CUDB: A simple document-based NoSQL DBMS

ECE464 Final Project

Jonathan Lam, Derek Lee, Victor Zhang

December 11, 2021

Contents

This document will include information pertinent to the functional and architectural design decisions, as well logistics of the final project for the sake of the class. The project source and build/test instructions can be found on GitHub at jlam55555/cudb. The API documentation can be found at `https://jlam55555.github.io/cudb`.

1 Abstract

CUDB (Cooper Union DataBase; alternatively CUDA++) is a simple document-based, MongoDB-inspired NoSQL database for learning purposes. The database is in the form of a Rust library that supports basic CRUD operations similar to MQL and a document-based data model. CUDB supports basic database features such as indexing using B-trees and persistence, but does not support many advanced features found in commercial DBMSes.

2 Changes from proposal

The original proposal can be found here. Surprisingly few changes have been made from the original proposal; though we were shrouded in ambiguity when writing the proposal, many of the proposed changes and much of the proposed schedule has stood up to plan. In particular, we support the following features:

- CRUD operations
- File-backed persistence using the OS's page cache similar to MongoDB's mmapv1
- Manually-created indices
- B-tree indexing
- Index- and table-scan lookups
- Nested queries using field paths

and don't support the following features:

- Clustering (MongoDB doesn't support this either)
- Transactions/ACID
- Concurrency
- Distributed store/sharding
- Schema definition or enforcement
- Aggregate operations and cross-table operations (e.g., `$lookup`)

The only changes from the original proposal were minor: we do not use BSON (instead using Rust's default binary serialization library), and the block allocation scheme does not have a residency map like mmapv1 (and is thus much simplified – more details can be found on the repo README). Both of these changes were made for the sake of simplicity, so that the project can be finished in time. The switch from BSON may make the storage format less portable but more performant (because it doesn't have to perform the conversion). The simplification to the block allocation scheme will cause a large loss of (space) efficiency when a database is delete- or update-heavy. In a insert-only database, it will not be inefficient.

Additionally, the original proposal was somewhat vague in describing the data structures and API regarding queries: over the course of the project, this had to be refined. The technical details of the final design can be found in Section 4, and on the online API documentation.

3 Functional overview

This section describes the CUDB public API and user-facing data structures, including idiosyncrasies and other design decisions.

3.1 Data structures

3.1.1 Documents and values

The `cudb::document::Document` struct allows for a "document" similar to that in MongoDB or Javascript/JSON. It is implemented as a `HashMap` mapping strings to `cudb::value::Value`s.

`cudb::value::Value` is an enum of possible value types. Its current definition is shown in Figure 1. Note that recursive documents (a.k.a., nested documents, or subdocuments) and array types are allowed. Note also that only scalar types of the same variant (i.e., two ints or two strings, but not a string and an int) are comparable using relational comparators or equality; comparisons between different variants or with a document or array type will fail. (This partial comparison will necessitate typed indices, which will be described in a later section.)

3.1.2 Collections

CUDB supports organizing documents into collections (`cudb::db::Collection`), but does not support organizing collections into databases like MongoDB. (This is not a problem, as cross-database operations are hardly, if ever, used.) Collections can be opened and closed using the management API shown in Figure 2. Each collection is backed by a file, and is opened by specifying the file path. If the file does not exist, then a new collection will be created (this is similar to MongoDB's behavior). The specifics of the query data structures will be discussed in the next section.

```rust
pub enum Value {
  /// Special type for unique identification,
  /// similar to MongoDB's `_id`.
  Id(String),
  /// Fixed-size integer type.
  Int32(i32),
  /// Arbitrary-length strings.
  String(String),
  /// Recursive documents (hashtables).
  Dict(Document),
  /// Array types.
  Array(Vec<Value>),
}
```

Figure 1: `cudb::value::Value` definition

```rust
impl Collection {
  /// Create a collection from a path.
  pub fn from(path: &str) -> Collection {}

  /// Close collection and underlying file pointer.
  pub fn close(self) {}

  /// Drop collection data and indices.
  pub fn drop(self) {}
}
```

Figure 2: `cudb::db::Collection` management API

Collections support the ordinary CRUD operations. The definition for the `cudb::db::Collection` API is shown in Figure 3.

