Analisis dan Implementasi Graph Indexing Pada Graph Database Menggunakan Algoritma Lindex

Astrid F. Septiany[1], Kemas Rahmat S. W.[2], Adiwijaya[3]
[1.2.3.]Fakultas Informatika, Telkom University
Jalan Telekomunikasi No.1, Dayeuh Kolot, Bandung 40257
astridfrillya@gmail.com[1], bagindok3m45@gmail.com[2], kang.adiwijaya@gmail.com[3]

Abstrak

Database yang awalnya disebut sebagai penyimpanan data terintegrasi mengalami perkembangan untuk memenuhi kebutuhan data yang kompleks.Namun basis data relasional tidak dapat selalu diandalkan karena seiring berjalannya waktu aplikasi-aplikasi pengguna layanan basis data relasional mengalami perlambatan kecepatan akses. Sebagai alternatif dari permasalahan tersebut maka basis data relasional dikembangkan ke dalam representasi graf yang termasuk dalam kelompok NoSQL yaitu graph database. Graph database semakin berkembang dan memerlukan metode untuk memproses query agar lebih efisien. Oleh karena itu dibutuhkan algoritma indexing. Untuk data dengan skala yang besar, permasalahan indexing yang terletak pada filtering dan verification merupakan hal yang penting. Lindex dipilih untuk penelitian ini karena berdasarkan sumber yang dirujuk, Lindex memiliki waktu index construction yang lebih singkat dibandingkan Gindex dan memiliki kekuatan filtrasi yang lebih bagus dari SwiftIndex. Tipe data graf yang akan digunakan pada penelitian kali ini adalah molekul karena memiliki node yang berlabel, edge yang tak berbobot dan tak berarah, sesuai dengan sumber yang dirujuk, serta ukuran basis data yang besar dan sesuai dengan permasalahan di atas. Dilakukan pengujian terhadap 4 buah dataset yang memiliki 3 buah path length yang berbeda. Dari keempat dataset yang diuji, dilakukan analisis mengenai waktu index construction, waktu pencarian candidate set, dan waktu response time dari masing-masing path length yang berbeda.

Kata kunci: *graph, graph database, graph indexing, Lindex, filtering, verification, path length*

1. PENDAHULUAN

Selama bertahun-tahun *database* selalu diandalkan sebagai media penyimpanan data-data berjumlah besar. Namun tidak selamanya basis data relasional dapat selalu diandalkan karena seiring berjalannya waktu seluruh aplikasi pengguna layanan basis data relasional mengalami perlambatan kecepatan akses. Hal tersebut tidak lain dikarenakan oleh kinerja basis data relasional yang cenderung memburuk ketika harus berhadapan dengan *dataset* dalam jumlah yang besar dan saling terhubung [4].

Sebagai alternatif dari permasalahan tersebut maka basis data relasional dikembangkan ke representasi graf dimana yang semula data disimpan pada beberapa tabel di dalam sebuah *database* menjadi sekumpulan *node* dan *edge* yang saling berhubungan satu dengan yang lainnya yang termasuk dalam kelompok NoSQL yaitu *graph database* [4]. Penggunaan *graph database* semata-mata bukan untuk menggantikan keberadaan basis data relasional melainkan untuk melengkapi segala kekurangan salah satunya permasalahan seperti yang telah dijelaskan di atas. Selain itu *graph database* juga dapat diandalkan dalam masalah fleksibilitas serta agilitas [4].

Dalam penggunaannya kini *graph database* pun semakin berkembang sehingga mulai dibutuhkannya efisiensi *query* untuk mempercepat proses pencarian. Untuk data dengan skala yang besar permasalahan *indexing* yaitu *filtering* dan *verification* merupakan hal yang penting agar proses *query* dapat menjadi lebih efisien. Berdasarkan hal tersebut terdapat 2 kategori algoritma *indexing graph database* yaitu algoritma berbasis *feature* dan algoritma berbasis *non-feature*[10]. Kinerja algoritma berbasis *non-feature* tidak sebaik algoritma berbasis *feature* dalam memaksimalkan *filtering* [11]. Hal ini dikarenakan untuk membangun *index*, algoritma berbasis *feature*akan mengambil *feature set* terlebih dahulu kemudian membangun *inverted index* untuk memfasilitasi filtrasi **[**10**]**. Dari sekian metode pada algoritma berbasis *feature* seperti Gindex [9], TreePi [12], dan SwiftIndex [5], *indexing* berbasis *Lattice* atau Lindex merupakan metode yang sangat cocok digunakan karena mengutamakan *power of filtering* untuk *indexing* [10].

