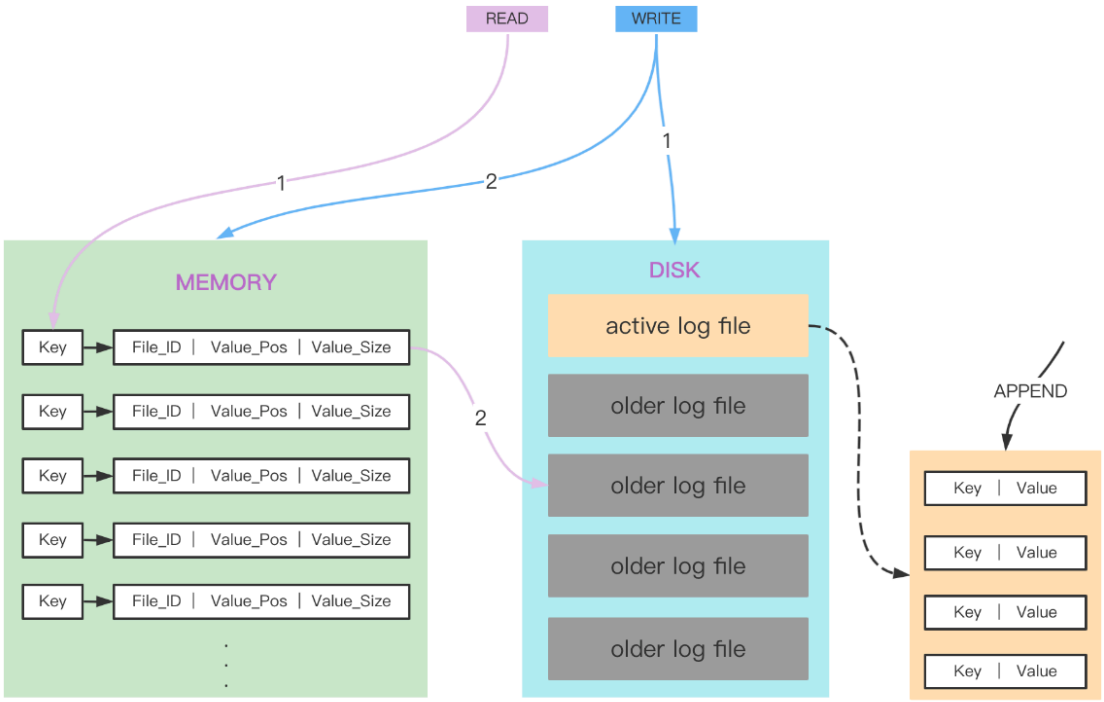
Vector Index Design Doc

Currently with the rise of LLM, it has been important for databases to provide in-time information that LLM models can understand. However, many key-value stores lack the ability to query such information efficiently. In this project, we are going to integrate vector query indexes to existing databases RoseDB. RoseDB is a bitcask based lightweight key-value database, which can be embedded to many existing systems. By empowering RoseDB, we also help systems that depend on it.

In this doc, we present the specific architectural design for our project. We first describe the existing architecture. Then we propose our solution and the milestones.

Existing Architecture

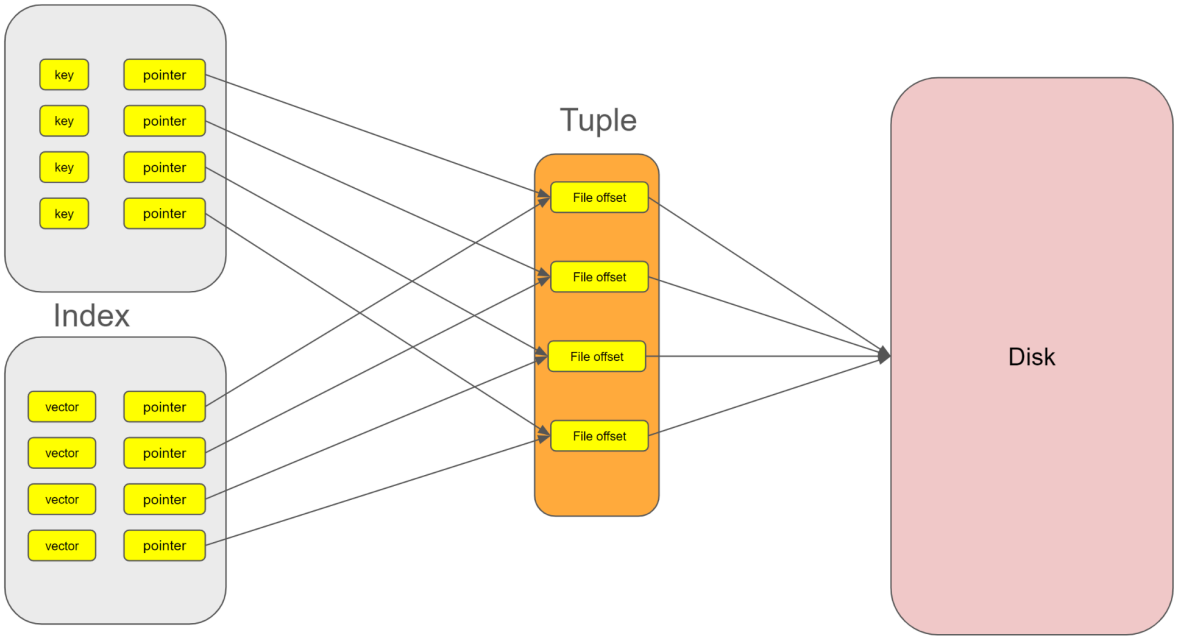
Rose DB primarily uses a lightweight architecture to support basic insert, delete and search queries. The general architecture looks like the following:



As we can see, the data is persisted on the disk as the log file. To maximize the write performance, all the log files are append-only. To speed up the search, there will be an in-memory index, which maps the key to the file offset where actual data is located. The typical write operations are handled like the following. First, the value will be appended to the end of current log files. The databases will then insert a new entry to the in-memory index. The updates will be handled in a similar way because the log file is append-only. The new value will

be appended to the end of the log, and the index will be updated such that it will point to the updated value.
When handling the read request, the databases will look up the index and find the corresponding log file offset. The value will be fetched from the disk and returned to users.

Proposed Solution



Our solution will respect the existing workflow in Rose DB. The vector index will be added as a secondary index. Also, the reads and writes process will remain the same. However, there is one noticeable change. Both primary and secondary indexes will not directly map key to the file offset. Instead, they will use a tuple pointer to locate the value position. We make this design choice because it will reduce the engineering complexity. There is no need for the vector index to explicitly manage the file offset, since the primary index already handle it for us.

The interface of our vector index will look like the following:

```
type VectorIndex interface {
    Get(vec []int64, res_num int64) [][]int64
    Insert(vec []int64) err
}
```

There are only two operations supported by our vector index, Get and Insert. The Get function will take a vector, and number of results that the client wants. The return will be all the vectors that are mathematically “similar” to the input vector. The insert function will be more

straightforward, where the function takes the input vector, and returns only if any error occurs. Noticeably, there will not be any update or delete functions, because it is meaningless to do updates or deletes. The vector index is constructed only for queries for vectors that are “close” to each other. The updates and deletes only make sense for scalar operations, which will be handled by the primary index accordingly. Our design decoupled primary and secondary index responsibility so that we only need to focus on logics of vector index.

Milestone

There are three main parts of our project milestone: vector index, integration with existing systems and benchmark testing.