Report

by. HoJoong Lee (2011171061)

Used Datastructure

Hash

- **개요**
  프로젝트에서 요구하는 기능들중 **검색**이 많은 비중을 차지한다. 예를 들면 특정 user, tweet를 찾아 세부값을 이용하는 경우가 있다. 하지만 user나 tweet의 정보량은 무궁무진하게 커질 수 있기 때문에, 그에 적당한 자료구조로 **Hash** 채택했다. 비록 Hash Table의 메모리 할당량이 클 수 있으나 알맞은 key값을 설정하면 원하는 데이터를 검색하는데 O(1)~O(n)의 빠른 처리를 기대할 수 있다.
- **쓰임**
  1. **UserDB.list** : User 데이터 묶음
     주어진 데이터 파일들을 참고해보면 UserProfile의 Identification Number(ID)로 유저를 언급(reference)하는 것을 확인할 수 있다. 그러므로 ID값을 key로 하는 Hash 가 필요함을 느꼈다.
  2. **TweetDB.list** : Tweet 데이터 묶음
     기능들중 단어를 검색을 요구하기에 각 글자마다 정수로 변환하여 처리한 특수값을 key로 지정한 Hash를 만들었다. 이때 특수한 경우 다른 단어지만 같은 key값을 공유하는 경우를 방지하고자, Hash Table 접근은 변환한 특수값으로 하되 그 이후 **Linked List**에서의 검색은 단어를 key로 하였다.

Linked List

- **개요**
  기본적으로 **Hash**를 구현하기 위해 필요한 자료구조다. 또한 메모리를 차지하는 비중이 적기 때문에 단순한 정보 축적을 위한 도구로 쓰기 좋다. 이때 눈여겨 볼 점은 Linked List를 구성하는 LNode에 **key값을 추가**했다는 것이다. Hash나 다른 상황에서 **검색**을 할때 검색값(key)과 다른 반환값(v)이 기대하는 경우가 있다. 예를 들어 특정 id값을 갖고 있는 User 인스턴스를 찾고 싶을때 key는 id가 되고 v는 해당 인스턴스가 되는 것이다. 그러므로 Linked List를 구성하는 노드 LNode는 다음값으로 구성되어 있다.
  - LNode.next
  - LNode.key
  - LNode.v
- **쓰임**
  1. **Hash** : Hash Table 구성요소
  2. **Tweet.userList** : 해당 Tweet 단어를 사용한 User 목록
     소스코드에서 서로다른 유저가 같은 단어로 Tweet을 할 경우 해당 Tweet.userList 목록에 축적되도록 구현되어 있다. 요구하는 기능에서 Tweet를 한 유저를 검색할 경우 단일이 아닌 모든 유저의 반환을 기대하며 할뿐만 아니라 **중복**을 허용하기 한다. 즉, 모든 User 목록을 검사해야하므로 Hash가 아닌 Linked List를 채택했다.
  3. **Edge.list** : Edge(유저간의 관계, A->B) 목록
     2번과 마찬가지로 Edge들을 특정 유저의 관계를 검색할때 단일한 key로 검색을 못하며(검색한 유저가 A 또는 B인 경우), **중복**을 허용한다. 모든 Edge 목록을 검사해야하므로 Hash가 아닌 Linked List를 채택했다.

List (python)

- **개요**
  파이썬에서 기본적으로 제공해주는 자료구조로 다양한 기능을 제공한다. 가장 큰 장점은 array의 형태로 특정 값 접근이 빠르며, 제공하는 sort의 Time Complexity가 O(n log n)이다(링크). 비교적 많은 메모리 할당에 불구하고 위 자료구조를 사용하는 이유는 접근이 용이하며, sort가 빠르기 때문이다.
- **쓰임**
  1. **Hash** : Hash Table 구조
  2. **UserDB.followRank** / **UserDB.tweetRank** ... : 순위표

Overall Class

Nodes.py

Data파일의 정보를 담고 있는 가장 작은 단위

- User
  - id : 번호
  - userName : 텍스트
  - tweetCount : User가 작성한 Tweet 개수
  - followCount : User가 Friend한 User 개수
- Tweet
  - word : 텍스트
  - userList : 해당 Tweet를 작성한 User 목록 [Linked List]
  - tweetCount : 해당 Tweet를 작성 개수(중복 인정)
- Edge
  - A : Follower의 UserID
  - B : Followed의 UserID