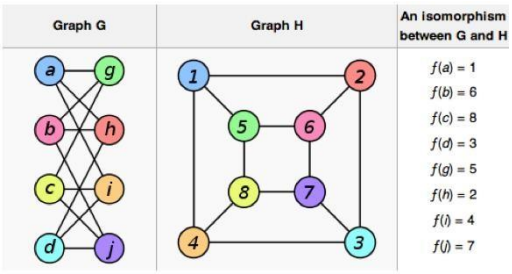
Berdasarkan pengujian yang telah dilakukan, *index construction* Lindex dengan *index structure* yang sama dengan Gindex, menunjukkan waktu yang lebih cepat dibandingkan dengan waktu *index construction* dari Gindex [11]. Waktu filtrasi dari SwiftIndex tidak sebagus Lindex ketika menjalankan proses filtrasi karena SwiftIndex tidak menambah *filtering power* pada saat memproses *query* yang berjumlah sedikit [11]. Oleh karena itu metode Lindex dipilih untuk penelitian disamping karena Lindex dapat menggunakan segala jenis substruktur *feature set* [11].Tipe data graf yang akan digunakan pada penelitian kali ini adalah molekul [4] karena memiliki *node* yang berlabel, *edge* yang tak berbobot dan tak berarah, sesuai dengan sumber yang dirujuk [11], serta ukuran basis data yang besar dan sesuai dengan permasalahan di atas.

2. LANDASAN TEORI

2.1 Isomorfisme Subgraf

Secara intuitif, dua graf G1 dan G2 adalah sama jika salah satu dari dua graf tersebut, misalnya G2 dapat digambar kembali, sedemikian hingga G2 identik dengan G1 dan hal ini disebut dengan graf isomorfik [7].Untuk menegaskan bahwa kedua graf dapat dikatakan isomorfik apabila mengikuti beberapa aturan berikut ini :

1. Mempunyai jumlah simpul yang sama.
2. Mempunyai jumlah sisi yang sama.
3. Mempunyai jumlah simpul yang sama berderajat tertentu.

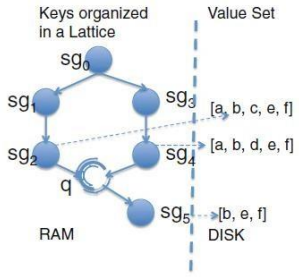


Gambar 1 Isomorfisme Subgraf [3]

Ketiga syarat ini ternyata belum cukup menjamin. Pemeriksaan secara visual perlu dilakukan.

2.2 Lindex

Lindex terdiri dari sekumpulan *key* dan *value*. *Key* merupakan sekumpulan subgraf yang terkandung di dalam graf tertentu, sedangkan *value* merupakan sekumpulan graf yang mengandung subgraf tertentu.



Gambar 2 Contoh Lindex [11]

Lindex [11] merupakan salah satu teknik *graph indexing* yang berbasis *feature*. Struktuktur index Lindex terdiri dari pengaturan *feature* di dalam *lattice*. Proses pencarian *query* dalam Lindex didukung oleh pencarian *maximal subgraph query* serta pencarian *minimal supergraph query*. Selain itu dalam proses pembangunan *index*, Lindex tidak membutuhkan tambahan *memory* karena hanya mengandalkan keterhubungan antar *feature*. Proses berjalannya metode ini dimulai dengan *index construction*, kemudian mengidentifikasi *maximal subgraph*, lalu mencari titik pertemuan dari *maximal subgraph*,selanjutnya menentukan *candidate set*, lalu melewati tahapan *pruning candidate set* hingga akhirnya menemukan *answers set*.

2.2.1 *Index Construction*

Index construction pada Lindex menggunakan *feature set* yang telah dipilih sebagai masukkan dan kemudian akan menghasilkan *lattice*. Langkah pertama yaitu mencari seluruh relasi keterikatan (*containment relationship*) antar *features*. Selanjutnya *mapping* akan disimpan. Setelah itu, *features* akan diurutkan berdasarkan jumlah sisinya mulai dari yang terbesar hingga yang terkecil. Tahap selanjutnya yaitu memutuskan relasi antar *feature* dan mulai membangun *lattice*. Untuk setiap verteks yang memiliki *child* akan dicek satu persatu apakah setiap *child* merupakan supergraf dari *parent*. Apabila *child* merupakan supergraf dari *parent*, makan akan disambungkan oleh sisi berarah. Pengecekan tersebut akan terus berulang hingga semua verteks telah selesai dicek. Setelah itu, *lattice* akan diberikan label dengan ditelurusi secara *Depth First Search.*

