IMPLEMENTASI *GRAPHQL* UNTUK MENGATASI *UNDER-FETCHING* PADA PENGEMBANGAN SISTEM INFORMASI PELACAKAN ALUMNI POLITEKNIK NEGERI MALANG

Annisa Taufika Firdausi[1], Dhebys Suryani Hormansyah[2], Fany Ervansyah[3]

[1,2,3] Program Studi Teknik In*form*atika, Jurusan Teknologi Informasi, Politeknik Negeri Malang
[1]annisa.taufika@polinema.ac.id, [2] dhebys.suryani@polinema.ac.id, [3]fanyervansyah.9c@gmail.com

Abstrak

Pendistribusian data pada sistem informasi memiliki berbagai macam cara yang digunakan. Diantaranya adalah dengan menggunakan metode *REST API*. Namun, terdapat beberapa kekurangan yang menjadi masalah pada *REST API*.Salah satunya adalah masalah under-fetching, yaitu masalah dimana bagian *frontend* harus melakukan lebih dari satu kali *request* ke *backend* untuk memenuhi kebutuhan data yang diperlukan oleh *frontend*. Tentunya *under-fetching* ini dapat menurunkan performa sistem yang dibuat. Salah satu cara untuk mengatasi *under-fetching* dapat dilakukan dengan menerapkan *GraphQL* sebagai metode pendistribusian data. Tujuan dari penelitian kali ini adalah untuk membandingkan performa antara sistem informasi dengan *REST API* dan sistem informasi dengan *GraphQL*. Studi kasus pada penelitian ini adalah pada pengembangan sistem informasi pelacakan alumni Politeknik Negeri Malang dimana dibuat dua versi sistem informasi yang metode pendistribusian data REST dan menggunakan *GraphQL*.Hasil penelitian ini berupa perbandingan performa antara sistem informasi dengan metode pendistribusian data *REST API* dan sistem informasi dengan metode pendistribusian data *GraphQL*. Berdasarkan hasil percobaan yang dilakukan *GraphQL* menunjukkan performa yang baik pada jumlah data yang besar dan kompleks, serta ketika terdapat banyak pengguna yang mengakses data dalam waktu yang bersamaan. Sedangkan untuk data yang sederhana dan sistem informasi yang tidak memiliki banyak pengguna yang akan mengakses data secara bersamaan, maka *REST API* masih lebih unggul.

Kata kunci : Sistem Informasi, *GraphQL, under-fetching*

1. Pendahuluan

Distribusi data merupakan hal yang sangat penting dalam pengembangan sebuah sistem informasi. Dalam pendistribusian data, beberapa *web*site menerapkan metode yang berbeda-beda. Mulai dari menggabungkan antara bagian yang bertugas menampilkan informasi ke pengguna (*Frontend*) dan bagian yang mengatur bagaimana data diolah (*Backend*) hingga menyediakan layanan distribusi data seperti *REST API*, untuk sistem dengan struktur *frontend* dan *backend* yang terpisah.

REST API adalah salah satu teknik yang populer karena merupakan sebuah teknik yang tidak tergantung pada protokol spesifik, dapat menggunakan *HTTP* dan menggunakan *form*at *JSON* untuk berkomunikasi (Doglio, 2018). Namun, dalam metode *REST API*, terdapat suatu masalah yang disebut *under-fetching*, dimana bagian *frontend* perlu untuk melakukan permintaan data lebih dari 1 kali ke bagian *backend* (Porcello & Banks, 2018). Hal itu akan meningkatkan latensi, yang membuat pengakses halaman *web* harus menunggu lebih lama sebelum data dikirim pada bagian *frontend* dengan sempurna. Selain itu, kompleksitas program juga semakin bertambah karena bagian *frontend* harus menambahkan satu permintaan data lagi ke bagian *backend*.

Salah satu cara mengatasi *under-fetching* pada *REST API* adalah dengan membuat *endpoint* baru yang melakukan pengambilan data sesuai dengan apa yang diminta oleh *frontend*. Namun, jika terdapat banyak data yang mirip dan memiliki *endpoint* masing-masing, maka bentuk kode pada bagian *backend* akan menjadi kompleks dan kurang baik diakibatkan oleh adanya kode yang memiliki fungsi mirip namun ditulis lebih dari 1 kali.

Pada tahun 2015, secara publik, Facebook meluncurkan sebuah *query language* yang menjadi metode baru dalam mengatur pendistribusian data. Nama *query* language tersebut adalah *GraphQL*. Salah satu masalah yang dapat diatasi oleh *GraphQL* adalah masalah *under-fetching*.

