Acolyte: An In-Memory Social Network Query System

Ze Tang, Heng Lin, Kaiwei Li, Wentao Han, and Wenguang Chen

Department of Computer Science and Technology, Tsinghua University
Beijing 100084, China
{tangz10,linheng11,lkw10,hwt04}@mails.tsinghua.edu.cn
cwg@tsinghua.edu.cn

Abstract. WISE 2012 Challenge provides a data set crawled from Sina Weibo, which consists of hundreds of millions of anonymized following relationships and microblogs. In the performance track, 19 typical queries on the data set are given. Most of the queries are to get the top n users or microblogs which have the largest statistical numbers of some features during a specific period. The solution should focus on both throughput and latency in the execution of these queries. In this report, we present Acolyte, which is an in-memory query system that can solve the performance track efficiently.

1 Introduction

Social networking service grows rapidly in recent years. Popular SNS websites like Facebook and Twitter have more than one hundred million active users. The huge amount of user data and contents bring a lot of new challenges to the storage and query system.

The performance track of WISE 2012 Challenge provides a data set crawled from Sina Weibo, a popular microblog service website in China. The data set has $265,580,802$ pairs of following relationships and $369,797,719$ microblogs. The participants are required to build a system which can deal with 19 given queries. Most of the queries are to get the top n users or microblogs which have the largest statistical numbers of some feature during a specific period. The system should have good performance in both throughput and latency.

To achieve high performance, Acolyte, the system we built, loads all the data *in memory*. We use C++ programming language to write Acolyte for both fine-grained control on data structures and efficient execution. The data structures used in Acolyte were carefully designed to get small footprint, while not affect the performance of query. We also optimize several kinds of slow queries.

The rest of this report is organized as follows. Section 2 describes the design of our system, as well as implementation details. Section 3 elaborates optimization we take on several non-trivial queries. Section 4 presents and analyzes the experimental results. And finally, Section 5 discusses our experiences in doing this track, giving some related works.

2 System Design and Implementation

Our goal is to build a system which can process the 19 typical queries as quickly as possible on the given data set. Since the original data we need to query from is very large, commodity database systems such as MySQL[2] and MongoDB[1] do not work very efficiently. To achieve low latency and high throughput, our system remaps, reschedules all the data in the original data set and loads all the data in memory. We also choose a proper selection order and algorithm for each query.

In our design, the server side runs on one machine with comparably large memory (64 GB), and clients run on another machine which connect to server to get the query result.

As Figure 1 shows, we have 4 steps for each query. On the first step, the client transports the query information to the server. On the second step, the server parses the query and transforms all parameters of the query to integers. On the third step, the server creates a new thread to compute the result data. On the last step, the server transforms all the result indices to original strings and sends them back to the client.

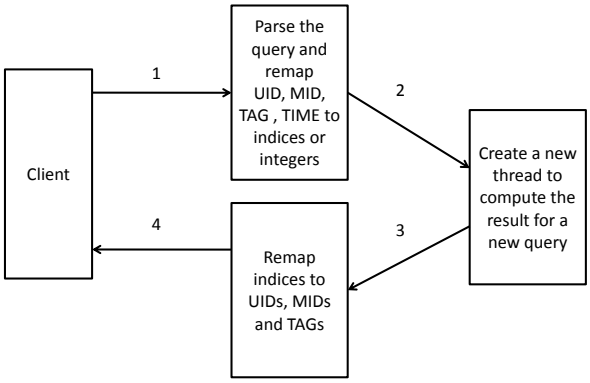

Fig. 1. The architecture of Acolyte

Acolyte does the following things when initializing.

1. Parsing the original data file, and generating the tables defined in the official documentation.
2. Remapping all the parsed tables, and generating new mapped data files.

3. Sorting and rescheduling all the mapped files to get the final version of data files Acolyte needs.
4. Reading the final data files into memory before any queries, and getting ready for processing queries from clients.

2.1 Parsing the Original Data File

We parse the original data file and generate 5 table files: *FRIENDLIST-TABLE*, *MICROBLOG-TABLE*, *EVENT-TABLE*, *MENTION-TABLE*, and *RETWEET-TABLE*. Each table file represents a table layout defined in the official documentation.

2.2 Remapping the Table Files

All the elements in the table files such as UID, MID, TAG, TIME are strings. If we simply load these strings into memory, it will cost a lot of space and a lot of time to locate or compare.

To reduce the response time as well as save memory, we remap UID, MID and TAG to a 32-bit integer. We collect all the UIDs, MIDs and TAGs appeared in the table-files, sort and count these strings, then map the original strings to 32-bit integers.

Take the *FRIENDLIST-TABLE* in Figure 2 as an example. The *UID-MAP-LIST* and the *FRIENDLIST-MAPPED-TABLE* corresponding to it are shown on the right.

