TUGAS AKHIR – KI141502

OPTIMASI SPARQL QUERY MENGGUNAKAN GRAPH DATABASE DENGAN MODEL LABELED PROPERTY GRAPH

NAFIAR RAHMANSYAH
NRP 05111440000109

Dosen Pembimbing I
Nurul Fajrin Ariyani, S.Kom.,M.Sc.

Dosen Pembimbing II
Abdul Munif, S.Kom.,M.Sc.

DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya 2018

TUGAS AKHIR – KI141502

OPTIMASI SPARQL QUERY MENGGUNAKAN GRAPH DATABASE DENGAN MODEL LABELED PROPERTY GRAPH

NAFIAR RAHMANSYAH
NRP 05111440000109

Dosen Pembimbing I
Nurul Fajrin Ariyani, S.Kom.,M.Sc.

Dosen Pembimbing II
Abdul Munif, S.Kom.,M.Sc.

DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya 2018

[Halaman ini sengaja dikosongkan]

UNDERGRADUATE THESES – KI141502

OPTIMASI SPARQL QUERY MENGGUNAKAN GRAPH DATABASE DENGAN MODEL LABELED PROPERTY GRAPH

NAFIAR RAHMANSYAH
NRP 05111440000109

Supervisor I
Nurul Fajrin Ariyani, S.Kom.,M.Sc.

Supervisor II
Abdul Munif, S.Kom.,M.Sc.

DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHNOLOGY
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA 2018

[Halaman ini sengaja dikosongkan]

LEMBAR PENGESAHAN

OPTIMASI SPARQL QUERY MENGGUNAKAN GRAPH DATABASE DENGAN MODEL LABELED PROPERTY GRAPH

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Manajemen Informasi
Program Studi S-1 DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember

Oleh
NAFIAR RAHMANSYAH
NRP : 05111440000109

Disetujui oleh Dosen Pembimbing Tugas Akhir:

1. Nurul Fajrin Ariyani, S.Kom.,M.Sc. ..... (Pembimbing 1)
   NIP: 198607222015042003
2. Abdul Munif, S.Kom.,M.Sc. ..... (Pembimbing 2)
   NIP: 198608232015041004

SURABAYA
JULI, 2018

[Halaman ini sengaja dikosongkan]

OPTIMASI SPARQL QUERY MENGGUNAKAN GRAPH DATABASE DENGAN MODEL LABELED PROPERTY GRAPH

Nama Mahasiswa : NAFIAR RAHMANSYAH
NRP : 05111440000109
Departemen : Informatika FTIK-ITS
Dosen Pembimbing 1 : Nurul Fajrin Ariyani, S.Kom.,M.Sc.
Dosen Pembimbing 2 : Abdul Munif, S.Kom.,M.Sc.

Abstrak

Web semantik menyediakan kerangka kerja umum yang memungkinkan datanya dibagikan dan digunakan ulang secara lintas aplikasi. Model data RDF sendiri sudah digunakan untuk berbagai macam aplikasi web semantik yang berguna untuk mesin pencarian publik, rekayasa pengetahuan, penyimpanan data hasil penelitian, dan proses-proses bisnis aplikasi lainnya. Karena data yang disimpan sangat penting dan dituntut ketahanannya, data RDF yang ada di internet saat ini ukurannya sudah sangat besar dan akan semakin membesar. Hal ini menyebabkan proses query data pada file RDF memakan waktu yang cukup lama.

Pada tugas akhir ini, permasalahan tersebut akan ditangani dengan mengusulkan metode optimasi SPARQL query dengan menggunakan database graf yang menerapkan model labeled property graph. Berdasarkan model data RDF yang sudah ada, labeled property graph dapat mengurangi jumlah node yang dihasilkan dari file RDF. Oleh karena itu metode ini diharapkan dapat meningkatkan kecepatan dalam melakukan proses traverse pada data graf. Pada tugas akhir ini, penulis akan membandingkan performa model Labeled

Property Graph dengan Triple Store dalam menghadapi SPARQL query.

Pengujian yang dilakukan menunjukkan bahwa metode ini dapat memberikan hasil running time yang lebih singkat dengan running time pada model Labeled Property Praph yang jauh lebih cepat jika dibandingkan dengan model Triple Store saat menghadapi SPARQL query.

Kata kunci: Web Semantik, Optimasi SPARQL Query, Label Property Graph, Basis data Graf.

SPARQL QUERY OPTIMIZATION USING GRAPH DATABASE WITH LABELED PROPERTY GRAPH MODEL

Student's Name	**: NAFIAR RAHMANSYAH**
Student's ID	**: 05111440000109**
Department	**: Informatika FTIK-ITS**
First Advisor	**: Nurul Fajrin Ariyani, S.Kom.,M.Sc.**
Second Advisor	**: Abdul Munif, S.Kom.,M.Sc.**

Abstract

The Semantik Web provides a common framework that allows data to be shared and reused across applications. The RDF data model itself already in use for variety semantik web applications which is useful to public search engines, is also used for knowledge management, and another business process. Because of the data stored is very important and demanded endurance, the RDF data that exists on internet today is already very large and will be larger.This causes the process time of querying dataset in RDF files become a considerable issue.

In this research, mentioned issues will be handled by proposing SPARQL query optimization method using graph database with Labeled Property Graph model. Based on the existing RDF data model, the Labeled Property Graph model can reduce the number of nodes generated from RDF file. Therefore this method is expected to increase the speed in conducting traverse process on graph data. In this research, I will compare the performace of Labeled Property Graph model with Triple Store model in facing certain SPARQL query.

Test conducted show that this method can provide faster running time with Labeled Property Graph model that is faster when compared to Triple Store model in facing certain SPARQL query.

Keywords: Modular Software, Academic Information System

KATA PENGANTAR

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

Alhamdulillahirabbil'alamin, segala puji bagi Allah SWT, yang atas izin dan karunianya penulis dapat menyelesaikan Tugas Akhir yang berjudul **"OPTIMASI SPARQL QUERY MENGGUNAKAN GRAPH DATABASE DENGAN MODEL LABELED PROPERTY GRAPH".**

Dalam pengerjaan tugas akhir ini, penulis mendapatkan banyak sekali ilmu baru dan memperdalam ilmu-ilmu yang sebelumnya telah diajarkan selama masa perkuliahan di Teknik Informatika ITS.

Terselesaikannya Tugas Akhir ini tidak lepas dari bantuan dan dukungan beberapa pihak. Sehingga pada kesempatan ini penulis mengucapkan syukur dan terima kasih kepada:

1. Allah SWT dan Nabi Muhammad SAW. Karena atas berkah dan karunianya penulis dapat menyelesaikan tugas akhir dengan lancar.
2. Ibu Nurul Fajrin Ariyani, S.Kom.,M.Sc. dan Bapak Abdul Munif, S.Kom.,M.Sc. selaku dosen pembimbing penulis yang telah memberikan ide, nasihat dan arahan sehingga penulis dapat menyelesaikan tugas akhir dengan tepat waktu.
3. Orang tua penulis, Ayah, dan Ibu yang telah memberikan dukungan moral, spiritual dan material serta senantiasa memberikan doa demi kelancaran dan kemudahan penulis dalam mengerjakan tugas akhir.
4. Saudara kembar penulis, Rafiar yang selalu mengingatkan penulis dalam kesalahan dan kekurangan pengerjaan tugas akhir serta adik penulis, dan keluarga besar yang telah memberikan dukungan secara langsung maupun tidak langsung.

5. Teman-teman seperjuangan tugas akhir di laboratorium Pemrograman: Rina, Sekbay, Hari, Afiif, Nobby, dan Dita yang telah sama-sama berjuang dengan saling memberi masukkan dan saling mengingatkan dalam pengerjaan tugas akhir.
6. Adik-adik admin dan penghuni laboratorium Pemrograman yang telah menemani penulis selama pengerjaan tugas akhir.
7. Pihak-pihak lain yang tidak bisa penulis sebutkan satu-persatu.

Penulis menyadari masih ada kekurangan dalam penyusunan tugas akhir ini. Penulis memohon maaf atas keasalahan, kelalaian, maupun kekurangan dalam penyusunan tugas akhir ini. Kritik dan saran yang membangun dapat disampaikan sebagai bahan perbaikan kedepannya

Surabaya, Juli 2018

Penulis

DAFTAR ISI

[Halaman ini sengaja dikosongkan]

DAFTAR GAMBAR

DAFTAR TABEL

[Halaman ini sengaja dikosongkan]

DAFTAR KODE SUMBER

BAB I
PENDAHULUAN

1.1 Latar Belakang

Web semantik menyediakan kerangka kerja umum yang memungkinkan datanya dibagikan dan digunakan ulang secara lintas aplikasi. Kerangka kerja tersebut berupa model data RDF, model data RDF sendiri sudah digunakan untuk berbagai macam aplikasi semantik *web*. Contohnya YAGO2 dan DBpedia yang mengekstrak data-data pada Wikipedia dan menyimpannya dalam format RDF agar data yang telah disimpan dapat didapatkan kembali dengan menggunakan *query* yang terstruktur. Selain itu *web* semantik juga berguna untuk mesin pencarian publik, rekayasa pengetahuan, penyimpanan data hasil penelitian, dan proses-proses bisnis aplikasi lainnya. Karena data yang disimpan sangat penting dan dituntut ketahanannya, data RDF yang ada di internet saat ini ukurannya sudah sangat besar dan akan semakin membesar.

Secara umumnya, data RDF merupakan kumpulan *metadata* terhadap suatu hal yang dapat direpresentasikan sebagai kumpulan *triples* yang dilambangkan sebagai SPO (*Subject Predicate Object*). Data yang ada pada sebuah RDF ditulis menggunakan sintaks *encoding* XML. Sebuah *resource* yang terdapat dalam RDF secara umum berupa apapun yang dinamai dengan *URI*. Bentuk data berupa link *URI* ini bertujuan agar data RDF dapat mendukung fungsi dari aplikasi *web* semantik yang dapat mengekspose dan berbagi data terhadap aplikasi *web* semantik lainnya. Namun kelebihan ini justru memberikan kekurangan tersendiri terhadap model data RDF. Merepresentasikan beberapa meta data menjadi sebuah string panjang bukanlah masalah yang besar, namun ketika data yang harus ditampung oleh sebuah RDF mencapai jutaan *record* data, hal ini menjadi kekurangan yang menyakitkan bagi para *developer*.

Dalam *web* semantik SPARQL merupakan Bahasa standar yang digunakan untuk melakukan *query* data *graph* yang direpresentasikan dalam bentuk *triple* RDF. Walaupun cara memanajemen data RDF sudah dipelajari dalam satu dekade terakhir, kebanyakan solusi yang sudah ada tidak dapat menghadapi data RDF dalam skala yang besar dan tidak dapat menjawab SPARQL *query* yang kompleks secara efisien. Untuk menghadapi masalah tersebut terdapat beberapa Teknik yang sudah diusulkan sebelumnya, contohnya *vertical partitioning* [1], *multiple indexing* [2], *sextuple indexing* [3], *graph partitioning* [4]. Walaupun begitu *query* engine yang sudah ada tersebut seperti SW-*Store* [5], Hexa*Store* [3], RDF -3x [2] memiliki batasan seperti berikut : (1) sistem-sistem tersebut tidak dapat menghadapi SPARQL dengan *wildcard query* (2) pada beberapa sistem yang ada sangat sulit untuk menangani *update* pada sebuah repositori RDF, sehingga memaksa sistem tersebut untuk memproses *dataset* ulang ketika ada *update* data.

Hasil Tugas Akhir ini diharapkan dapat memberikan metode yang lebih baik dalam *indexing* pada basis data graf sehingga mampu menghadapi permasalahan di atas dengan optimal dan dapat memberikan masukan dalam perkembangan *web* semantik.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam tugas akhir ini dapat dipaparkan sebagai berikut:

1. Bagaimana cara membuat basis data untuk menyimpan data graf yang didapatkan dari *file* RDF?
2. Bagaimana cara membuat *indexing* pada basis data graf untuk mengoptimalkan performa *update* dan *query* ?
3. Bagaimana cara memetakan model data RDF menjadi model data *Labeled Property Graph?*
4. Bagaimana cara membuat basis data yang dapat menjawab SPARQL dengan *wildcard query*?

1.3 Batasan Masalah

Permasalan yang dibahas dalam tugas akhir ini memiliki beberapa batasan, di antaranya sebagai berikut:

1. Membandingkan basis data yang sudah dibuat menggunakan Neo4j dengan Apache Jena Fuseki.
2. Percobaan yang dilakukan berupa membandingkan *running time* dalam waktu satuan detik.
3. *Dataset* yang digunakan adalah *foodfacts open data*.
4. Percobaan yang dilakukan menggunakan beberapa SPARQL *query* yang dapat ditangani oleh basis data yang sudah dibuat.

1.4 Tujuan

Tujuan dari pengerjaan tugas akhir ini adalah sebagai berikut:

1. Mengetahui cara membuat basis data untuk menyimpan data *graph* yang didapatkan dari *file* RDF.
2. Mengetahui cara membuat basis data yang dapat menjawab SPARQL dengan *wildcard query*.
3. Mengetahui cara membuat basis data yang menerapkan *indexing* sehingga dapat menangani *update* baru pada sebuah *dataset*.

1.5 Manfaat

Hasil pengerjaan tugas akhir ini diharapkan dapat memberikan manfaat:

1. Dalam pengembangan *website* semantik, sebagai masukkan dari metode yang dapat digunakan untuk penyimpanan data *file* RDF yang besar.

2. Dalam pengembangan aplikasi yang memanfaatkan teknologi Neo4j, sebagai masukan dalam membangun struktur *query* cypher untuk *traverse* data graf.
3. Dalam pemrosesan SPARQL *query*, sebagai metode yang dapat digunakan dalam menangani *wildcard query* terhadap *dataset* RDF yang besar.

1.6 Metodologi

Pada tahap ini akan dijelaskan secara singkat tahapan-tahapan yang akan dilakukan dalam proses pengerjaan tugas akhir ini.

1.6.1 Studi literatur

Pada tahap ini dilakukan pengumpulan data dan studi terhadap sejumlah referensi yang diperlukan dalam pengerjaan tugas akhir. Referensi tersebut didapatkan dari beberapa jurnal yang dipublikasikan. Selain dari jurnal, studi literatur juga dilakukan melalui pencarian referensi dari internet yang membahas mengenai informasi yang dibutuhkan, seperti optimasi pemrosesan SPARQL *query*, metode untuk memasukkan data RDF kedalam basis data graf, struktur Bahasa *query* pada basis data graf.

