CS286A Metadata Repository Architecture

Enrico Tanuwidjaja, Feiyang Xue, Jimmy Su

Overview

Our group worked on building a repository that can store metadata passed from the Data Mover. When discussed during the beginning of the semester, the goal was for the repository to:

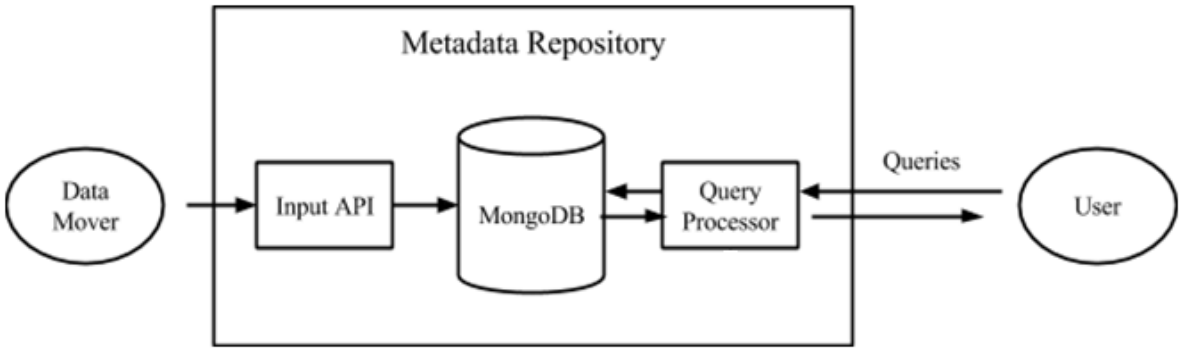
- have a schema to capture relevant information from a set of prototypical tasks and tools
- be extensible to new tasks and tools with varying degrees of opacity
- be able to scale up to large volumes of metadata and high access rates.

The metadata repository is built with MongoDB as its supporting foundation. MongoDB is an open-source document database that both performs dynamic queries and supports nested key-value pairs, and this is important because early on our team as a whole was collectively targeting ways to pass and store data in a JSON format: a key-value standard that was universal across our three separate components.

We chose an essentially flat scheme of storing data: a commit consists of JSON data, where the value for *metadata* is a set of nested JSON data one level deep. We decided on this scheme (1) due to time constraints, and (2) due to the fact that keys can be path names, so if we ever did need to store metadata further than one level deep, we can just request that the metadata can have a pointer to other metadata to maintain the flatness.

As we accomplished our goal, we also defined the API that allows the stored metadata to be queried and/or manipulated. Given a limited time frame, we opted to implement those commands that are more common and practical among data queries. (See the section on “Input API/Query Processor” in the Description of Major Components".)

Architecture



Description of Major Components

MongoDB: (as opposed to Apache CouchDB)

- Queries are easier with MongoDB
- MongoDB does dynamic queries (so you don’t need to know the keys in advance); in general, we believe this means that MongoDB is more flexible

- CouchDB is for occasionally-changing data, so if data changes too much, use MongoDB instead (i.e. CouchDB is useful when versioning is important)

Input API / Query Processor:

- *Main.java:* main program; connects to the machine where the metadata repository is stored
- *MetadataRepo* class: a class that connects to a MongoDB database and handles all interactions with the database
    - *commit*: user can enter metadata into the database
      `syntax: commit [filename] [metadata] [timestamp](optional)`
      `e.g. commit foo.csv {a:"b", hello:"world"} 01/30/1992`

    - *dump*: user can view all metadata in the database
      `syntax: dump`
      `e.g. dump`

    - *show*: user can view a particular file from the database
      `syntax: show [timestamp](optional) [filename]`
      `e.g. show 01/30/1992 foo.csv`

      ** Note: *show* takes advantage of the built-in query capabilities in MongoDB so that memory is not wasted to manually iterate through all files

    - *find*: user can view all files with a particular key-value pair from the database
      `syntax: find [timestamp](optional) [query]`
      `e.g. find 01/30/1992 owner = Joe and cell_count > 50`

      ** Note: *find* takes advantage of the built-in query capabilities in MongoDB so that memory is not wasted to manually iterate through all files

      ** Note: the [filename] of a commit is stored at the same level as the [metadata], but the metadata ALSO contains the filename in its metadata. So, a user could query to find the filename (i.e. the full filename path).

    - *rm*: user can remove a file
      `syntax: rm [filename]`
      `e.g. rm foo.csv`

      ** Note: the *rm* command is ideally for administrative use only (e.g. testing purposes), and not directly accessible to a user (since it permanently deletes the whole history of a file)

- *clear*: user can delete a namespace from the database
  ```
  syntax: clear [namespace]
  e.g. clear bar
  ```

- *namespace*: user can change the current namespace
  ```
  syntax: namespace [namespace]
  e.g. namespace bar
  ```

- *Parser* class: a utility class that provides a function to parse queries (used by the *MetadataRepo* class)
  - *parseTime:* parse a string (formatted as "MM/dd/yy" or number of ms since epoch) into a Date
  - *parseGlob:* parse UNIX-style glob string. Now only supports * and ? wildcard characters
  - *parseExpression:* parse an expression. For any query expression, this should be the start of the grammar
  - *parseKey:* parse a string that represents a key in the metadata key-value pairs
  - *parseValue:* parse a string that represents a value in the metadata key-value pairs; currently only supports string literals, decimal, and floating point

Interface to Data Mover

The Data Mover group will ensure that incoming metadata from the Crawler group is of a JSON key-value format. They will then commit the data to the metadata repository following the syntax of the *commit* command that we specified. (See the section on “Input API/Query Processor” in the Description of Major Components".)
Some fundamental parameters to have with every commit include the filename, the metadata itself, and the timestamp of the commit. (Note that by having a timestamp, our queries can look at data farther back to specific points in time.)

Version 2 Considerations

- Need to think about implementing additional user queries such as:
  - select
  - join
- Implement timestamp to not only be in the form MM/dd/yyyy, but to also include hours, minutes, and seconds
- Add user-defined functions, like max() and min()