Maximilian Lai (ml155, ml575); Nathan Chou (nwc1, nathanchou21); Shivam Pathak (sp151, pathaks1)

Design Description:

We have one package for our skip-list implementation. This package makes no assumptions about how the skip list will be used and imports no other packages. It simply defines a concurrent safe and efficient data structure that can store key value pairs and methods that allow them to be found, inserted or updated, removed, or returned in a list based on minimum and maximum bonds for key values.

We also have a documents package to represent documents in our system. The document package makes no assumptions about how it is used and imports no other packages. The creation of a document requires some kind of concurrent safe data structure of ordered keys paired with values to be passed in in order to manage collections. This allows concurrency management to always be the sole responsibility of the data structure, in our case skip-list. This data structure must conform to an interface which will allow the document package to manipulate it. The data structure is created and injected through main, a process which also sets the data structure's key and value generics to the appropriate types. Additionally, the type of collection stored inside documents is not even known or relied upon by the documents package, as it also uses a generic controlled by main to define what it contains.

Databases and collections are implemented very similarly in another package. Like with documents, the collections package relies on the injection of a concurrent safe data structure (in our case skiplist) to manage the documents it will contain. The type of the documents is defined by a generic in both the data structure and the collections package.

In general, our approach was to set our generics in main to match interfaces defined in our handler package. Our handler package manages http-requests, and by doing this we allow the handler package to not need to import anything, just be injected with factories to items which are pre-set to return types defined by handler interfaces. The handler package is instantiated by main with a concurrent safe (empty) data structure of outermost databases. As it receives requests, it passes them off to appropriate methods within the handler class which operate on this data structure. These methods generally call helper functions to parse and traverse over the incoming url path, and then call methods defined by handler interfaces as appropriate, with the factories ensuring all items the handler interacts with conform to handler interfaces.

Our subscribers are all stored in their respective collection (or parent collection/database). This enables subscribers to specific documents (document name) to remain subscribed to that document even after that document gets deleted; in other words, updates would still come if a document with the same name is recreated. This is implemented through "intervals" where if a subscriber subscribes to a specific document, then the interval is set to be just that document and intervals are set accordingly for collection subscriptions.

The modification of document data using PATCH operations is handled using the visitor pattern; as this is significantly different from all other operations, we thought it would be necessary to separate this functionality into its own package, patchvisitors.

Design Principles:

We incorporated the dependency inversion principle by decoupling every package from every other package (other than importing other packages, and packages using the provided json package). Our strategy for decoupling centers around the needs of the handler package. The handler package uses interfaces to define all the methods it needs from other packages, with these interfaces always referencing other handler interfaces. For example, the handler Collectioner interface requires a find method which will return a handler Documenter interface. In order to conform with these interfaces, the methods in packages like collection and document must have headers that return handler interfaces. For creating new documents, databases, and collections, this is accomplished using factory methods which Main creates to return handler interfaces, thus matching requirements in the handler factory interfaces. Additionally however, when main is calling methods to create new documents, collections, etc as it sets up the factory methods it is also setting generics within these methods to be handler interfaces. These generics are used within packages like document and collection to make sure method headers are compatible with handler interfaces. For example, because it has a generic of the handler's Documenter interface, the find method in the collection package will be able to return this interface, thus conforming to the handler's Collectioner interface. Additionally, the call to NewDocument or NewCollection within Main passes in a dbindex(a skip list) that was also instantiated with the appropriate handler interface for the values it will be storing. In ways like these, we allow our packages to be standalone, with the handler package simply defining interfaces with the functionality it needs, and other packages knowing the appropriate types they will need to return through generics passed through main.