3.1.3 Queries

A stereotypical simple (without joins or aggregate operations) query is shown in SQL, MongoDB (MQL), and CUDB (in RON format) in Figure 4. This simple query format shares a common format: a set of constraints (the SQL `WHERE` clause), a set of projections (the SQL `SELECT` clause), and a sort order (the SQL `ORDER BY` clause). In MQL, the constraints and projections are positional arguments in the `Collection::find()` function; however, we make them explicit in the `cudb::query::Query` struct. In particular, the `cudb::query::Query` struct comprises three fields: a `cudb::query::ConstraintDocument` (type-aliased to `HashMap<cudb::query::FieldPath, cudb::query::Constraint>`); a `cudb::query::ProjectionDocument` (type-aliased to `HashMap<cudb::query::FieldPath, cudb::query::F` and an optional sort ordering (`Option<Vec<cudb::query::ResultOrder>>`). Updating and deleting documents requires a constraint, and sort order may be provided for the `cudb::db::Collection::updateOne()` and `cudb::db::Collection::deleteOne()` operations. See the API documentation for more details.

3.1.4 Indices

To speed up queries, users can create an index (`cudb::index::IndexSchema`) on a collection. An index is a non-empty set of field names. An index may be used to speed up a query to logarithmic (B-tree search) rather than linear time (linear scan), if the index matches the query's constraints. There is no restriction on the number of indices on a collection. Indices must be created manually by the user.

A `cudb::index::IndexSchema` comprises an ordered set of field specifications (`cudb::index::FieldSpec`). Each field specification includes a field path (`cudb::query::FieldPath`) and a default value (`cudb::value::Value`). The default value serves both as the placeholder for missing values and the type specification for that field. The `cudb::db::Collection` API supports functions to create, list, and delete an index. See the documentation for more details.

Some notes about CUDB's indices:

Immutable indices The default values for fields in an index are immutable. To change the default values, the index must be deleted and re-created with the desired default value.

Failing to create an index due to a document If a document has a field contained in the index, and its value is of a different type than the default value specified in the index, then creating the index will fail.

Failing to create a document due to an index If an index on the collection contains one of the fields that is contained in a document to insert or update, and the type of the default value for that field in the index is

```rust
impl Collection {
  /// Insert one document.
  pub fn insert_one(&mut self, doc: Document) {}

  /// Insert a vector of documents.
  pub fn insert_many(&mut self, docs: Vec<Document>) {}

  /// Fetch at most one document matching the query.
  pub fn find_one(&mut self, query: Query)
    -> Option<Document> {}

  /// Fetch a vector of documents matching the query.
  pub fn find_many(&self, query: Query)
    -> Vec<Document> {}

  /// Update at most one document that matches the query.
  pub fn update_one(&self, query: ConstraintDocument,
                    update: UpdateDocument) {}

  /// Update all documents matching the query.
  pub fn update_many(&self, query: ConstraintDocument,
                     update: UpdateDocument) {}

  /// Replace at most one document that matches the query.
  pub fn replace_one(&self, query: ConstraintDocument,
                     replace: Document) {}

  /// Delete at most one document that matches the query.
  pub fn delete_one(&self, query: ConstraintDocument) {}

  /// Delete all documents that match the query.
  pub fn delete_many(&self, query: ConstraintDocument) {}
}
```

Figure 3: `cudb::db::Collection` CRUD API

```sql
SELECT name, dob
FROM students
WHERE gpa>3.0 AND grade<>9
ORDER BY gpa DESC;
```

(a) SQL

```
db.students.find({
  gpa: { $gt: 3.0 },
  grade: { $ne: 9 }
}, {
    name: 1,
    dob: 1
}).sort({ gpa: -1 });
```

(b) MQL

```
Query(
  constraints: {
    ["gpa"]: Constraint::GreaterThan(Value::Double(3.0))),
    ["grade"]: Constraint::NotEquals(Value::Int32(9)))
  },
  projection: {
    ["name"]: Projection::Include,
    ["dob"]: Projection::Include
  },
  order: Some([
    ResultOrder::Desc(FieldPath)
  ])
)
```

(c) CUDB

Figure 4: Stereotypical SQL, MQL, and CUDB queries

different than the type of the value in the document to insert or update, the insert or update operation will fail.