2.2.2 *Maximal Subgraph Search*

Untuk mencari *candidate set*, *lattice* akan menelusuri seluruh *node feature* untuk mencari informasi mengenai dari *parent* mana sebuah node berasal. Setelah *query* dibagi menjadi beberapa *path* yang kemudian menjadi sekumpulan *feature* sesuai dengan masukkan *user*, setiap *feature* akan ditelusuri pada *lattice* dan *mapping* dari sgi dimana sgi adalah anggota dari *node lattice*, akan disimpan. Pertama *lattice* akan mulai ditelusuri dari *root* menuju ke *child root* untuk melihat apakah terdapat *mappings* dari *child root* ke *query* atau M(Sgc,q). Apabila *child root* tidak mengandung *query*, maka penelusuran akan dilanjutkan ke *child* dari *child root* atau Sgc' dan *mappings*-nya kemudian akan disimpan. Penelusuran akan terus berulang hingga seluruh *mapping node* menuju *query* telah ditemukan. Setelah ditemukan, selanjutnya *maximal subgraph* dari *query* akan ditentukan dari *super-set* maxSub untuk *query* yang diberikan.

2.2.3 *Minimal Supergraph Search*

Minimal supergraph search digunakan untuk menghemat langkah tes isomorfisme subgraf. Hal ini dikarenakan ketika sebuah graf merupakan supergraf dari *query*, graf tersebut tentu akan mengandung semua subgraf *query* yang ada di dalam *lattice*. Oleh karena itu *value set minimal supergraph* dari masing-masing *feature query* akan ditelusuri kemudian digabungkan. Sebuah *node index* Sgi dikatakan *minimal supergraph* dari *query* apabila Sgi merupakan supergraf dari *query* dan tidak ada lagi subgraf dari Sgi selain *query*.

$$\text{[illegible]} = \{ \text{[illegible]} \mid \text{[illegible]} \subset \text{[illegible]} \nexists x \in P \text{[illegible]} \subset x \subset \text{[illegible]} \} \quad (1)$$

$$\text{[illegible]} = \{ \text{[illegible]} \mid \text{[illegible]} \subset \text{[illegible]} \nexists x \in P \text{[illegible]} \subset x \subset \text{[illegible]} \} \quad (2)$$

Setelah hasil *union* dari *minimal supergraph* dari masing-masing *feature query* telah ditemukan, selanjutnya akan mengalami *intersection* dengan *value set* hasil *intersection maximal subgraph* seluruh *feature query*. Hasil perpotongan *minimal supergraph query* beserta *maximal subgraph query* ini yang kemudian menjadi *candidate set.*

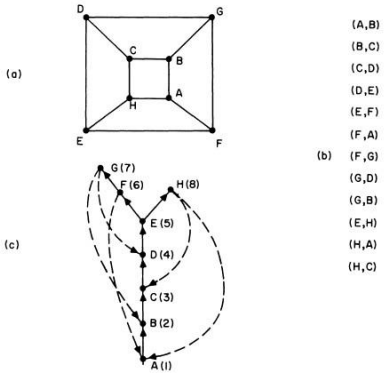
2.3 Algoritma Ullman

Algoritma Ullman [3] merupakan salah satu algoritma *matching* yang kanonikal eksak. Sebagai salah satu masalah *NP-Complete*, satu-satunya cara agar kita dapat menambah kecepatannya yaitu dengan mengenalkan beberapa aturan. Berdasarkann metode *brute-force*, algoritma Ullman menggunakan beberapa aturan dan memperbaiki cara untuk mengurangi ruang pencarian dan menambah kecepatan.

2.4 Algoritma *Depth First Search*

Misalnya G adalah graf yang ingin ditelusuri. Inisialisasi seluruh verteks dari G yang belum ditelusuri. Dimulai dari sebuah verteks dari G dan pilihlah sisi untuk diikuti. Telusuri sisi yang mengarah ke sebuah verteks baru. Kemudian, setiap langkah pilihlah sisi yang belum pernah dikunjungi yang berasal dari verteks yang telah dikunjungi dan telusuri sisi tersebut.Sisi mengarah ke beberapa verteks, baik yang baru ataupun yang sudah dikunjungi. Apabila verteks yang telah dikunjungi tidak memiliki sisi lagi, jika ada, pilihlah verteks lain yang belum dikunjungi, dan memulai penelusurin baru dari titik ini. Pada akhirnya kita hanya akan menelusuri seluruh sisi dari G, dan masing- masing sisi ditelusuri tepat satu kali.