Maximilian Lai (ml155, ml575); Nathan Chou (nwc1, nathanchou21); Shivam Pathak (sp151, pathaks1)

Single-responsibility was a core part of how we organized our packages. For example, managing concurrency safety of editing files or the file structure in a concurrent safe way was the sole responsibility of one package (skiplist), allowing the packages of Document and Collection to not have to worry about concurrency and only have the responsibility of their specific roles. With respect to functions, we incorporated the single-responsibility principle by laying out the steps of each main functionality and considering whether or not each step fell into the category of "function" or "logic". A "function" would be something that modifies existing data or creates something new from existing data (but not something like reading in data from URL), while "logic" would be something that can be represented by if statements or loops. If a step was a "function" and was not just a few lines of code, we gave it a helper function; this approach ensured that functions were only responsible for one high-level piece of functionality and also completed that functionality thoroughly. With respect to structs, we created structs if a) the struct represented some kind of concrete concept that bundles multiple data together and/or operate in a specific way using that data, b) if we needed somewhere to unmarshal data into (especially in testing), or c) if we needed to match an interface's methods (concrete visitors for the visitor pattern). In all of these cases, each struct would be responsible for unmarshaling one kind of data or handling methods related to a single concept or pattern.

We incorporated the interface segregation principle by making sure that every time we require a struct from an outside package, we create an interface with only the relevant struct methods that we will use in the current package, even if the actual struct has more methods. In doing this, we ensure that our interfaces don't depend on unnecessary behaviors yet still match to the struct(s) they need to match to, making our interfaces precise and focused.

Concurrency:

Our concurrency of our project hinges on the existence of a concurrent safe data structure of key value pairs, in our case, skiplist. Concurrency management for file management is the sole responsibility of this package, allowing other packages to simply call the methods in this package and make the assumption that the linearization point for the action they are taking is within the function call to this package.

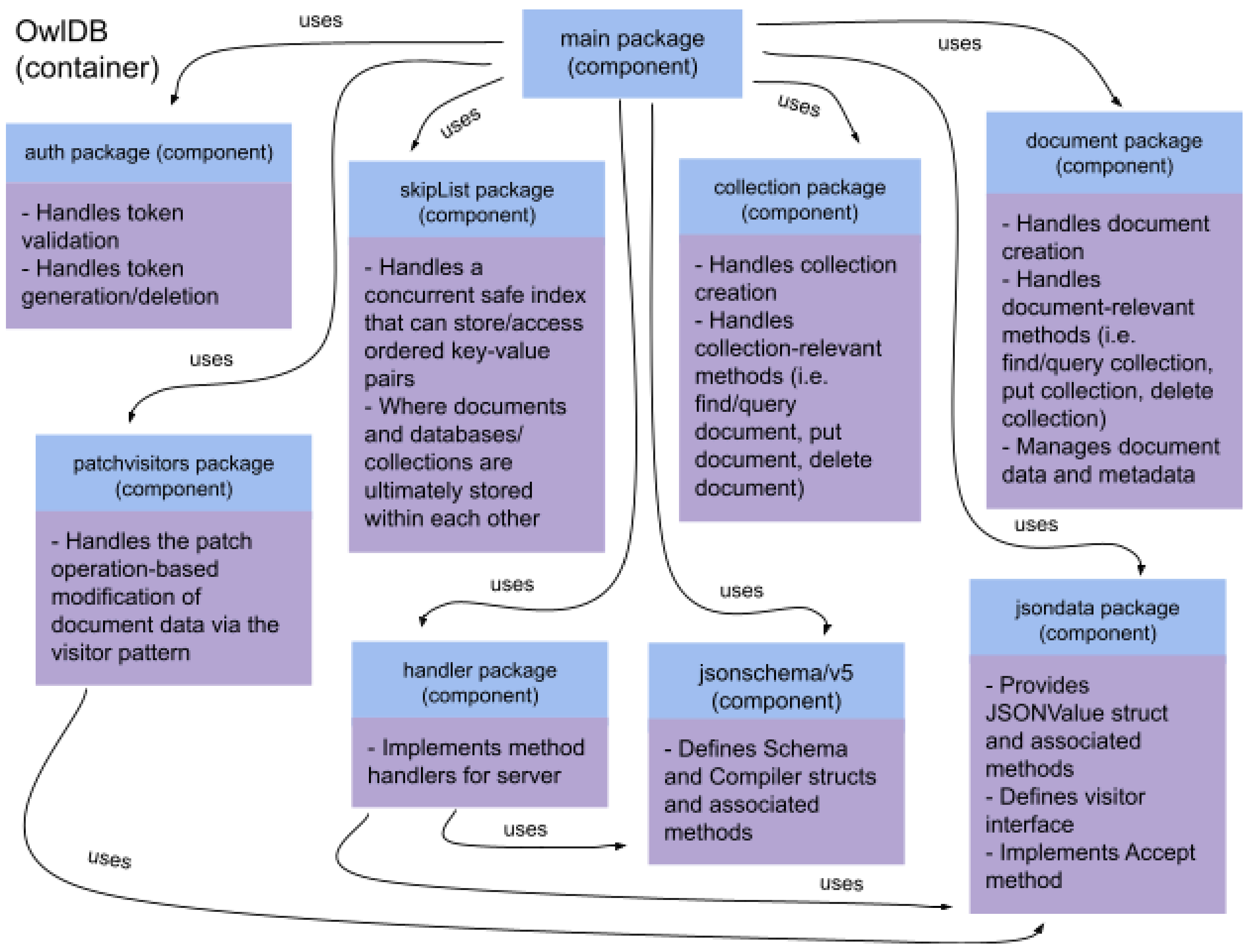
The skiplist is structured like a linked list, except instead of each node pointing to a single next node, nodes point to a list (of variable size) of next nodes. Given a node *b*, the node at any given level *n* of *b's* next list must be the node with the next largest key that has at least n levels. There are also start and end nodes (with minimum and maximum keys) who have next lists that are always at least one larger than the largest possible size of any other node's next list.