Failing to create an index due to an index If an existing index contains a field that the new index contains, and the fields have different type, then creating the new index will fail.

Indices and missing values Creating an index will not fail if a document does not contain one of its fields. Instead, it will be treated as if that field were given the specified default value.

Index matching An index will "match" a query if all of its fields are used in the query's constraints. Note that this is not optimal: in a B-tree, we can actually match an index to any query where the (unordered) constaint fields form some prefix of the (ordered) index fields. Our approach is taken for simplicity.

Index selection The selection process is fairly simple for our database: an index with the maximal number of matching fields (i.e., a matching index with the maximal number of fields). A better database should have better cost metrics due to cataloging or manual user input.

Primitive values Index fields can only include scalar types (i.e., not subdocuments or arrays). However, field paths can traverse subdocuments; only the final path component has to be a scalar type.

3.2 Sample usage

See the README on the GitHub repository for sample client code.

3.3 Expected performance characteristics

See Table 1. I represents the number of indices on a collection, and C indicates the cardinality of a collection. D represents the average size of a document in the collection.

Inserts Insertions require two steps: writing to the file and inserting into all indices. Writing to a file involves finding the insertion offset ($O(1)$ due to the simple buffer allocation scheme). Writing to each index (B-tree) takes $\log C$ time, for a total of $O(I \log C)$ time. The constant-time file allocation is simple and fast but memory-inefficient, because it does not attempt to reclaim memory.

Read/Update/Delete (RUD) Each RUD operation first involves looking for the best matching index, which takes time $O(I)$. Without a matching index, read/update/delete (RUD) updates require a full table scan. With a matching index, RUD operations are roughly logarithmic (due to a B-tree implementation). If all of the constraint fields are used in the index,

Operation	Time	Memory
Insert	$O(I \log C)$	$O(D)$
RUD (no index, unsorted)	$O(I + C)$	$O(CD)$
RUD (index, unsorted)	$\approx O(I + \log C)$	$\approx O(\log CD)$
RUD (no index, sorted)	$O(I + C \log C)$	$O(CD)$
RUD (index, sorted)	$\approx O(I + (\log C)(\log \log C))$	$\approx O(\log CD)$

Table 1: Expected performance characteristics

then the lookup is logarithmic; if there are additional fields in the index that are not contained in the index, then there needs to be an additional linear scan on the index-matching documents to filter out documents that do not match the remaining fields in the constraint. Sorts are implemented using the default Rust unstable sort (linearithmic time and in-place).

4 Architectural overview

4.1 Physical storage

A `cudb::db::Document` is file-backed. The data structures and routines to handle this physical storage are located in `cudb::mmapv1`. The physical representation of a collection is a "pool" (`cudb::mmapv1::Pool`), which is stored in a regular file.

The desired operations on a pool are a pool scan, pool insert, pool write, pool read, and pool delete (at some offset). To achieve this, the pool file is structured as a sequence of contiguous "blocks" (`cudb::mmapv1::block::Block`). Each block has an offset (in bytes from the beginning of the pool), and a size (a power of two between 32 and 1MB). Contained in each block are a fixed-size header (containing the following metadata: whether the block is (soft-)deleted, the size of the data in the block, and the total block size) and a serialized `cudb::document::Document`. The pool operations may return a wrapped version of a document (`cudb::mmapv1::TopLevelDocument`) including its block information. The pool also maintains a pointer to the end of the file as the insert location. With this setup, the pool operations are implemented as follows:

Read document at offset Seek to the offset in the file. Read the block header to get the block capacity and data size. Read the serialized document, deserialize it, and return the document.