1.6.2 Analisis dan perancangan

Adapun pembagian tahap analisa kebutuhan dan perancangan dari sistem yang akan dibuat sebagai berikut:

1. Menganalisa permasalahan yang ada dan memberikan solusi dari permasalahan tersebut.
2. Mengusulkan rancangan sistem yang akan dibuat berdasarkan solusi yang telah didapatkanb dari analisis permasalahan.

1.6.3 Implementasi

Sistem yang akan dibangun berupa *database* yang dibuat menggunakan Neo4j, dengan menggunakan *plugin* Neosemantiks dalam proses memasukkan data RDF kedalam *database*. Selain itu *database* pada Apache Jena Fuseki juga disiapkan sebagai pembanding sistem yang telah dibuat. Selain menyiapkan *database* diperlukan aplikasi Java yang digunakan untuk mengubah SPARQL *query* menjadi bentuk Cypher *query*.

1.6.4 Pengujian dan evaluasi

Pada tahap ini dilakukan pengujian terhadap sistem yang telah dibuat. Pengujian dan evaluasi akan dilakukan dengan melihat performa *running time* sistem yang sudah dibuat dalam satuan waktu. Tahap ini dimaksudkan juga untuk mengevaluasi jalannya sistem, mencari masalah yang mungkin timbul dan mengadakan perbaikan jika terdapat kesalahan. Skenario pengujian menggunakan beberapa *dataset* RDF dengan jumlah data yang berbeda-beda dan beberapa jenis *query* pada setiap *dataset* kemudian dihitung *running time* untuk mendapatkan hasil *query*. Setelah itu dilakukan skenario yang sama terhadap sistem lainnya yaitu Apache Jena Fuseki. Hasil akhir berupa *running time* dibandingkan antara sistem yang sudah dibuat dan Apache Jena Fuseki.

1.6.5 Penyusunan buku tugas akhir

Pada tahap ini dilakukan penyusunan laporan yang menjelaskan dasar teori dan metode yang digunakan dalam tugas akhir ini serta hasil dari implementasi aplikasi perangkat lunak yang telah dibuat. Sistematika penulisan buku tugas akhir secara garis besar antara lain:

1. Pendahuluan
    a. Latar Belakang
    b. Rumusan Masalah
    c. Batasan Tugas Akhir
    d. Tujuan
    e. manfaat
    f. Metodologi
    g. Sistematika Penulisan
2. Tinjauan Pustaka
3. Analisis dan Perancangan
4. Implementasi
5. Pengujian dan Evaluasi
6. Kesimpulan dan Saran
7. Daftar Pustaka

1.7 Sistematika Penulisan Laporan Tugas Akhir

Buku tugas akhir ini bertujuan untuk mendapatkan gambaran dari pengerjaan penelitian ini. Selain itu, diharapkan dapat berguna untuk pembaca yang tertarik untuk melakukan pengembangan lebih lanjut. Secara garis besar, buku tugas akhir terdiri atas beberapa bagian seperti berikut ini:

Bab I Pendahuluan

Bab yang berisi mengenai latar belakang, tujuan, dan manfaat dari pembuatan tugas akhir. Selain itu permasalahan, batasan masalah, metodologi yang digunakan, dan sistematika penulisan juga merupakan bagian dari bab ini.

Bab II Tinjauan Pustaka

Bab ini berisi penjelasan secara detail mengenai dasar-dasar penunjang dan teori-teori yang digunakan untuk mendukung pembuatan tugas akhir ini.

Bab III Analisis dan Perancangan

Bab ini berisi tentang analisis dan perancangan desain sistem yang akan digunakan untuk optimasi SPARQL *query*.

Bab IV Implementasi

Bab ini membahas implementasi dari desain yang telah dibuat pada bab sebelumnya. Penjelasan berupa kode yang digunakan untuk proses implementasi.

Bab V Uji Coba dan Evaluasi

Bab ini membahas tahap-tahap uji coba, kemudian hasil uji coba dievaluasi untuk kinerja dari sistem yang dibangun.

Bab VI Kesimpulan dan Saran

Bab ini merupakan bab terakhir yang menyampaikan kesimpulan dari hasil uji coba yang dilakukan dan saran untuk pengembangan perangkat lunak ke depannya.

[Halaman ini sengaja dikosongkan]

BAB II
TINJAUAN PUSTAKA

Bab ini berisi tinjauan pustaka yang terdiri dari dasar teori yang digunakan dalam mengerjakan penelitian ini, dan penelitian sebelumnya yang menjadi referensi dalam mengerjakan Tugas Akhir ini.

2.1 Penelitian Terkait

Karena semakin berkembangnya basis data graf, bentuk strukur penyimpanan data pada basis data graf yang sudah ada saat ini semakin beragam. Karena menggunakan *file* RDF dalam penyimpanan datanya, dalam perkembangan *web* semantik hanya basis data dengen model *Triple Store* yang dapat menyimpan data dari *file* RDF. Saat ini terdapat model lain dalam penyimpanan data graf yaitu dengan menggunakan model *Labeled Property Graph* dimana penyimpanan data graf nya lebih natural berdasarkan bentuk graf yang sesungguhnya. Pada artikel yang dituliskan oleh Jesus Barrasa, bulan Juni 2017 ini pada [6], mengajukan metode untuk *import file* RDF *agar* dapat disimpan pada basis data graf Neo4j yang menerapkan model *Labeled Property Graph.* Metode yang diajukan berupa pemetaan aturan-aturan yang akan diterapkan pada data *triples* agar dapat diubah menjadi bentuk data pada Neo4j.

Berdasarkan kondisi-kondisi *triples* pada *file* RDF, untuk memetakan data RDF akan ditetapkan 4 aturan agar data yang ada bisa diubah menjadi model data *Labeled Property Graph* Neo4j dengan bentuk yang natural. Berikut adalah aturan-aturan yang diajukan oleh Jesus Barrasa pada [6],

1. *Subject* dari *triples* dipetakan menjadi sebuah *node* pada Neo4j. sebuah *node* pada Neo4j merepresentasikan sebuah *resource* RDF yang akan dilabelkan :*Resource* dan memilki properti "*uri*" dengan nilai berupa *URI* dari *resource* tersebut.

2. Predikat dari *triples* dipetakan ke properti *node* di Neo4j jika *object* dari *triple* adalah *literal* .
3. *Predicates* dari *triples* dipetakan menjadi *relationships* dalam Neo4j bila *object* pada *triples* adalah sebuah *resource*.
4. *Statements* dengan rdf:type dipetakan menjadi kategori dalam Neo4j.

Artikel ini diimplementasikan dalam bentuk *plug-in* pada basis data Neo4j yang dapat digunakan dengan menjalankan Cypher *query* pada Neo4j *command line*. Contoh Cypher *query* untuk menjalankan *import* data menggunakan neosemantiks dapat dilihat pada Kode Sumber 2.7.

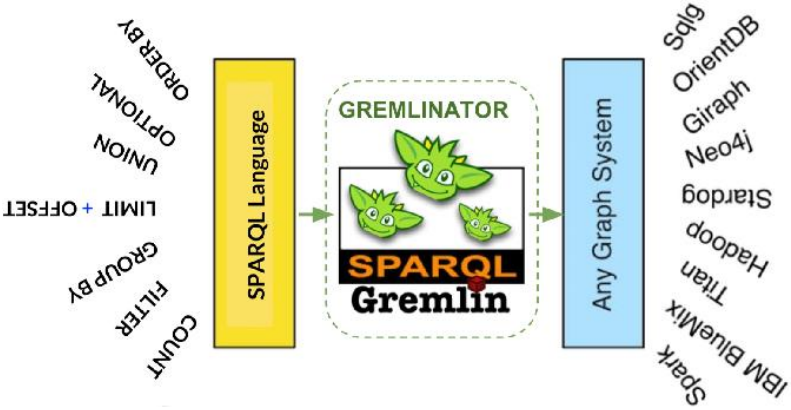
```
1.    CALL semantiks.importRDF("file:///path/to/some-
      file.ttl", "Turtle", { shortenUrls: true })
```

Kode Sumber 2.1 Contoh sintaks *import* data menggunakan neosemantiks.

Kelebihan yang ada dari metode ini dalam optimasi SPARQL *query* antara lain, menawarkan strktur baru untuk menyimpan data *triples*, dan sebagai metode mapping berupa aturan-aturan untuk mengubah struktur *triples* menjadi *Labeled Property Graph*. Namun dalam hal optimasi *query* masih ada kekurangan dari metode yang diajukan yaitu, belum adanya metode untuk membentuk *query* pada struktur data baru yang sudah diajukan, dan belum adanya *benchmarking* performa basis data yang baru dibuat dengan basis data graf yang mengikuti model data *Triple Store*.

Metode untuk merubah SPARQL *query* menjadi bentuk *query* lainnya diajukan pada [7]. Pada penelitian [7] didasari dari permasalahan yang ada pada perkembangan basis data graf. Graf pengetahuan yang populer pada satu dekade terakhir sangat bergantung pada RDF dengan model *Triple Store* atau basis data graf dengan model *Property Graph* (PG). Namun Bahasa *query* yang digunakan pada kedua model data ini berbeda, dengan SPARQL untuk model *Triple Store* dan Bahasa *query* Gremlin

untuk contoh model data *Property Graph*, keduanya tidak saling interoperabilitas. Pada penelitian [7] mengajukan penerjemah Bahasa SPARQL *query* untuk dipetakan menjadi Bahasa Gremlin *query* dengan menggunakan Gremlinator. Metode ini mengusulkan ide untuk membuat penerjemah Bahasa *query* berdasarkan bentuk *Graph Pattern* pada SPARQL untuk dirubah menjadi bentuk *traverse* graf pada Bahasa *query* Gremlin. Pada Gambar 2.1 merupakan alur Gremlinator menerima *input* berupa SPARQL *query* yang akan diubah menjadi bentuk *query* untuk basis data graf lainnya. [8]



Gambar 2.1 Alur penerjemahan pada Gremlinator berdasarkan [8]

Kelebihan yang ada dari metode ini dalam optimasi SPARQL *query* antara lain, merupakan metode untuk merubah SPARQL *query* menjadi Bahasa *query traverse* Gremlin, metode ini merubah bentuk *query* dengan membagi 2 jenis bentuk SPARQL *query* menjadi *Basic Graph Pattern* dan *Complex Graph Pattern*. Namun dalam optimasi *query* pada penelitian yang akan dilakukan, meotode ini masih memiliki kekurangan yaitu, metode yang diajukan tidak dapat merubah SPARQL *query* menjadi Bahasa Cypher *query* milik Neo4j, metode ini juga

belum dapat menjawab SPARQL *query* yang menggunakan variabel pada bagian *predicate* nya.

Dari metode yang sudah diajukan pada [6], yang mengajukan metode *import* data dari *file* RDF menjadi bentuk *Labeled Property Graph* milik Neo4j, dan pada [7], yang mengajukan metode untuk mengubah bentuk SPARQL *query* menjadi Gremlin *query* akan dijadikan dasar dalam pengerjaan penelitian pada buku ini. Pada penelitian yang akan dilakukan, penulis akan menggunakan kelebihan dari metode yang diajukan pada kedua penelitian tersebut dan menjawab kekurangan dari kedua metode tersebut. Pada [6], selanjutnya disebut penelitian 1, penulis akan menggunakan metode *import file* RDF untuk merubah struktur data *file* RDF menjadi bentuk *Labeled Property Graph* dan mengajukan metode *query* yang belum diajukan pada penelitian 1 dengan cara mengubah SPARQL *query* menjadi bentuk Cypher *query* pada Neo4j. Pada [7], selanjutnya akan disebut penelitian 2, penulis akan menggunakan masukan dari metode mengubah SPARQL *query* menjadi Gremlin *query* untuk membuat metode convert SPARQL *query* menjadi Cypher *query*, selain itu pada penelitian 2 tidak dapat menjawab permasalahan pada penelitian 1 karena struktur data yang digunakan adalah *Property Graph* bukan *Labeled Property Graph*. Dari hipotesa berikut diringkas dalam Tabel 2.1.

Tabel 2.1 Perbandingan terhadap metode dalam optimasi SPARQL *query*

Perbandingan	Penelitian 1	Penelitian 2	Penelitian yang akan dilakukan
Merubah struktur data penyimpanan menjadi *Labeled Property Graph*	Dapat melakukan dengan menerapkan aturan-aturan	Data baru yang dibuat bukan dengan bentuk *Labeled Proerty Graph* melainkan dengan *Property Graph*	Tidak mengajukan metode baru, namun menggunakan metode pada penelitian 1
Metode query dari data baru yang sudah dibuat	Tidak mengajukan metode *query* dari data baru yang sudah dibuat	Detode *query* data baru dengan mengubah bentuk SPARQL *query* menjadi Gremlin *query*	Metode *query* data baru dengan mengubah bentuk SPARQL *query* menjadi Cypher *query*
Metode *convert* SPARQL *query* menjadi Cypher *query*	Tidak mengajukan metode *convert* query	Merubah bentuk SPARQL *query* berdasarkan *Graph Pattern* , namun diperuntukan untuk Bahasa Gremlin *query*	Merubah bentuk SPARQL *query* berdasarkan kondisi SPARQL *query* , variabel, dan *return value* menjadi Cypher *query*

2.2 *Web* Semantik

Web semantik adalah sebuah visi, ide atau pemikiran dari bagaimana memiliki data pada *web* yang didefinisikan dan dihubungkan dengan suatu cara dimana dapat digunakan oleh mesin tidak hanya untuk tujuan *display*, tetapi untuk otomatisasi, integrasi dan penggunaan kembali data diantara berbagai aplikasi [9]. *Web* semantik menyediakan kerangka kerja umum (berbasis RDF format) yang memungkinkan datanya dibagikan dan digunakan ulang secara lintas aplikasi.

Web semantik adalah sebuah *web* yang menyediakan data, seperti layaknya sebuah basis data global. Pendekatan

Semantik *web* mengembangkan bahasa untuk mengekspresikan informasi dalam bentuk yang dapat diproses oleh mesin (*machine processable*). Ide dasarnya adalah untuk membuat *web* agar memiliki definisi dan link data sehingga dapat digunakan lebih efektif untuk mencari, otomasi, integrasi dan *re-use* informasi pada berbagai aplikasi.

