Database Analytics in Social Networks

Database Analytics (CSE 190), Project

Jay Ceballos (A09338030) Matthias Springer (A99500782)

March 22, 2014

1 Analytic Queries on Single Node

In this section, we present our implementation of the *Social190* database and certain optimizations.

1.1 Database Schema

```
CREATE TABLE members (
     id              integer PRIMARY KEY,
     name            varchar(255),
     nation          varchar(255),
     birthday        date,
     created         timestamp
);

CREATE TABLE topics (
     id              integer PRIMARY KEY,
     name            varchar(255)
);

CREATE TABLE posts (
     id              integer PRIMARY KEY,
     author          integer references members(id),
     title           varchar(255),
     text            varchar(255),
     topic           integer references topics(id),
     created         timestamp
);

-- friend relationship is symmetric
CREATE TABLE friends (
     x               integer references members(id),
     y               integer references members(id),
     created         timestamp
);

CREATE TABLE viewed (
     reader          integer references members(id),
     post            integer references posts(id)
);
```

- `members` stores data about registered members of Social190.
- `topics` are used for posts. Every post has to be associated to a topic.
- Members can create posts about certain topics. These posts are stored in `posts`.
- Members can be friends with other members. The friend relationship is symmetric, i.e. if x is a friend of y, then y is also a friend of x. We store the friend relationship in `friends`. Every time we add a friendship, two tuples are added to the table.
- Posts can be viewed by friends of the poster. This is captured in `posts`. The table may contain a tuple (r, p) only if p's author is a friend of r.

1.2 Unoptimized Queries

In this section we describe two kinds of analytic queries that are executed on the database.

Query 1: view ratio of friend's posts by friend Given a member r, for every friend a, report a tuple (a, r, v) where v is the ration of the number of posts by a that are read by r, divided by the total number of posts by a. We execute this query by first calculating the number of posts of a (`denominator`), and the number of read posts of a by r (`numerator`), and dividing both numbers.

```
SELECT (1.0 * numerator.cnt / denominator.cnt) AS v, numerator.author as a, numerator.
   reader as r
FROM (SELECT posts.author AS author, viewed.reader AS reader, COUNT(*) AS cnt
          FROM posts, viewed
          WHERE posts.id = viewed.post AND viewed.reader = 504
          GROUP BY posts.author, viewed.reader) AS numerator,
     (SELECT posts.author AS author, COUNT(*) AS cnt
          FROM posts
          GROUP BY posts.author) AS denominator
WHERE numerator.author = denominator.author;
```

Query 2: view ration of friends' posts from nation by friend Given a member r, for every nation n, report a tuple (n, r, v) where v is the ratio of the number of posts read by r that have been posted by friends from nation n, divided by the total number of messages posted from friends from nation n in the newsfeed of r.

This query is similar to Query 1, but takes into account a bigger number of friends instead of just one single friend. We execute the query by first calculating the number of posts of all friends grouped by nation (`denominator`), and the number of posts of these friends that were read by r, and dividing both numbers.

```
SELECT (1.0 * numerator.cnt / denominator.cnt) AS v, numerator.nation as n, numerator.
   reader as r
FROM (SELECT members.nation AS nation, viewed.reader AS reader, COUNT(*) AS cnt
          FROM posts, viewed, members
          WHERE posts.id = viewed.post AND viewed.reader = 504 AND members.id = posts.
             author
          GROUP BY members.nation, viewed.reader) AS numerator,
     (SELECT members.nation AS nation, COUNT(*) AS cnt
          FROM posts, members, friends
          WHERE posts.author = members.id AND friends.x = 504 AND friends.y = members.id
          GROUP BY members.nation) AS denominator
WHERE numerator.nation = denominator.nation;
```

1.3 Table Index Optimization

In this section, we present indices that improved the performance of these two queries. In general, hash indices turned out to be faster than B^+ tree indices in most cases, because hash indices perform better for random accesses, whereas B^+ are better for range queries.

Indices for both queries

- `CREATE INDEX ON viewed using hash (reader);`
  In both queries, we select tuples $\sigma_{reader=504}$ `viewed`, therefore we create a hash index for that attribute. After selecting one or multiple tuples of that table, *viewed.posts* is fixed.