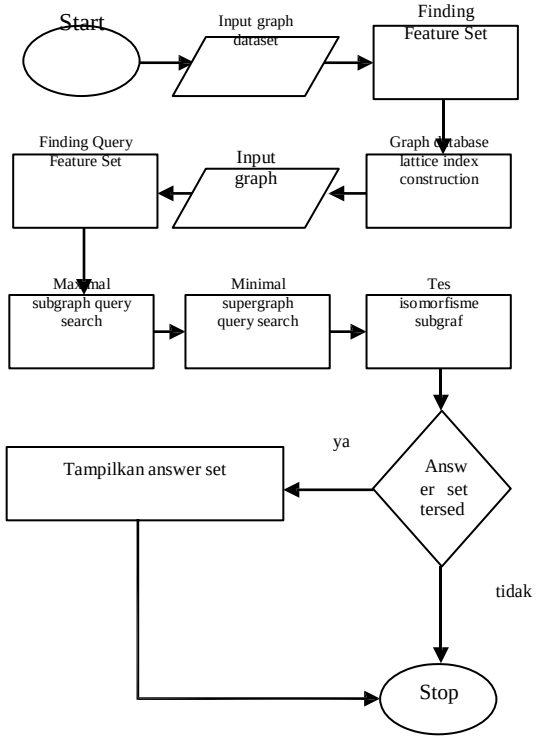
Banyak cara untuk mencari graf, tergantung bagaimana cara pemilihan sisi yang akan dicari. Berdasarkan aturan : ketika memilih sebuah sisi untuk ditelusuri, selalu pilih sisi dari verteks yang baru saja dikunjungi yang memiliki sisi yang belum ditelusuri. Pencarian seperti ini dinamakan *depth first search*. Sekumpulan verteks yang telah dikunjungi dengan kemungkinan sisi yang belum dikunjungi akan disimpan pada tumpukan. Maka dari itu DFS sangat mudah di program dibandingkan secara *iterative* maupun rekursif.



Gambar 3 (a) Graf yang akan ditelusuri, (b) Keterhubungan antar Verteks, (c) Palm Tree yang Dihasilkan DFS

3. PERANCANGAN DAN IMPLEMENTASI

Saat sistem dijalankan, pertama *user*akan memasukkan graf *dataset* molekul berformat SMILES. Setelah itu pada tahap *index constructiondatabase* sistem akan memproses *dataset* hingga terbangun menjadi *lattice*. Kemudian setelah user memasukkan *query* yang diinginkan, sistem akan segera melakukan *maximal subgraphquery* kemudian mencari *minimal supergraph query* untuk menentukan *candidate set*. Selanjutnya sistem akan melakukan tes isomorfisme subgraf untuk melakukan proses verifikasi *candidate set* yang mengandung *query*. Apabila *answer set* ditemukan maka sistem akan menampilkan *answer set*. Namun jika tidak ditemukan maka sistem akan segera berhenti bekerja.



Gambar 4 Perancangan Sistem

4. PENGUJIAN DAN ANALISIS

4.1 Skenario Pengujian

Berikut merupakan skenario pengujian yang dilakukan dalam penelitian kali ini :

1. Pengujian pengaruh *path length* terhadap waktu *index construction*.
2. Pengujian pengaruh *path length* terhadap waktu pencarian *candidate set*.
3. Pengujian pengaruh *path length* terhadap *response time* seluruh proses *query*.

4.1.1 Pengujian Pengaruh *Path Length* terhadap Waktu *Index Construction*

Pada pengujian ini akan dilakukan pengujian pengaruh *path length* terhadap waktu pembentukan *index* pada *graph database* yang akan digunakan. Pengujian pembentukan *index* akan dilakukan pada *path length* yang telah ditentukan yaitu 3, 6, dan 9 untuk masing-masing *dataset* dengan data berjumlah 250, 502, 751, dan 1001. *Path length* 9 dipilih karena *path length* tersebut merupakan *path* terlengkap dengan waktu yang lebih singkat dibandingkan *path length* maksimal atau *path length* 10. *Path length* 3 dipilih karena *path length* tersebut memiliki *path* yang cukup lengkap dibandingkan *path length* 1 dan 2 dan memiliki waktu tercepat dibandingkan *path length* lain yang akan diuji. Sedangkan *path length* 6 dipilih untuk membuat pemilihan *path length* agar berselang 3, yang berguna agar dapat melihat perkembangan dan pola waktu pengujian dari *path length* terkecil dengan waktu tercepat hingga *path length* terbesar dengan waktu terlama. Begitupula untuk pemilihan jumlah *dataset*. Jumlah *dataset* dipilih agar dapat melihat dan menganalisis hasil pengujian dari *dataset* dengan jumlah terkecil hingga jumlah terbesar.