2.3 RDF

RDF merupakan singkatan dari *Resource Description Framework*. RDF adalah dasar yang digunakan untuk pemrosesan *metadata* yang dapat menyediakan interoperabilitas antar aplikasi dengan saling bertukar informasi yang *machine-understandable* pada jaringan *website*. RDF menekankan fasilitas untuk memungkinkan pemrosesan sumber data *Web* secara otomatis. *Metadata* RDF sendiri dapat digunakan dalam berbagai bidang aplikasi; misalnya: dalam penemuan sumber daya untuk menyediakan kemampuan mesin pencarian yang lebih baik; kemampuan sebagai kamus untuk menggambarkan konten dan hubungan konten yang tersedia di situs *web*, sebuah halaman *web*, atau perpustakaan digital tertentu; oleh agen perangkat lunak cerdas untuk memfasilitasi berbagi pengetahuan dan pertukaran; dalam rating konten; dalam menggambarkan koleksi halaman yang mewakili "dokumen" logis tunggal; untuk menggambarkan hak kekayaan intelektual halaman *Web*, dan banyak lainnya. RDF dengan tanda tangan digital akan menjadi kunci untuk membangun "*Web* of Trust" untuk perdagangan elektronik, kolaborasi, dan aplikasi lainnya.

Metadata adalah "data terkait sebuat data" atau secara khusus dalam konteks data RDF "yang menggambarkan *resource web*." Perbedaan antara "data" dan "*metadata*" bukanlah hal yang pasti, hal itu adalah perbedaan yang dibuat terutama oleh aplikasi tertentu. Seringkali kali sumber daya yang sama akan ditafsirkan dalam dua cara secara bersamaan. RDF mendorong pandangan ini dengan menggunakan XML sebagai sintaks pengkodean untuk *metadata*. Sumber daya yang dijelaskan oleh RDF adalah, secara

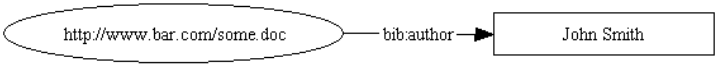
umum, apa pun yang dapat dinamai melalui *URI*. Tujuan luas RDF adalah untuk mendefinisikan cara untuk mendeskripsikan *resource* dengan tidak membuat asumsi tentang domain aplikasi tertentu, atau mendefinisikan hubungan terhadap domain aplikasi apa pun. Definisi mekanisme harus netral domain, namun mekanisme tersebut harus sesuai untuk menggambarkan informasi tentang domain apa pun [10].

Pada Kode Sumber 2.1 merupakan potongan kode XML sederhana pada sebuah RDF

```
1.  <?namespace href=http://docs.r.us.com/bibliography-
    infoas="bib"?>
2.  <?namespace href="http://www.w3.org/schemas/rdfsche
    ma"as="RDF"?>
3.  <RDF:serialization >
4.     <RDF:assertions href="http://www.bar.com/some.do
       c">
5.        <bib:author>John Smith</bib:author>
6.     </RDF:assertions>
7.  </RDF:serialization>
```

Kode Sumber 2.2 Berkas XML sederhana

Dari potongan kode XML pada Kode Sumber 2.1 dapat dipetakan menjadi data *graph* yang dapat digambarkan seperti pada Gambar 2.2.



Gambar 2.2 Bentuk data graf RDF

2.4 SPARQL

Sama seperti SQL yang menyediakan bahasa *query* yang (relatif) standar di seluruh sistem *database* relasional, SPARQL menyediakan bahasa *query* yang standar untuk data graf pada RDF. SPARQL *query* berupaya mencocokkan *pattern* dalam graf

dan mengikat variabel *query* ketika menemukan solusi yang *match*.

SPARQL menyediakan empat bentuk *query*: *SELECT*, CONSTRUCT, ASK, dan DESCRIBE. Semuanya digunakan untuk menentukan menampilkan data berdasarkan *traverse pattern* pada graf, dan semua bentuk *query* memiliki struktur *traverse pattern* yang sama. Walaupun SPARQL adalah rekomendasi W3C, banyak platform semantik hanya mendukung beberapa bentuk SPARQL *query* - namun, semantik platform apa pun yang layak dipertimbangkan setidaknya harus mendukung *SELECT* dari bentuk SPARQL. [11]

Pada Kode Sumber 2.2 merupakan potongan kode dari SPARQL *query*.

```
1.  PREFIX food: <http://data.lirmm.fr/ontologies/food#>
2.  select ?s where {
3.      ?s food:name "Almondmilk".
4.  }
```

Kode Sumber 2.3 Contoh SPARQL *query*.

2.5 Cypher

Cypher adalah bahasa *query* basis data graf yang ekspresif . Meskipun saat ini khusus untuk Neo4j, pendekatannya yang mirip dengan kebiasaan kita dalam merepresentasikan graf sebagai diagram membuatnya ideal untuk menggambarkan graf secara program. Cypher dirancang untuk mudah dibaca dan dipahami oleh pengembang, profesional basis data, dan pemangku kepentingan bisnis. Kemudahan penggunaannya berasal dari fakta bahwa cypher sesuai dengan cara kita secara intuitif menggambarkan graf menggunakan diagram. Cypher memungkinkan pengguna (atau aplikasi yang bertindak atas nama pengguna) untuk meminta *database* untuk menemukan data yang

cocok dengan pola tertentu. Contoh Cypher *query* dapat dilihat pada Kode Sumber 2.3.

```
1.  MATCH (you:Person {name:"You"})-[:KNOWS]-
    (:Database {name: "Neo4j"})
2.  RETURN you
```

Kode Sumber 2.4 Contoh Cypher *query*.

2.6 SPARQL *Query* Dengan *Wildcard*

Dalam aplikasi nyata, sangat tidak praktis apabila pengembang aplikasi harus mengetahui secara pasti *object* yang ingin didapatkan. Oleh karena itu, hal ini tidak memungkinkan untuk menentukan kriteria *query* secara tidak spesifik. Sebagai contohnya kita mungkin tahu bahwa seorang politisi penting lahir pada 12 Februari dan meninggal pada 15 April, namun kita tidak mengetahui kelahiran dan kematiannya secara detil. Dengan kasus ini kita harus melakukan *query* dengan *wildcard*. [12]

```
2.  select ?name where {
3.     ?m <hasName> ?name .
4.     ?m <BornOnDate> “02-12*” .
5.     ?m <DiedOnDate> “04-15*” .
6.  }
```

Kode Sumber 2.5 Contoh SPARQL *query* dengan *wildcard*

Meskipun ada teknik untuk mendukung SPARQL *query* dengan *wildcard* dan cara untuk mengelola kumpulan data RDF yang besar, tidak ada teknik untuk mendukung keduanya, yaitu kemampuan untuk melakukan SPARQL *query* dengan *wildcard* dengan cara yang terukur. Sistem penyimpanan RDF yang ada, seperti Jena, Yars2, dan Sesame 2.0, tidak dapat bekerja dengan baik dalam *dataset* RDF yang besar (seperti *dataset* Yago). SW-*Store*, RDF-3x, dan Hexa*Store* dirancang untuk mengatasi skalabilitas, namun mereka hanya dapat mendukung SPARQL

query yang tepat, karena semuanya menggantikan semua *literal* (dalam *triples* RDF) dengan id menggunakan kamus pemetaan.

2.7 SPARQL *Query* Dengan *FILTER regex*

SPARQL menyediakan fungsi *regex* dalam sintaks *FILTER* yang digunakan untuk memeriksa hasil string menggunakan *regular expression*. Fungsi ini memiliki kemampuan pemeriksaan hasil yang mirip seperti pemeriksaan hasil menggunakan sintaks LIKE pada SQL, walaupun sintaks dari *regular expression* nya berbeda. Dalam melakukan pemeriksaan hasil string, sintaks ini membutuhkan hasil *query* dari kondisi sebelumnya untuk diperiksa menggunakan *regular expression* sehingga penggunaan nya tidak melalui pemeriksaan *index*. Sebagai contoh seperti yang dapat dilihat pada Kode Sumber 2.4, bila mencari kemunculan dari string "printer" dari rdf:label pada kumpulan ex:Product, maka proses pencarian hasil dilakukan dengan memeriksa seluruh label yang terpilih dan membandingkan dengan kondisi *regular expression* nya.

```
1.    PREFIX ex: < http: //www.example.org/resources#>
2.    PREFIX rdfs: < http: //www.w3.org/2000/01/rdf-
      schema#>
3.
4.    SELECT ?s ?lbl WHERE {
5.       ?s a ex: Product;
6.            rdfs:label ?lbl
7.       FILTER regex( ? lbl, "printer", "i")
8.    }
```

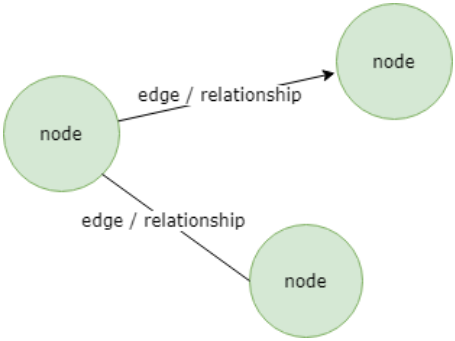
Kode Sumber 2.6 Contoh SPARQL *query* dengan *FILER regex*.

2.8 *Graph Database*

Sebuah *graph database management system* (selanjutnya, basis data graf) adalah sistem manajemen basis data online dengan metode *Create*, *Read*, *Update*, dan *Delete* (CRUD) yang

mengekspos model data graf. Basis data graf umumnya dibuat untuk digunakan dengan sistem transaksional (OLTP). Dengan demikian, mereka biasanya dioptimalkan untuk kinerja transaksional, dan direkayasa dengan integritas transaksional dan ketersediaan operasional dalam pikiran.

Pada umumnya basis data graf adalah basis data yang datanya dapat direpresentasikan kedalam bentuk graf. Terlepas dari bentuk data yang disimpan, data pada basis data graf memiliki satu komponen penting berupa *relationship* atau *edge* yang dapat menghubungkan sebuah data kepada data yang lainnya atau *node* pada *node* lainnya. Seperti yang digambarkan pada Gambar 2.3



Gambar 2.3 Bentuk data graf pada basis data graf

Ada dua properti dari basis data graf yang harus kita pertimbangkan ketika menyelidiki teknologi basis data graf:

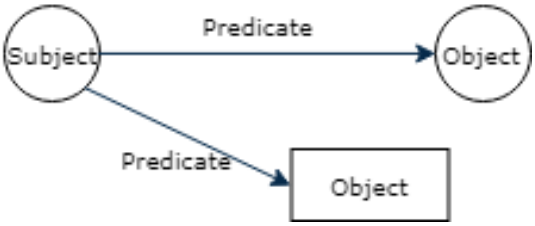
Berdasarkan penyimpanan yang mendasarinya. Beberapa basis data graf menggunakan penyimpanan graf secara *native* yang dioptimalkan dan dirancang untuk menyimpan dan mengelola graf. Namun tidak semua teknologi basis data graf menggunakan penyimpanan graf *native*. Beberapa meyimpan data graf ke dalam konsep basis data relasional, basis data berorientasi objek, atau beberapa penyimpanan data dengan tujuan umum lainnya.

Berdasarkan mesin pengolah. Beberapa definisi menyatakan bahwa basis data graf seharusnya menggunakan *index-free adjacency*, yang berarti bahwa *node* yang terhubung secara fisik "menunjuk" satu sama lain dalam basis data. Di sini kita mengambil pandangan yang sedikit lebih luas: setiap basis data yang dari perspektif pengguna berperilaku seperti basis data graf (yaitu, memaparkan model data graf melalui operasi CRUD) memenuhi syarat sebagai basis data graf. Telah diakui keuntungan performa yang signifikan dari *index-free adjacency*, dan oleh karena itu menggunakan istilah pemrosesan graf *native* untuk menggambarkan basis data graf yang memanfaatkan *index-free adjacency*.
Relationship adalah unsur utama dari model data graf ini, dan tidak berlaku pada sistem manajemen basis data lainnya, di mana kita harus menyimpulkan hubungan antara entitas menggunakan hal-hal seperti *foreign key* atau pemrosesan out-of-band seperti *map-reduce*. Dengan menyusun abstraksi sederhana dari *node* dan *relationship* ke dalam struktur yang terhubung, basis data graf memungkinkan kita untuk membangun model-model canggih sebebas-bebasnya yang mendekati permasalahan dari domain yang kita hadapi. Model yang dihasilkan lebih sederhana dan pada saat yang sama lebih ekspresif daripada yang dihasilkan menggunakan basis data relasional tradisional dan penyimpanan NOSQL yang lain [13].

2.9 *Triple Store*

Triple Store berasal dari perkembangan *Web* Semantik dan cara penyimpanan data dalam format *triples*. *Triples* sendiri merupakan struktur data yang terdiri dari *subject-predicate-object* seperti yang digambarkan pada Gambar 2.4. Dengan menggunakan *triples*, kita bisa mendapatkan fakta seperti "*Ginger dances with Fred*" dan "*Fred likes ice cream*." Secara individual, *single triples* tidak terlalu berguna secara semantik, tetapi secara keseluruhan, *triples* menyediakan kumpulan data

yang kaya untuk mendapatkan pengetahuan dan menyimpulkan koneksi.



Gambar 2.4 Bentuk data graf pada model *Triple Store*

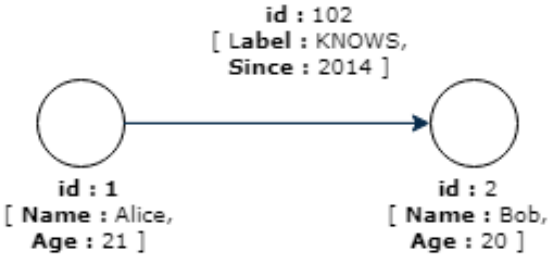
Triple Store dimodelkan berdasarkan spesifikasi *Resource* Description Framework (RDF) yang ditetapkan oleh W3C, menggunakan SPARQL sebagai bahasa *query* nya. Data yang diproses oleh *Triple Store* cenderung dihubungkan secara logical, sehingga *Triple Store* termasuk dalam kategori basis data graf. Namun, *Triple Store* bukan merupakan basis data graf "*native*" karena dia tidak mendukung *index-free adjacency*, juga mesin penyimpanannya tidak dioptimalkan untuk menyimpan *property* graf.

Triple Store menyimpan *triples* sebagai elemen independen, yang memungkinkan mereka untuk horizontal scalling tetapi mengurangi kemampuan untuk *traverse* graf secara cepat. Untuk melakukan *query* graf, *Triple Store* harus membuat koneksi dari fakta individu dan independen - menambahkan latensi terhadap setiap *query*. Karena timbal-balik ini dalam hal scalling dan latensi, kasus penggunaan paling umum untuk *Triple Store* adalah analitik offline dan bukan untuk transaksi online. [13].

