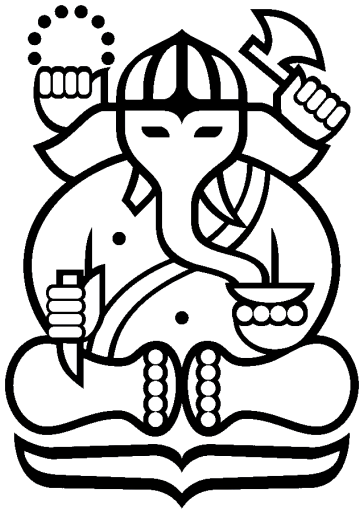
Laporan Tugas Besar IF4031

Analisis dan Eksperimen Arsitektur Aplikasi Social Network



Disusun oleh

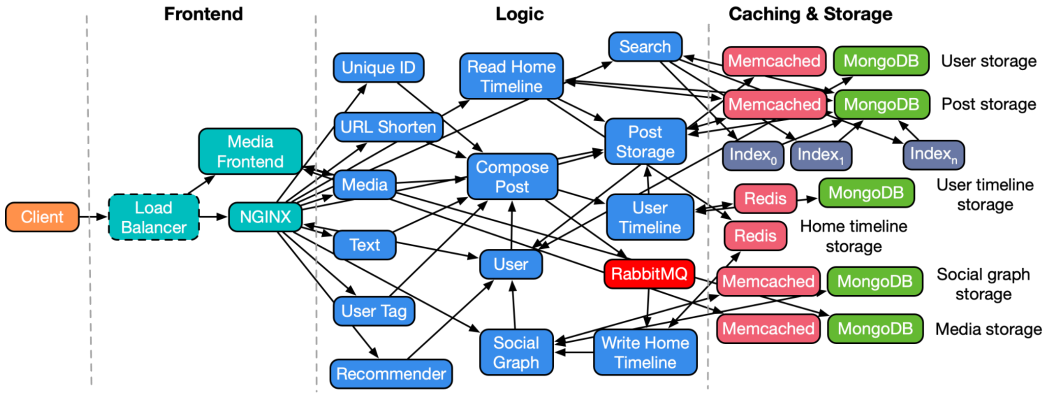
Kelompok 19

13521083 Moch. Sofyan Firdaus
13521091 Fakih Anugerah Pratama
13521170 Haziq Abiyyu Mahdy

Program Studi Teknik Informatika

Institut Teknologi Bandung

1. Arsitektur Aplikasi Social Network



Aplikasi Social Network menggunakan arsitektur *microservice* yang dapat dibagi menjadi tiga *layer*.

1. *Frontend*

   Bagian ini terdiri atas Media Frontend yang menggunakan NGINX yang menyediakan fitur untuk mengakses dan meng-*upload file* media seperti foto dan video. *File* dapat diakses oleh pengguna melalui URL `/get-media/?filename={file_name}`. *Upload file* dapat dilakukan melalui URL `/upload-media`. File media disimpan di MongoDB (Media Storage)

   Terdapat pula Web Server NGINX yang berfungsi untuk menyediakan file statik (HTML dan JS) untuk diakses pada UI. Selain itu, Web Server ini juga berfungsi sebagai *API Gateway* untuk meneruskan *request* ke *service* yang bersangkutan pada *logic layer*. Apabila *request* dilakukan terhadap URL dengan format `/file_name` seperti `/main.html`, maka Web Server akan mengembalikan file yang diakses pada URL. Namun apabila *request* dilakukan terhadap URL dengan prefix `/api`, maka Web Server akan menjalankan *script* Lua yang sesuai untuk melakukan invokasi RPC terhadap *service* yang berada pada *logic layer*.

2. *Logic*

   Bagian ini terdiri atas 14 *service* yang menggunakan Apache Thrift sebagai *framework* RPC. Kode implementasi dari seluruh *service* pada *repository* ini menggunakan C++. Akan tetapi, Apache Thrift menyediakan fitur untuk men-*generate* kode *stub* dengan berbagai bahasa pemrograman, seperti Python, Java, Lua, dsb. Dengan demikian, kode implementasi dari masing-masing

service dapat menggunakan bahasa pemrograman yang berbeda-beda, sesuai kebutuhan dari *service* tersebut. Kegunaan dari masing-masing *service* akan dijelaskan lebih lanjut di bagian *use-case* di bawah.

3. *Caching & Storage*
   Bagian ini menyediakan *persistent storage* dengan MongoDB serta *caching* dengan Redis dan Memcached. Masing-masing *service* pada *logic layer* dapat terhubung *cache* dan *persistent storage*.