Banyak penelitian sebelumnya yang membahas mengenai pemanfaatan *GraphQL* dalam menyelesaikan masalah *under-fetching*, antara lain dilakukan oleh (Hartig & Pérez, 2017), pada penelitian tersebut disimpulkan bahwa bahasa Graph milik *GraphQL* memiliki kompleksitas yang rendah, selanjutnya ada juga penelitian yang membandingkan antara REST dan *GraphQL* yang dilaksanakan oleh (Eizinger, 2017).

Penelitian lain memanfaatkan *GraphQL* dalam riset mengenai kemampuan *GraphQL* dalam berkomunikasi dan bertukar data pada ekosistem yang menggunakan teknologi data *driven*, pada penelitian tersebut disimpulkan bahwa *GraphQL* menunjukkan peningkatan dalam fleksibelitas, maintainabilitas dan performa (Vazquez-Ingelmo, Cruz-Benito, & García-Penalvo, 2017).

Peneliti lain juga membahas mengenai performa pengambilan data yang mengkombinasikan antara Falcor dan Relay + *GraphQL* para penelitian tersebut masih belum dapat menentukan mana yang memberikan latency yang lebih rendah antara Relay + *GraphQL* atau *REST API* namun berhasil menemukan bahwa Falcor dan Relay+*GraphQL* dapat menurunkan jumlah *request* untuk data flow yang paralel dan sekuensial untuk single *request* (Cederlund, 2016).

GraphQL juga dapat dimanfaatkan sebagai content delivery pada Kentico Cloud, ada penelitian yang mencoba mengganti content delivery dari *REST API* ke *GraphQL* pada sebuah CMS online, penelitian tersebut menyimpulkan bahwa *GraphQL* dapat digunakan sebagai pengganti *REST API* namun tidak terdapat perbedaan signifikan untuk penurunan data flow pada jaringan antara *GraphQL* dan *REST API* (Čechák, 2017).

Penelitian lain membahas mengenai perbandingan performa antara *GraphQL* dan *REST API* pada sistem informasi, penelitian tersebut menyimpulkan bahwa *GraphQL* dapat menjadi pilihan yang tepat ketika data yang dibutuhkan sering berubah (Hartina, Lawi, & Panggabean, 2018).

Mengkombinasikan antara *GraphQL* dengan teknologi lain juga menjadi topik yang sering diteliti antara lain (Mukhiya, Rabbiab, Punax, Rutle, & Lamo, 2019) dimana pada penelitian tersebut dilakukan evaluasi proses pertukaran informasi pada sebuah healthcare system menggunakan kombinasi *GraphQL* dan HL7 FHIR.

Penelitian mengenai kemampuan *GraphQL* dalam melakukan *query* data menyimpulkan bahwa *GraphQL* dapat melakukan *query* dengan efektif sesuai situasi dan tidak mengirimkan data yang tidak berlebihan jika dibandingkan dengan *REST API* (Jeon, Liuhaoyang, & Hwang, 2019). Sampai sekarang evaluasi dan perbandingan kemampuan antara *GraphQL* dan *REST API* masih terus menjadi topik yang menarik untuk diteliti (Brito & Valente, 2020) melakukan eksperimen membandingan *GraphQL* dan *REST API*, (Lee, Kwon, & Yun, 2020) mengukur performa *GraphQL* pada ESS (Energy Storage System) dan yang terbaru (Vadlamani, Emdon, Arts, & Baysal, 2021) meneliti apakah *GraphQL* dapat menggantikan *REST API*.

Pada penelitian ini akan dilakukan evaluasi implementasi *GraphQL* untuk mengatasi *under-fetching* pada Pengembangan Sistem Informasi Pelacakan Alumni Politeknik Negeri Malang yang akan penulis kembangkan. Dengan adanya *GraphQL*, diharapkan dapat meningkatkan performa situs *web* Sistem Informasi Pelacakan Alumni Politeknik Negeri Malang.

2. Tinjauan Pustaka

2.1 *Underfetching*

Under-fetching menjadi masalah pada suatu sistem informasi karena memiliki dampak pada penurunan performa suatu situs *web*. Penurunan performa tersebut terjadi dikarenakan adanya proses `meminta kembali` suatu data pada *backend* di *endpoint* yang berbeda. Hal ini menyebabkan adanya tambahan aktivitas pada sistem. Secara teori, hal itulah yang menyebabkan performa sistem informasi menjadi terhambat.

Sebagai contoh, jika dalam suatu sistem terdapat halaman yang melakukan *request* ke 2 *endpoint* yang berbeda, misalnya /*user*/id untuk mendapatkan data pengguna dan /*user*s/id/*item*s untuk mendapatkan data barang yang dimiliki pengguna, maka sistem memiliki 2 aktivitas yang harus dilakukan, yaitu meminta data pada *endpoint* /*user*s, sekaligus meminta data pada *endpoint* /*user*s/*item*s.