For the same reason, we transform a TIME format string to a 32-bit integer representing the number of seconds from 1970-01-01 00:00:00 UTC (the epoch time).

After remapping all the table files, we generated 5 mapped files: *FRIENDLIST-MAPPED-TABLE*, *MICROBLOG-MAPPED-TABLE*, *EVENT-MAPPED-TABLE*, *MENTION-MAPPED-TABLE*, *RETWEET-MAPPED-TABLE*, and 3 map list files: *UID-MAP-LIST*, *MID-MAP-LIST*, and *TAG-MAP-LIST*.

Each of the mapped table has the same data as the original table, besides all the strings are transformed to integers. And the map list files tell us how to transform a UID, MID, or TAG to an integer, and vice versa.

FRIENDLIST-TABLE	
UID	FRIENDID
HELEN	JOHN
HELEN	TOM
TOM	JERRY

UID-MAP-LIST	
UID	UID-index
HELEN	0
JOHN	1
TOM	2
JERRY	3

FRIENDLIST-TABLE-MAPPED	
UID-index	FRIENDID-index
0	1
0	2
2	3

Fig. 2. Remapping example, transforming UID from strings to integers

2.3 Rescheduling the Table-Mapped Index Files

To reduce the latency of all queries, we have to reschedule the remapped table files and choose proper data structures to store all these files.

The Relationship Data Structure The *FRIENDLIST-MAPPED-TABLE* file describes all the following relations in key-value pairs. This structure doesn't contain the graph features, so it is not very efficient for some types of queries. For example, when we ask all the followees of a given user U, we may need to scan all the rows.

We use adjacency lists to store the relations among users. We have three kinds of adjacency list: followee adjacency list, follower adjacency list, and R-friend adjacency list.

Figure 3 shows the adjacency list of a given example. In the followee adjacency list, each UID points to an array which represents the followees of it.

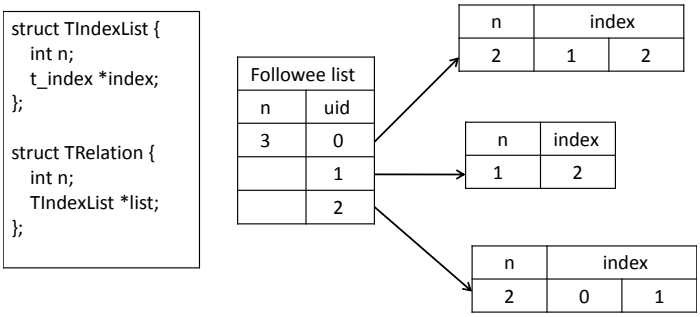

Fig. 3. Data structure for relationship

The Microblog Related Data Structure The Microblog Table, Event Table, Mention Table and Retweet Table all have a primary field named MID. So we create a structure named *TMicroblog_all* combining these four tables in one list. *TMicroblog_all* has 6 attributes: *UID*, *MTIME*, *REMID*, *BE_RET_MID*, *MENTION_ID*, *TAG*. Each of the attributes is either an integer or an array. For the array attribute, we sort the elements of it to reduce the data locate time.

The C++ struct of *TMicroblog_all* is shown in Figure 4.

The Re-schedule List The structure in Section 2.3 can be used to easily get all the information relative to a given microblog, but sometimes we need to get information relative to a given user or a given tag. So we need to generate some

```
struct TMicroblog_all {
            int n;
            int *uid;
            int *mtime;
            int *remid;
            TIndexList  **be_ret_mid;
            TIndexList  **mention_uid;
            TIndexList  **tag,
};
```

Fig. 4. Microblog related data structure

reschedule tables to achieve these goals. Figure 5 shows the extra lists that we generate.

As shown in Figure 5, *microblog_time* stores all the MID indices and sorts these indices by the microblog times. We can use binary search to get all the MIDs in a given timespan very quickly.

The *miroblog_uid*, *mention_uid*, *event_tag* represent microblogs created by a user, mentioned by a user or attached a given tag. The MID indices in *microblog_uid.list[UID_i]*, *mention_uid.list[UID_i]*, *event_tag.list[TAG_i]* are sorted by microblog timestamps.

```
struct TMicroblog_time {
    int n;
    int *mid;
};

struct TList {
    int n;
    TIndexList *list;
};

Tmicroblog_time  microblog_time;
TList  microblog_uid;
TList  mention_uid;
TList  event_tag;
```

Fig. 5. Reschedule list data structure

```
struct  TUID_desc {
      int n;
      int *uid;
};

TUID_Desc   Mention_uid_desc;
TUID_Desc   Retweet_uid_desc;
```

Fig. 6. Two special lists used for Query 7 and Query 10

Figure 6 shows another 2 special lists used for optimizing Query 7 and Query 10.

Mention_uid_desc stores all UIDs which are mentioned by at least one microblog. And the UIDs are in decreasing order of the times being mentioned.