- `CREATE INDEX ON posts using hash (id);`
  Both queries join `posts` $\bowtie_{posts.id=viewed.post}$ `viewed`, where *viewed.posts* is fixed.

- `CREATE INDEX ON posts using btree (author);`
  In Query 1, we group by authors in the two inner queries, therefore an index on this attribute might be useful. In Query 2, in the second inner query, we select on *posts.author*, after joining `friends` $\bowtie$ `members` $\bowtie_{members.id=posts.author}$ `posts`, where *members.id* is fixed[1].

Query 2-specific indices

- `CREATE INDEX ON members using hash (nation);`
  In both inner queries, we group by *nation*, therefore an index on that attribute might be useful.

- `CREATE INDEX ON members using hash (id);`
  In the first inner query, we join `viewed` $\bowtie$ `posts` $\bowtie_{posts.author=members.id}$ `members`, where *posts.author* is fixed[2]. The same applies for another join in the second inner query.

- `CREATE INDEX ON friends using hash (x);`
  In the second inner query, we select all friends with a certain attribute x.

1.4 PostgreSQL Materialized Views

PostgreSQL supports materialized views that cache the result of a query. However, they need to be refreshed manually. In order to have the most recent data all the time, we triggered a refresh of the materialized views after every insertion in one of the tables. However, this causes the whole view to be regenereated completely, even though only few tuples might have changed. Queries, on the other hand, performed significantly better. We did not investigate the use of materialized views any further in this project.

1.5 Incremental Views

In this section, we present our implementation of materialized views that are updated incrementally, i.e. not the whole view is being regenerated but only the tuples that actually change. Specifically, we assume that the materialized view is in a consistent state before an update/insert is issued. We make sure, that after the operation, the materialized view is again in a consistent state.

[1] *friend.x* is given, fixing *friend.y*, therefore also fixing *members.id* after the first join.
[2] We fix *posts.author* after joining `viewed` $\bowtie$ `posts`, where the join attribute of `viewed` is fixed.

1.5.1 Database Schema

```
CREATE TABLE ivm_arv (
     author          integer,
     reader          integer,
     numerator       integer,
     denominator     integer
);

CREATE TABLE ivm_nrv (
     nation          varchar(255),
     reader          integer,
     numerator       integer,
     denominator     integer
);
```

We use `ivm_arv` for Query 1 and `ivm_nrv` for Query 2. In both cases, we store the numerator and the denominator instead of the divided value (average), in order to allow for incremental view updates, since average is not an associative function but sum is.

1.5.2 Queries

```
SELECT author AS a, numerator * 1.0 / denominator AS v, reader AS r FROM ivm_arv WHERE
   reader = 504;
```

```
SELECT nation AS n, numerator * 1.0 / denominator AS v, reader AS r FROM ivm_nrv WHERE
   reader = 504;
```

For both queries, we simply select the single tuple that already contains the precomputed result from the IVM table and calculate the division.

1.5.3 Incremental View Maintenance (Triggers): Query 1

For the incremental view maintenance, we basically have to consider four kinds of insertions: insertions into `views`, `members`, `posts`, and `friends`. However, we do not need a trigger for inserting tuples into `friends` or `members` for Query 1.

Insert into `viewed` We first handle the case where we insert tuples into `viewed`.

```
CREATE OR REPLACE FUNCTION f_insert_view() RETURNS trigger AS
$f_insert_view$
     DECLARE
          num_posts                INTEGER;
          author_index             INTEGER;
     BEGIN
          author_index := (SELECT posts.author FROM posts WHERE posts.id = NEW.post);

          UPDATE ivm_arv
               SET numerator = numerator + 1
               WHERE reader = NEW.reader AND author = author_index;
          IF NOT FOUND
```

```
                num_posts := (SELECT COUNT(*) FROM posts WHERE author = author_index);
                INSERT INTO ivm_arv
                     VALUES (author_index, NEW.reader, 1, num_posts);
           END IF;

           RETURN NEW;
     END;
$f_insert_view$
LANGUAGE plpgsql;

CREATE TRIGGER insert_view AFTER INSERT ON viewed FOR EACH ROW EXECUTE PROCEDURE
   f_insert_view();
```

The following list describes the steps for the trigger.