Masalah ini dapat diatasi dengan cara menggabungkan 2 data yang berbeda tersebut menjadi 1 *endpoint* saja. Namun, hal ini juga memiliki kekurangan dimana jika sistem hanya akan menggunakan data diri pengguna saja, sistem juga akan menerima data *item* pengguna yang tidak diperlukan. Sehingga, terjadilah pengambilan data yang berlebihan.

Masalah inilah yang diselesaikan oleh *GraphQL*. *GraphQL* membantu sistem untuk menentukan data apa yang harus diambil berdasarkan *query* yang tersedia dari *backend*. *Backend* menentukan data apasaja yang boleh diambil, dan bagian *frontend* juga bisa menentukan data apa saja yang bisa diambil sesuai dengan apa yang disediakan oleh *backend*.

Misalnya, terdapat data pengguna yang terdiri dari *name*, *email*, *role*, dan password. Bagian *backend* menyediakan *query* untuk mengambil data pengguna yang berupa *name*, *email*, dan *role*. Maka, bagian *frontend* dapat mengambil data dari *query* tersebut, entah mengambil *name* saja, *name* dan *email* saja, dan sebagainya. Namun, bagian *frontend* tidak akan bisa mengambil *password* karena *query* tidak disediakan oleh *backend*.

GraphQL juga dapat mengambil data dari beberapa *query* sekaligus. Misalkan terdapat *query user* dan *item*, maka bagian *frontend* dapat melakukan pengambilan data *user* dan data *item* dalam 1 kali *request*. Hal inilah yang menjadikan *GraphQL* mampu untuk menyelesaikan masalah *under-fetching*.

3. Metode Penelitian

Penelitian dibagi menjadi tiga tahapan besar yaitu tahap pengambilan data, tahap pengembangan sistem dan tahap pengujian.

3.1 Metode Pengambilan Data

Pengambilan data dilakukan dengan cara scraping pada situs *Linkedin* dan input secara manual melalui sistem informasi yang akan dibuat.

Pada pengambilan data di *Linkedin*, data yang diambil adalah data mahasiswa yang sudah mendaftarkan Politeknik Negeri Malang sebagai riwayat pendidikan mereka. Mekanisme pengambilan datanya adalah dengan menggunakan bot untuk melakukan login terotomatisasi, kemudian masuk ke halaman Politeknik Negeri Malang, lalu membuka halaman detail setiap card dari mahasiswa yang muncul. Dari halaman detail yang sudah dibuka, akan diambil data-data yang diperlukan.

Pada pengambilan data dari input manual, pengambilan data dilakukan dengan mengisikan data alumni melalui *form* yang sudah disediakan di sistem. *Form* tersebut memiliki *field* diantaranya berupa nama, tahun masuk dan lulus dari Politeknik Negeri Malang, jurusan yang diambil, pekerjaan saat ini, jabatan/posisi yang dipegang pada pekerjaan saat ini, dan *email*.

3.2 Metode Pengembangan Sistem

Metode pengembangan aplikasi yang penulis gunakan adalah metode *Agile* dengan *framework Kanban*. Alasan penulis memilih metode *Kanban* karena metode ini memiliki *WIP (Work-In-Progress)* yang memberi batasan berapa banyak tugas yang bisa diselesaikan (Kniberg & Skarin, 2010). Sehingga, pengembang dapat fokus pada beberapa tugas terlebih dahulu. Untuk teknologi pada bagian *frontend*, pennulis akan menggunakan *React.js*, yaitu sebuah *library* untuk mengembangkan bagian *frontend* dari suatu *web* (Alex & Eve, 2017). Untuk *backend*, penulis akan menggunakan *Node.js* karena *Node.js* mampu membuat *server-side* program dengan menggunakan bahasa *javascript* (Brown, 2014).

3.3 Metode Pengujian

Setup pengujian di lakukan pada *website* sistem informasi alumni yang sudah *dipublish* ke *sebuah virtual private server*. Sistem informasi alumni ini terdapat dua versi yaitu versi yang memanfaatkan *GraphQL* dan yang menggunakan *REST API*.

Pengujian pertama akan dilakukan dengan cara memuat suatu halaman yang sama dan memiliki masalah *under-fetching*, pada Sistem Informasi yang menggunakan *GraphQL* maupun yang menggunakan REST sebagai metode pendistribusian datanya. Kemudian, halaman akan di-*refresh* sebanyak 20 kali. Pada masing-masing sesi *refresh*, akan ada beberapa data yang dicatat pada masing-masing sistem informasi. Diantaranya adalah jumlah *request* ke *backend*, waktu muat *request* hingga selesai, dan waktu total memuat halaman. Dari data-data tersebut, akan dirata-rata pada masing-masing sistem informasi dan akan dibandingkan. Sehingga, dapat diketahui kelebihan/kekurangan dari *GraphQL* dan REST jika digunakan dalam pendistribusian data.