4.1.2 Pengujian Pengaruh *Path Length* terhadap Waktu Pencarian *Candidate Set*

Pada pengujian ini akan dilakukan pengujian pengaruh *path length* terhadap waktu pencarian *candidate set*. Waktu pencarian *candidate set* merupakan penjumlahan waktu pencarian *maximal subgraph* dan *minimal supergraph* dari *query* dengan waktu *intersection value set* dari *maximal subgraph* dan *minimal supergraph query*.

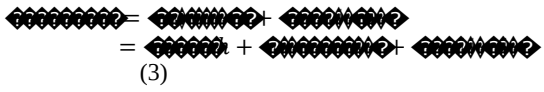
Adapun*query* yang akan menjadi masukkan yaitu berjenis tanpa siklik dan siklik dengan jumlah *node* yang sama, yaitu sebagai berikut :

Tabel 1 Query Pengujian

Q	*Query*	Jenis Graf	Jumlah *Node*
Q1	BrC(C)C(Br)	Tanpa siklik	5
Q2	s1cc[nH0]c1	Siklik	5

4.1.3 Pengujian Pengaruh *Path Length* terhadap *Response Time* Seluruh Proses *Query*

Pengujian ini akan menghitung *response time* dari *query processing* yang dilakukan pada setiap *path length* yang telah ditentukan dari setiap *graph database*. *Response time* merupakan penjumlahan dari waktu pencarian *maximal subgraph query*, *minimal supergraph query*, dan irisan *value set* (filtrasi) dengan lamanya tes isomorfisme subgraf (verifikasi).



(3)

4.2 Analisis Hasil Pengujian

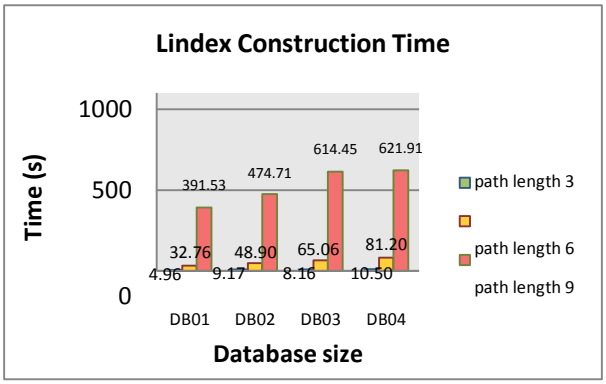
Di bawah ini akan dijelaskan mengenai hasil analisis pengujian yang telah dilakukan beserta hasil pengujian berdasarkan skenario pengujian.

4.2.1 Analisis Hasil Pengujian Pengaruh *Path Length* terhadap Waktu *Index Construction*

Pada pengujian akan ditentukan *path length* untuk setiap *dataset* yang diujikan untuk mengukur waktu yang dibutuhkan dalam *index construction*. Maka didapatkan hasil seperti di bawah ini :

Tabel.2 Hasil Pengujian Pengaruh Path Length terhadap Index Construction

Path Length	DB01	DB02	DB03	DB04
3	4.962	9.1674	8.1554	10.5036
6	32.7552	48.8986	65.0616	81.1989
9	391.5313	474.7086	614.4475	621.9135



Gambar.5: Grafik Pengujian Pengaruh Path Length terhadap Waktu Lindex Construction