2.10 *Labeled Property Graph*

Labeled Property Graph merupakan model yang terdiri atas *node*, *relationship*, *property*, dan label. Pada model ini sebuah *node* atau *relationship* dapat dinyatakan jenisnya dengan

menggunakan label. Sebuah *node* dapat digambarkan sebagai sebuah dokumen yang didalam nya terdapat keterangan pasangan *key-value* yang dapat dinyatakan secara bebas. Secara sederhana bentuk data dari basis graf model *Labeled Propert Graph* dapat digambarkan sesuai dengan Gambar 2.5.



Gambar 2.5 Bentuk data graf pada model *Labeled Property Graph*

Node dapat ditandai dengan menggunakan satu atau lebih label. Dalam hal ini label memberikan keterangan untuk mengelompokkan jenis *node* dan peran yang dimiliki *node* tersebut. *Relationship* menghubungkan *node-node* dan membentuk struktur pada graf. Sebuah *relationship* selalu memiliki arah, sebuah nama, *start node* dan *end node*. Bila digabungkan arah *relationship* dan penamaan memberikan kejelasan semantik dari proses membuat struktur pada *node-node*. Seperti sebuah *node*, *relationship* juga bisa memiliki *property*. Kemampuan menambahkan keterangan *property* pada sebuah *relationship* memberikan kegunaan yang sangat berguna dalam menyediakan *metadata* tambahan yang bisa digunakan dalam algoritma graf, menambahkan semantik tambahan untuk *relationship*, dan pemberian *constraint* pada saat *runtime*. [13].

2.11 Neo4j

Neo4j merupakan NoSQL *open-source* berupa basis data graf *native* yang menyediakan backend transactional dengan

menerapkan konsep ACID. Pengembangan Neo4j dimulai pada tahun 2003, mulai tersedia secara publik pada tahun 2007. Neo4j dibangun dengan menggunakan Java dan Scala yang tersedia di Github, dengan komunitas yang semakin bertumbuh.

Neo4j disebut sebagai basis data graf *native* karena mengimplementasikan model *Property Graph* secara efisien ke tingkat penyimpanan. Berbeda dengan graf *processing* atau *in-memory libraries*, Neo4j menyediakan karakteristik basis data lengkap termasuk patuh akan konsep transaksi ACID, dukungan cluster, dan *running time failover*, sehingga cocok untuk menggunakan data graf dalam skenario *production*. [14]

2.12 Apache Jena Fuseki

Apache Jena Fuseki adalah basis data *graph Triple Store*. Jena Fuseki bisa berjalan sebagai layanan sistem operasi, seperti aplikasi *web* Java (*file* WAR), dan sebagai *server* mandiri. Jena Fuseki juga menyediakan keamanan (menggunakan Apache Shiro) dan memiliki antarmuka pengguna untuk pemantauan dan administrasi *server*. Jena Fuseki menyediakan protokol SPARQL 1.1 untuk *query* dan *update* serta protokol SPARQL *Graph Store*. Fuseki terintegrasi dengan TDB untuk menyediakan lapisan penyimpanan persisten yang kuat dan transaksional, dan menggabungkan kueri teks Jena dan kueri spasial Jena. Jena Fuseki dapat digunakan untuk menyediakan mesin protokol untuk *query* dan sistem penyimpanan RDF lainnya [15].

2.13 Apache Jena

Apache Jena adalah kerangka kerja Java untuk membangun aplikasi *web* semantik. Jena menyediakan banyak *library* Java yang luas untuk membantu *developer* mengembangkan kode yang menangani RDF, RDFS, RDFa, OWL dan SPARQL sesuai dengan rekomendasi W3C yang diterbitkan. Jena menyertakan mesin inferensi berbasis aturan untuk melakukan penalaran berdasarkan ontologi OWL dan RDFS, dan berbagai strategi

penyimpanan untuk menyimpan RDF *triples* dalam memori atau pada disk. [15].

2.14 *Index* pada Neo4j

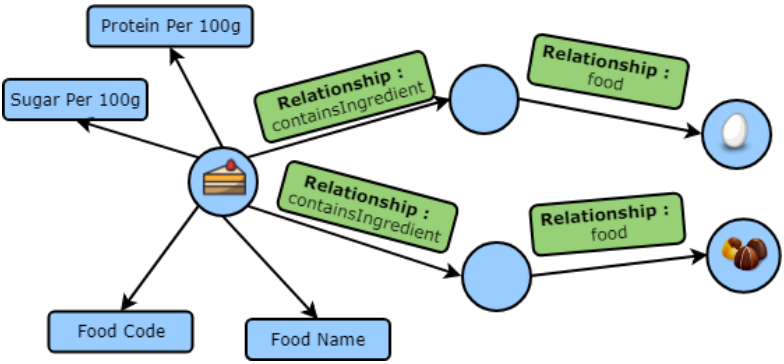
Pada Neo4j terdapat *index* yang akan membentuk struktur data baru pada basis data Neo4j. Data ini akan disimpan secara terpisah dari tempat yang menjadi penyimpanan data utama pada Neo4j. Pada Neo4j terdapat 2 jenis *index*, yaitu *single-property index* dan *composite index*. Pada *single-property index*, *indexing* terhadap *node* pada Neo4j akan dilakukan terhadap salah satu *property* yang merupakan *node* dengan label tertentu, sedangkan *composite index* dilakukan terhadap beberapa *property* dari *node* yang memiliki label tertentu. Pada Kode Sumber 2.5 merupakan contoh *create index* untuk *single-property index* pada Neo4j. [14]

```
1.  create index on:Resource(uri)
```

Kode Sumber 2.7 Contoh sintaks *create index* untuk *single-property index*.

2.15 *Open Food Facts*

Open food facts merupakan basis data dari produk-produk makanan yang ada di seluruh dunia yang datanya bisa didapatkan bebas secara online. Data yang ada pada *open food facts* dibuat berdasarkan kontribusi orang-orang yang ingin membagikan informasi terkait produk-produk makanan yang ada pada seluruh dunia. Data *open food facts* digunakan sebagai *dataset* dalam proses pengujian basis data graf yang akan dibuat. Pada Gambar 2.6 merupakan gambar diagram representasi dari *entity* utama yang ada pada data *open food facts*. [16]



Gambar 2.6 Representasi diagram dari *entity* utama pada *open food facts*

[Halaman ini sengaja dikosongkan]

BAB III
ANALISIS DAN PERANCANGAN

Bab ini berisi analisis permasalahan dan perancangan sistem yang akan dibangun. Tahap analisis membahas mengenai analisis permasalahan yang menjadi dasar dari tahap perancangan.

3.1 Analisis

Tahap analisis menjelaskan permasalahan yang ada pada model penyimpanan data RDF. Pada tahap ini juga akan dijelaskan usulan metode yang akan digunakan dalam mengoptimalkan SPARQL *query* pada model data RDF.

3.1.1 Analisis permasalahan

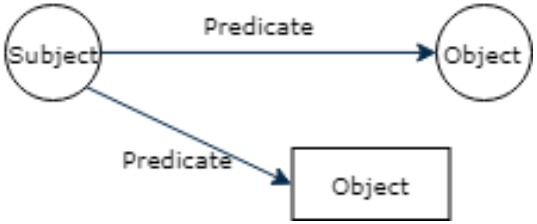
RDF merupakan standar dasar yang ditetapkan oleh W3C sebagai model yang digunakan untuk pertukaran data pada pengembangan *web* semantik. RDF memahami dunia dalam hal entitas yang terhubung, dan menjadi sebuah hal yang populer di awal abad ini karena artikel *The Semantik Web* yang diterbitkan di *Scientific American* [17]. Dalam artikel ini dijelaskan visi yang ingin dicapai oleh para penulis dari internet, dimana orang-orang akan mempublikasikan data mereka dalam sebuah format yang terstruktur dengan semantik yang *well-defined* dengan cara dimana *software-agents* dapat melakukan hal-hal yang pintar dengan model RDF tersebut.

Saat ini, konsep penyimpanan model data RDF menjadi hal yang populer, dengan konsep penyimpan yang disebut sebagai *triples stores*. Namun model data RDF tidak diperuntukkan sebagai cara tertentu dalam menyimpan data. Data yang ada pada RDF dapat disimpan dengan berbagai cara berupa tabel relational, dokumen-dokumen, pasangan *key-value, property graph, triples graph* namun tetap dipublish atau pertukaran data yang dilakukan sebagai model RDF. Metode penyimpanan RDF yang sudah ada sangat terpaku akan

konsep *index-base*, sehingga proses pencarian data pada penyimpanan tersebut terpaku dengan memindai *index* yang telah dibuat.

Perkembangan pada metode *Triple Store* disebabkan karena karakteristik model *dataset* RDF di masa kini yang bersifat aditif, dan cenderung dituntut ketahanannya dengan ukuran data yang sangat besar. Seperti *dataset* YAGO3 yang datanya didapatkan dengan mengekstrak data pada wikipedia dan saat ini sudah memiliki kurang lebih 120 juta *triples* dalam datanya. Dengan karakteristik data yang seperti ini model data RDF akan beresiko pada masa yang akan datang dalam pemrosesan datanya.

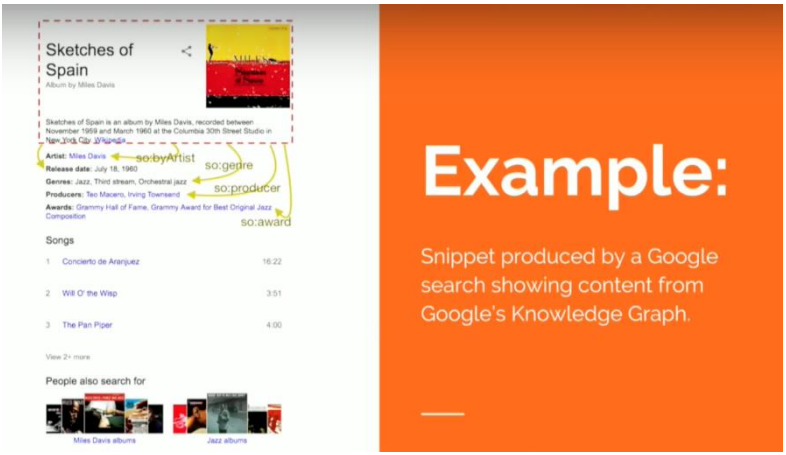
Seperti yang kita ketahui bahwa sebuah graf disusun oleh 2 komponen, yaitu *vertice* dan *edge* yang menghubungkan mereka. Dalam model data RDF, vertex yang ada pada data graf bisa berupa 2 hal. Inti dari RDF adalah gagasan tentang *triples*, yang merupakan *statement* yang terdiri dari tiga elemen yang mewakili 2 *vertice* yang dihubungkan dengan *edge*. Hal ini disebut sebagai *subject-predicate-object*. Sebuah *subject* akan berupa *resource* atau *vertice* dalam graf. *Predicate* akan merepresentasikan sebuah *edge*, dan *object* dapat menjadi *node / vertice* lain berupa *resource* atau berupa nilai *literal* [18] keterangan ini dapat dilihat pada Tabel 3.1. Bentuk struktur graf dari model data RDF dapat dilihat pada Gambar 3.1.



Gambar 3.1 Struktur graf model RDF

Tabel 3.1 Tabel Keterangan Struktur graf

vertice	
Resource	Berupa *URI*
Nilai atribut	Berupa nilai *literal*
Edge	
Relationship	Berupa *URI*



Gambar 3.2 *Snippets* hasil pencarian *Sketches of Spain* di Google [15]

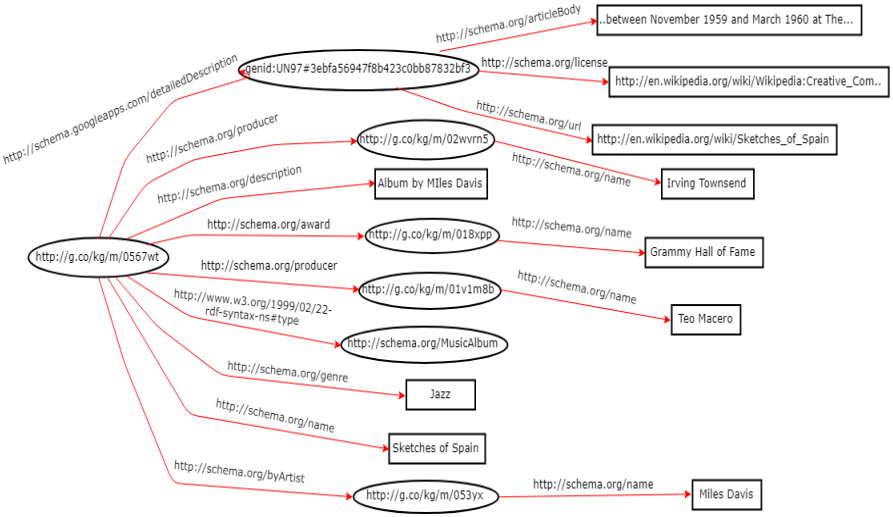
Pada Gambar 3.2 merupakan contoh *snippets* yang akan ditampilkan oleh google apabila kita mencari *Sketches of Spain.* Pencarian ini mengembalikan deskripsi album dengan atribut-atribut seperti artis, tanggal rilis, genre, produser dan beberapa penghargaan. Berikutnya merupakan dua model bentuk representasi dari hasil pencarian tersebut.

```
1.    @prefix schema: <http://schema.org/>.
2.    @prefix nso: <http://schema.googleapis.com/>.
3.        INSERT DATA {
4.            <http://g.co/kg/m/0567wt>
5.            schema:name 'Skectches of Spain';
6.                a schema: MusicAlbum;
7.            schema: description 'Album by Miles Davis';
8.            schema: genre 'Jazz';
9.            ns0: detailedDescription[
10.   schema: liscense 'Creative_Commons_Attribution-
      ShareAlike_3.0_Liscense';
11.   schema: url 'http://en.wikipedia.org/wiki/Sketches_of_Spain
      ';
12.   schema: articleBody '_between Nov 1959 and Mar 1960 at the
      Columbia 30th St Studio in NY City'];
13.           schema: award <http: //g.co/kg/m/018xppx>;
14.           schema: byArtist <http: //g.co/kg/m/053yx>;
15.           schema: producer <http: //g.co/kg/m/01vm8b>, <http:
      //g.co/kg/m/02wvrn5>.
16.           <http://g.co/kg/m/018xpp> schema:name 'Grammy Hall
      of Fame'.
17.           <http://g.co/kg/m/053yx> schema:name 'Miles Davis'.

18.           <http://g.co/kg/m/01v1m8b> schema:name 'Teo Macero'
      .
19.           <http://g.co/kg/m/02wvrn5> schema.name 'Irving Town
      send'.}
```

Kode Sumber 3.1 Data RDF (sintaks turtle) dari snippets Sketches of Spain

Dapat dilihat pada Kode Sumber 3.1 bahwa *triples* diidentifikasi oleh *URI*, yang merupakan *subject*. Predikatnya adalah nama dan objeknya *Sketches of Spain*, yang jika digabungkan menjadi urutan *triples*. Jadi ini adalah hal-hal yang harus ditulis jika ingin memasukkan data ke dalam *triple store*.