DB.py

Nodes를 묶어 놓는 Database

- UserDB
  - list : User Node 목록 [Hash]
  - followRank : User의 Follow 많은 순서대로 나열한 User 목록 [list]
  - tweetRank : User의 Tweet 많은 순서대로 나열한 User 목록 [list]
  - totalUser : User 총 개수
- TweetDB
  - list : Tweet Node 목록 [Hash]
  - wordRank : Tweet한 User의 개수가 많은 순서대로 나열한 Tweet 목록 [list]
  - totalTweet : Tweet 총 작성 개수(중복 인정)
- EdgeDB
  - list : Edge Node 목록 [Linked List]
  - totalEdge : Edge 총 개수
- MasterDB
  - UserDB
  - TweetDB
  - EdgeDB

Expected Performance

0. Read data files

Time Complexity : O(n) * O(Hash)

> Check : MasterDB.readFile()

이는 제공하는 데이터 파일의 신뢰도에 따라 Time Complexity를 조절할 수 있다. 작성한 소스코드에서는 User 데이터를 추가할때마다 중복 ID가 있는지 확인을 한다(참고 : *UserDB.addUser(key,n)*). 또한 Word 데이터를 추가할때도 이미 같은 단어가 있는 Tweet가 있는지 확인한다(참고 : *TweetDB.addTweet(key,n)*). 이때 중복확인을 위한 검색은 Hash를 이용하기 때문에 해당 Hash의 효율 O(Hash)에 따라 TimeComplexity가 좌우된다.

1. Display statistics

Time Complexity : O(n log n)

> Check : UserDB.updateFollowRank() / UserDB.updateTweetRank()

작성한 소스코드는 User, Tweet 또는 Edge가 추가 될때마다 통계가 업데이트 된다. 그렇기 때문에 매번 User의 Tweet횟수나 Friend개수를 부를때 O(1)이 걸린다. 하지만 Min, Max 를 구하기 위해서는 업데이트된 통계에 대한 sort가 필요다. 그러므로 이를 list(python)에 대입하고 sort를 하므로 O(n log n)이 걸린다. Sort는 매번 요청할때만 작동하며, 그 이유는 통계 값는 자주 바뀔 수 있기 때문이다.

2. Top 5 most tweeted words

Time Complexity : O(n log n)

> Check : TweetDB.updateWordRank()

1번과 같은 방식으로 동작한다.

3. Top 5 most tweeted users

Time Complexity : O(n log n)

> Check : UserDB.updateTweetRank()

1번과 같은 방식으로 동작한다.

4. Find users who tweeted a word

Time Complexity : O(n)

> Check : TweetDB.searchTweet(word) / Tweet.getUserList()

단어와 일치하는 Tweet를 검색한 다음 해당 Tweet의 userList를 분석하는 방식이다. 우선 검색은 Hash를 사용하기 때문에 O(Hash)이며, 받은 userList는 list(python) 자료구조로 처음부터 끝까지 읽어가며 중복된 UserID들도 세므로 O(n)이다.

5. Find all people who are friends of the above users

Time Complexity : O(n^2)

> Check : EdgeDB.getFollowID(id) / UserDB.getUser(id)

User 목록을 갖고 있는 경우, 한 User 마다 그 Friends를 찾기 위해서는 EdgeDB의 LinkedList를 전부 돌며 검색해야한다. LinkedList 검색 O(n)을 매 유저마다 하니 O(n^2)가 된다. 만약 EdgeDB의 Edge목록이 Hash 자료구조였으면 Time Complexity가 더 감소하지 않았을까 한다.

6. Delete all mentions of a word

Time Complexity : O(n^3)

> Check : TweetDB.deleteTweet(word) / Tweet.getUserList()

Hash로 이루어진 TweetDB에서 원하는 Tweet를 찾아 TweetDB.list(Hash)에서 지울 뿐만아니라 Tweet.userList에 있는 User들의 User.tweetCount 값을 조정한다. Hash를 사용하므로 Tweet 제거뿐만 아니라 검색까지 O(Hash)가 걸리며, Tweet.getUserList를 구하여 해당 User들에 변경사항 적용까지 O(n)이 걸려 전체 Time Complexity는 O(n)이다.