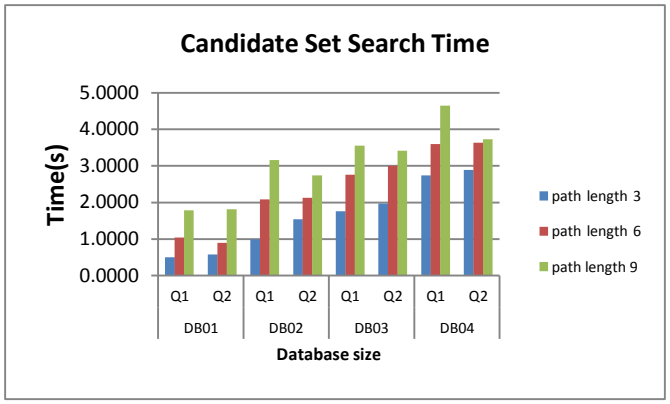
Berdasarkan hasil pengujian pada Tabel 2 dan Gambar 6, dapat disimpulkan bahwa, semakin besar panjang *path* yang dipilih maka akan semakin lama waktu *index construction*. *Path* dengan *length path* yang lebih besar akan memakan waktu yang lebih lama karena untuk mencari *path* dengan panjang tertentu maka dibutuhkan *path* dengan panjang yang sebelumnya. Misalnya jika kita ingin memilih *path length* 4, maka sistem akan mencari dan menampilkan *path* di bawah *path* dengan panjang 4, yaitu *path length* 1,2, dan 3. Hal ini menyebabkan jumlah *feature set* yang akan dimasukkan ke dalam *index construction* juga akan semakin banyak. Sehingga waktu penyusunan *lattice* dan pemberian label pada setiap *node* yang akan dibangun pada *lattice* akan semakin besar.

4.2.2 Analisis Hasil Pengujian Pengaruh *Path Length* terhadap Waktu Pencarian *Candidate Set*

Untuk melakukan pengujian pengaruh *path length* terhadap waktu pencarian *candidate set*, maka telah ditentukan 2 buah *query* yang memiliki jumlah *node* yang sama. Waktu pencarian *candidate set* diperoleh dari penjumlahan waktu pencarian *maximal subgraph* dan *minimal supergraph*, serta *waktu intersection value set*.
Hasil yang didapatkan dari pengujian pengaruh *path length* terhadap waktu pencarian *candidate set* yaitu seperti di bawah ini.

Tabel.3 Hasil Pengujian Pengaruh Path Length terhadap Waktu Pencarian Candidate Set

Path Length	DB01		DB02		DB03		DB04	
	Q1	Q2	Q1	Q2	Q1	Q2	Q1	Q2
3	0.5027	0.5782	0.9888	1.5441	1.7571	1.9660	2.7418	2.8851
6	1.0392	0.8982	2.0807	2.1274	2.7553	3.0029	3.5950	3.6312
9	1.7852	1.8127	3.1612	2.7433	3.5525	3.4127	4.6425	3.7244



Gambar.6 Grafik Pengujian Pengaruh Path Length terhadap Waktu Pencarian Candidate Set

Berdasarkan hasil pengujian pada Tabel 3 dan Gambar 7, maka dapat disimpulkan bahwa, *path length* sangat berpengaruh pada waktu pencarian *candidate set*. Semakin besar panjang *path* yang dipilih maka akan semakin lama waktu pencarian *candidate set*. *Path* dengan *length path* yang lebih besar akan memakan waktu yang lebih lama karena untuk mencari *path* dengan panjang tertentu maka dibutuhkan *path* dengan panjang yang sebelumnya. Misalnya jika kita ingin memilih *path length* 4, maka sistem akan mencari dan menampilkan *path* di bawah *path* dengan panjang 4, yaitu *path length* 1,2, dan 3. Hal ini menyebabkanpenyusunan *node index* pada *lattice* semakin banyak sesuai *feature* yang dimasukkan pada tahap *index construction*. Sehingga waktu penelusuran *index* akan lebih lama dibandingkan dengan *index* dengan *path length* yang lebih kecil.
Disamping itu dapat terlihat bahwa perbedaan waktu pencarian *candidate set* antara *query* 1 dan *query* 2 sangatlah signifikan. Hal ini disebabkan oleh kompleksnya *query* yang dimasukkan. Jenis *query* pada *query* 1 lebih sederhana karena terbentuk tanpa siklik dan bentuk molekul lainnya. Sehingga waktu pengecekan antara setiap *feature set* pada *query* 1 dengan setiap *feature set* pada *node index* akan berlangsung lebih cepat dibandingkan dengan waktu pengecekan antara setiap *feature set* pada *query* 2 dengan setiap *feature set* pada *node index*.

