**Resource**

1. <https://pypi.python.org/pypi/pycurl/7.43.0>
2. http://flask.pocoo.org/

**Design and Implementation**

**Assumptions:**

* Query server implemented as web server with rest api interface for data insertion and querying
* To insert data we can use clients or use curl command
* Data insertion and response is all JSON based
* Data is defined as {“name”:<string>, “rank”:<int>}
* The name field of data can be case sensitive
* Duplicate names are acceptable, just that they override previous insertion
* For simplicity we are not solving issues related to synchronization of writes and reads

**Data Insertion Protocol**

Send Data as Post message with data encoded as [{“name”:<string>, “rank”:int}]. Once the data is for inserted, we inform the user about the number of records inserted.

API :: <IP>:<PORT>/insert

E.g. curl example::

curl -H "Content-Type: application/json" -X POST -d '[{"name":"hallow\_hello\_hello123", "rank":200}, {"name":"meow\_abc", "rank":400}]' http://localhost:8900/insert

**Data Query Protocol**

Send query as Get request with query defined as follows: <IP>:<PORT>/query/<query\_val>. The response to this request are top 10 highest ranking prefix as per logic.

We have given use the alternative to get performance data as <IP>:<PORT>/perf\_query/<query\_val>. The response to this request are top 10 highest ranking prefix as per logic and the time taken to run this query.

The server internally maintains a data structure which is a prefix tree defined as follows:

* Each node contains {dictionary of child nodes aka node dictionary, dictionary of data}
* Dictionary of child nodes consists of key=<character>, value=<node>
* Dictionary of data consists of key=<name>, value=<rank> if a particular nodes is a match for a given prefix
* The dictionary of data is empty for root node
* By using data dictionary, we make sure that when duplicate entries comes, we just override the previous ones

**Insertion Process**

Before insertion starts, for a given name we find the possible number of substrings that can match it. This is possible since a ‘name’ can be of type ‘str1\_str2\_str3’. So any time, we have query with possible value ’str1’ or ’str2’ or ’str3’, there will be match. Based on this fact, it is safe to assume that ‘str1’, ‘str2’, ‘str3’ are the possible search strings, that can match a prefix search against the name ‘str1\_str2\_str3’. So we take each such sub-string and insert it for the data set <name, rank>. For such insertion, we traverse the prefix by iterating through each character in the substring and reaching a node where the last character matches or there are parts of it remaining. In case where full match occurs, we insert the <name, rank> to the data dictionary for the node, otherwise we continue creating nodes for each remaining character of the substring with data dictionary being empty. When we reach the last node, we define the data dictionary with <name, rank>.

Time Complexity:: Insertion requires us to traverse O(n) where n is the length of the substring. Since there can be k such substrings in a name, we define the process as k\*O(n). We are assuming that searching each node of the tree is using a dictionary so the operation is O(1) since the key is formed of constant number of characters.

**Querying Process**

For querying we recursively search the prefix tree such that each node’s node-dictionary has a character in substring matching it. In case, matches are not found, we return empty list of data. Once the match is found, we use the node as a the root node to traverse and capture all data elements from every nodes’ data dictionary. As we are collecting this information, we compute the top Nth ranked matches to our query.

Time Complexity:: Querying requires us to go through O(n) times where n is length of the prefix query. Once we have a match we have to collect all node information for the sub-tree of the node where we find the complete match. So we have to go to all possible K nodes. And each K node requires to maintain the top 10 ranked ‘names’. This requires 10 K iterations. So we need ~ O(n + 10 K).

**Implementation Requirement**

* Language :: python 2.7.1
* Web Technology:: Flask
* OS :: Mac or Linux
* External Libraries for python:: flask. Pycurl, pyaml

**Code Directory**

**src/lib/**

* client.py :: Logic for streaming data as JSON to query server
* query\_server.py :: Logic for defining query service via FLASK
* index\_utils.py :: Logic for defining index data structure and its operations
* logger.py :: logging utility

**scripts/**

* run\_server.py --port=<port> :: script for running server

**tests/**

* test\_indexing.py :: tests for indexing data structure
* test\_integration.py:: end to end smoke tests for testing query server

**Logging**

The system generated run\_log.log and error\_log.log for for debugging needs.

**Alternatives**

The system is designed like an in-memory system. To make the system persistent, we might choose to stream the index to a file. We can shorten the size of storage by segmenting the index at the root node since each child node is mapped to a unique character. Also, we can choose to distribute the data to different servers where each server matches the beginning of all possible characters. This helps improve performance since we do not have to lock the index when we are having multiple writes. Also, querying is faster since we can distribute the load across the servers.

Another approach would be to use mysql as the backend relational database and store the data with substring, list of data <name, rank>. Since substring is unique, we can build a primary index on top of it for searching. For indexing we will pick B-Tree as our choice. MySQL will take care of our synchronization requirements as well.

**Deployment Plan**

We have provided a script to start the server as follows:

1. Install Server :: make install
2. Run Server :: make run\_server --port=PORT\_NUMBER
3. Stop Server :: make stop\_server
4. Run Tests :: make smoke

For simplicity, upgrades are planned to be offline

**Test Plan**

* Test Indexing feature
* End to end smoke test