Retweet_uid_desc is similar to *Mention_uid_desc*, it stores all UIDs whose microblogs has retweeted by other microblogs at least once. The UIDs in *Retweet_uid_desc* are in decreasing order of the times being retweeted.

2.4 Initialization

On initialization step, we load all the data file described in Section 2.3 into memory. This needs about 50 GB memory. Then the server listens to incoming requests from clients. We use C++ to write the server side code and use Pthreads[5] to do multi-threading tasks.

3 Query Optimization

Using the data structures described above, we can get the data from memory very quickly. For each query, we can divide it into several steps. On each step, we read data from one list or calculate the data from previous steps. In this section, we emphasize the optimization taken to some queries. The other queries are straight-forward.

3.1 Query 6

Steps to get the answer:

- Use *Microblog_uid.list[UID_i]* and *Microblog_all.mention_uid[UID_i]* to get all UIDs which mentioned by A's microblogs
- Use *Mention_uid.list[UID_i]* to get all MIDs which mentioned these UIDs
- Use *Microblog_all.uid[MID_i]* to get all UIDs belong to the MIDs we get from the last step.

3.2 Query 7

We consider the UIDs in *Mention_uid_desc* one by one. For each *UID_i*, use binary search to count the MIDs which mentioned *UID_i* and in the given timespan. We use a heap to stores these candidate UIDs and the count of MIDs. The top of the heap stores the candidate UID which has the minimal count.

If the number of candidates in the heap is not larger than β (returncount), we can just insert the new UID and count number into the heap.

If the heap is full (number of candidates equals β), we compare the number of MIDs which mentioned *UID_i* (not consider the timespan) to the count number of heap-top candidate, if the total number is not larger than the count number of the heap-top candidate, we need not scan all the remaining UIDs in *Mention_uid_desc* from now on, because the total number of mentioned MIDs in *Mention_uid_desc* is in decreasing order. Each of the remaining UIDs cannot be insert into the heap later. We can break the scan loop to save time. Otherwise,

we compare the MID count number of new UID to the heap-top one and replace the heap top by the new UID if it is better, then re-organize the heap.

We can easily find that the time complexity of these algorithm is $O(n_{\text{UID}} \cdot (\log \beta + \log n_{\text{mention\_MID}})$. While n_{UID} is the number of UIDs which be mentioned at least one microblog, and $n_{\text{mention\_MID}}$ is the maximal number in all of *Mention_uid_desc.list[UID_i].n*, for any *UID_i*. As described above, for β is always much smaller than n_{UID}, so in most cases, we don't need to scan all the UIDs in *Mention_uid_desc.list[UID_i]*. Thus, the actual time complexity always much smaller than theoretical value.

The pseudocode of the algorithm is shown below:

```
initialize candidate_heap;
for (int i=0; i<Mention_uid_desc.n; i++) {
    UID_i = Mention_uid_desc.uid[i];
    Use binary search to count the number of MIDs which mentioned UID_i and
    in given timespan, name the count number c_MID.
    if (candidate_heap.size() < x) {
       candidate_heap.insert(pair<UID_i, c_MID>);
    } else
    if (Mention_uid.list[UID_i].n <= candidate_heap.top().c_MID) {
       break;
    } else
    if (c_MID > candidate_heap.top().c_MID) {
       candidate_heap.remove(candidate_heap.top());
       candidate_heap.insert(pair<UID_i, c_MID>);
    }
}
return candidate_heap;
```

3.3 Query 8

In this query, we first read the followee list to get all of A's two-level followees. For each user *UID_i* in A's two level followees, we can read *Microblog_uid.list[UID_i]* to get all microblogs belong to *UID_i*. Since the microblogs in *Microblog_uid.list[UID_i]* have been sorted by the microblog time, we just need to consider the last β ones.

We use a heap to store the candidate MIDs, the number of the candidates in the heap is not larger than β and the top of the heap stores the candidate MID which has the minimal microblog time in the heap. If the new MID can't insert to the candidate heap, we don't need to consider the remaining MIDs belong to this followee, because they are earlier than this one.

The pseudocode of the algorithm are shown below:

```
Select all of A's two level follwees
initialize candidate_heap;
For (UID in A's two level followees)
    For (in decreasing order, get last x MID_i in microblog_uid.list[UID]) {
```

```
        mtime = microblog_all.mtime[MID_i];
        If (candidate_heap.n < x) {
            candidate_heap.insert(pair<MID_i, mtime>)
            continue;
        }
        If (candidate_heap.top().mtime < microblog_all.mtime[MID_i]) {
            candidate_heap.remove(candidate_heap.top());
            candidate_heap.insert(pair<MID_i, mtime>);
        } else break;
    }
return candidate_heap;
```

4 Evaluation

We ran our experiment on a server with 2 Intel Xeon E5-2680 processors (4 physical cores and 8 logical cores each) and 64 GB main memory.