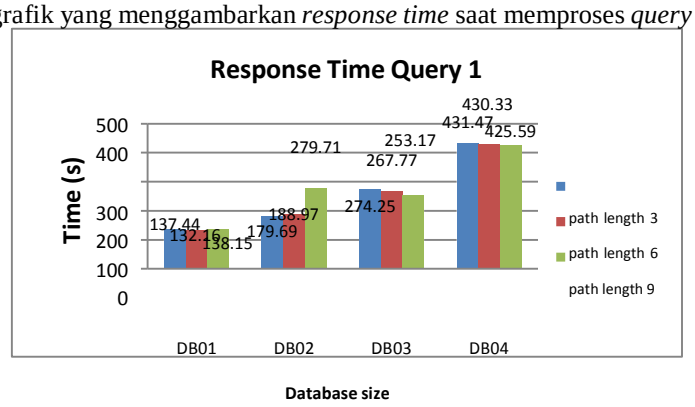
4.2.3 Analisis Hasil Pengujian Pengaruh *Path Length* terhadap *Response Time* Seluruh Proses *Query*

Seperti yang sudah dijelaskan di atas, *response time* merupakan penjumlahan waktu filtrasi dan verifikasi. Berdasarkan pengujian pengaruh *path length* terhadap *response time* seluruh proses *query*, maka didapatkan hasil seperti berikut ini.

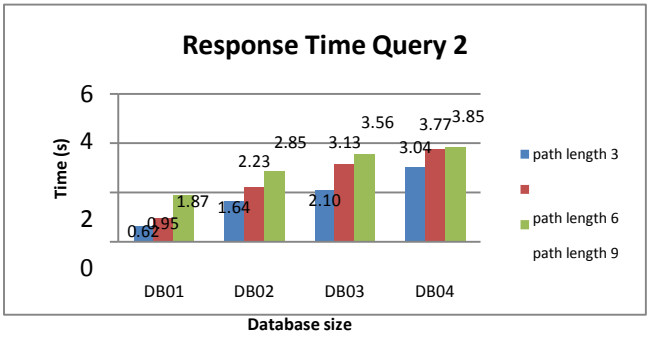
Tabel.4 Hasil Pengujian Pengaruh Path Length terhadap Response Time Seluruh Proses Query

	3		6		9	
	Q1	Q2	Q1	Q2	Q1	Q2
DB01	137.438	0.6232	132.1643	0.9465	138.1506	1.8695
DB02	179.691	1.6431	188.967	2.2252	279.7098	2.8522
DB03	274.245	2.1023	267.7711	3.1349	253.1664	3.5587
DB04	431.473	3.0387	430.3285	3.7712	425.5937	3.8503

Di bawah ini merupakan grafik yang menggambarkan *response time* saat memproses *query* 1 dan *query* 2.



Gambar 7 Grafik Pengujian Pengaruh Length Path terhadap Response Time Query 1



Gambar.8 Grafik Pengujian Pengaruh Length Path terhadap Response Time Query 2

Berdasarkan Tabel 4 dan Gambar 8 serta Gambar 9, dapat dilihat bahwa perubahan *response timequery* 1 atau pun 2 dari DB01, DB02, DB03, dan DB04, terlepas dari jumlah *path*, selalu menunjukkan perkembangan. Hal ini menunjukkan bahwa *path length* tidak selalu berpengaruh bagi *response time* Lindex. Ukuran *database* lebih berpengaruh dibandingkan *path length* karena jumlah *candidate set* yang ditemukan pada setiap *database* dapat berbeda. Sehingga hal ini menyebabkan waktu proses isomorfisme pada setiap *database* tidak menentu. Dan apabila kita perhatikan bahwa perbedaan *response time query* 1 dan 2 cukup besar.*Response time* terbesar pada *query* 1 yaitu 431,473 s sedangka *response time* terbesar pada *query* 2 hanya 3.8503 s.
Berikut ini merupakan jumlah *candidate set* pada *query* 1 dan *query* 2 beserta waktu tes isomorfisme.

Tabel.5 Jumlah Candidate Setdan Waktu Tes Isomorfisme pada Setiap Path Length di Query 1 dan 2

	Path Length	Jumlah *Candidate Set*	Iso *Test* (T)
Query 1	3	23	136.9348
	6	23	131.1251
	9	23	136.3654
	3	47	178.7026
	6	47	186.8863
	9	47	276.5486
	3	71	272.4874
	6	71	265.0158
	9	71	249.6139

	3	94	428.7311
	6	94	426.7335
	9	94	420.9512
Query 2	3	1	0.045
	6	1	0.0483
	9	1	0.0568
	3	2	0.099
	6	2	0.0978
	9	2	0.1089
	3	3	0.1363
	6	3	0.132
	9	3	0.146
	3	3	0.1536
	6	3	0.14
	9	3	0.1259

Berdasarkan tabel di atas dapat disimpulkan bahwa jumlah *candidate set* sangat mempengaruhi waktu tes isomorfisme subgraf. Dan selain itu dapat terlihat pula bahwa *path length* mempengaruhi jumlah *candidate set* pada *query* 2 yang berjumlah 8 buah *node* dengan 1 buah siklik. Dengan ini dapat ditarik kesimpulan bahwa waktu tes isomorfisme subgraf bergantung pada jumlah *candidate set* dan jumlah *candidate set* bergantung pada jumlah *path length* yang dipilih.