Gambar 3.3 Bentuk graf dari *snippets Sketches of Spain* pada model *Triple Store*

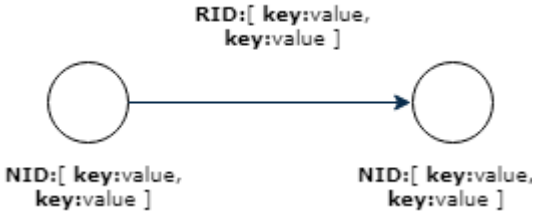
Pada Gambar 3.3 *Node* di sebelah kiri mewakili album, yang memiliki satu set tepian yang keluar darinya. Persegi panjang mewakili nilai *literal* - deskripsi: album tentang Miles Davis, genre, judul - dan memiliki koneksi ke hal-hal lain yang berada dalam elips yang mewakili *resource* lain (yaitu, *node* lain dalam graf) yang memiliki sifat dan atribut.

Artinya, dengan merepresentasikan data dalam RDF dengan *triples*, kita menjabarkan datanya secara maksimum. Dengan ini kita melakukan dekomposisi *atomic* lengkap dari data tersebut, dan akhirnya kita menemukan *node* dalam graf yang merupakan *resource* dan juga nilai *literal*.

3.1.2 Solusi yang diusulkan

Berdasarkan permasalahan yang ada pada penyimpanan data RDF, maka diusulkan metode untuk optimasi SPARQL *query* menggunakan basis data graf dengan model *Labeled Property Graph.*

Salah satu permasalahan yang dihadapi dalam penyimpanan data *triples* pada *file* RDF adalah bentuk data *triples* yang sudah disimpan akan membentuk graf yang tidak efisien, oleh karena itu *Labeled Property Graph* diusulkan dalam penyimpanan data graf. Dalam model *Labeled Property Graph*, *vertice* disebut *node*, yang memiliki ID unik yang dapat diidentifikasi dan kumpulan pasangan *key-value*, atau properti, yang mencirikannya. Pada Gambar 3.4 dengan cara yang sama, *edge*, atau koneksi antar *node* yang kita sebut *relationhip* memiliki ID. Hal ini penting agar kita dapat mengidentifikasi *relationship* secara unik, dan *relationship* juga memiliki jenis dan sekumpulan pasangan *key-value* - atau properti yang mencirikan koneksi.



Gambar 3.4 Struktur Graf *Labeled Property Graph*

Hal yang penting untuk diingat di sini adalah bahwa baik *node* dan *relationship* memiliki struktur internal, yang membedakan model ini dari model RDF. Menurut struktur internal, maksudnya adalah kumpulan pasangan *key-value* yang mendeskripsikannya.

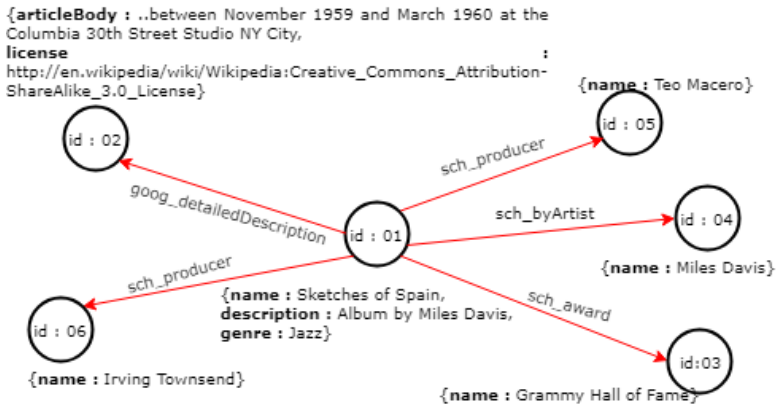
Berdasarkan contoh data *snippets* yang diambil dari hasil pencarian *Sketches of Spain* pada Gambar 3.2, pada Kode Sumber 3.2 merupakan *query* Cypher yang dibutuhkan untuk membuat *Labeled Property Graph* berdasarkan data tersebut [18].

```
1.    CREATE(sos: Resource: MusicAlbum { name: "Sketches of Spain
      ", description: "Album by Miles", genre: "Jazz"})
2.
3.    CREATE(dd { liscense: "Creative_Attribution-
      ShareAlike_3.0_Liscense", url: "http://en.wikipedia.org/wik
      i/Sketches_of_Spain", articleBody: "_bettween Nov 1959 and
      Mar 1960 at the Columbia 30th St Studio in NY City" })
4.
5.    CREATE(sos)-[:goog_detailedDescription]->(dd)
6.
7.    CREATE(sos)-[:award]->({ name: "Grammy Hall of Fame"})
8.    CREATE(sos)-[:byArtist]->({ name: "Miles Davis" })
9.    CREATE(sos)-[:aproducer]->({ name: "Teo Macero" })
10.   CREATE(sos)-[:producer]->({ name: "Irving Townsend" })
```

Kode Sumber 3.2 Bentuk sintaks *CREATE* pada *query* Cypher Neo4j

Secara semantik strukturnya masih sama. Tidak ada format serialisasi standar, atau cara untuk mengekspresikan *Labeled Property Graph*, tetapi lebih pada urutan *CREATE* yang dibutuhkan.

Pada sebuah *node*, diwakili dengan tanda kurung, dan kemudian atribut atau struktur internal dalam kurung kurawal: nama, deskripsi dan genre. Demikian pula, koneksi antara *node* dijelaskan dengan kurung kurawal.



Gambar 3.5 Bentuk graf pada neo4j dari *Snippets Sketches of Spain*

Berdasarkan Gambar 3.5 kesan pertama yang terpikirkan adalah model ini jauh lebih ringkas. Meskipun memiliki elemen-elemen tertentu yang sama dengan graf RDF, hal ini benar-benar berbeda karena *node* memiliki struktur internal dan nilai-nilai atribut ini tidak mewakili *node* dalam graf. Hali ni memungkinkan untuk membentuk struktur yang lebih singkat. Graf ini masih memiliki *node* album di pusat, yang terhubung ke sejumlah entitas, tetapi judul, nama, dan deskripsi tidak diwakili oleh *node* terpisah. Itu adalah perbedaan yang paling utama.

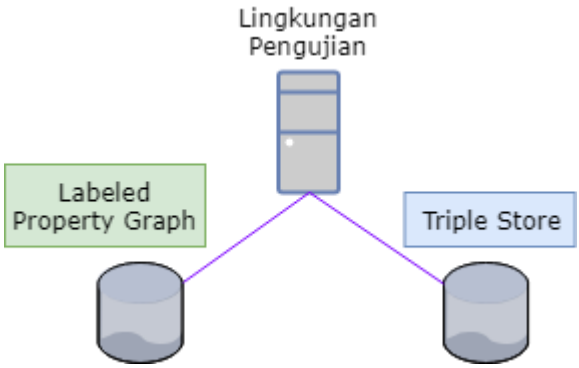
3.2 Perancangan Sistem

Arsitektur perangkat lunak yang dibahas meliputi arsitektur secara utuh. Tahap perancangan sistem membahas mengenai arsitektur sistem yang akan digunakan dalam proses implementasi dan pengujian, kemudian akan dijelaskan rencana yang akan dilakukan untuk memetakan data RDF menjadi model data *Labeled Property Graph*. Terakhir akan dijelaskan mengenai rencana pengujian terhadap sumber daya lingkungan pengujian terhadap *dataset* yang akan digunakan.

3.2.1 Arsitektur sistem

Pada penelitian ini akan dilakukan perbandingan performa 2 model basis data graf, yaitu model data *Triple Store* dan model data *Labeled Property Graph*. Model data *Labeled Property Graph* akan digunakan sebagai struktur data baru pengganti model data *Triple Store* sebagai metode untuk mengoptimalkan performa SPARQL *query*. Membandingkan performa basis data model *Labeled Property Graph* dengan model *Triple Store* bertujuan untuk membandingkan struktur data baru dengan struktur data yang digunakan sampai saat ini. Arsitektur sistem pengujian dapat dilihat pada Gambar. 3.6. Pada Gambar. 3.6. dalam lingkungan pengujian akan diapsang basis data graf dengan model *Triple Store* dan *Labeled Property Graph*. Kemudian, pada kedua basis data akan dijalankan SPARQL *query* pada model *Triple Store* dan *query* yang sesuai dengan bahasa *query*

pada model *Labeled Property Graph* seperti yang dapat dilihat pada Gambar 3.8.



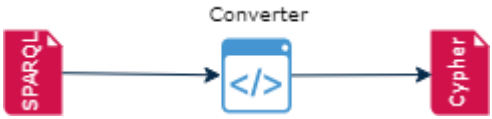
Gambar 3.6 Arsitektur Sistem

Berdasarkan Gambar 3.6, pada penelitian ini akan dipasang 2 basis data graf secara bedampingan yaitu Apache Jena Fuseki yang menerapkan model *Triple Store*, dan Neo4j sebagai basis data graf yang menerapkan model L*abeled Property Graph.* Dua basis data graf tersebut akan digunakan sebagai manajemen sistem basis data untuk menyimpan data dari *file* RDF yang akan diuji dan untuk pemrosesan *query* yang akan dijalankan untuk mendapatkan *output* dari *file* RDF yang sudah disimpan.

Pada basis data graf Neo4j akan dipasang *plugin* neosemantiks yang akan digunakan untuk memasukkan *file* RDF. *Plugin* neosemantiks tersebut akan memetakan data RDF menjadi model data *Labeled Property Graph* sehingga data RDF dapat diproses pada basis data Neo4j.

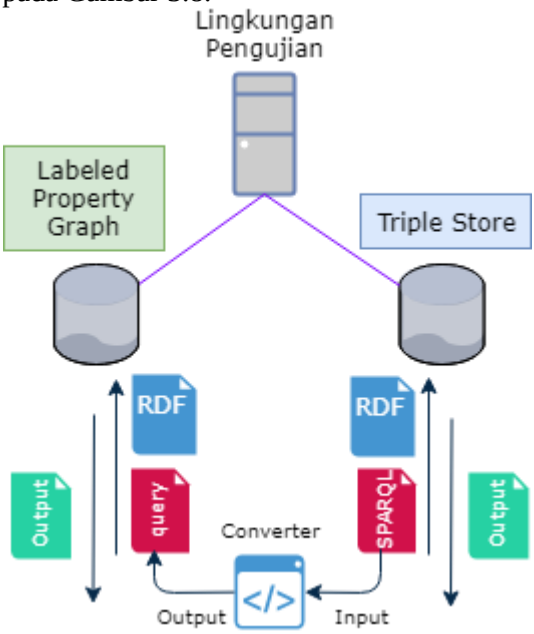
Selain akan dibangun sistem manajemen basis data graf dengan model *Triple Store* dan model *Labeled Property Graph*, pada tugas akhir ini juga dibuat program *converter* dari SPARQL *query* yang akan diubah menjadi bentuk Cypher *query*. Perubahan SPARQL *query* menjadi Cypher *query* karena Pada program *converter* tersebut dibangun dengan menggunakan Bahasa pemrograman Java dengan *input*an berupa *file* berisikan SPARQL *query* dan akan menghasilkan

output Cypher *query*. Pada Gambar 3.7 merupakan arsitektur dari program *converter* yang akan dibangun.



Gambar 3.7 Program Converter

Secara keseluruhan arsitektur sistem yang akan dibangun dapat dilihat pada Gambar 3.8.



Gambar 3.8 Arsitektur sistem secara keseluruhan

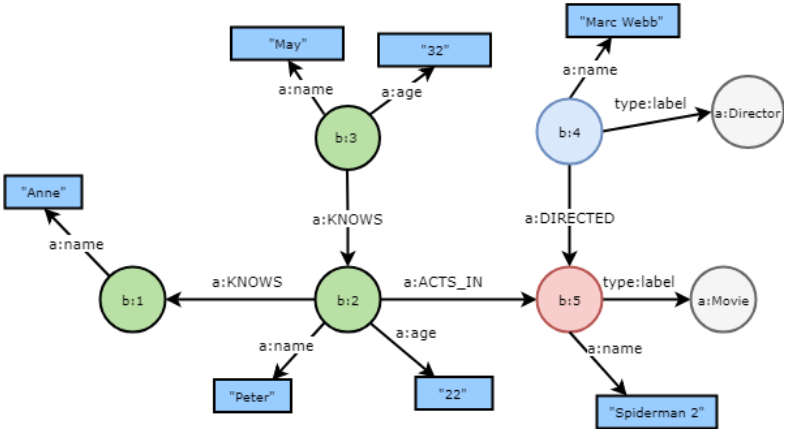
Pada Gambar 3.8, setiap basis data graf yang sudah dipasang pada lingkungan implementasi akan diberikan masukan berupa *file* RDF yang sama. *File* RDF yang sama disini berdasarkan pada skenario pengujian yang akan dilakukan dimana setiap skenario memiliki ukuran *file* yang berbeda-beda. Pada sistem yang dibangun

setiap basis data graf juga akan memproses *query* terhadap data yang sudah disimpan sebelumnya. *Query* yang dijalankan sesuai dengan mesin pemrosesan masing–masing basis data, pada Neo4j pemrosesan *query* menggunakan Cypher *query* yang sudah diubah dari SPARQL *query*. Sedangkan pada basis data graf Apache Jena Fuseki pemrosesan *query* akan langsung dijalankan menggunakan SPARQL *query* yang diuji. Bedasarkan arsitektur tersebut sistem yang sudah dibangun diharapkan dapat menghasilkan *output* yang sesuai pada kedua basis data.