Uji coba kedua adalah dengan menggunakan bantuan *tool* bernama K6, yaitu *tool* untuk melakukan berbagai macam load testing. Penulis akan menguji kemampuan *backend* dari masing-masing metode pendistribusian data dalam menangani jumlah *user* yang melakukan permintaan data secara bersamaan. Jumlah *user* yang disimulasikan sejumlah 50 *user*s dan batas durasi yang diberikan adalah 1 menit. Sehingga, 50 *user*s akan melakukan *request* secara bersamaan dan selama 1 menit, akan dicatat berapa *user* yang mendapatkan respon dari *backend*.

GraphQL dikatakan berhasil menyelesaikan masalah *under-fetching* jika sistem yang dikembangkan dengan mengimplementasikan *GraphQL* dapat melakukan pengaksesan *endpoint* lebih sedikit daripada sistem dengan metode *REST API* dalam memenuhi kebutuhan datanya atau jika server dengan *GraphQL* dapat merespon sejumlah *user* yang lebih banyak dibandingkan dengan *REST API*.

4. Hasil dan Pembahasan

4.1 Hasil Uji Muat Halaman *Dashboard*

Tabel 1 Hasil Uji Muat Halaman *Dashboard*

Metode	Jumlah *Request* ke *backend*	Kecepatan rata-rata
REST API	5	623 ms
GraphQL	1	635,7 ms

Pada halaman *dashboard* di sistem informasi dengan REST, terjadi 5 kali *request* dengan data yang sederhana, yaitu berupa total dari beberapa kategori data, dan ditampilkan sekaligus.

Hasil akhir uji muat halaman *dashboard* menunjukkan bahwa *GraphQL* berhasil mengatasi masalah *under-fetching* dengan hanya memerlukan 1 *endpoint* saja untuk memenuhi semua kebutuhan datanya. Sedangkan REST membutuhkan 5 kali *request* ke *backend* untuk memenuhi kebutuhan datanya. Namun, kecepatan muat halaman pada sistem informasi dengan REST masih lebih cepat dibandingkan sistem informasi dengan *GraphQL*. Selisih waktu dari masing-masing rata-rata kecepatan muat halaman adalah 12,7 milisekon dengan sistem informasi yang REST lebih unggul.

Dari hasil tersebut, terlihat bahwa *GraphQL* dapat mengatasi masalah *under-fetching*. Namun, untuk kasus data yang sederhana, kasus *under-fetching* tidak memberikan dampak yang mempengaruhi performa. Sehingga, *REST API* masih lebih cepat dibandingkan *GraphQL* dalam pemuatan data untuk ditampilkan pada halaman yang dimaksud.

4.2 Hasil Uji Muat Halaman List Alumni

Tabel 2 Hasil Uji Muat Halaman List Alumni

Metode	Jumlah *Request* ke *backend*	Kecepatan rata-rata
REST API	2	1.398 ms
GraphQL	1	944,5 ms

Pada halaman list alumni, terdapat masalah *under-fetching* pada sistem informasi dengan REST. Dimana, sistem harus melakukan 2 kali *request* ke *backend* untuk memenuhi kebutuhan yang berupa data beberapa alumni dengan sumber yang berbeda, pengaturan halaman, total halaman dan total data yang didapat sekaligus. Hal ini menunjukkan bahwa pada halaman list alumni, terdapat data yang lebih kompleks dan besar dibanding pada halaman *dashboard*, namun dengan kasus *under-fetching* tidak sebesar pada halaman *dashboard* karena hanya terdapat 2 kali *request* ke *backend*.

Hasil uji coba muat halaman list alumni menunjukkan bahwa *GraphQL* berhas il mengatasi masalah *under-fetching* dengan hanya melakukan 1 kali *request* ke *backend* untuk memenuhi kebutuhan datanya. Sedangkan sistem informasi dengan REST membutuhkan 2 kali *request* ke bagian *backend* untuk memenuhi kebutuhan data yang akan ditampilkan. Perbandingan rata-rata kecepatan muat halaman pada masing-masing sistem menunjukkan bahwa *GraphQL* lebih cepat dibandingkan dengan *REST API*. Selisih dari rata-rata kecepatan muat halaman dari masing-masing sistem informasi adalah 453,5 milisekon dengan dengan *GraphQL* yang lebih unggul.