- A tuple (r, p) is inserted into `viewed`.
- Increment the numerator of the tuple in the IVM table for that author and that reader. If no such tuple exists, insert a new one with the number of posts of that author.

Insert into `posts` In case we insert a tuple into `posts`, we execute the following trigger. We simply increase the denominator for every tuple of the author that created the post, because every reader has now one more post to read.

```
CREATE OR REPLACE FUNCTION f_insert_posts() RETURNS trigger AS
$f_insert_posts$
     BEGIN
           UPDATE ivm_arv
                SET denominator = denominator + 1
                WHERE author = NEW.author;

           RETURN NEW;
     END;
$f_insert_posts$
LANGUAGE plpgsql;

CREATE TRIGGER insert_posts AFTER INSERT ON posts FOR EACH ROW EXECUTE PROCEDURE
   f_insert_posts();
```

1.5.4 Incremental View Maintenance (Triggers): Query 2

For Query 2, we need triggers for inserting tuples into `viewed`, `friends`, and `posts`.

Insert into `viewed` In case we insert a tuple (r, p) into `viewed`, we first check whether there is a tuple for the reader r and the nation of p's author in the IVM table. If that is the case, we update the numerator for that tuple by 1. Otherwise, we have to insert a tuple and calculate the number of posts by all friends of r who have the nation of p's author first.

```
CREATE OR REPLACE FUNCTION f_insert_view() RETURNS trigger AS
$f_insert_view$
     DECLARE
          num_posts_nation         INTEGER;
          view_nation              VARCHAR(255);
     BEGIN

          UPDATE ivm_nrv
               SET numerator = numerator + 1
               FROM posts, members
               WHERE reader = NEW.reader AND ivm_nrv.nation = members.nation AND members
                  .id = posts.author AND NEW.post = posts.id;

          IF NOT FOUND THEN
               view_nation := (SELECT nation FROM members, posts WHERE posts.author =
                  members.id AND posts.id = NEW.post);
               num_posts_nation := (SELECT COUNT(*) FROM members, posts, friends WHERE
                  members.id = friends.x AND friends.y = NEW.reader AND posts.author =
                  members.id AND members.nation = view_nation);

               INSERT INTO ivm_nrv
                    VALUES (view_nation, NEW.reader, 1, num_posts_nation);
          END IF;

          RETURN NEW;
     END;
$f_insert_view$
LANGUAGE plpgsql;

CREATE TRIGGER insert_view AFTER INSERT ON viewed FOR EACH ROW EXECUTE PROCEDURE
   f_insert_view();
```

Insert into `friends` In contrast to Query 1, we have to update the IVM table if we insert tuples into the `friends` table. In that case, a member's read ratio for a given nation changes when we add a friend who has already some posts: in that case, the number of posts that he could have read increases.

We update the IVM table by first getting the new friend's nation and his posts and then updating the IVM table for that nation by increasing the denumerator. Note, that this happens only if there is already such a tuple in the IVM table, i.e. we do not create null tuples with this approach.

```
CREATE OR REPLACE FUNCTION f_insert_friends() RETURNS trigger AS
$f_insert_friends$
     DECLARE
          num_posts           INTEGER;
          friend_nation       VARCHAR(255);
     BEGIN
          num_posts := (SELECT COUNT(*) FROM posts WHERE author = NEW.x);
          friend_nation := (SELECT nation FROM members WHERE id = NEW.x);

          UPDATE ivm_nrv
               SET denominator = denominator + num_posts
               WHERE nation = friend_nation AND reader = NEW.y;

          RETURN NEW;
     END;
$f_insert_friends$
```

```
LANGUAGE plpgsql;

CREATE TRIGGER insert_friends AFTER INSERT ON friends FOR EACH ROW EXECUTE PROCEDURE
   f_insert_friends();
```