Each node has a mutex as part of its struct, and two atomic booleans representing if the node is marked for deletion and if the node has finished being inserted. Additionally, the list of next pointers is also atomic. This atomicity allows us to increase efficiency by minimizing the nodes we need to lock down, since the "status" of a node and the pointers of the list can be changed and read atomically. Atomics allow atomic reads, making find atomic. They also allow atomic stores/removes in inserts/deletes. Locks make sure that two processes don't modify existing values at the same time in interleaved order.

To make the *Find* method concurrent safe, skiplist relies on each nextlist being a list of atomic next pointers. Because of this atomicity, Find can navigate the skip list in O(log n) style and if it finds an unmarked and fully inserted node, that node was in the skiplist at some point during the find. Insert and delete only need to lock all the predecessors to the node of interest to ensure all the predecessors point to the same new/old node, and updating also locks the node of interest so only one update can change it at a time. This ensures that other concurrent insert/delete/update operations don't interleave, creating different orders in the respective levels of the skiplist. Query relies on repetitively trying to make two passes over the given range of the skiplist, so that if the two passes have the same result we know that the entire result was in the skiplist at some point.

Subscriptions handle their concurrency using channels. Each subscriber has a "message channel" and "done channel". When someone subscribes to a document or collection, message and done channels are created for this subscriber and stored in the corresponding collection. Whenever updates or deletes happen to a document, the corresponding message is sent to all the subscribers (where a copy of these subscribers are created to ensure concurrent safety) corresponding to this document via the message channel. The subscription request continuously tries to read from the message channel until the subscriber closes. Whenever the subscriber closes, the done channel also closes and the subscriber is deleted from the map of subscribers stored in the corresponding collection. Whenever an update or delete tries to write to the message channel, it also tries to read from the done channel (via select/case where closed channels always output on reads) so that it does not get deadlocked by trying to write to a channel that may never be read from again. Since only one message at a time is read from the message channel, the writes to the subscriber will also not be interleaved and will thus be concurrent safe.

Maximilian Lai (ml155, ml575); Nathan Chou (nwc1, nathanchou21); Shivam Pathak (sp151, pathaks1)


OwlDB (container)
main package (component)
uses
auth package (component)
- Handles token validation
- Handles token generation/deletion
skipList package (component)
- Handles a concurrent safe index that can store/access ordered key-value pairs
- Where documents and databases/ collections are ultimately stored within each other
collection package (component)
- Handles collection creation
- Handles collection-relevant methods (i.e. find/query document, put document, delete document)
document package (component)
- Handles document creation
- Handles document-relevant methods (i.e. find/query collection, put collection, delete collection)
- Manages document data and metadata
patchvisitors package (component)
- Handles the patch operation-based modification of document data via the visitor pattern
handler package (component)
- Implements method handlers for server
jsonschema/v5 (component)
- Defines Schema and Compiler structs and associated methods
jsondata package (component)
- Provides JSONValue struct and associated methods
- Defines visitor interface
- Implements Accept method