Dari hasil tersebut, dapat dilihat bahwa, untuk data yang besar dan lebih kompleks dibandingkan dengan halaman *dashboard*, kasus *under-fetching* mulai memberikan dampak pada performa suatu halaman *web* dan *GraphQL* mampu mengatasi masalah tersebut sekaligus meningkatkan performa suatu halaman *web*. Sehingga, waktu muat halaman *web* pada sistem informasi dengan *GraphQL* dapat lebih singkat dibandingkan dengan waktu muat halaman *web* dengan *REST API*.

4.3 Hasil Uji Muat Halaman Edit Alumni

Tabel 3 Hasil Uji Muat Halaman Edit Alumni

Metode	Jumlah *Request* ke *backend*	Kecepatan rata-rata
REST API	2	738,85 ms
GraphQL	1	779,7 ms

Pada halaman ini, terdapat kasus *under-fetching* dimana sistem informasi dengan REST melakukan 2 kali *request* dengan data yang tidak terlalu besar dibandingkan dengan halaman list alumni, yaitu meminta detail dari data alumni yang akan diubah dan meminta list jurusan yang sudah terdaftar pada sistem. Artinya, pada halaman edit alumni ini terdapat kasus *under-fetching* yang sama dengan halamn list alumni, yaitu 2 kali *request*, namun datanya tidak sebesar data pada halaman list alumni.

Hasil uji muat halaman edit alumni menunjukkan bahwa *GraphQL* dapat mengatasi masalah *under-fetching* dengan hanya membutuhkan 1 kali *request* saja ke *backend* untuk memenuhi data yang diperlukan. Sedangkan sistem informasi dengan *REST API* membutuhkan 2 kali *request* untuk memenuhi kebutuhan data yang akan ditampilkan. Selisih rata-rata waktu muat halaman pada sistem informasi dengan *REST API* dan sistem informasi dengan *GraphQL* adalah 40,85 milisekon dengan *REST API* lebih unggul dibandingkan *GraphQL*.

Dari data tersebut, dapat dilihat bahwa untuk kasus data yang lebih kecil dibandingkan pada halaman list alumni dengan jumlah *request* yang sama, yaitu 2 kali *request*s, kasus *under-fetching* tidak memberikan pengaruh yang besar pada performa halaman *web*, namun lebih memberikan pengaruh pada efisiensi penggunaan kuota data internet. Sehingga, meskipun kasus *under-fetching* dapat diatasi, kecepatan muat halaman pada sistem informasi dengan *GraphQL* masih sedikit lebih lambat dibandingkan dengan sistem informasi dengan *REST API*.

4.4 Hasil Load Testing *Endpoint* Halaman *Dashboard*.

Tabel 4 Hasil *Load Testing* Halaman *Dashboard*

Metode	Iterasi	Kecepatan rata-rata/iterasi (s)	Total data dikirim (B)	Total data diterima (B)
REST API	151	16	361524	1012434
GraphQL	156	16	108636	222666

Hasil load testing *endpoint* halaman *dashboard* pada sistem informasi dengan REST dan *GraphQL* terdiri dari beberapa kategori dengan perlakuan yang sama pada masing-masing sistem. Dilakukan permintaan ke *backend* oleh 50 *user*. Ketika 1 *user* telah menyelesaikan *request* data yang dibutuhkan dan mendapatkan respon dari *backend*, *user* akan segera melakukan *request* kembali ke

backend. Perlakuan ini akan terulang secara terus-menerus selama 1 menit.

Hasil dari uji coba pada *backend* dengan *REST API* menunjukkan bahwa terjadi 151 iterasi permintaan dari 50 *user* dengan jumlah *request* sebanyak 755 dan semua permintaan berhasil mendapatkan respon sukses dari *backend*. Rata-rata waktu yang dibutuhkan untuk masing-masing iterasi adalah 16 detik dengan waktu maksimal 21 detik dan waktu minimal 4 detik.

Hasil dari load testing *backend* dengan *GraphQL* menunjukkan bahwa terjadi 156 iterasi pengguna dengan jumlah *request* sebanyak 156 kali dan semua iterasi mendapatkan respon sukses dari *backend*. Rata-rata waktu yang dibutuhkan untuk setiap iterasi adalah 16 detik dengan waktu maksimal 20 detik dan waktu minimal 2 detik.

Dari segi eifisiensi penggunakan kuota internet, jumlah data yang diterima pada sistem informasi dengan *REST API* adalah 1012434 *bytes* dan jumlah data yang dikirimkan adalah 361524 *bytes*. Pada sistem informasi dengan *GraphQL*, data yang diterima adalah 222666 *bytes* dan data jumlah data yang dikirmkan adalah 108636 *bytes*.