Berikut merupakan beberapa *use-case* yang disediakan pada aplikasi Social Network.

1. Create text post (optional media: image, video, shortened URL, user tag)
   Membuat sebuah *post* berisi teks yang dapat secara opsional menerima lampiran berupa *image, video, shortened URL,* dan *user tag*.
   Saat pengguna membuat text post, Social Network App akan memanggil service Compose Post untuk memproses masukan text dan lampirannya. Selanjutnya, service Post Storage akan dipanggil untuk menyimpan masukan ke dalam *storage Post* bersamaan dengan dipanggilnya service User Timeline untuk menerima masukan Post baru User untuk juga disimpan dalam *storage User Timeline.*
2. Read home timeline
   Menampilkan post *followee* yang direkomendasikan oleh service dalam sebuah linimasa yang dapat diakses melalui halaman utama */home*
   Saat App memanggil logic Read Home Timeline, service akan membaca *storage home timeline* untuk menentukan *post* apa saja yang sekarang seharusnya ada di *home timeline* pengguna dan memanggil layanan Post Storage untuk melihat konten *post* yang akan ditampilkan
3. Read entire user timeline
   Menampilkan seluruh *post* yang pernah dibuat oleh pengguna dalam sebuah linimasa yang dapat diakses melalui halaman akun personal
   Saat App memanggil logic User Timeline, service akan membaca *post* apa saja yang harusnya tampil dalam *user timeline* lalu memanggil service Post Storage untuk melihat konten yang dimiliki oleh *post* tersebut.
4. Receive recommendations on which users to follow
   Service merekomendasikan *user* tertentu yang dapat diikuti oleh pengguna

Service ini masih dalam tahap *planned*, sehingga kenyataannya service ini masih diimplementasi secara *hardcoded* (*pre-defined followee candidate*).

5. Search database for user or post

   Mencari *user* atau *post* tertentu berdasarkan kata kunci masukan.

   Saat logic Search dipanggil, service akan mencari Post dalam *storage* menggunakan index yang telah di-cache sebelumnya.

6. Register/Login using user credentials

   Mendaftar sebagai pengguna baru dengan memasukkan username, nama, email, dan password lalu *login* menggunakan *credential* akun yang sebelumnya pernah dibuat.

   Saat melakukan register/login, service User akan dipanggil untuk melakukan tugas yang ditentukan lalu melakukan pencarian atau perubahan terhadap *storage User*.

7. Follow/Unfollow user

   Pilihan untuk mengikuti pengguna tertentu disertai pilihan untuk batal mengikuti

   Saat pengguna melakukan *follow/unfollow* terhadap salah satu user, App akan melakukan update dengan memanggil service Social Graph yang selanjutnya akan mengubah informasi yang disimpan dalam *social graph storage*.

2. Lingkungan Eksperimen

Berikut merupakan spesifikasi perangkat yang digunakan untuk menjalankan eksperimen ini.

- Processor : 11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz
- Installed RAM : 32.0 GB (31.7 GB usable)
- System type : 64-bit operating system, x64-based processor
- OS : Windows 11

Untuk menjalankan aplikasi, digunakan WSL dengan distribusi Ubuntu, serta Docker Desktop.

3. Perencanaan Eksperimen

Berikut adalah langkah-langkah yang dilakukan untuk melakukan eksperimen pada aplikasi Social Network

- Instalasi dan menjalankan aplikasi

1. Install seluruh *pre-requisite* di bawah di perangkat yang akan menjalankan eksperimen
    a. Docker
    b. Docker-compose
       (bisa menggunakan docker compose sebagai alternatif)
    c. Python 3.5+ (with asyncio and aiohttp)
    d. libssl-dev (apt-get install libssl-dev)
    e. libz-dev (apt-get install libz-dev)
    f. luarocks (apt-get install luarocks)
    g. luasocket (luarocks install luasocket)
2. Jalankan *service* menggunakan *docker compose*
3. Lakukan *populate* pada *social graph* menggunakan perintah python3 scripts/init_social_graph.py --graph=<socfb-Reed98, ego-twitter, soc-twitter-follows-mun> (pilih antara socfb-Reed98, ego-twitter, soc-twitter-follows-mun)