Insert into `posts` Similarly to Query 1, we might have to increment the denominator of some IVM tuples. We iterate over all friends of the post's author and increase their tuples' denominator for the author's nation by one, if it exists.

```
CREATE OR REPLACE FUNCTION f_insert_posts() RETURNS trigger
AS $f_insert_posts$
    DECLARE
        author_nation            VARCHAR(255);
        friend_id                INTEGER;
    BEGIN
        author_nation = (SELECT nation FROM members WHERE NEW.author = members.id);

        FOR friend_id IN (SELECT members.id FROM members, friends WHERE members.id =
           friends.x AND friends.y = NEW.author)
        LOOP
            UPDATE ivm_nrv
                SET denominator = denominator + 1
                WHERE nation = author_nation AND reader = friend_id;
        END LOOP;

        RETURN NEW;
    END;
$f_insert_posts$
LANGUAGE plpgsql;

CREATE TRIGGER insert_posts AFTER INSERT ON posts FOR EACH ROW EXECUTE PROCEDURE
   f_insert_posts();
```

Remarks All the triggers that we presented in this section do only generate tuples in the IVM table if the reader has actually read a post by that author/nation. This is in accordance with the suggestion in the problem description not to generate NULL tuples in the IVM table.

In a later section, we present another way of maintaining IVM tables that includes generating NULL tuples and we argue that there are workloads where this approach might turn out to be more efficient in terms of runtime performance.

1.5.5 Incremental Views with Indices

In this section, we present indices that improved the performance of our implementation.

Indices for both queries

- `CREATE INDEX ON posts using hash (id);`
  In both queries, we can utilize this index when selecting the post for a given ID ($\sigma_{id=NEW.post}$ `posts`) after inserting a tuple into `viewed`.

- `CREATE INDEX ON posts using btree (author);`
  In Query 1, we can utilize this index when retrieving the number of posts of a specific author after inserting a tuple into `viewed`. This involves the selection $\sigma_{author=...}$ `posts`. The trigger for inserting into `viewed` in Query 2 contains a join `members` $\bowtie_{members.id=posts.author}$ `posts`, where $members.id$ is fixed.

Query 2-specific indices

- `CREATE INDEX ON friends using hash (y);`
  When inserting a tuple into `viewed`, we can utilize this index in the trigger when we get the number of posts by friends from a specific nation, since the subquery contains a selection $\sigma_{y=...}$ `friends`.

- `CREATE INDEX ON members using hash (id);`
  When inserting a tuple into `viewed`, we can utilize this index in the trigger when we get the nation for a post, because the subquery contains a join `members` $\bowtie_{members.id=posts.author}$ `posts`, where $posts.author$ is fixed.

- `CREATE INDEX ON members using btree (nation);`
  When inserting a tuple into `viewed`, we can utilize this index in the trigger when we count the number of posts by a specific nation, because the subquery contains the selection $\sigma_{nation=...}$`members`. Although, the members are fixed at this point, this might allow for a more efficent query processing based on generating the intersection of the tuples found so far with the result of index querying.

Trivial IVM indices The indices shown so far increase the speed of IVM updates. In addition, we added indices on the attributes that are used for querying the IVM tables (and to some degree also during IVM updates). This mainly increased the performance of the queries.

- `CREATE INDEX ON ivm_arv using hash (reader);`
- `CREATE INDEX ON ivm_arv using hash (author);`
- `CREATE INDEX ON ivm_nrv using hash (reader);`
- `CREATE INDEX ON ivm_nrv using hash (nation);`

1.6 Performance Evaluation

The following table shows the performance of our implementation without and with indices, and with and without IVM. Every number represents the runtime of the fastest out of 3 runs with PostgreSQL 9.3 on an Intel i7 2.8 GHz Notebook with 16 GB RAM.