Dari percobaan tersebut, dapat dilihat bahwa *GraphQL* lebih mampu dalam memberikan respon ke *user* dengan jumlah respon sebanyak 156 kali. Jumlah ini lebih banyak 5 iterasi dibandingkan dengan respon yang berhasil diberikan oleh *backend* dengan *REST API*. Untuk kecepatan respon, *backend* dengan *GraphQL* mapun *backend* dengan *REST API* memiliki rata-rata kecepatan yang sama. Dari segi efisiensi kuota data internet, dapat dilihat bahwa *GraphQL* mengirimkan dan menerima data lebih efisien dari *REST API*. Hal ini terjadi karena pada masing-masing *request* di sistem informasi dengan REST, terjadi validasi token dan token yang sama dikirimkan berkali-kali ke *backend*. Karena mengakses banyak *endpoint*, maka sistem informasi dengan REST menerima berbagai data dengan tambahan header data pada masing-masing respon. Sedangkan pada *GraphQL*, karena hanya terjadi 1 kali *request*, maka token hanya dikirmkan sekali dan juga mendapatkan keseluruhan data dalam 1 respon.

4.5 Hasil Load Testing *Endpoint* Halaman List Alumni.

Hasil load testing *endpoint* halaman list alumni pada sistem informasi dengan REST dan *GraphQL* terdiri dari beberapa kategori dengan perlakuan yang sama pada uji coba sebelumnya. Dilakukan permintaan ke *backend* oleh 50 *user*. Ketika 1 *user* telah menyelesaikan *request* data yang dibutuhkan dan mendapatkan respon dari *backend*, *user* akan segera melakukan *request* kembali ke *backend*. Perlakuan ini akan terulang selama 1 menit.

Tabel 5 Hasil *Load Testing* Halaman *List* Alumni

Metode	Iterasi	Kecepatan rata-rata/iterasi (s)	Total data dikirim (B)	Total data diterima (B)
REST API	291	9,5	232650	5220098
GraphQL	309	8,9	321321	4263453

Hasil dari percobaan pada *backend* dengan REST menunjukkan bahwa terjadi 291 iterasi permintaan dari 50 *user* dengan jumlah *request* sebanyak 602 *request*, dengan 582 *request* berhasil mendapatkan respon sukses dari *backend*, sedangkan sisanya masih belum selesai diproses. Rata-rata durasi waktu yang dibutuhkan untuk masing-masing iterasi adalah 9,5 detik dengan waktu maksimal 13,7 detik dan waktu tersingkat 7 detik.

Pada *backend* dengan *GraphQL*, terjadi 309 iterasi permintaan dari 50 *user* dengan jumlah *request* 309 kali serta keseluruhannya mendapat respon sukses dari *backend*. Rata-rata durasi waktu yang diperlukan untuk masing-masing iterasi adalah 8,9 detik dengan waktu maksimal 17,8 detik dan waktu minimal 1,7 detik.

Dari segi efisiensi kuota internet, jumlah data yang diterima pada sistem informasi dengan REST adalah 5220098 *bytes* dan data yang dikirimkan adalah 232650 *bytes*. Sedangkan pada sistem informasi dengan *GraphQL*, data yang diterima adalah 4263453 *bytes* dan data yang dikirimkan adalah 321321 *bytes*.

Dari hasil di atas, dapat dilihat bahwa *GraphQL* mampu menangani jumlah *user* yang lebih banyak dibanding dengan *REST API* di mana *GraphQL* mampu merespon hingga iterasi ke 309, sedangkan REST mampu merespon *request* hingga iterasi ke-291. untuk durasi respon, *GraphQL* dan *REST API* memiliki selisih 0,6 detik dengan *GraphQL* yang lebih unggul. Dari segi efisiensi kuota data, dapat dilihat bahwa untuk data yang diterima, *GraphQL* lebih hemat sumber daya. Namun, pada bagian data yang dikirmkan, *REST API* lebih unggul. Hal ini terjadi karena *GraphQL* mengirimkan data berupa *query* pada *backend*, sedangkan REST hanya mengirimkan *endpoint* apa yang dituju. Sehingga, REST menggunakan kuota data lebih sedikit dibandingkan dengan *GraphQL*. Namun, karena *GraphQL* mengirimkan *query* ke *backend*, maka *GraphQL* dapat menentukan data apasaja yang dibutuhkan oleh *frontend*. Sehingga, tidak terjadi kelebihan data dari jumlah data yang diperlukan. Hal inilah yang membuat *GraphQL* dapat menghemat kuota lebih baik dari REST ketika menerima data.