- Menjalankan beberapa *workload*

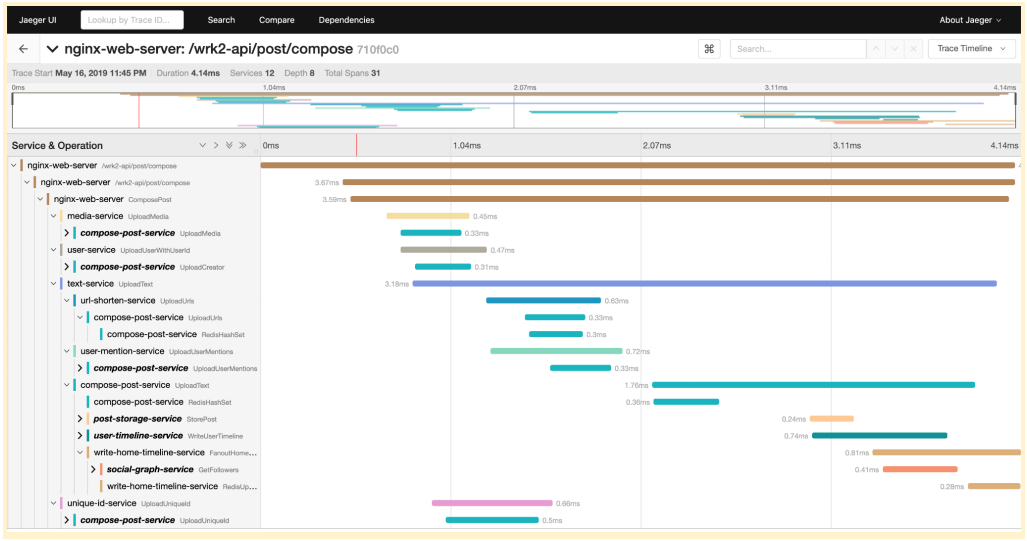
  Untuk memudahkan eksperimen, kita akan men-*generate traffic* untuk menghasilkan laporan *packet trace jaeger*. *Traffic* dapat dihasilkan menggunakan *HTTP workload generator wrk2.* Untuk menggunakan *wrk2* lakukan langkah berikut

  1. Pastikan direktori *wrk2/wrk2/* ada. Jika tidak, *resolve submodule* dengan menggunakan perintah *git submodule update --init --recursive*
  2. Lakukan *make* untuk menghasilkan *binary wrk2*
  3. Jalankan perintah wrk2 untuk men-*generate payload*, contohnya

     ```
     ../wrk2/wrk -D exp -t 12 -c 400 -d 300 -L -s
     ./wrk2/scripts/social-network/compose-post.lua
     http://10.109.126.103:8080/wrk2-api/post/compose -R 10
     ```

- Melihat *trace*

  Lihat *trace* hasil *workload* yang di-*generate* dengan mengakses *endpoint jaeger* di alamat *http://localhost:16686*



- Melihat struktur *database*

  Untuk membantu proses eksperimen, kita dapat meneliti struktur dari setiap database yang digunakan. Menggunakan MongoDB compass yang dikoneksikan dengan endpoint database berikut, kita bisa melihat bagaimana struktur dari setiap dokumen yang ada di masing-masing database.

  - localhost:27018 – Social Graph Database

    ```
    _id: ObjectId('6775489d5bbda356493eb1e2')
    user_id : 0
    followers : Array (73)
    followees : Array (73)
    ```

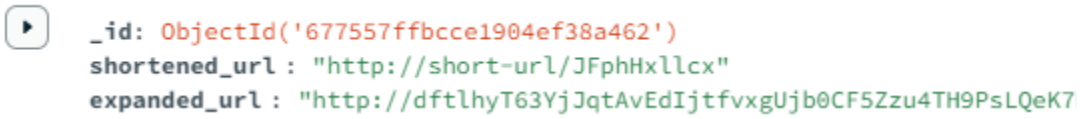
  - localhost:27019 – Post Database

    ```
    _id: ObjectId('677557ff7574821605051832')
    post_id : 1180848231076417537
    timestamp : 1735743487825
    text : "EKMTtpCAQsHiXtvHDpj0KDSimOehy4r0D1J9U7BLMLQ3XcI4xmCI6kQKtv6
    req_id : 77290788143150864
    post_type : 0
    creator : Object
    urls : Array (3)
    user_mentions : Array (6)
    media : Array (3)
    ```

  - localhost:27020 – User Timeline Database

    ```
    _id: ObjectId('677557ff0d6671ffb94ceead')
    user_id : 534
    posts : Array (10)
    ```

- localhost:27021 – URL Shortener Database



- localhost:27023 – User Database

4. Analisis Hasil Eksperimen dan Implementasi Solusi Pengembangan

Berikut merupakan analisis hasil eksperimen terhadap beberapa *use-case* atau *workload* yang dijalankan serta implementasi solusi pengembangan.