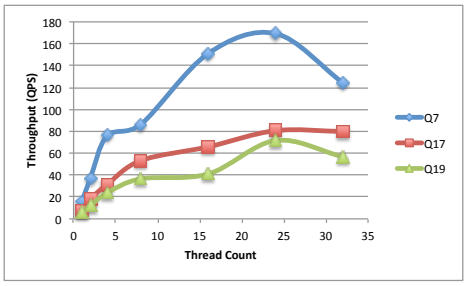

Fig. 7. Throughput under different thread count

We choose $\beta = 50$, $\gamma = \mathrm{w}$, and choose Query 7, Query 17, and Query 19 to observe their throughputs when α increases. As shown in Figure 7, the throughputs get increasing with α. When α reaches 25, the throughputs begin to drop. Since the server has just 16 physical cores, larger α means more costs in scheduling and shared processor units.

To evaluate the performance of each query, we calculate the average latency of all property setting cases for each query and each of the 19 latencies are shown in Figure 8. We can find that most of the average latency is less than 100 ms. All of the queries except Query 6 and Query 13 have an average latency less than 1000 ms.

Besides the average latency and throughputs, we also consider the maximum latency of all queries. Figure 9 shows the maximum latency of all property setting cases for each query. In the worst case, most of the queries can be finished within 5000 ms. Unfortunately, for some individual case, such as Query 6, the worst case needs the client to wait more than one minute. We have tried some simple parallel

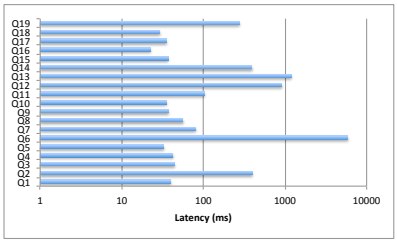

Fig. 8. Average latency of all the queries

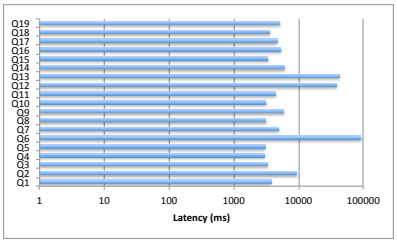

Fig. 9. Maximum latency of all the queries

programming technology such as OpenMP to optimize these bad queries, but the speed-up isn't good enough and the optimizing can also reduce the throughput when more than one client are connecting to the server.

We also compare Acolyte to the solutions based on commodity databases MySQL and MongoDB. Figure 10 and Figure 11 show the latency and throughput on Query 5 when running Acolyte, MongoDB and MySQL. We can see that Acolyte is about 10 times faster than MongoDB, and 40 times faster than MySQL. On other more complicated queries such as Query 7, Acolyte has even more speed-up. Although the solutions based on MySQL and MongoDB are quite preliminary, the results here indicate the performance of Acolyte is rather good.

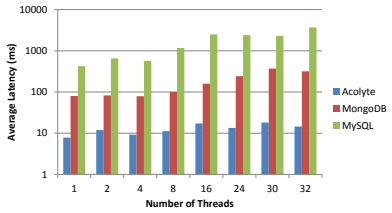

Fig. 10. Average latency of Query 5 using Acolyte, MongoDB and MySQL

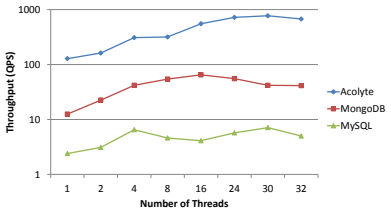

Fig. 11. Throughput of Query 5 using Acolyte, MongoDB and MySQL

5 Discussion

In this report, we present Acolyte, the system for fast social network query. The original data are transformed to get small memory footprint and fast access. All the data reside in memory, so it's much faster than disk-based solutions, which are the common cases for commodity databases. We use C++ programming language to write server side code to get more control over memory management and high performance. Some slow queries are further optimized to improve the overall performance.

There is some opportunity to leverage caching to get even better performance. To do so, the data of following relations and microblogs need to be rearranged by the graph structure. If the total amount of data goes beyond the main memory, flash disks and high-speed local area networks can be used.

Many recent works focus on large-scale graph related processing. Pregel[4] is a system built by Google to process large graphs in a distributed environment. Grace[6] is a graph-aware system which optimizes for fast computation based on graph traverse. WebGraph[3] is a high-ratio compression method to store graph structure and data. We believe that this topic of research is still growing in following years.

References

1. MongoDB. http://www.mongodb.org/.
2. MySQL. http://www.mysql.com/.
3. P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.
4. G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146. ACM, 2010.
5. Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.
6. V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX conference on USENIX annual technical conference*, pages 182–193. USENIX Association, 2012.