4.6 Hasil Load Testing *Endpoint* Halaman Edit Alumni

Load testing pada *endpoint* halaman edit alumni dilakukan dengan menerapkan perlakuan yang sama pada load testing *endpoint* halaman list alumni dan halaman *dashboard*, dimana terdapat 50 *user* yang
akan melakukan *request* ke bagian *backend*. Setiap 1 *user* selesai mendapatkan respon dari *backend*, *user* tersebut akan segera melakukan *request* kembali. Hal ini akan terus berulang selama 1 menit.

Hasil yang diberikan oleh *backend* dengan REST menunjukkan bahwa terjadi 410 iterasi dengan jumlah *request* sebanyak 821 dan sebanyak 820 *request* berhasil direspon dengan sukses oleh *backend*. Sedangkan sisanya belum selesai direspon. Rata-rata kecepatan tiap iterasi adalah 6,8 detik dengan waktu terlama 8,7 detik dan waktu tercepat 1,8 detik.

Tabel 6 Hasil *Load Testing* Halaman *Edit* Alumni

Metode	Iterasi	Kecepatan rata-rata/iterasi (s)	Total data dikirim (B)	Total data diterima (B)
REST API	410	6,8	316560	1697150
GraphQL	415	6,8	344355	956845

Sedangkan hasil yang diberikan oleh *backend* dengan *GraphQL* menunjukkan bahwa terjadi 415 iterasi dengan jumlah *request* 415 kali dan keseluruhannya mendapatkan respon sukses dari *backend*. Rata-rata durasi tiap iterasi adalah 6,8 detik dengan durasi terlama 8,5 detik dan durasi tersingkat adalah 0,5 detik.

Dari segi efisiensi kuota internet, sistem informasi dengan *REST API* menerima data sebesar 1697150 *bytes* dan mengirimkan data sebesar 316560 *bytes*. Pada sistem informasi dengan *GraphQL*, data yang diterima sebesar 956845 *bytes* dan data yang dikirimkan adalah 344355 *bytes*. Dari data tersebut, dapat dilihat bahwa *GraphQL* mampu menghemat penggunaan data yang diterima hampir 2 kali dari *REST API*.

Dari hasil uji coba di atas, dapat dilihat bahwa *GraphQL* dapat melakukan lebih banyak iterasi dibandingkan *REST API* yang artinya *GraphQL* dapat memanajemen *request* dari *user* dengan baik ketika terdapat banyak *user* yang mengakses data secara bersamaan. Untuk rata-rata kecepatan, kedua sistem *backend* memiliki rata-rata kecepatan yang sama. Dari segi efisiensi penggunaan kuota internet, *GraphQL* cukup unggul dibandingkan *REST API*.

5. Kesimpulan

Dari hasil load test sistem informasi yang sudah dibuat dengan *GraphQL* dan sistem informasi dengan *REST API*, dapat diambil kesimpulan bahwa, pada proses pemuatan halaman dengan data yang besar dan beragam, kasus *under-fetching* memberikan dampak pada durasi pemuatan halaman dan *GraphQL* mampu menangani pemuatan halaman lebih cepat. Pada kasus pemuatan halaman dengan data yang sederhana seperti pada halaman *dashboard* dan edit alumni, masalah *under-fetching* tidak memberikan dampak pada kecepatan muat halaman dan *REST API* masih lebih unggul.

Pada load test *endpoint* masing-masing *backend*, dapat diambil kesimpulan bahwa *GraphQL* mampu melayani lebih banyak *user* dibandingkan dengan *REST API*. Ini dibuktikan dengan lebih banyaknya iterasi yang terjadi pada hasil load test *backend GraphQL* dibandingkan dengan jumlah iterasi yang terjadi pada hasil load test *backend* dengan *REST API*. Kemudian, dari segi efisiensi penggunakan kuota internet, *under-fetching* memberikan masalah berupa pengiriman data pada beberapa respon. Sehingga, selain *frontend* harus mengirimkan *token* yang sama untuk divalidasi ke *backend* pada masing-masing *endpoint*, *frontend* juga akan mendapatkan data dari berbagai *respon* dengan berbagai *header*. Hal ini menyebabkan pembuangan kuota internet untuk mengirimkan dan mendapatkan data yang tidak perlu. *GraphQL* dapat mengatasi hal ini dengan menggunakan 1 *endpoint* dan mengirimkan *query* yang berisikan kebutuhan data apa saja yang kita inginkan untuk digunakan di *frontend*.