Type	**INS** `members`	**INS** `friends`	**INS** `posts`	**INS** `views`	**Query 1**	**Query 2**	**INS Mix**	**Query 1 Mix**	**Query 2 Mix**
unoptimized	3.008	6.121	2.700	2.949	33.817	31.390	9.285	30.962	24.562
indices	3.180	6.652	3.099	3.297	31.599	10.985	10.426	26.176	11.931
IVM	3.550	9.144	18.787	45.118	9.560	9.439	44.644	6.916	6.246
IVM + indices	3.546	9.290	16.291	8.091	3.265	3.254	27.042	3.357	3.333

We can see that the performance for the IVM maintenance after inserting tuples into `posts` is bad, even after applying indices. We try to address this issue in the next section.

We ran two types of benchmarks. The columns **INS** ... are insertions of a lot of tuples of that type (sequentially). We started with an empty database, inserted 10000 members, then 10000 friends, then 15000 topics, then 7500 posts, then 10000 views. The numbers are the runtime of the execution of the SQL script in seconds. We ran all scripts 3 times and took the minimum value.

The column **INS Mix** denotes insertions of 25000 tuples into different tables, independently at random. We inserted members with a probablity of 0.1, friends with 0.2, topics with 0.05, posts with 0.25, and views with 0.4. Afterwards, we executed 10000 different instances of Query 1 and Query 2 sequentially (**Query 1/2 Mix**).

The row *unoptimized* is an unoptimized version without IVM or triggers. The row *indices* is a version without IVM but with triggers. The row *IVM (+indices)* is an IVM version (with indices). It is surprising that Query 1 with indices is slower than Query 2 with indices. We double checked our results and made sure that we used as many triggers for Query 1 as possible: the query execution plan for Query 1 uses index scans instead of sequential scans. We can also rule out a measuring error, since the runtime is big enough and measuring jitter is usually in the area of less than half a second.

1.7 Deferred Updates of Incremental Views

In this section, we present ideas for increasing the performance of IVM maintenance after inserting tuples into one of the four tables. The idea is to defer updates of materialized views by storing tuples in separate tables (*temp* tables) and accounting for them when running the queries in a special way. Only when the temp tables reach a certain size, the tuples are merged into the IVM tables.

Query 1 Consider a single insertion into `posts`. Since there may be multiple readers in `ivm_arv` for the post's author, we might have to update a lot of tuples, even if we insert only one tuple into `posts`[3]. We can optimize the trigger as follows.

- Create a table `posts_temp`(*author*, *new_posts*).
- When we insert a tuple into `posts`, increment *new_posts* in `posts_temp` or create a new tuple with value 1 for that author if none exists (trigger).
- When `posts_temp` reaches n tuples, write back the changes to `ivm_arv`, using code from `f_insert_posts`. However, increase the denominator by *num_posts* instead of 1. Clear the table `posts_temp`.
- Query 1 needs to account for tuples in `posts_temp` by adding the number of posts of an author to the denominator[4].

We think that creating further temp tables for the other triggers is not useful for the following reasons.

- There is no trigger for `members` or `friends` at all.
- Inserts into `viewed` are already quite fast due to indices. Since we cannot perform better than the unoptimized version, there is not much space to improve.
- Updates into `viewed` trigger an increment of only one tuple in the IVM table. Adding a tuple to a temp table is not much faster than updating the numerator value of a single IVM tuple, considering that we have indices on the attributes of the IVM table.

[3]This corresponds to the idea that we have to increase the denominator of tuples for all friends who have one more post they can read now.

[4]The SQL code could be similar to this: `SELECT author AS a, numerator / (denominator + (SELECT counter FROM posts_temp WHERE author = a))AS v, reader AS r FROM ivm_arv WHERE reader = 8937;`

Query 2 For this query, the same optimization on `posts_temp` can be applied. Similarly to Query 1, it does not pay off to add a temp table for `viewed`, because only one value is being incremented when inserting a tuple. Query 2 has an additional trigger for updating `friends`. However, an insertion into `friends` triggers only one update in the IVM table. Therefore, it does not pay off to generate a temps table.

Note, that if we consider the IVM of Query 1 and Query 2 at the same time, an insertion into `viewed` triggers an update of a value in both IVM tables. From that point of view, a single insertion might be faster than two updates.