Pool scan Start with an initial offset of zero. If you are not at the end of the pool, read the document header to get the block data size, soft-deleted status, and capacity. If the document is deleted, ignore it; otherwise, read at the current offset and push the document into a vector of documents. Advance the current offset by the block size and repeat.

Insert document Serialize the document and find the size d of the serialized data. Find the smallest power of two n such that $n > d + h$, where h is the (fixed) header size, also in bytes. Write a new block at the end of the file, and update the pool size. Return the pool offset of the document as a `cudb::mmapv1::TopLevelDocument`.

Write/update document at offset Serialize the document. Read the header at the specified offset. If the block size is too small to contain the updated document, soft-delete the document (see below) and insert the document as if it were a new document (allocating a suitably-sized block in the process). Return the (potentially-updated) pool offset of the document as a `cudb::mmapv1::TopLevelDocument`.

Delete document at offset Update the header at the specified offset by setting the soft-deleted flag to true.

The pool can be closed by simply closing the file. The pool can be deleted by simply deleting the file. When reopening the file, the only metadata/state that has to be restored is the current insert index, which is the end of the file (the size of the file in bytes), which is easy to calculate using some filesystem API.

This storage engine is named after MongoDB's old storage engine, mmapv1. CUDB's storage engine is similar to mmapv1 in several ways: documents are stored contiguously; documents are stored with power-of-two-sized allocations; and memory paging is managed by the OS. However, CUDB does not use `mmap` to bring pages into memory (despite the name), instead relying on the OS for its caching properties as well. This is a tradeoff of control over caching, in exchange for simplicity of implementation.

Note that there are a great number of possible error conditions if any of the following occur: pool file permissions changed, pool file corruption, or any operation at an invalid block offset. Thus, we assume that the filesystem is tamper-proof and that the user is never able to modify block offsets via encapsulation in the API's (e.g., they will never be exposed to a `cudb::mmapv1::block::Block` descriptor and thus not be able to perform an operation at an invalid offset). Of course, this is a very weak assumption, and only suitable for a simple academic project where we can assume no bad actors.

4.2 Indices and index schemas

There are two closely related data structures that are relevant to the discussion of "index" in CUDB. An "index schema" (`cudb::index::IndexSchema`) contains the definition of an index: an ordered collection of fields with a type and default value – this is what is referred to when saying "index" colloquially. An "index instance" (`cudb::index::Index`) is the ordered set of values from a document that correspond to the fields specified in a index schema. If the field is missing in the document, then it is given the specified default value.

As an example, consider the index schema shown in Figure 5. The index schema specifies three fields that we want to index by. If any constraint contains all three of these fields, then we may use this index schema to speed up that query. Note that each field path is specified by an array: i.e., the path to the field `"c"` within the subdocument `"b"` is given by the fieldpath `["b", "c"]`. The document is specified normally. The index instance is created by extracting the fields from the document, in the same order as in the index schema, and checking that the types are consistent. Note that the last field in the index schema does not exist in the document, so it is filled in with the default value specified by the schema.

For each index schema, a B-tree is created from all of the documents in the collection. The B-tree maps index instances to a set of block offsets of documents who have that index instance. A collection stores a hashmap of index schemas to corresponding B-trees. When a user declares and index schema, the B-tree for that index schema is created; if any of the documents have a type conflict with the index schema, building the index schema fails. Alternatively, if any other index schema has conflicting types, then the index schema creation also fails. Similarly, when a user inserts or updates a document, it must be updated in all index schemas; if the new document has any type conflicts with any of the existing index schemas, then the operation fails.

Note that indices are necessarily typed. This is because indices must be stored as (keys of) a B-tree, and thus must be totally-comparable. As mentioned when first discussing documents and values, only scalar values of the same variant (type) are comparable. Thus, the "type" of a field in the index schema is defined using a default value. Note that in the case of documents that are missing that field, they are treated and ordered as if their value was the default value.