3.2.2 Perancangan *indexing*

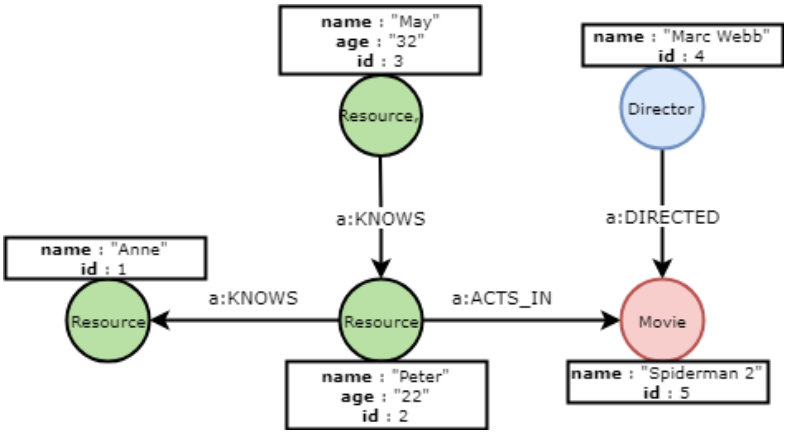
Indexing yang akan digunakan, akan dibuat pada Neo4j sebagai basis data graf yang akan digunakan sebagai metode optimasi SPARQL *query*. Berdasarkan data *file* RDF yang akan dipetakan menjadi *Labeled Property Graph* oleh *plugin* Neosemantiks, hal penting yang perlu diperhatikan adalah aturan pada Neosemantiks yang merubah bentuk *subject-predicate-object* menjadi *node* yang berisikan *key* dan *value*. Dalam aturan yang dibuat salah satunya adalah merubah *subject* dari bentuk *subject* yang dinyatakan dengan s pada bentuk *triples* statement { s p o . } dan *subject* nya berbentuk *resource URI* maka akan dipetakan menjadi *node* dengan label :*Resource* dan *string subject* yang akan dijadikan *value* pada *property* "*uri*" sebuah *node*. Selain aturan tersebut, aturan yang lain masih tidak pasti bergantung pada *string predicate* dan *object* yang ada pada sebuah *statement triples*. Oleh karena itu, *indexing* yang akan dibuat pada Neo4j, dibuat berdasarkan *node* dengan label :*Resource* yang memiliki *property* "*uri*". Implementasi *indexing* dapat dilihat pada Kode Sumber 4.4.

3.2.3 Perancangan metode *query*



Gambar 3.9 Contoh data graf pada *file* RDF

Untuk merancang metode pengubahan *query*, kita perlu mengamati terlebih dahulu bentuk data yang akan disimpan pada kedua basis data. Pada Gambar 3.9, merupakan contoh data berupa gambar graf yang akan dibentuk pada model data *Triple Store* sedangkan pada Gambar 3.10, merupakan bentuk graf yang akan dibentuk pada model data *Labeled Property Graph*. pada model *Labeled Property Graph* akan terlihat bentuk graf yang dibuat menjadi lebih sederhana. hal ini disebebkan kerena *node* pada model *Labeled Property Graph* dapat menyimpan data *predicate* dari model *Triple Store* menjadi pasangan *key-value* pada sebuah *node* yang menjadi *resource* untuk data *property* tersebut. aturan terpenting dalam merubah bentuk graf dari penelitian sebelumnya adalah pemetaan sebuah *node resource* berdasarkan hubungan dengan *node* lainnya. Hubungan yang dimaksud adalah hubungan antar *node* yang dilihat pada *node* destinasi nya, apakah *node* yang menjadi tujuan berupa *node resource* atau *node string literal*.



Gambar 3.10 Contoh data graf pada model Labeled Proprty *Graph*

Contoh pada Gambar 3.9 *node resource* b:2 yang memiliki hubungan a:*name* dengan *node* yang berisikan *string* "Peter" akan dipetakan menjadi *node* yang memiliki *property name* dengan nilai *string* "Peter" pada Gambar 2. sedangkan *node* b:2 yang memiliki hubungan a:KNOWS dengan *node* b:1 pada Gambar 3.9. akan dipetakan menjadi hubungan *node* antar *node* pada Gambar 3.10. Hal ini yang menyebabkan struktur graf pada model *Labeled Property Graph* yang akan dibuat bisa lebih sederhana dari data pada model *Triple Store file* RDF. selain itu pada penelitian sebelumnya, *node* yang berupa *resource* akan dipetakan menjadi sebuah *node* yang memiliki label tertentu. *node resource* b:4 pada Gambar 3.9. akan dipetakan menjadi *node* dengan label Director pada Gambar 3.10 berdasarkan hubungan type:label antara *node* b:4 dengan *node* a:Director.

Setelah data berhasil dipetakan, permasalahan yang dihadapi adalah *Graph Pattern Matching* (GPM). GPM merupakan persoalan dalam mendapatkan subgraf yang sesuai pada 2 buah graf yang berbeda. permasalahan GPM dapat dianggap sesuai tanpa memperhatikan hubungan graf yang saling terhubung secara terarah atau tidak terarah. GPM adalah proses komputasional dalam

mengevaluasi *pattern* graf pada sebuat basis data graf. bentuk paling sederhana dari GPM adalah *Basic Graph Pattern* (BGP). BGP berupa 1 buah statement yang akan digunakan dalam mengevaluasi sebuah subgraf yang terdapat dalam sebuah graf. dalam penelitian ini, permasalahan GPM yang dihadapi adalah pencarian bentuk *traverse* graf pada 2 buah bentuk graf yang berbeda. 2 bentuk graf yang dimaksud adalah bentuk graf *Triple Store* dan *Labeled Property Graph*. karena 2 model basis data yang digunakan berbeda, struktur data yang digunakan pun berbeda, maka bentuk *query* pada bahasa *query* yang digunakan di masing-masing basis data graf juga akan berbeda. metode pemetaan bentuk *query* yang akan dilakukan adalah dengan memetakan bentuk SPARQL *query* menjadi Cypher *query* yang memiliki hasil subgraf yang sesuai pada kedua bahasa *query* tersebut. Pemetaan yang dilakukan dengan memperhatikan bentuk BGP pada SPARQL *query* untuk diubah menjadi BGP pada Cypher *query*.

```
1.  SELECT ?s WHERE {
2.      ? s a: name“ Peter”.
3.  }
```

Kode Sumber 3.3 Contoh bentuk BGP dari SPARQL *query*

BGP dari SPARQL *query* yang terdapat pada graf Gambar 1. dapat dicontohkan dengan sintaks "WHERE { ?s a:name "Peter" . }". Pada contoh BGP tersebut, SPARQL *query* akan mencari *node* yang memiliki hubungan dengan *node string literal* "Peter" dengan hubungan a:*name*. pada bentuk BGP tersebut, terdapat variabel s yang dinyatakan dengan "?s". nilai dari variabel s yang akan dihasilkan berupa *string resource* dari *node* yang terhubung dengan *predicate* a:*name* terhadap *node* "Peter". apabila pada sintaks *SELECT* terdapat variabel s yang dinyatakan dengan "SELECT ?s ", maka hasil *query* pada kondisi yang dijelaskan sebelumnya akan ditampilkan nilai *string resource* "b:2".

```
1.  MATCH(s: Resource {
2.      a: name: “Peter”
3.  }) RETURN s.uri
```

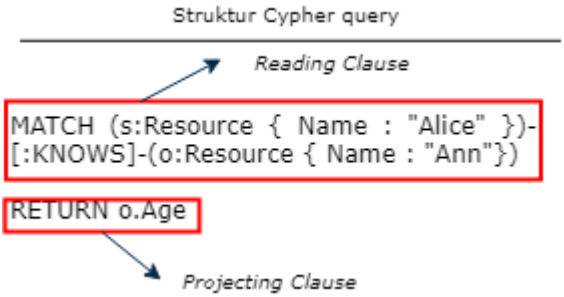
Kode Sumber 3.4 Bentuk sintaks *MATCH* dari perubahan BGP SPARQL *query* menjadi Cypher *query*

Dari bentuk BGP yang terdapat pada SPARQL *query* sebelumnya dapat dipetakan menjadi bentuk BGP pada Cypher *query*. sebagai contoh, dalam memetakan sintaks "*WHERE* { ?s a:*name* "Peter" . }" dapat didapatkan beberapa informasi penting sebelum merubahnya menjadi Cypher *query*. dari sintaks tersebut kita dapat mengetahui bahwa *subject* berupa *resource* dengan *string* b:2, dengan *predicate* a:*name*. selain itu, variabel yang terdapat pada sintaks tersebut adalah s yang dinyatakan dalam bentuk "?s". Dari kondisi tersebut kita dapat membuat *traverse* BGP untuk Cypher *query* nya. Untuk membuat GPM digunakan sintaks *MATCH*() yang berarti melakukan *traverse* untuk semua *node*. karena udah diketahui variabel yang ingin dicari ada pada *subject* nya dan nama variabelnya adalah s, maka sintaks *MATCH* nya menjadi “MATCH(s:Resource)”. Untuk mencari kondisi nama peter maka ditambahkan kondisi pada sintaks *MATCH* menjadi ”MATCH(s:Resource { a:name : “Peter” })”. Untuk *return value* nya karena yang diminta adalah variabel *subject*, maka sintaks *RETURN* nya adalah “RETURN s.uri*”* karena *string* dari *resource subject* disimpan pada *property uri*.

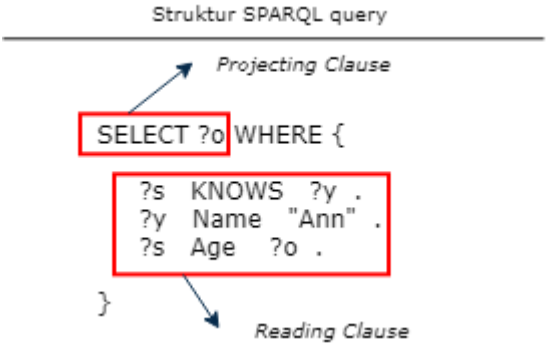
Pada proses *convert* SPARQL *query* menjadi Cypher *query* hal paling penting yang dapat kita sadari bahwa kedua *query* ini memiliki struktur yang mirip walaupun tidak sama. Dimana kedua *query* ini pasti dibuat dengan memiliki *projecting clause* dan *reading clause* pada setiap sintaks *query* nya. Yang dimaksud dengan *projecting clause* berdasarkan basis data Neo4j merupakan sintaks yang digunakan untuk menampilkan hasil dari proses *traverse* graf. Sedangkan *reading clause* merupakan kondisi yang harus dipenuhi dalam melakukan *traverse* pada graf yang dimiliki.

Pada sintaks Cypher *query*, *projection clause* berupa sintaks *RETURN* diikuti variabel-variabel yang ingin ditampilkan pada ouput

seperti yang dapat dilihat pada Gambar 4.8. Hal ini pun sama serperti pada SPARQL *query* namun pada SPARQL *query projecting clause* berupa sintaks *SELECT* diikuti variabel-variabel yang ingin ditampilkan seperti yang dapat dilihat pada Gambar 4.9.



Gambar 3.11 Struktur Cypher *query*



Gambar 3.12 Struktur SPARQL *query*

Kemudian pada sintaks Cypher *query*, *reading clause* terdapat pada proses *traverse* graf baik itu dalam sintaks *MATCH* atau penambahan kondisi seperti sintaks *WHERE* untuk mem*filter* hasil dari proses *traverse* graf seperti yang dapat dilihat pada Gambar 4.8. Sedangkan dalam SPARQL *query*, *reading clause* berupa kondisi-kondisi *triples* yang terdapat setelah sintaks *WHERE* seperti yang dapat dilihat pada Gambar 4.9. Seperti yang disebutkan sebelumnya

hal ini membuktikan bahwa struktur SPARQL *query* dan Cypher *query* memiliki struktur yang sama hanya pada penempatan strukturnya yang berbeda. Pada SPARQL *query*, *projecting clause* diletakkan di awal sedangkan pada Cypher *query* diletakkan di akhir setelah *reading clause* dinyatakan.

Dari dasar tersebut diaajukan metode untuk mengubah SPARQL *query* dengan menggunakan pemeriksaan kondisi SPARQL *query* yang ada untuk diubah menjadi bentuk Cypher *query* yang sesuai. Kondisi-kondisi yang ada akan dipetakan menjadi Cypher *query* berdasarkan *return value* yang yang diminta, variabel yang terdapat pada statement dan bentuk SPARQL *query* yang ingin diubah. Kondisi-kondisi tersebut dapat dilihat pada Tabel 3.2.

Tabel 3.2 Kondisi-kondisi dalam mengubah SPARQL *query* menjadi Cypher *query*

No	*Return Value*	*Variable*	Bentuk SPARQL *query*	Bentuk Cypher *query* yang dihasilkan
1	*subject*	*subject*	where { ?s p:*predicate* "string" . }	MATCH(s:*Resource* { `p:*predicate*` : "string" }) RETURN s.*uri*
2	*subject*	*subject*	where { ?s <p:*predicate*> <*resource*:*object*> . }	MATCH(s:*Resource*)-[`p:*predicate*`]-(:*Resource* {*uri* : "*resource*:*object*"}) RETURN s.*uri*
3	*subject*	*subject*	where { ?s rdf:type <*resource*:category> . }	MATCH(s:`*resource*:category`) RETURN s.*uri*
4	*subject*	*predicate*	where { <*resource*:*subject*> ?p <*resource*:*object*> . }	MATCH(:*Resource* {*uri* : "*resource*:*subject*})-[p]-(:*Resource* { *uri* : "*resource*:*object*"}) RETURN type(p)
5	*subject*	*subject, object*	where { ?s <p:*predicate*> ?o . FILTER	MATCH(s:*Resource*) WHERE s.`p:*predicate*` =~ "String.*"

No	*Return Value*	*Variable*	Bentuk SPARQL *query*	Bentuk Cypher *query* yang dihasilkan
			regex(?o, "^String") . }	RETURN s.*uri*
6	*object*	*object*	where { <*resource*:*subject*> <p:*predicate*> ?o . }	MATCH(s:*Resource* {*uri* : "*resource*:*subject*"}) WHERE s.`p:*predicate*` is not null RETURN s.`p:*predicate*` as o UNION ALL MATCH(:*Resource* {*uri* : "*resource*:*subject*"})- [:`p:*predicate*`]-(o) RETURN o.*uri* as o
7	*subject, object*	*subject, object*	where { ?s <p:*predicate*> ?o . }	MATCH(s:*Resource*)- [:`p:*predicate*`]-(o:*Resource*) RETURN s.*uri* as s, o.*uri* as o
8	*subject, predicate*	*subject, predicate*	where { ?s ?p <*resource*:*object*> . }	MATCH(s:*Resource*)-[p]- (o:*Resource* {*uri* : "*resource*:*object*"}) WITH s.*uri* as s, type(p) as p RETURN s, p

Berikut penjelasan lebih lanjut dari kondisi-kondisi yang digunakan untuk merubah SPARQL *query* menjadi Cypher *query*.