1.8 Generating NULL tuples

The solution presented in the previous sections does not generate NULL tuples, i.e. we do not add a tuple for readers that have read 0% of the posts. Considering the typical workload of a social network it might, however, pay off to generate NULL tuples.

- We claim that in a social network, there are usually many more insertions into `viewed` than into any of the other tables.
- Note, that the triggers for inserting into `viewed` are quite complex. In particular, they contain a check whether a certain tuple exists or not. If we generate NULL tuples when inserting tuples into the other tables, we can get rid of these checks, making insertions into `viewed` faster, but the other insertions slower. For a typical workload, a large number of operations (inserts into `viewed`) becomes a little bit faster, whereas a small number of operations becomes slower.
- For example, when inserting a friend, we have to insert a new tuple for (x, y) into the IVM table for Query 1. In the query code we have to check for NULL values, though, otherwise we might get a division-by-zero exception, in case one the of the friends has no posts yet.
- This approach might increase the performance by a very small factor. However, it wastes a lot of space, because we insert a lot of tuples that we do not actually need, i.e. NULL tuples.

1.9 Further Optimizations

The triggers for both Query 1 and Query 2 involve operations that count the number of posts by an author or a nation. For Query 1, we tried optimizing this operation by storing the number of posts per author in another table and reusing that value instead of counting the posts every time we insert a new tuple into the IVM table. However, it turned out that this optimization did not increase the performance of our implementation because we had already an index on *posts.author* which is good enough for counting, since we do not actually have to take a look at the tuples themselves.

For Query 2, it might pay off to cache the number of posts per nation and member, since this calculation involves two joins. However, consider, that this slows down insertions into `posts` and requires additional space that is proportional to the number of members and the number of nations.

2 Parallel Programming

In this section, we analyze slightly modified versions of Query 1 and Query 2. We do not want to output the result for a single member r, but for every member r.

2.1 Data Partitioning

In this section, we present strategies to partition the tables presented in the last section onto n nodes.

2.1.1 Query 1: Partition by Author

We propose a partitioning of `members` in a round-robin way, and a derived partitioning of the other tables, resulting in no communication overhead at all.

- Distribute `members` round-robin.
- Store a post on the node of the associated member, i.e. on the node where $post.author = members.id$.
- Store views on the node of the associated post's author, i.e. on the node where $view.post.author = members.id$.
- We do not care about `friends` or `topics` for Query 1 at all, since these tables are not needed for answering this query.
- Query 1 can be answered locally by executing the exact same query that we presented in the previous section for every author a and every reader r on a's node. Note, that we store all views and posts that are associated with an author on the author's node. Therefore, no communication is necessary.
- This approach does not imply any replication.

2.1.2 Query 2: Partition by Nation

We propose a partitioning of `members` according to the nation, as described in the following.

- Store all members of a given nation on a designated node, i.e. partition by nation. A single node can hold members for multiple nations.
- Store a post on the node of the associated member, i.e. on the node where $post.author = members.id$.
- Store views on the node of the associated post's author, i.e. on the node where $view.post.author = members.id$.
- Store the part of the `friends` table on a node, where $friends.x = members.id$.
- We do not care about the `topics` table for Query 2.
- Query 2 can be answered locally by executing the exact same query that we presented in the previous section, for all nations *nat* and every reader r on *nat*'s node. Note, that we store all views and posts that are associated with a nation on the nation's node. We also store friends of members of that nation on that node (only the `friends` table). Therefore, no communication is necessary.
- At first glace, it seems as if the `friends` table is being replicated. However, note that `friends` is symmetric and even in the previous section we had two tuples for every friendship. Since we only store tuples based on attribute y, no data is actually replicated.
- This approach is susceptible to skew. We can try to avoid skew by gathering histrograms/statistics about the distribution of the members by nation before doing the partitioning. We should try to keep the number of members per node constant. For example, a good partitioning might put members from China on a single node and put members from Europe on another node. However, there might still be many more members on the node for China than on any other node.
- This approach does not scale. Scaling is limited by the number of nations, because all members from one nation have to be stored on one single node.

2.1.3 Query 2: Partition by Author

We propose an alternative scheme of partitioning `members` in a round-robin way.