a. Compose post

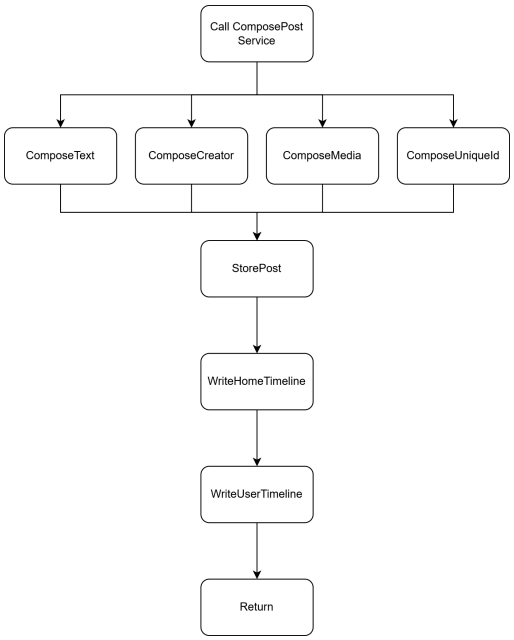
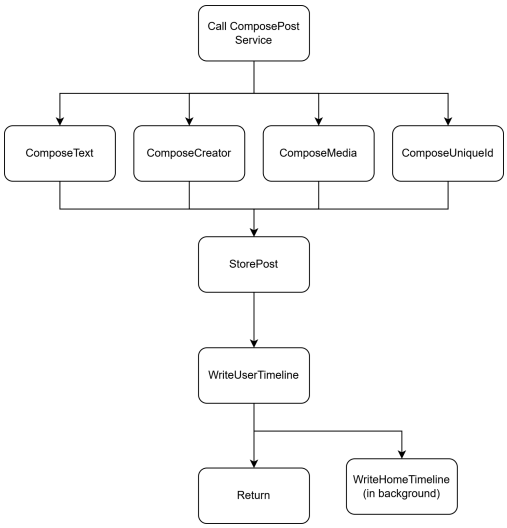
Fungsi `ComposePost` memanggil 4 fungsi secara paralel, yaitu:

- `ComposeText`
- `ComposeCreator`
- `ComposeMedia`
- `ComposeUniqueId`

Setelah keempat fungsi tersebut dijalankan, *post* siap untuk disimpan ke *database* dan dimasukkan ke *user timeline* dan *home timeline*. *User timeline* adalah tampilan *timeline* pada halaman profil *user* berisi seluruh *post* yang dibuat oleh *user* tersebut, sedangkan *home timeline* adalah tampilan halaman utama yang berisi seluruh *post* yang dibuat oleh *followee*.

Penulisan *post* ke dalam *home timeline* dapat menjadi *bottleneck*, karena *post* tersebut harus dituliskan ke *home timeline* dari seluruh *follower*. Fungsi `ComposePost` harus menunggu proses penulisan yang panjang tersebut sebelum *return*. Padahal, *user* tidak perlu menunggu *post*-nya masuk ke *home timeline* seluruh *follower*-nya. *User* hanya perlu menunggu *post*-nya masuk ke *user timeline* miliknya. Oleh karena itu, memasukkan *post* ke *home timeline* dapat dilakukan di *background* menggunakan *thread* terpisah tanpa perlu ditunggu

hingga selesai (*fire-and-forget*), sehingga fungsi `ComposePost` dapat *return* lebih cepat. Berikut merupakan perbandingan diagram alur mekanisme awal dengan mekanisme baru.