4.3 Explanation of querying

It has already been discussed how to perform operations on a pool when a document offset is known. However, arguably the most important role of databases is to be able to query data based on some query constraint. This section will be an overview of the steps taken to perform a query operation with the aid of indices. (A query that does not use any indices is the degenerate case, in which the matched index is empty.)

Once we have the ability to query, the CRUD operations are straightforward to implement. In other words, querying can be thought of as a function that takes a constraint and returns a set of offsets, which are used as inputs for pool operations. The CRUD operations can be summarized as the following combination of querying and pool operations shown in Table 2.

The first three steps (access path selection, creating B-tree ranges, and the linear scan over all fields referenced in the constraints[1]) are solely concerned

[1] We don't have to scan over every field. We only have to scan over all fields that weren't in the index and all fields that were in the index that had exclusive bounds. The reason for the latter is explained in Section 4.3.2.

```
// Index schema
index_schema = [
  ( field_path: ["a"], default: Value::Int(0) ),
  ( field_path: ["b", "c"], default: Value::String("World") ),
  ( field_path: ["b", "e"], default: Value::String("some value") ),
]

// Document
document = {
  "a": Value::Int32(42),
  "b": Value::Dict({
    "c": Value::String("Hi"),
    "d": Value::Int32(-2)
  })
}

// Index instance for this schema and document
index_instance = [
  Value::Int32(42),
  Value::String("Hi"),
  Value::String("some value")
]
```

Figure 5: Illustrative example of an index schema and an index instance (using RON-like syntax for illustrative purposes)

Operation	Query?	Pool operation
Create/Insert	No	Insert
Read/Find	Yes	Find at location
Update	Yes	Update at location
Delete	Yes	Mark deleted at location

Table 2: Summary of CRUD operations

with the constraints document. The last step (sorting and projection) deals with the specified result order and field projection, and may not be applicable to all CRUD operations.

4.3.1 Access path selection (a.k.a. finding the best matching index)

The "access path selection" step of DBMS query optimizers is usually used to select the best method to find the requested records. In the case of CUDB, there is only one form of access path: find the best matching index, and then linearly scan to filter out documents that don't match the constraints[2]. Thus, the only access path selection lies in selecting the best matching index.

This step is very simple. A matching (candidate) index schema is one that has all of its fields match the constraint document (including type matching); i.e., the fields in the constraint document should subsume the index schema. A maximal candidate index schema is chosen as the "best" index schema. If there are multiple maximal candidate index schemas, an arbitrary one is chosen.

Note that this is about the simplest query optimization scheme possible, and could be greatly improved with catalog information. It would also have to be greatly revamped if there is a more advanced set of query access paths.

4.3.2 Creating B-tree ranges to scan

B-trees can provide range searches: if the key type is some totally-ordered set S, then we can easily query all documents in a specified interval $[S_1, S_2]$. The time complexity of finding this range and iterating over its documents is $O(\log C + |C[S_1 : S_2]|)$, where $|C[S_1, S_2]|$ represents the number of documents in the collection in the specified interval. Note that equality is a degenerate form of range search, in which $S_1 = S_2$.

In our case, our key type is a n-tuple of values, sorted lexicographically. Thus a range must be specified as a pair of n-tuples. Assuming we have an index schema with integer fields $[a, b, c]$ (we relax the notation in this section to illustrate B-tree properties), and a constraint $(a \leq 3) \wedge (b = 2) \wedge (c \geq 5)$, then this translates to the following n-tuple range:

$$\begin{bmatrix} -\infty \\ 2 \\ 5 \end{bmatrix} \leq \begin{bmatrix} a \\ b \\ c \end{bmatrix} \leq \begin{bmatrix} 3 \\ 2 \\ \infty \end{bmatrix}$$

This approach is relatively straightforward, and is how range searches on multi-field indices are constructed, but there are a few difficulties with this approach:

Mixing inclusive and exclusive constraints Assume in the above example that the constraint included the exclusive bound $(a < 3)$. Then the above problem could not be expressed using an inequality, because there is a mix

[2]Theoretically, we only have to scan over all non-index fields. However, due to a quirk in our implementation, we must scan over every field. The reason for this is explained in Section 4.3.2.

of inclusive and exclusive constraints in the upper bound of the range. To solve this, CUDB changes all fields to inclusive bounds, and then in the linear scan re-checks the constraints on index fields to filter out elements which might have been falsely satisfied due to the incorrect bounds.