7. Delete all users who mentioned a word

Time Complexity : O(n^3)

> Check : MasterDB.deleteUser(id)

가장 복잡한 함수로, 아쉬운 부분이 많은 부분이다. 각 User를 제거할때, UserDB에 list(Hash) 검색하여 제거하면 O(Hash)이지만, TweetDB의 모든 Tweet의 Tweet.userList 를 검사하여 User 제거해서 O(n^2)이며, EdgeDB의 관계 Edge를 제거하되 그 User를 Friend로 하고 있는 'User들의 'User.countFollow도 조절해줘야 하므로 O(n)로 한 User당 O(n^2)다. 이를 고려하여 Time Complexity O(n^3)이다. 만약 User 마다 User.tweetList가 있어 쉽게 Tweet에 접근했으면 O(n^2)으로 상향 시킬 수 있었을 것이다.

Improvement

User에 TweetList 추가

7번을 보면 알 수 있듯이, 특정 User가 작성한 모든 Tweet를 추적하는데 많은 과정을 거쳐야 했다. 그 이유는 해당 정보가 User에게 종속되어 있지 않기에 모든 Tweet를 거쳐가며 User의 작성 여부를 확인해야 했기 때문이다. User에 TweetList를 추가하되 Tweet가 추가 또는 제거되는 과정에서 처리를 조심하면 될 것이다.

같은 내용 Tweet 구별

가장 많이 사용된 Tweet등을 쉽게 구하고자하는 의도로 다른 User라도 같은 단어를 사용하면 동일한 Tweet 간주하도록 처리했다. 하지만 이는 User를 지울때 해당 User가 작성한 Tweet를 추적하는 과정에 많은 어려움을 줬다. 왜냐하면 해당 Tweet의 userList를 읽어 중복된 개수까지 고려하여 처리

를 해야하기 때문이다. 차라리 같은 User의 같은 Tweet라도 따로 관리를 한다면, 비록 추가 sort에 따른 2번과 4번의 Time Complexity O(n log n)로 증가하더라도 7번 처리를 더 빠르게 할 수 있을 것이다.

EdgeDB의 list를 Hash로 사용

EdgeDB의 list를 LinkedList로 한 이유는 어차피 모든 Edge노드의 A와 B를 확인해야 해서 마땅한 key값을 설정하기 어렵기 때문이었다. 하지만 하나는 A를 key로 하고 나머지는 B를 key로 하는 Hash를 만들면 비록 메모리 공간을 조금 더 차지하더라도 검색 TimeComplexity를 현저히 줄일 수 있을 것이다.

Update 추적

User, Tweet나 Edge 값의 변동이 있지 않은한 Statistics 또는 Top5 등의 값은 바뀔 일이 없다. 그러므로 이미 한번 값을 도출한 이후 그 다음 변동사항이 있기 전까지 불필요한 반복 활동을 방지할 수 있는 bool 값이 있으면 좋다. 특정 값(예를 들어 Statistics)를 구한 이후 해당 up-to-date 변수(bool)를 true로 바꾸어 false가 되기 전까지 추가 계산 없이 저장된 값을 반환하는 형식을 사용할 수 있다.

Strongly Connected Components와 Shortest Path 구현

시간이 부족한 관계로 위 두 기능을 구현하지 못했다. **Strongly Connected Component**(SCC) 같은 경우, SCC기능을 **DFS**를 사용하여 구현한 클래스를 우선 작성하고, User.id를 SCC.addNode에 Edge.A&B를 SCC.addEdge에 넣어 DB의 데이터를 SCC로 넘겨서 도출한 결과를 다시 DB로 반환하는 구조를 생각했다. **Shortest Path**(SP)도 마찬가지로 SP기능을 **Dijkstra Algorithm**을 사용하여 구현한 클래스를 우선 작성하고, DB데이터를 옮기되 User.followCount를 SP.addEdge의 weigth로 넘겨줘 도출한 결과를 다시 DB로 반환하는 구조를 생각했다.