- Distribute `members` round-robin.
- Store a post on the node of the associated member, i.e. on the node where $post.author = members.id$.
- Store views on the node of the associated post's author, i.e. on the node where $view.post.author = members.id$.
- Store the part of the `friends` table on a node, where $friends.x = members.id$.
- We do not care about the `topics` table for Query 2.
- For executing Query 2, we need to send some data around. We evaluate two different execution schemes here.

  1. *Send to designated node, grouped by nation*
     On every node, for every distinct reader, we count the number of views of posts of friends (note that we have that portion of the friends table on that node) grouped by nation. For every author, we also count the number of posts, grouped by nation. This results in a list of values: multiple values for a lot of readers and nations, and a some values for all authors. We send the list to a designated node for that nation, e.g. by hashing *nation*. On the target node, we sum the number of posts per member and the number of reads of posts of friends from a given nation per member. Note, that at this point, we have all data concerning a specific nation on a single node, and we can apply the techinque shown in the previous section. Note, that scaling is limited by the number of nations (reducers).
  2. *Send to designated node, grouped by reader*
     Apply the same technique as shown before, but, instead of having a designated node per *nation*, have a designated node per reader. This turns out to be a bad idea, because we have to send the number of posts of every member to every node, that will have a friend of the author. Note, that in the first approach, we sent that number only to the node for that nation. However, this approach might scale better since less data is sent to a single node. We do not evaluate this approach any further.

- Note, that the second idea is less susceptible to skew and scales better. However, it requires some communication.

2.2 Map-Reduce Implementation

In this section, we present Map-Reduce implementations that are based on the data partitioning shown in the previous section.

2.2.1 Query 1

```
map(key, value):
     for each member.id in members:
     num_posts = SELECT COUNT(*) FROM posts WHERE author = member.id

        for each (SELECT distinct(viewed.reader) FROM viewed, posts WHERE viewed.post =
           posts.id AND posts.author = member.id):
          EmitIntermediate((member.id, viewed.reader), (1, num_posts))
```

```
reduce((author, reader), Iterator values):
     EmitTuple(author, reader, values.size() / values[0].second)
```

The Map-Reduce code for this query is trivial. In fact, we do not need Map-Reduce to execute this query efficiently if we assume that the data is partitioned as described in the previous section.

In the mapper, we do not need key and value. We just iterate over all authors and get their number of posts. Then we iterate over all readers of posts of that author and emit a tuple. We have a reducer for every author-reader pair that just generates the sum, divides the numerator and the denominator, and outputs the tuple. In fact, we could already calculate the sum in by using the `count` keyword in the mapper or inside a combiner and let the reducer just output the result, to be more efficient.

2.2.2 Query 2

We assume that the data is partitioned according to the author's nation, as discussed in Section 2.1.2.

```
map(key, value):
     for each reader in (SELECT distinct(viewed.reader) FROM viewed):
         for each x in (SELECT x FROM friends WHERE y = reader):
              num_posts = (SELECT COUNT(*) FROM posts WHERE author = x)
              num_read = (SELECT COUNT(*) FROM viewed, posts WHERE viewed.post = posts.
                 id AND posts.author = x AND viewed.reader = reader)
              EmitIntermediate((x.nation, reader), (num_read, num_posts))

reduce((nation, reader), Iterator values):
     num_read = 0
     num_posts = 0

     for each v in values:
          num_read += v.first
          num_posts += v.second

     EmitTuple(nation, reader, num_read/num_posts)
```

The Map-Reduce code for this query is similar to the code from Query 1. In the mapper, we first get all distinct readers, i.e. readers that read posts from members that are stored on this node. Then we get the reader's friends and get, for every friend, his number of posts and the number of views of his posts. Then we send these two number to a reducer, where the key is a combination of the author's nation and the reader's id.

In the reducer, we get tuples for every nation and every reader. We just have to sum the reads (numerator) and the post (denominator) and output the divided value. Note, that this approach works even if the data is partitioned according to authors, as described in Section 2.1.3. The only crucial thing is that the part of the `friends` table is present, such that we can get all friends per author.