Upper and lower bounds on unbounded types In the above example, values for negative and positive infinity are required for lower and upper bounds of one-sided inequalities. This can be solved in two ways: by establishing a minimum and maximum value for each value type, and substituting that value for infinity; or by establishing special value types `Value::PosInf` and `Value::NegInf` that always compare larger and smaller than any other value, respectively (care must be taken that these values may never exist in a document). The former approach was taken, but the latter approach was thought of after the fact and is more robust in the case of unbounded types such as strings, for which there is no true maximum value and only an approximation for a maximum value can be used.

Conjunction (intersection) and disjunction (union) of constraints Consider a conjunctive constraint on a single field such as $(d > 3) \wedge (d < 5)$. This requires merging the ranges of the left and right subconstraints. Also consider a disjunctive constraint on a single field such as $(e < 3) \vee (d > 5)$. In this case, we have to split the query into two ranges. In general, we can expect to represent any set of range constraints as a set of disjoint continuous ranges in disjunctive normal form (DNF); each continuous range is represented by a single or pair of constraints, and these may all be connected together via disjunctions.

Additional research into how range searches with a diverse set of constraints are implemented on multi-field indices may be greatly helpful for simplifying this process. An alternative proposed scheme is to use chained B-trees, where each field has its own level of B-tree and thus may be constrained independently; however, this is prohibitive in the number of B-trees it spawns. It is unknown how MongoDB implements this.

4.3.3 Linear scan over non-index fields

The previous step filters out all documents that don't match the constraint on the fields specified in the index. An additional step is required to filter out all documents that don't match the constraint on the fields not specified in the index.

Note that if no index is declared or if no index matches the constraint, this is simply searching using a linear scan on the entire collection.

Note that this step should also check the constraints on the index fields, due to problems with perfect matching on index fields in the previous step. In other words, the previous step will produce a superset of the matching documents, but some of those documents may be false positives (in the case of documents

missing index fields, where we use the default value instead, or the problem with inclusive/exclusive bounds).

Note that this step can also check more complicated constraints than index constraints. Recall that indices were constrained to scalar types (which have a total ordering, assuming all values of the field have the same type). In this step, CUDB can check constraints such as the `cudb::query::Constraint::In` constraint, which checks for the existence of the value in a set of values.

In this step, the idea of what it means for a constraint to match an inconsistent value becomes relevant. At this step, some fields are either missing or of the wrong type. For simplicity of implementation, if a value is either missing or of the wrong type, then it will not match the document. Note that this is even true of the not equals constraint; in other words, neither checking equality or inequality against a value will return a document. This may feel inconsistent, but it is an issue with the "three-valued boolean logic" or nullable values. Future work can be done making this behavior more intuitive.

4.3.4 Sorting and projection

Sorting is performed on the documents matching the constraints using Rust's standard `Vec::sort_unstable` function. Projection is performed by mapping the result set through a projection function guided by the projection constraints.

5 Evaluation

At the time of writing this document, we still have to finish queries/CRUD operations on the DBMS. Evaluations of CUDB for speed and correctness will be added to the README once this is complete.

6 Known bugs/incomplete features

6.1 Assumptions

Unit and integration tests were written to test for common bugs and use cases. The assumption is that CUDB will be run under intended conditions. E.g., the following are not accounted for:

Data corruption We assume that the OS/block device is robust against bitrot, and that there are no malicious actors modifying the backing store file.

Concurrent database access We assume that only a single process is accessing a collection at a time. This may be enforced using a filesystem lock (?), but we haven't done this yet.

Insufficient disk space We assume that the inserted data will not exhaust available disk space. (Note that this may be a poor assumption due to the simplistic block allocation scheme.)

