiving submitted bulk updates and streaming requested ones back out with some precomputation of materialized view updates.

3 Schemas

A database schema is composed of types, relations, dependencies, and plans. It is divided into the logical aspect, which expresses permissible events, a history of migrations, and queries of interest over the event log; and the non-logical aspect, which expresses base tables recording information modelled by the event log, materialization of queries as materialized views and indices, and auxiliary tables and queries used in incremental querying and view maintenance, as well as cached query plans.

Types, relations, and dependencies have both logical and non-logical variants, which are much the same except for their relationship to time. Plans only exist in the non-logical world.

3.1 Types

Types will be a subset of the types definable in first-order logic; that is, they will not be dependent on other types. There will be base types typical of a database, that is, character strings, small binary blobs, fixed (including integral) and floating point numbers, and possibly some other application-specific types. There will be dependent types representing decidable predicates and relations on base types; these will all compute away to either a unit or empty type and will act as what are usually called domain or CHECK constraints. There will also be first-order quotient inductive-inductive types built on top of these.

Question 3. How will Amar Hadzihasanovic's work affect this?

Question 4. How specially should we treat record types?

3.2 Relations

A *relation* consists of an abstract set (the set of *rows*) equipped with a mapping to a concrete type, the latter specified when the relation is first created. An existing relation can also be upgraded to an *insertable relation* which renders it able to receive insert events. A relation can also be *tombstoned* at which point it can no longer receive insert events (if it ever could). These alone comprise the logical content of relations.

Even logical relations also possess non-logical content, including the materialization flag and cached query plans; these aspects will be addressed below. Meanwhile, it is not meaningful for a non-logical relation to be insertable or tombstoned.

Question 5. How can we represent the abstract set in a way that enables portability of events? The ideal is to derive the identifier(s) for a row from the event(s) that created them via the dependencies that served as intermediaries. However, with this approach, the number of valid identifiers for a row can be large, perhaps even infinite, and the individual identifiers themselves can contain a lot of entropy, so some more sophisticated approach may be needed.

3.3 Dependencies

Dependencies are used to express semantic constraints, queries, and migrations. Semantically, they are primarily expressed as sequents in a subset of geometric logic.

In order to support incremental querying, the language of dependencies must be strong enough to express all the following:

1. Tuple generating dependencies, that is, rows that must exist given a logical condition. Semantics for generation of new rows by rows that are "duplicates"–that is, distinguished solely by row identifiers–will be handled according to the calculus of spans.

2. Functional dependencies, including keys (functional dependency of the row identifier on some part of the row content).

3. Aggregates of some sort.

4. Invertibility of dependencies.

Question 6. How are aggregates to be supported? And are functional dependencies to be distinguished from tuple generating dependencies? This would probably be helpful for implementation, but how could the geolog-lang elaborator support this?

3.4 Plans

Plans are somewhat internal to the query engine, but they reference non-logical types, relations, and dependencies. It is also useful for them to be introspectable (see below).

The content and interpretation of plans will be elaborated upon in section 6.1.

3.5 Non-logical aspects of relations

Relations can be toggled between materialized and non-materialized; materialized relations have all their rows directly expressed as a structure on disk, or sometimes in memory. In addition, they have a cached plan for querying, and sometimes one for insertion, which are created by the query optimizer and can change with the evolution of the schema.

Question 7. Do we want to support materialization of relations covering only some logical moments?

3.6 Introspection of the schema

The schema itself will be exposed via the *system catalog*, a subschema in a special namespace that will itself be available in-schema at all moments. Migrations of the system catalog that come with new versions of the geolog software will occur at special *reserved moments*.

3.7 `geolog-lang` and the schema

The tool `geolog-lang` will be used to elaborate from a user-friendly and compositional source language for expressing schemata to the form expressed in the system catalog.

3.8 Non-logical restrictions on schema evolution

In order to prevent problems, there will be mechanisms to restrict evolution of the schema (including its logical portion)

4 Namespaces and security

Namespaces are the primary unit of security for the logical database. There will be namespaces where logical changes cannot be made by the AI.

Question 8. How can we assure this?

There may also be cases where, before verification of an answer received from the AI, the event log of the accumulated database is rewritten into a controlled schema or a simpler serialization format to reduce the exposure of the verifier to attacks from the AI.

Question 9. Will this actually be helpful? What are the details?

4.1 The session namespace

There will be a special namespace, pronounced "session", for relations and such created and used during the session. Any events that reference the session namespace will be translated into events that use only persistent namespaces, as if they were issued that way in the first place. As this can lead to different results on merge than using things with the same definitions in the persistent namespace, use with caution.

5 Events and transactions

5.1 Events

Events are how data enters the database. Each event happens at (or strictly, just before) a moment, and is associated with a relation and a value of the relation's associated row type. *Named events* are how data exits the database, and are dependent tuples of an event with a row identifier for its relation.

5.2 Transactions

A *transaction* creates a new moment. It consists of a set of predecessor moments and a set of events. Transactions can be created incrementally, but all the predecessor moments must be specified before any events. An open transaction can have read-only queries associated with it, which are evaluated incrementally in order to provide (temporary?) row identifiers for use in formulating further events for the transaction. An open transaction may be discarded or closed. In the latter case, it becomes a moment.

Question 10. How do we formulate this facility to prevent fragility? Specifically, how do we record row identifiers that reference newly created rows in the underlying event log?

Question 11. Do we want to support some kind of distributed partitioning of event logs? If so it calls into question the entire transaction mechanism, as well as the idea of merges.

A *named transaction* describes a moment. It consists of a moment identifier, a set of predecessor moments, and a set of named events ordered in dependency order. These are the usual results of read-only queries, although read-only queries internal to a transaction simply return sets of named events incrementally.

6 Queries and their optimization

6.1 Kinds of queries

A *read query* is specified as a set of logical relations decorated with conditions, and a range of moments, and returns a set of named transactions. An *insert query* is specified as an event in the context of an open transaction and returns a set of named events corresponding to monitor queries. A *monitor query* is specified as a logical relation decorated with conditions in the context of an open transaction and returns nothing.

There will be a facility to easily rerun a read query with different time bounds.

Question 12. Instead of monitor queries, should some kind of "INSERT RETURNING" construct be used as a substitute?

There are also special queries. A *clock query* is specified as a moment and ongoingly returns new moments coming after that moment as they enter the database.

6.2 Semantics of query execution

All normal queries consist of optimized versions of a dependency chase. Dependencies will be chased during a read query when they may recursively lead to creation of a row under the purview of the read query.

For insert queries, dependencies will be chased when they may recursively lead to any of:

1. Creation of a row subject to an active monitor query.
2. Creation of a row in a materialized relation.
3. A constraint that can fail, like a check constraint or functional dependency.

In addition, when adding a second or subsequent predecessor moment to an open transaction, dependencies will be chased as if for an insert.

6.3 Optimization of queries

There are three primary ways you can optimize a dependency chase:

1. By chasing dependencies in bulk and ordering the phases of dependency discovery and bulk chasing appropriately, redundant chasing can be reduced on an intra-query basis and locality of disk access improved.

   This form of optimization is similar to join order optimization and can use similar techniques and data structures.

2. By materializing relations, redundant chasing can be reduced on an inter-query basis and depending on the data structure into which they are materialized, the former method may become more efficient by taking advantage of the characteristics of the data structure(s) to optimize discovery.

3. By adding auxiliary relations and dependencies and pruning the original dependencies that become logically redundant, better opportunities are created for the above two processes to work. This is especially important for incremental and/or time-range querying.

References