5. PENUTUP

5.1 Kesimpulan

Berdasarkan pengujian yang telah dilakukan pada Bab 4, maka diperoleh beberapa kesimpulan seperti berikut ini :

1. Semakin besar panjang *path* yang dipilih maka akan semakin lama waktu *index construction*. Hal ini dikarenakan apabila panjang *path* yang dipilih semakin besar, maka jumlah *feature* yang akan dimasukkan ke dalam *index construction* juga akan semakin banyak.
2. Semakin besar panjang *path* yang dipilih maka akan semakin lama waktu pencarian *candidate set*. Hal ini dikarenakan *node index* yang tersusun pada *lattice* akan semakin banyak sesuai *feature* yang dimasukkan pada tahap *index construction*.
3. Panjang *query* menentukan waktu pencarian *candidate set*. Semakin panjang *query* maka waktu yang diperlukan untuk mencari *candidate set* akan semakin lama.
4. *Path Length* tidak selalu berpengaruh bagi *response time* Lindex. Ukuran *database* lebih berpengaruh dibandingkan *path length* karena jumlah *candidate set* yang ditemukan pada setiap *database* dapat berbeda. Sehingga hal ini menyebabkan waktu proses isomorfisme pada setiap *database* tidak menentu.
5. Waktu tes isomorfisme subgraf bergantung pada jumlah *candidate set* [8] dan jumlah *candidate set* bergantung pada jumlah *path length* yang dipilih.

5.2 Saran

Setelah proses pengerjaan tugas akhir ini, berikut ini adalah beberapa saranuntuk penelitia kedepannya, khususnya untuk ruang lingkup *graph indexing* :

1. Untuk penelitian *graph indexing* berikutnya diharapkan dapat menggunakan metode berbasis *feature* lainnya seperti SwiftIndex.
2. Untuk penelitian selanjutnya diharapkan dapat menggunakan *graph database* dan jenis *query* yang lebih kompleks, misalnya dengan siklik yang lebih dari 1.

3. Untuk penelitian selanjutnya diharapkan dapat menggunakan tipe *dataset* dan struktur graf yang berbeda dengan *dataset* molekul.
4. Untuk penelitian selanjutnya diharapkan dapat menggunakan *dataset* yang berukuran lebih besar besar.

DAFTAR PUSTAKA

[1] Anon., 2011. Daylight Theory Manual. Aliso Viejo, CA: Daylight Chemical Information Systems, Inc.

[2] Deo, N., 1992. Graph Teory with Applications to Engineering and Computer Science. New Delhi: Pretince Hall of India.

[3] Dongoran, E.S.S., 2015. Analisis dan Implementasi Graph Indexing Pada Graph Database Menggunakan Algoritma GraphGrep.

[4] Ian, R., Weber, J. & Eifrem, E., 2013. Graph Databases. O'Reilly Media.

[5] Shang, H., Zhang, Y., Lin, X. & Xu Yu, J., 2008. Taming Verification Hardness: an Efficient Algorithm for Testing Subgraph Isomorphism. Proceedings of the VLDB Endowment, pp.364-75.

[6] Singh, H. & Sharma, R., 2012. Role of Adjacency Matrix & Adjacency List in Graph Theory. International Journal of Computers & Technology, III(1).

[7] Suryadi, D. & Priatna, N., 2010. Representasi Graph dan Beberapa Graph Khusus. In Pengantar Teori Graph. Ch. 2.

[8] Tarjan, R.E., 1972. Depth-First Search and Linear Graph Algorithms. Siam Journal on Computing - SIAMCOMP, 1, pp.146-60.

[9] X., Y., Philip S., Y. & J., H., 2004. Graph Indexing: A Frequent Structurebased Approach. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data., 2004. ACM.

[10] Yuan, D. & Mitra, P., 2011. A Lattice-based Graph Index for Subgraph Search. In WebDB., 2011. WebDB.

[11] Yuan, D. & Mitra, P., 2013. A Lattice-based Graph Index for Graph Database. The VLDB Journal 22 (2), pp.229-52.

[12] Zhang, S., Hu, M. & Yang, J., 2007. Treepi: A novel graph indexing method. 2007 IEEE 23rd International Conference on Data Engineering, pp.966-75.