Sebelum	Sesudah
	

Berikut adalah perubahan kode yang diterapkan

Sebelum

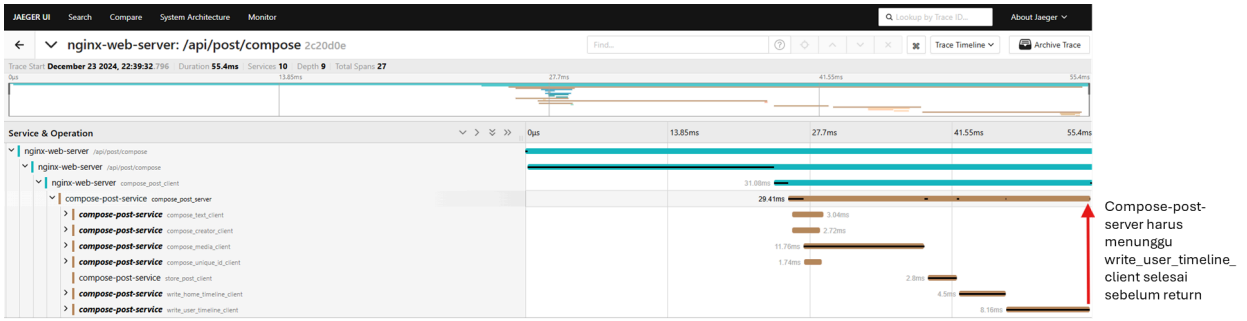
```
auto post_future =
    std::async(std::launch::async, &ComposePostHandler::_UploadPostHelper, this, req_id, post, writer_text_map);

auto user_timeline_future = std::async(std::launch::deferred, &ComposePostHandler::_UploadUserTimelineHelper, this,
                                       req_id, post.post_id, user_id, timestamp, writer_text_map);
auto home_timeline_future = std::async(std::launch::deferred, &ComposePostHandler::_UploadHomeTimelineHelper, this,
                                       req_id, post.post_id, user_id, timestamp, user_mention_ids, writer_text_map);

post_future.get();
home_timeline_future.get();
user_timeline_future.get();

span->Finish();
```

Sesudah

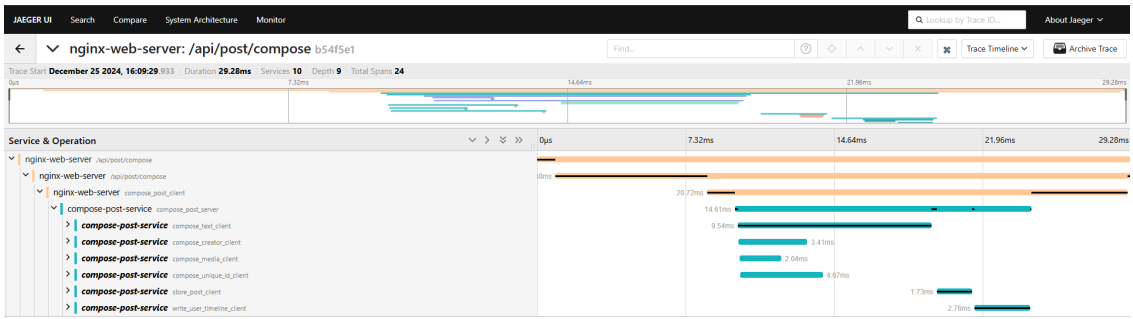
```cpp
auto post_future =
    std::async(std::launch::async, &ComposePostHandler::_UploadPostHelper, this, req_id, post, writer_text_map);

auto user_timeline_future = std::async(std::launch::deferred, &ComposePostHandler::_UploadUserTimelineHelper, this,
                                       req_id, post.post_id, user_id, timestamp, writer_text_map);

post_future.get();
user_timeline_future.get();

auto home_timeline_future = std::async(std::launch::async, &ComposePostHandler::_UploadHomeTimelineHelper, this,
                                       req_id, post.post_id, user_id, timestamp, user_mention_ids, writer_text_map);

span->Finish();
```

Berikut adalah perubahan *trace* yang terjadi

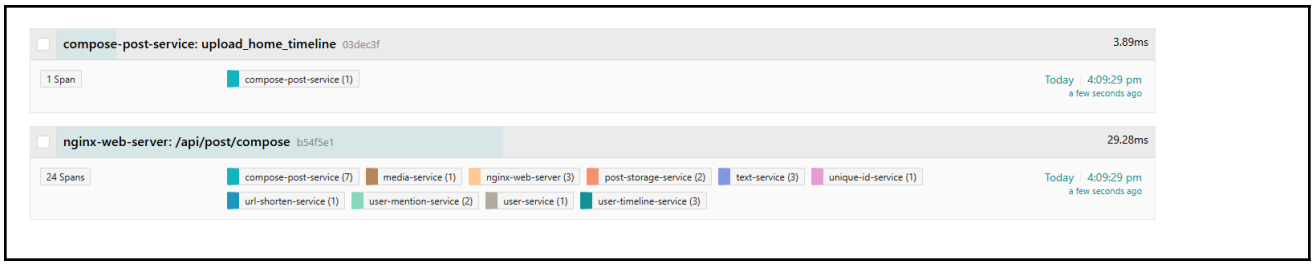
Sebelum (55.4ms)



Sesudah (29.28ms)



Compose-post-server return ketika write_user_timeline selesai, dan write_home_timeline dijalankan secara terpisah