Dengan begitu, dapat diambil kesimpulan bahwa, untuk suatu halaman *web* yang sederhana dan dengan jumlah *user* yang tidak terlalu banyak dalam satu waktu akses, *REST API* masih dapat diandalkan sebagai metode distribusi data. Namun, ketika aplikasi atau *web*site tersebut sudah mulai berkembang menjadi lebih kompleks dengan data yang cukup banyak dan dengan jumlah pengguna yang besar dalam satu waktu akses, serta berpeluang besar untuk terjadi *under-fetching*, *GraphQL* mulai diperlukan sebagai solusi dalam mengatasi *under-fetching* dan mengoptimalkan kemampuan server dalam memberikan respon secara cepat kepada pengguna serta menghemat penggunaan kuota data internet oleh *user*. Sehingga, pengguna akan lebih nyaman dan merasa betah berada di halaman *web* yang dibuat. Karena pengguna merasa lebih nyaman, maka tentu hal ini akan dapat meningkatkan reputasi dari halaman *web* yang dikembangkan.

Daftar Pustaka:

Alex, B., & Eve, P. (2017). *Learning React Functional Web Development With React and Redux*. 1005 Gravenstein Highway North, Sebastopol, CA 954721005 Gravenstein

Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc.

Biehl, M. (2015). *API Architecture: The Big Picture for Building APIs*.

Brito, G., & Valente, M. T. (2020). REST vs *GraphQL*: A controlled experiment. *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020*, (Dcc), 81–91. https://doi.org/10.1109/ICSA47634.2020.00016

Brown, E. (2014). *Web* development with Node & Express. In *O'Reilly Media, Inc.* (Vol. 1). 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc.

Buna, S. (2021). *GraphQL in Action*. Shelter Island, NY 11964: Manning Publication Co.

Čechák, D. (2017). Using *GraphQL* for Content Delivery in Kentico Cloud. *Is.Muni.Cz*. Retrieved from https://is.muni.cz/th/qm0cs/thesis.pdf

Cederlund, M. (2016). *Performance of frameworks for declarative data fetching : An evaluation of Falcor and Relay+GraphQL*. (2016:92), 105.

Doglio, F. (2018). *REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development*. Retrieved from https://isbnsearch.org/isbn/9781484237144

Eizinger, T. (2017). *API Design in Distributed Systems: A Comparison between GraphQL and REST*. 1–64. Retrieved from https://eizinger.io/assets/Master-Thesis.pdf

Hartig, O., & Pérez, J. (2017). An initial analysis of facebook's *GraphQL* language. *CEUR Workshop Proceedings*, *1912*.

Hartina, D. A., Lawi, A., & Panggabean, B. L. E. (2018). Per*form*ance Analysis of *GraphQL* and RESTful in SIM LP2M of the Hasanuddin University. *Proceedings - 2nd East Indonesia Conference on Computer and Information Technology: Internet of Things for Industry, EIConCIT 2018*, 237–240. https://doi.org/10.1109/EIConCIT.2018.8878524

Hillar, G. C. (2016). *Building RESTful Python Web Services*. Birmingham: Packt Publishing Ltd.

Jeon, D. C., Liuhaoyang, & Hwang, H. (2019). Design of Hybrid Application Based on *GraphQL* for Efficient *Query* for PHR. *ICTC 2019 - 10th International Conference on ICT Convergence: ICT Convergence Leading the Autonomous Future*, 381–383. https://doi.org/10.1109/ICTC46691.2019.8940003

Kniberg, H., & Skarin, M. (2010). *Kanban And Scrum Making The Most Of Both* (1st ed.). C4Media Inc.

Lee, E., Kwon, K., & Yun, J. (2020). Per*form*ance Measurement of *GraphQL* API in Home ESS Data Server. *International Conference on ICT Convergence*, *2020-October*, 1929–1931. https://doi.org/10.1109/ICTC49870.2020.9289569

Mukhiya, S. K., Rabbiab, F., Punax, V. K. I., Rutle, A., & Lamo, Y. (2019). A *GraphQL* approach to healthcare in*form*ation exchange with hl7 fhir. *Procedia Computer Science*, *160*, 338–345. https://doi.org/10.1016/j.procs.2019.11.082

Porcello, E., & Banks, A. (2018). Learning *GraphQL*. In *O'Reilly Media, Inc.O'Reilly Media, Inc.* (1st ed.). 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc.

Sharma, S. (2021). *Modern API Development with Spring and Spring Boot*. Birmingham: Packt Publishing Ltd.

Vadlamani, L., Emdon, B., Arts, J., & Baysal, O. (2021). *Can GraphQL Replace REST? A Study of Their Efficiency and Viability*. 10–17. https://doi.org/10.1109/SER-IP52554.2021.00009

Vazquez-Ingelmo, A., Cruz-Benito, J., & García-Penalvo, F. J. (2017). Improving the OEEU's data-driven technological ecosystem's interoperability with *GraphQL*. *ACM International Conference Proceeding Series*, *Part F1322*.