3.2.3.1 Kondisi *Query* 1

Pada Kode Sumber 3.5 merupakan bentuk SPARQL *query* untuk mendapatkan data *uri node* dengan *property* berupa *string* "*predicate*". Pada Kode Sumber 3.6 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.5.

```
1.  select ?s where {
2.      ?s <relationship:predicate> "predicate" .
3.  }
```

Kode Sumber 3.5 Kondisi SPARQL *Query* 1

```
1.  MATCH(s:Resource { `relationship:predicate`: "predicate" })
2.  RETURN s.uri
```

Kode Sumber 3.6 Kondisi Cypher *Query* 1

3.2.3.2 Kondisi *Query* 2

Pada Kode Sumber 3.7 merupakan bentuk SPARQL *query* untuk mendapatkan data *uri node* yang memiliki hubungan "*relationship*:*predicate*" dengan *resource* "*resource*:*object*". Pada Kode Sumber 3.8 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.7.

```
1.  select ?s where {
2.      ?s <relationship:predicate> <resource:object> .
3.  }
```

Kode Sumber 3.7 Kondisi SPARQL *Query* 2

```
1.  MATCH(s:Resource)-[:`relationship:predicate`]-(:Resource
    {uri : "resource:object"})
2.  RETURN s.uri
```

Kode Sumber 3.8 Kondisi Cypher *Query* 2

3.2.3.3 Kondisi *Query* 3

Pada Kode Sumber 3.9 merupakan bentuk SPARQL *query* untuk mendapatkan data *uri node* yang memiliki kategori *resource*:*object*. Pada Kode Sumber 3.10 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.9.

```
1.  select ?s where {
2.      ?s rdf:type <resource:object>.
3.  }
```

Kode Sumber 3.9 Kondisi SPARQL *Query* 3

```
1.  MATCH(s:`resource:object`)
2.  RETURN s.uri
```

Kode Sumber 3.10 Kondisi Cypher *Query* 3

3.2.3.4 Kondisi *Query* 4

Pada Kode Sumber 3.11 merupakan bentuk SPARQL *query* untuk mendapatkan hubungan yang dimiliki antara dua *node* dengan *node* pertama memiliki *uri* dengan *string* "*resource*:*subject*" dan *node* yang terhubung memiliki *uri* dengan *string* "*resource*:*object*". Pada Kode Sumber 3.12 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.11.

```
1.  select ?p where {
2.      <resource:subject> ?p <resource:object>.
3.  }
```

Kode Sumber 3.11 Kondisi SPARQL *Query* 4

```
1.  MATCH(s:Resource {uri : "resource:subject"})-[p]-(:Resource
    {uri : "resource:object"})
2.  RETURN type(p)
```

Kode Sumber 3.12 Kondisi Cypher *Query* 4

3.2.3.5 Kondisi *Query* 5

Pada Kode Sumber 3.13 merupakan bentuk SPARQL *query* untuk mendapatkan *node* dengan *property food*:*name* dengan kondisi *wildcard*, dengan *string wildcard* "Almond*" . Pada Kode Sumber 3.14 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.13.

```
1.  select ?s where {
2.      ?s food:name ?o .
3.      FILTER regex(?o, "^Almond") .
4.  }
```

Kode Sumber 3.13 Kondisi SPARQL *Query* 5

```
1.  MATCH (s:Resource)
2.  WHERE s.`http://data.lirmm.fr/ontologies/food#name` =~
    "Almond.*"
3.  RETURN s.uri
```

Kode Sumber 3.14 Kondisi Cypher *Query* 5

3.2.3.6 Kondisi *Query* 6

Pada Kode Sumber 3.15 merupakan bentuk SPARQL *query* untuk mendapatkan *node uri* ataupun nilai dari *property* pada sebuah *node*. Pada Kode Sumber 3.16 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.15.

```
1.  select ?o where {
2.      <resource:subject> <relationship:predicate> ?o .
3.  }
```

Kode Sumber 3.15 Kondisi SPARQL *Query* 6

```
1.  MATCH(s: Resource { uri: "resource:subject" })
2.  WHERE s.`relationship:predicate` is not null
3.  RETURN s.`relationship:predicate` as o
4.  UNION ALL MATCH(: Resource { uri: "resource:subject" })-
    [:`relationship:predicate`]-(o)
5.  RETURN o.uri as o
```

Kode Sumber 3.16 Kondisi Cypher *Query* 6

3.2.3.7 Kondisi *Query* 7

Pada Kode Sumber 3.17 merupakan bentuk SPARQL *query* untuk mendapatkan *node uri* dari kedua *node* yang memiliki hubungan :`*relationship*:*predicate*`. Pada Kode Sumber 3.18 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.17.

```
1.  select ?s ?o where {
2.      ?s <relationship:predicate> ?o .
3.  }
```

Kode Sumber 3.17 Kondisi SPARQL *Query* 7

```
1. MATCH(s)-[:`relationship:predicate`]-(o)
2. RETURN s.uri as s, o.uri as o
```

Kode Sumber 3.18 Kondisi Cypher *Query* 7

3.2.3.8 Kondisi *Query* 8

Pada Kode Sumber 3.19 merupakan bentuk SPARQL *query* untuk mendapatkan *node uri* dan jenis *relationship* nya yang memiliki hubungan dengan *node* yang memiliki *string uri* "*resource*:*object*". Pada Kode Sumber 3.20 merupakan bentuk Cypher *query* berdasarkan SPARQL *query* pada Kode Sumber 3.19.

```
1. select ?s ?p where {
2.     ?s ?p <resource:object> .
3. }
```

Kode Sumber 3.19 Kondisi SPARQL *Query* 8

```
1. MATCH(s:Resource)-[:p]-(:Resource  {uri :
   "resource:object"})
2. WITH s.uri as s, type(p) as p
3. RETURN s, p
```

Kode Sumber 3.20 Kondisi Cypher *Query* 8

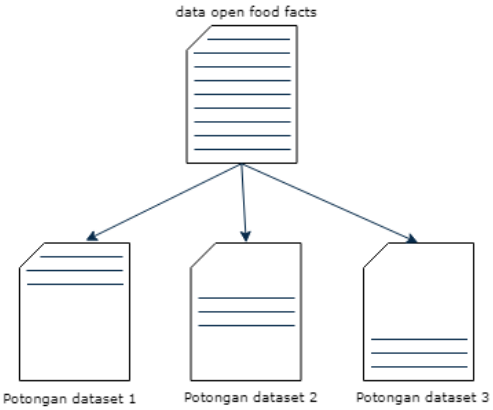
3.2.4 Perancangan pengujian

Pada perancangan pengujian akan dijelaskan cara *preprocess dataset* yang akan digunakan dan penghapusan *cache* dalam proses pengujian.

3.2.4.1 *Preprocess dataset*

Pengujian akan dilakukan dengan menggunakan *dataset* yang disediakan oleh *open food facts*. *Dataset open food facts* memiliki ukuran *file* 1.7 GB, *dataset* ini digunakan karena memiliki data yang cukup besar dan memungkinkan untuk digunakan dalam lingkungan pengujian . Selain itu pengujian juga akan dilakukan dengan menggunakan SPARQL *query* yang menyesuaikan dengan data yang

ada pada *dataset* yang akan diujikan. Data yang digunakan dari *open food facts* akan dibagi menjadi beberapa *file dataset* dengan ukuran yang berbeda-beda. Pemotongan data berdasarkan baris kode XML yang terdapat pada RDF *open food facts*. Proses pemotongan data dapat dilihat pada Gambar 3.11.



Gambar 3.13 Diagram kelas akses data

Pemotongan *dataset* ini perlu dilakukan karena sumber daya pada linkungan pengujian tidak dapat memproses data RDF secara keseluruhan dalam satu kali proses. Pemotongan setiap sejumlah kurang lebih 6 juta baris yang ada pada RDF *open food facts*. Dengan demikian pada skenario pengujian pertama akan menggunakan potongan *dataset* 1, kemudian untuk skenario berikutnya akan mengguanakan potongan *dataset* 1 ditambah potongan *dataset* 2, dan seterusnya.

3.2.4.2 *Reset Cache*

Pada basis data graf Neo4j dan Apache Jena Fuseki, memiliki kemampuan *caching* untuk menyimpan hasil *query* pada memory setelah operasi *query* dilakukan. Hal ini mempengaruhi proses pengujian, untuk mendapatkan hasil pengujian performa basis data

graf terdapat skenario yang mengujikan performa dengan menjalankan operasi *query* atau *update* yang sama. Dalam melakukan operasi *query* atau *update* yang sama ini, perlu dilakukan *reset cache* pada masing-masing basis data agar hasil operasi *query* atau *update* yang didapatkan merupakan hasil yang valid karena operasinya sama seperti basis data yang baru dijalankan pertama kalinya. Untuk mendapatkan hasil *query* yang valid perlu dilakukan *reset cache* dengan cara menjalankan ulang basis data kemudian menghapus *cache* yang sudah dibuat saat basis data graf dijalankan sebelumnya.

BAB IV
IMPLEMENTASI

Pada bab ini berisi implemetasi yang akan menjelaskan mulai dari lingkungan sistem yang akan diimplementasikan. Kemudian penulis akan menjelaskan implementasi yang dilakukan dalam menjawab permasalahan yang menjadi dasar pengerjaan tugas akhir ini

4.1 Lingkungan Implementasi

Berikut kakas bantu yang digunakan pada proses implementasi perangkat lunak ini:

1. Linux Ubuntu 18.04 64 bit sebagai sistem operasi
2. Intellij IDEA Ultimate 2018.1 dengan *plugin Graph Database Support* sebagai *Integrated Development Environment* (IDE)
3. Java JDK 8.1 sebagai pustaka Java dan lingkungan *Runtime*.
4. Neo4j java driver 1.4.4 sebagai *library* untuk menjalankan *query* Cypher.
5. Apache Jena 3.6 sebagai *library* untuk menjalankan SPARQL *query*.
6. Apache Jena Fuseki 3.7.0 sebagai sistem manajemen basis data graf dengan model *Triple Store*.
7. Neo4j 3.4.0 sebagai sistem manajemen basis data graf dengan model *Labeled Property Graph*.

4.2 Cara Membuat Basis Data Graf untuk Menyimpan Data RDF

Pada sub bab ini akan dijelaskan cara membuat basis data graf agar dapat menyimpan data *triples* dari *file* RDF. Basis data yang akan digunakan adalah basis data graf dengan model *Triple Store* dan basis data graf dengan model *Labeled Property Graf.*

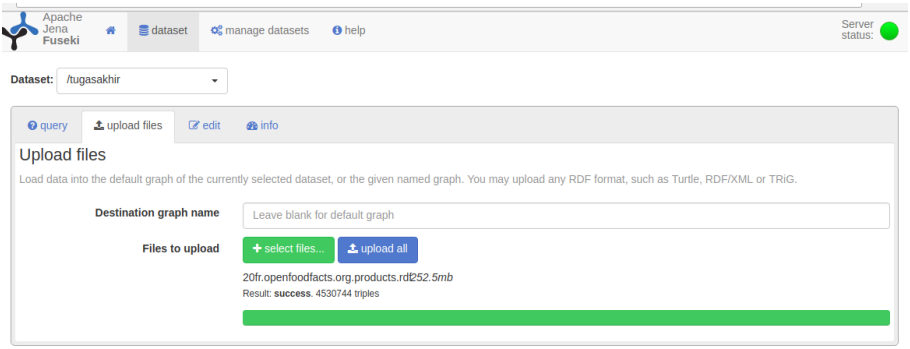
4.2.1 Basis data graf model *Triple Store*

Pada model basis data graf ini, basis data yang digunakan adalah Apache Jena Fuseki. Untuk membuat basis data pada Apahce Jena Fuseki, pertama-tama kita harus menjalankan perintah pada linux *command line* menggunakan perintah yang tertera pada Kode Sumber 4.1.

```
1.  ./fuseki-server --loc=DB /tugasakhir
```

Kode Sumber 4.1 *Command line* untuk menjalankan Apache Jena Fuseki

Kemudian, bila Apache Jena Fuseki sudah berjalan, maka kita dapat mengakses tampilan basis data Apache Jena Fuseki pada *browser* dengan *host* : *localhost* dan *port* : 3030. Kemudian untuk *import* data, buka menu *Add Data* pada baris yang memiliki nama *dataset* yang akan kita gunakan. Pada menu ini, klik tombol *Upload Files* kemudian pilih *file* yang ingin digunakan sebagai data untuk dimasukkan. Kemudian untuk memasukkan data ke Apache Jena Fuseki klik tombol *upload now*. Apabila proses *import* data berhasil maka akan muncul tampilan seperti pada Gambar 4.1. Pada Gambar 4.1. proses *import* data berhasil dengan penambahan 4.530.744 *triples* yang berhasil ditambahkan ke dalam basis data.



Gambar 4.1 Berhasil *import* data pada Apache Jena Fuseki

4.2.2 Basis data graf model *Labeled Property Graph*

Pada basis data graf ini, basis data yang digunakan adalah Neo4j. Untuk membuat basis data pada Neo4j, pertama-tama kita harus menjalankan perintah pada linux *command line* menggunakan perintah yang tertera pada Kode Sumber 4.2. Kode Sumber 4.2 merupakan perintah untuk memulai Neo4j pada ubuntu linux.

```
1.  sudo service neo4j start
```

Kode Sumber 4.2 *Command line* untuk memulai Neo4j

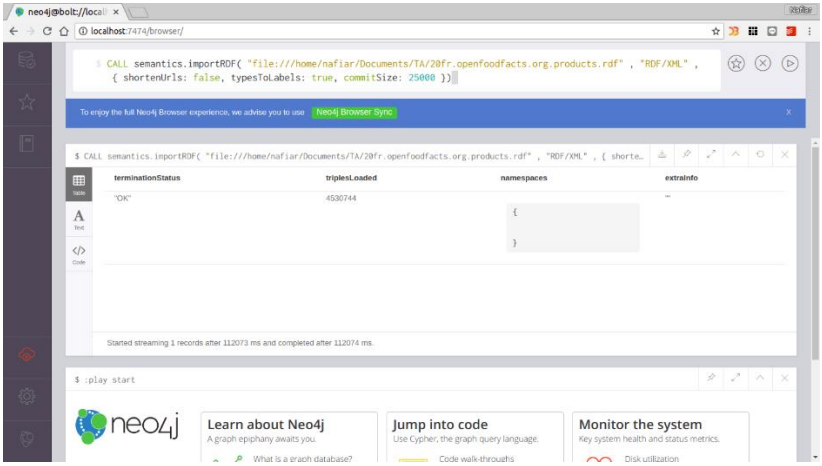
Setelah menjalankan perintah di atas, maka Neo4j akan berjalan sebagai *service* pada ubuntu linux. Untuk membuka tampilan basis data Neo4j kita dapat mengakses nya pada *browser* dengan memasukkan alamat "localhost:7474/browser" pada *browser*. Pada basis data graf Neo4j sudah tersedia basis data *default* yang dapat digunakan untuk penyimpanan data graf. Pada basis data *default* yang sudah ada ini akan digunakan untuk *import* data dari *file* RDF. Sebelum *import* data, harus dilakukan instalasi *plugin* Neosemantiks pada Neo4j. Setelah *plugin* selesai dipasang, selanjutnya harus dibuatkan *index* pada *property uri*

dari *node* dengan label *Resource* sebelum proses *import* data. Berikutnya untuk melakukan proses *import* data dari *file* RDF dapat dilakukan dengan menjalankan Cypher *query* pada tampilan basis data Neo4j. Pada Kode Sumber 4.3 merupakan *query* yang digunakan untuk *import* data RDF menggunakan *plugin* Neosemantiks.

```
1.  CALL semantiks.importRDF("file:///home/nafiar/Document
    s/TA/20fr.openfoodfacts.org.products.rdf", "RDF/XML",
    {
2.      shortenUrls: false,
3.      typesToLabels: true,
4.      commitSize: 25000
5.  })
```

Kode Sumber 4.3 *Query* Cypher untuk *import* data

Setelah *query* tersebut dijalankan maka akan muncul tampilan seperti pada Gambar 4.2. Gambar 4.2 merupakan hasil yang ditampilkan setelah memproses *query* Cypher untuk *import* data.