Disclaimer: Eksperimen dilakukan dengan melakukan *post* pada akun yang hanya memiliki 1 *follower*. Kedua *post* memiliki konten yang sama. Total waktu eksekusi dapat mengandung *noise* sehingga perbandingan mungkin tidak benar-benar *apple-to-apple*. Akan tetapi, apabila kita hanya membandingkan waktu eksekusi fungsi `ComposePost` saja, waktu eksekusi yang diperlukan pada mekanisme baru (14.61ms) masih lebih cepat dari waktu eksekusi yang diperlukan pada mekanisme lama (29.41ms). Meskipun begitu, diperlukan eksperimen lebih lanjut dengan kondisi lebih banyak *follower* untuk melihat signifikansi dari optimisasi tersebut. Selain itu, perlu diketahui bahwa optimisasi ini hanya mengurangi latensi tetapi tidak mengurangi beban komputasi.

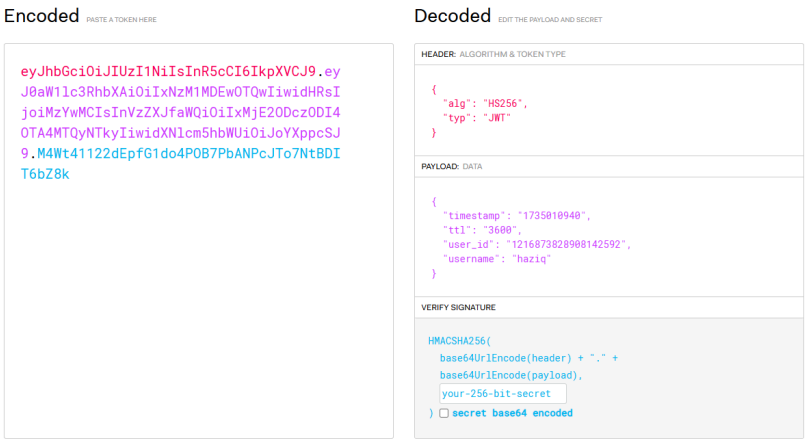
b. *Follow/Unfollow*

Sebelum melakukan *follow*, program menjalankan dua *task* berikut pada fungsi `FollowWithUsername`.

1. Mendapatkan user_id berdasarkan *username* yang melakukan *follow.*
2. Mendapatkan followee_id berdasarkan *username* yang akan di-*follow*.

```
try {
  user_id = user_id_future.get();
  followee_id = followee_id_future.get();
} catch (const std::exception &e) {
  LOG(warning) << e.what();
  throw;
}

if (user_id >= 0 && followee_id >= 0) {
  Follow(req_id, user_id, followee_id, writer_text_map);
}
```

Kedua *task* tersebut sudah cukup efisien karena dilakukan secara paralel. Namun, *task* pertama sebetulnya tidak perlu dilakukan, karena user_id sudah tersimpan pada *cookie* dengan JWT.



Oleh karena itu, alih-alih menggunakan fungsi `FollowWithUsername`, kita dapat menggunakan fungsi `Follow` yang menerima parameter `user_id` dan `followee_id`. Parameter `user_id` sudah didapat dari *login token* pada *cookie*, sehingga kita hanya perlu mencari `followee_id`. Berikut perbedaan kode sebelum dan sesudah optimisasi.

Sebelum

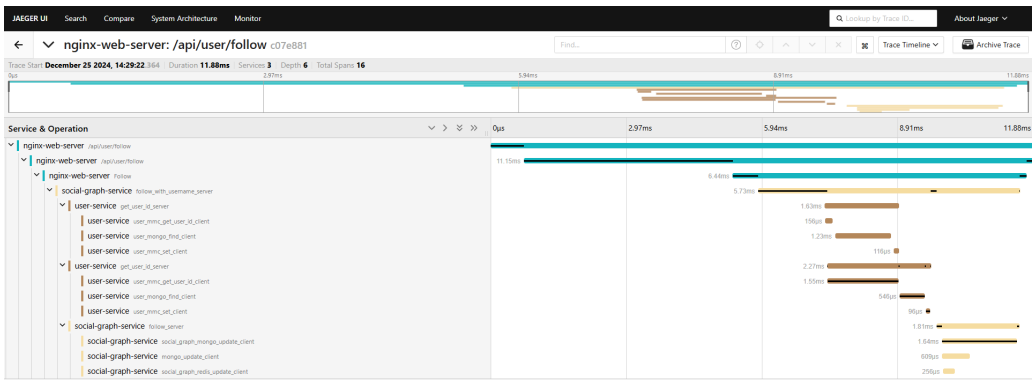
```
-- Follow with user name and followee name
status, err = pcall(client.FollowWithUsername, client,req_id,
    post.user_name, post.followee_name, carrier)
```

