Timeline Prototype Designs

Suggested Ideal Solution: Postgre Design

Because:

- Though slower for reads(2.5x slower), it is much faster(10x faster) for writes. Since we need a lot of both, Postgre wins.
- Commonly used with django models
- Plentiful documentation

Potential Postgre solution (not ideal normalized solution, but rather for given constraint of 1 table only):

- Structure
  - 1 table for all Scientific Projects.
    - It will hold ALL your current info (title, wiki, project id, date, ...)
    - 1st row for a given project id will be complete. Rest can be sparse
      - after some constant k updates, must store snapshot. Allows for constant time lookup for both historical and current view project.
    - rows will be sorted by project id and then by date ← easily done in django.
- Benefits
  - writes are fast. Must faster than Mongo version (10x faster based on jmeter tests)
  - Because of transactions, no incorrect overwrites.
  - ACID all the way
  - supported by django and django models. Thus lots of resources available.
    - Helps when teaching new interns
    - helps when hiring contractors
  - for numerous tasks, can look at postgres, oracle, mysql or any database's solutions. Thus multiple different sources for solutions to problems.
- Flaws
  - reads are slower(2.5x slower based on Jmeter tests)
  - documentation isn't as nice as mongo
  - If we eventually want to optimize so that the historical searching is done inside the database, it is harder to work with sql syntax than javascript
  - does not linearly scale

Potential MongoDB solution:

- Structure
  - 1 document for each Scientific Project.
    - It will hold ALL your current info (title, wiki, project id, most recent update date...).
    - It will also hold a list of History documents.
      - History documents will hold title, wiki...
      - should be a sorted list
- Benefits
  - Reads are very fast for
  - Fast migrations to and from key-document based databases
  - Because of single document transactions, you will be sure that no incorrect overwrites occur.
  - Fast migrations to and from key-document based databases
  - linearly scales
- Flaws
  - BIGGEST problem is that mongodb is NOT yet officially supported
  - writes are slower
  - if you have a problem, there arent as many mongo specific resources. Nosql databases defer very much in syntax and inner workings.

Potential Hybrid solution:

- Structure
  - 1 table for current version of each Scientific Project in Postgre
    - holds ALL current info (title, wiki, project id, most recent update date...).
  - 1 collection for all historical versions in Mongo
    - holds historical info (title, wiki, project id, date...)
    - Sorted by date
  - when reading current version, only look at postgre
  - when reading historical version, first look at postgre then at mongo
  - when updating, store recent updates in postgre and then batch send to mongo
- Benefits
  - scales linearly for historical part (bulk of data so very good)
- Flaws
  - mongodb is NOT yet officially supported
  - difficult to make django models use both databases
  - for updates, still have to look at both databases
    - a way you can get around this is by having a batch process that moves all your recent updates in the postgres databases to mongo.
  - Becomes complex very quickly
    - not good for new interns since complex
    - not good for django contractors since it becomes highly customized

Key differences in Django between Postgre and Mongo versions:

- Mongo does not use django serializers, unlike Postgre
  - thus making mongo faster, but less secure
- Mongo does not require migrations
- MongoEngine(django model module I used) written on top of pymongo. It is not officially supported by mongodb or django.
  - There is a push by the community to allow for nosql db's in django models. Not there yet, but making progress. It is required in todays world.
  - MongoEngine requires django nonrel (django version for non-relational db's)
  - MongoEngine requires djangotoolbox (utilities for non-relational django)
  - can't use same models as normally used to do specialized queries or deletions.
    - Had to use pymongo itself (not the mongoengine wrapper I used to make mongo and postgre versions nearly identical) to delete projects. This meant establishing a new connection to the database.
- Postgres is well tested and used with Django
- Issues with having hybrid in Django
  - there are two ways to have your models use different databases
    - 1) use a customized database router for your models
      - have problems with this because you are using an different fork of django with mongo.
    - 2) force your query to use a specific database at each call.
      - Very bad since it requires you to customize all your calls. Plus there are lots of easily made mistakes possible due to this.
      - For example, you can create a model object using user input, then commit it to 1 db. Commiting will force extra information onto your object which will mess up your commit to second db.
        - Many such mistakes make this is a bad option. Not recommended on Django website either.

Structure and process of my Timeline samples

Structure of Postgre Version of Timeline Prototype:

- Timeline table stores title, author, wiki, project id, and date
  - sparse table. Except for 1st entry, any can be not completely filled.
- Timeline is not technically sorted internally but since it is added sequentially, it sort of is.
  - Forced sort by date can be easily done using django models.

Structure of Mongo Version of Timeline Prototype:

- Timeline document contains title, author, wiki, project id, date, and history fields.
  - Timeline stores current values of each of the documents.
  - history is a list of History documents
- History contains title, author, wiki, and date.
  - History is not sorted internally within the database, though it is easily possible to do so using django models.

Project creation, project view, historical project view, and updates are almost the same in both versions that I actually wrote. The differences are only in storing the information as per its structure. The below describes each.

Project Creation

- input is project id, date, title, wiki, author. None are optional.
- Checks to see if project by input project id exists. If not, then creates new project.
  - Mongo – Creates a Timeline Document with initial History Document with the input data.
  - Postgre – inserts regularly into Timeline table. 1St row must be full.

View current version of Project

- input is project id
- if project id exists, returns title, wiki, author.
  - Mongo - you can just get it quickly since current is stored seperatly.
  - Postgres – must sort and traverse. Slow.

View Historical version of Project

- input is project id and date
- django python attains the historical versions of the project, sorts them by date, then finds value of author, wiki, title closest but previous to input date
  - mongo – looks in seperate list, thus can ignore all historical data with projections for normal non-historical views
  - Postgres – looks through entire list. NOT ignoring any data in this case. Partial cause for slow historical reads.
- since we take snapshots of the historical project after every 10 updates, going up db to get appropriate version is done in constant time.
  - Does take extra space
- the sorting step still takes a long time in current version. BUT, both can simply be

forced to be internally sorted ie. store in sorted list. Makes slow updates but faster reads.
    - Idea that we can have a hybrid that has fast writes due to postgre and fast reads due to mongo is tough since you will always have to store your updates.
        - With that said, we can follow leveldb and store 2 trees of information. One for current-ish and one for historical. We can have store all current+say 1 month of data in a shorter database so you can in fact have fast writes in it simply because there is less data. Then, we send this data to our larger historical database in large batches.

Update Project

- Input project id, date, and optionally any of wiki, title, authors.
- Validates and stores in databases
    - Mongo – updates Timeline values AND stores new version into history as well
    - Postgre – just stores regularly in Timeline
- every 10 updates, backtraces database and creates a snapshot. Stores this snapshot (which includes the most recent update) instead of just the most recent update.
- the key issue in mongo usually is the lack of transactions. However, with only 1 document for each project, it is not an issue