Indices fit in RAM While our data pages are distributed throughout page files and can be paged-in with random-access/seek operations in the file, we don't have the same functionality with our indices because we use the builtin B-tree implementation and have little control over how it's serialized in memory. Thus all B-tree indices are loaded into and assumed to fit within RAM.

Reasonable OS page cache scheme The buffer manager simply reads pages from disk when necessary. We assume that (in a *nix environment) blocks from the backing store file are cached in the page cache so that we can quickly retrieve information. We also assume that page caches are invalidated once available memory runs low (e.g., in a LRU fashion). Our mmapv1-like implementation exerts no explicit mmap caching (ironically, given its name), instead relying on the OS to perform it for us.

6.2 Known issues

Some known issues of CUDB are:

Inefficient page allocation scheme The memory/buffer manager is extremely simple for the purposes of this project. First of all, it relies on the underlying OS buffer manager and does not attempt to smartly pre-fetch or evict pages. Also, currently the buffer manager never reclaims deleted pages. This allocation scheme is simple because we simply keep a reference to the end of the pool as a place to allocate new buffers, and this is fine for insert-only buffers, but is highly inefficient for collections that involve much deleting or resizing. A smarter memory allocator would keep a list of free spaces (perhaps in a residency bitmap) and would have better alignment criteria (e.g., block-aligning records). The design of CUDB is fairly modular, and thus the memory allocator can be easily switched by replacing the `cudb::mmapv1` module.

Inelegant/inefficient query range selectors for multi-field indices All searches using an index are implemented as (inclusive) range searches over the index's B-tree, followed by a linear scan to match other fields that are not contained in the index. In the standard library B-tree implementation, there is a method to search over some interval (with optionally closed, open, or unbounded endpoints). However, in our case our B-tree keys are vectors of field values, sorted lexicographically; there is no (easy?) way to sort with different endpoint types on a per-field basis. Thus, the range search is performed over an inclusive range, which raises some issues. One issue is that inclusive upper- and lower-bounds have to be defined on all types, including unbounded types (e.g., strings; an arbitrary upper bound of string comprises 32 characters with the value 255 was chosen.). Another issue is that we cannot simultaneously accommodate inclusive and exclusive bounds, so we have to re-check all constraints on the index fields even

though we use the index's B-tree. More research is necessary to figure out how multi-field indices are implemented in MongoDB.

Overlapping ranges in conjunctive clauses An `OR` constraint in an index field will cause two ranges to be searched. For now, the two sub-constraints in the conjunction are assumed to be disjoint (non-overlapping). If they are non-disjoint, then the overlapping regions will be iterated twice. (This can be solved by checking for and merging non-disjoint regions. We just haven't gotten around to that yet.)

Poor error handling Much of the error handling is performed through the use of the `panic!()` macro. While this is considered "safe" error handling in Rust, it is not very intuitive and leads to a poor user experience. Rather, more graceful error handling should be done through by returning `Option` or `Result` types.

6.3 Missing features

Most of the important missing features are mentioned above. Some missing features that may be relatively simple to implement (given our existing progress) include:

Automatic ID's. Currently, there is an ID type supported but it is unused. To distinguish between documents with the same value, the user should manually create a unique ID field.

Collection schema Enforce a specification on each document in the collection. CUDB already enforces a specification on any field that is part of an index; this functionality can be furthur extended to the entire document. Of course, this will somewhat limit the flexibility of the documents.

Database-level organization Currently, there are only collection-level operations. CUDB does not support organization of collections into databases.

Aggregation operators CUDB does not support anything like the MongoDB Aggregation Framework, such as grouping operations or cross-table lookups.

7 Conclusion

It was a great learning experience for all of us. This project allowed us to learn about Rust, a programmer's favorite. Rust was a good choice for a combination of speed, high-level safety-features, and understandable documentation/error messages. We were also able to think through many of the difficult design decisions that govern schema-less, dynamically-typed, document-based DBMSes like MongoDB. While we did not implement many of the more advanced features related to scalable and reliable (distributed) systems that are characteristic of NoSQL DBMSes, we were also forced to consider some of those design details as well.