Sesudah

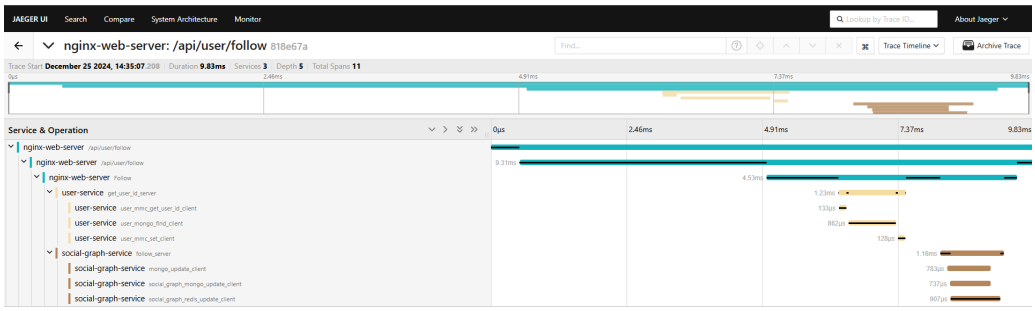
```
-- Get, verify, and unpack login token payload
local login_obj = jwt:verify(ngx.shared.config:get("secret"),
    ngx.var.cookie_login_token)
if not login_obj["verified"] then
  ngx.status = ngx.HTTP_UNAUTHORIZED
  ngx.say(login_obj.reason);
  ngx.exit(ngx.HTTP_OK)
end

-- Get user_id from payload
local user_id = login_obj["payload"]["user_id"]
local username = login_obj["payload"]["username"]

-- Get followee_id
status, result_or_error = pcall(userServiceClient.GetUserId,
    userServiceClient, req_id, post.followee_name, carrier)

if status then
    local followee_id = result_or_error
    -- Follow with user id and followee id
    status, err = pcall(client.Follow, client,req_id,
        tonumber(user_id), followee_id, carrier)
else
    -- Fallback if fail to get followee id
    status, err = pcall(client.FollowWithUsername, client,req_id,
    post.user_name, post.followee_name, carrier)
end
```

Berikut perbedaan *trace* sebelum dan sesudah optimisasi.

Sebelum (11.88ms)



Sesudah (9.83ms)



Perbedaan waktu eksekusi tidak begitu signifikan, karena pada versi sebelumnya, meskipun terdapat dua kali pengambilan id (user id dan followee id), keduanya terjadi secara paralel. Akan tetapi, optimisasi ini dapat mengurangi beban komputasi pada social-graph-service, karena tidak perlu mencari user id yang sudah tersimpan pada *cookie*. Berbeda dengan optimisasi pada bagian a, optimisasi ini justru tidak terlalu mengurangi latensi tetapi mengurangi beban komputasi.

c. *URL Shortener*

URL singkat dan URL asli disimpan dalam MongoDB dengan struktur berikut.

```
{
  "_id": {
    "$oid": "676bcee1be8f005416486832"
  },
  "shortened_url": "http://short-url/s04tmkvLxo",
  "expanded_url":
"http://sRxdgFrSU1MkQSSjBcw2Olb1G8RDjTZcXZAdSwMrjuxjSEh5Wpvngtr
```

```
7FhHoIhDp"
}
```

Dokumen ini sudah menggunakan index pada atribut shortened_url, sehingga mempercepat *lookup*. Terdapat satu kekurangan, yaitu shortened_url berisi *full URL* yang memiliki prefix yang sama, yaitu “http://short-url”. Hal ini dapat memboros penyimpanan. Oleh karena itu, dari pada menyimpan *full URL*, cukup menyimpan ID dari shortened_url nya saja (seperti `s04tmkvLxo`). Selain itu, index pada MongoDB umumnya menggunakan *B-tree*, sehingga membutuhkan waktu *O(log n)* untuk *lookup*. Tipe index bisa diganti menjadi *hashed*, sehingga waktu *lookup* rata-rata bisa dipercepat menjadi *O(1)*. Karena keterbatasan waktu, optimasi yang dilakukan hanya mengubah tipe index menjadi *hashed*. Berikut perubahan-perubahan yang dilakukan pada code.

```
diff --git a/socialNetwork/src/utils_mongodb.h
b/socialNetwork/src/utils_mongodb.h
index 5bbfa6c..cb37b3e 100644
--- a/socialNetwork/src/utils_mongodb.h
+++ b/socialNetwork/src/utils_mongodb.h
@@ -50,7 +50,8 @@ bool CreateIndex(
     mongoc_client_t *client,
     const std::string &db_name,
     const std::string &index,
-    bool unique) {
+    bool unique,
+    const std::string &index_type = "") {
   mongoc_database_t *db;
   bson_t keys;
   char *index_name;
@@ -61,7 +62,12 @@ bool CreateIndex(

   db = mongoc_client_get_database(client, db_name.c_str());
   bson_init (&keys);
-  BSON_APPEND_INT32(&keys, index.c_str(), 1);
+  if (index_type == "hashed") {
+    BSON_APPEND_UTF8(&keys, index.c_str(), "hashed");
+    unique = false;
+  } else {
+    BSON_APPEND_INT32(&keys, index.c_str(), 1);
+  }
   index_name = mongoc_collection_keys_to_index_string(&keys);
   create_indexes = BCON_NEW (
       "createIndexes", BCON_UTF8(db_name.c_str()),
@@ -78,6 +84,7 @@ bool CreateIndex(
```

```
  bson_free (index_name);
  bson_destroy (&reply);
  bson_destroy (create_indexes);
+  bson_destroy (&keys);
  mongoc_database_destroy(db);

  return r;
```

```
diff --git a/socialNetwork/src/UrlShortenService/UrlShortenService.cpp
b/socialNetwork/src/UrlShortenService/UrlShortenService.cpp
index 2b59ac2..760e735 100644
--- a/socialNetwork/src/UrlShortenService/UrlShortenService.cpp
+++ b/socialNetwork/src/UrlShortenService/UrlShortenService.cpp
@@ -60,7 +60,7 @@ int main(int argc, char* argv[]) {
  }
  bool r = false;
  while (!r) {
-    r = CreateIndex(mongodb_client, "url-shorten", "shortened_url", true);
+    r = CreateIndex(mongodb_client, "url-shorten", "shortened_url", true,
"hashed");
    if (!r) {
      LOG(error) << "Failed to create mongodb index, try again";
      sleep(1);
@@ -80,4 +80,4 @@ int main(int argc, char* argv[]) {

  LOG(info) << "Starting the url-shorten-service server...";
  server.serve();
-}
\ No newline at end of file
+}
```

Berikut perubahan pada performa sebelum dan sesudah perubahan. Performa dihitung dari rata-rata durasi *span* ***compose_urls_client*** dari total 50 *traces*.

Sebelum (avg. 1.91 ms)

```
❯ cat traces-btree.json | jq -s 'map(.data | map(.spans) | .[].[] | select(.operationName=="compose_urls
_client") | .duration/1000) | add/length'
1.9181199999999998
```

Sesudah (avg. 1.82 ms)

```
❯ cat traces-hashed.json | jq -s 'map(.data | map(.spans) | .[].[] | select(.operationName=="compose_url
s_client") | .duration/1000) | add/length'
1.82494
```

d. *Media Storage*

File media disimpan langsung di MongoDB oleh Media Frontend seperti ditunjukkan pada kode berikut.

```lua
function _M.UploadMedia()
  ...
  local conn = mongo()
  conn:set_timeout(1000)
  local ok, err = conn:connect("media-mongodb" .. k8s_suffix, 27017)
  if not ok then
    ngx.log(ngx.ERR, "mongodb connect failed: "..err)
  end
  local db = conn:new_db_handle("media")
  local col = db:get_col("media")

  local media = {
    filename = media_id .. '.' ..  media_type,
    file = media_file
  }
  col:insert({media})
  conn:set_keepalive(60000, 100)
  ngx.header.content_type = "application/json; charset=utf-8"
  ngx.say(cjson.encode({media_id = media_id, media_type = media_type}))

end

return _M
```

Hal ini kurang baik untuk dilakukan, karena mongoDB umumnya dioptimisasi untuk menyimpan dan meng-*query* dokumen (JSON), bukan *binary file*. Selain itu, *file* dapat memakan *storage* dan menyebabkan I/O yang lebih lama. *File* media sebaiknya disimpan pada *object storage* (seperti S3) yang memang dioptimisasi untuk menyimpan *file*. *Metadata* dari *file* dapat disimpan di MongoDB, tetapi *file* tersebut disimpan di S3. Optimasi ini tidak diimplementasikan karena keterbatasan waktu.

Lampiran

- Link Repository implementasi: https://github.com/haziqam/social-network-performance-engineering