Gambar 4.2 Tampilan pada Neo4j setelah menjalankan *query* *import* data

Pada Gambar 4.2 menandakan bahwa jumlah data *triples* yang dimasukkan dari *file* RDF adalah 4.530.744 *triples* dan berhasil disimpan sebagai *nodes* ke dalam basis data dengan jumlah *nodes* 1.120.073.

4.3 Cara Pemembuat *Index* pada Basis Data Graf Neo4j

Untuk mengoptimalkan performa basis data graf yang akan digunakan maka akan dibuat *indexing* untuk meningkatkan performa basis data dalam menjawab *query* maupun *update*. *Plugin* neosemantiks sendiri menuntut pembuatan *index property uri* pada *node* dengan label *Resource*. Hal ini dikarenakan untuk melakukan *query* pada SPARQL untuk mengakses sebuah *subject* yang berupa *resource* ataupun *object* yang berupa *resource*, *property* pada sebuah *node* yang akan diperiksa adalah *property uri* nya. Pada Kode Sumber 4.4 merupakan sintaks *create index* yang akan dijalankan pada Neo4j untuk membuat *index* pada *node* dengan label :*Resource* dan memiliki *property "uri*".

```
1.  CREATE INDEX on :Resource(uri)
```

Kode Sumber 4.4 sintaks Neo4j untuk membuat *index*

4.4 Cara Memetakan Data RDF Menjadi Model *Labeled Property Graph*

Berdasarkan kondisi-kondisi yang sudah dibuat pada bagian perencanaan untuk mengubah bentuk SPARQL *query* menjadi Cypher *query*. Pada Kode Sumber 4.5 merupakan *pseudo code* berdasarkan program yang sudah dibuat.

```
1.  if(is Wildcard Query) {
2.     filterHandler()
3.     matchQueryKondisi5(subject,predicate,object)
4.  } else {
5.     if(objectIsVariable)  {
6.            addReturnVariable(object)
7.         if(subject IsVariable)  {
8.            addReturnVariable(object)
9.
    matchQueryKondisi7(subject,predicate,object)
10.        } else {
11.
    matchQueryKondisi6(subject,predicate,object)
12.        }
13.    } else if (predicateIsVariable) {
14.           addReturnVariable(predicate)
15.           if(!is Literal Object) {
16.                  if(subject IsVariable){
17.                         addReturnVariable(object)
18.
       matchQueryKonidisi8(subject,predicate,object)
19.                 } else {
20.
       matchQueryKondisi4(subject,predicate,object)
21.                 }
22.
23.           }
24.    } else if (subject IsVariable) {
25.           addReturnVariable(subject)
26.           if (is Literal Object) {
27.
       matchQueryKondisi1(subject,predicate,object)
28.           } else {
29.
       matchQueryKondisi2(subject,predicate,object)
30.           }
31.    }
32. }
```

Kode Sumber 4.5 *Pseudo code* program *convert query*

Kondisi-kondisi yang sudah dibuat dipetakan menjadi sintaks *if* pada program dengan memperhatikan variabel yang terdapat pada sebuah *triples statement.*

4.5 Cara Membuat Basis Data yang Dapat Menjawab SPARQL *D*engan *Wildcard Query*

Pada Kode Sumber 4.6 merupakan contoh SPARQL *query* dengan *wildcard* yang ingin dijawab pada penelitian ini.

```
1.  select ?s where {
2.      ?s food: name "Almond*".
3.  }
```

Kode Sumber 4.6 SPARQL *query* dengan *wildcard* yang diharapkan

Pada basis data graf dengan model *Triple Store*, tidak semua jenis basis data dapat menjawab SPARQL *query* dengan *wildcard*. Pada Apache Jena Fuseki *query wildcard* dapat dijawab dengan menggunakan sintaks *FILTER regex* yang dapat memeriksa *string* berdasarkan *regular expression*, namun fungsionalitas ini masih belum terpenuhi pada jenis basis data graf dengan model *Triple Store* lainnya. Berdasarkan contoh SPARQL *query* dengan *wildcard* pada Kode Sumber 4.9, agar Apache Jena Fuseki dapat menghasilkan *output* yang sesuai dengan *query* tersebut, maka *query* tresebut akan dijalankan menggunakan sintaks *FILTER regex* seperti yang dapat dilihat pada Kode Sumber 4.7.

```
1.    select ?s where {
2.        ?s food:name ?o .
3.        FILTER regex(?o, “^Almond”) .
4.    }
```

Kode Sumber 4.7 SPARQL *query* dengan *FILTER regex*

Pada basis data graf Neo4j sudah mendukung untuk menjawab *query* dengan *wildcard*. Kode Sumber 4.8 merupakan kondisi yang digunakan untuk megubah *query wildcard* SPARQL menjadi Cypher *query* yang dapat menjawab *wildcard*.

```
1.  MATCH(s: Resource)
2.  WHERE s.`http://data.lirmm.fr/ontologies/food#name` =
    ~"Almond.*"
3.  RETURN s.uri
```

Kode Sumber 4.8 Cypher *query* untuk menjawab *wildcard*

BAB V
PENGUJIAN

Pada bab ini akan dijelaskan pengujian dan analisis dari hasil pengujian yang telah dilakukan pada basis data yang sudah dibangun. Pengujian dilakukan untuk mengetahui performa dari masing-masing basis data graf terhadap *dataset* yang sudah disiapkan.

5.1 Skenario Pengujian

Terdapat 2 skenario pengujian, yaitu skenario pengujian yang dilakukan dalam mendapatkan hasil performa basis data dalam menghadapi operasi *query* pada masing-masing basis data yang akan disebut pengujian *query* dan mendapatkan hasil performa basis data terhadap *indexing* yang sudah diterapkan yang akan disebut pengujian *indexing*.

5.1.1 Skenario Pengujian *Query*

Pada tahap pengujian akan dilakukan pengujian dengan langkah-langkah sebagai berikut :

1. Melakukan *import* data RDF pada basis data Neo4j dan Apache Jena Fuseki.
2. Melakukan Cypher *query* pada basis data Neo4j, dan SPARQL *query* pada basis data Apache Jena Fuseki.
3. Melakukan *restart* Apache Jena Fuseki dan Neo4j, kemudian menghapus *cache* yang sudah ada sebelumnya.
4. Melakukan Cypher *query* pada basis data Neo4j melalui program Java, dan Sparql *query* pada basis data Apache Jena Fuseki melalui program Java.

Dalam perngujian yang akan dilakukan, data yang digunakan adalah data RDF yang didapatkan dari *open food facts* yang berisikan informasi-informasi dari produk makan yang ada

di seluruh dunia. Seperti yang dijelaskan pada bagian perancangan pengujian, data RDF ini dibagi menjadi beberapa *file*, pemotongan dilakukan berdasrkan baris kode XML dengan rincian sebagai berikut :

1. *File* 1 : data *open food facts* yang berisikan 4,5 juta *triples statement*.
2. *File* 2 : data *open food facts* yang Berisikan 4,2 juta *triples statement*.
3. *File* 3 : data *open food fact*s yang Berisikan 4,1 juta *triples statement*.
4. *File* 4 : data *open food facts* yang Berisikan 4,2 juta *triples statement*.
5. *File* 5 : data *open food facts* yang Berisikan 4,2 juta *triples statement*.
6. *File* 6 : data *open food facts* yang Berisikan 4,1 juta *triples statement*.

Dalam pengujian performa basis data, akan digunakan 9 bentuk SPARQL *query*. Pada *query* 1 sampai dengan *query* 7 merupakan bentuk *query* yang digunakan untuk memastikan bahwa kondisi-kondisi untuk memetakan SPARQL *query* menjadi Cypher sudah benar. Sedangkan untuk *query* 8 dan *query* 9 digunakan untuk memeriksa performa masing-masing basis data terhadap *query* yang lebih kompleks. berikut merupakan SPARQL *query* yang akan digunakan :

1. *Query* 1

Query untuk mendapatkan *URI resource* dari produk makanan yang memiliki nama “Almondmilk”, dapat dilihat pada Kode Sumber 5.1.

```
1.  PREFIX food: <http: //data.lirmm.fr/ontologies/food#>

2.  select ?s where {
3.      ?s food:name "Almondmilk".
4.  }
```

Kode Sumber 5.1 SPARQL *query* 1.

2. *Query* 2

Query untuk mendapatkan *URI resource* dari sebuah produk makanan dengan nama menggunakan *string wildcard* "Almond*", dapat dilihat pada Kode Sumber 5.2.

```
1.  PREFIX food: <http://data.lirmm.fr/ontologies/food#>
2.
3.  select ?s where {
4.      ?s food:name ?o .
5.      FILTER regex(?o, "^Almond") .
6.  }
```

Kode Sumber 5.2 SPARQL *query* 2.

3. *Query* 3

Query untuk mendapatkan *URI resource* dari sebuah makanan yang memiliki data *property* "*containsIngredient*" dengan string *URI resource* dari tomat, dapat dilihat pada Kode Sumber 5.3.

```
1.  PREFIX food: <http://data.lirmm.fr/ontologies/food#>
2.
3.  select ?s where {
4.  ?s food: containsIngredient <http://fr.openfoodfacts.o
    rg/ingredient/tomatoes> .
5.  }
```

Kode Sumber 5.3 SPARQL *query* 3.

4. *Query* 4

Query yang untuk mendapatkan tipe *class* dari *URI resource dataset*, dapat dilihat pada Kode Sumber 5.4.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX void: <http://rdfs.org/ns/void#>

select ?s where {
    ?s rdf:type <http://rdfs.org/ns/void#Dataset> .
}
```

Kode Sumber 5.4 SPARQL *query* 4

5. *Query* 5

Query untuk mendapatkan *predicate* / *relationship* dari *URI resource solid dark chocolate fannie may* dengan *URI resource vanilla*, dapat dilihat pada Kode Sumber 5.5.

```
PREFIX food: <http://data.lirmm.fr/ontologies/food#>

select ?p where {
   <http://world-fr.openfoodfacts.org/produit/0052745953028/solid-dark-chocolate-fannie-may> ?p <http://fr.openfoodfacts.org/ingredient/vanilla> .
}
```

Kode Sumber 5.5 SPARQL *query* 5

6. *Query* 6

Query untuk mendapatkan bahan makanan apa saja yang terdapat pada *gourmet pizza sauce caputo*, dapat dilihat pada Kode Sumber 5.6.

```
PREFIX food: <http://data.lirmm.fr/ontologies/food#>

select ?o where {
    <http://world-fr.openfoodfacts.org/produit/0010096752660/gourmet-pizza-sauce-caputo> food:containsIngredient ?o .
}
```

Kode Sumber 5.6 SPARQL *query* 6

7. *Query* 7

Query untuk mendapatkan *subject* dan *predicate* dari *triples* yang memiliki *object* dengan *string URI resource* dari *onion*, dapat dilihat pada Kode Sumber 5.7.

```
1.  PREFIX food: < http: //data.lirmm.fr/ontologies/food#>
2.
3.  select ?s ?p where {
4.  ?s ?p <http://fr.openfoodfacts.org/ingredient/onions>
    .
5.  }
```

Kode Sumber 5.7 SPARQL *query* 7

8. *Query* 8

Query untuk mendapatkan *URI resource* dari sebuah produk makanan yang memiliki hubungan *containsIngredient* dengan makanan yang memiliki *URI resource* dari oregano dan mendapatkan *string uri* dari *node* intermediate antar kedua *resource* tersebut, dapat dilihat pada Kode Sumber 5.8.

```
1.  PREFIX food: <http://data.lirmm.fr/ontologies/food#>
2.
3.  select ?x ?y where {
4.  ?x food:containsIngredient ?y.
5.
    ?y food:food <http://fr.openfoodfacts.org/ingredient/o
    regano> .
6.  }
```

Kode Sumber 5.8 SPARQL *query* 8

9. *Query* 9 :

Query yang sama dengan *query* 8 dengan penambahan kondisi agar hasil yang didapatkan lebih sedikit dengan tambahan

kondisi kandungan protein per 100g berupa “9.55”, dapat dilihat pada Kode Sumber 5.9.

```
1.  PREFIX food: <http: //data.lirmm.fr/ontologies/food#>
2.
3.  select ?x ?y where {
4.  ?x food:containsIngredient ?y.
5.  ?y food:food <http://fr.openfoodfacts.org/ingredient/oregano> .
6.  ?x food:proteinsPer100g "9.55".
7.  }
```

Kode Sumber 5.9 SPARQL *query* 9

Sebelum *query* dijalankan dimasing-masing basis data, untuk bentuk Cypher *query* didapatkan dengan mengubah bentuk sintaks SPARQL *query* menggunakan program yang sudah dibuat. Berikut merupakan bentuk Cypher *query* yang akan dihasilkan berdasarkan urutan SPARQL *query* sebelumnya :

1. *Query* 1 :

Kode Sumber 5.10 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 1.

```
1.  MATCH
    (s:Resource { `http://data.lirmm.fr/ontologies/food#name`: "Almondmilk"})
2.  RETURN s.uri
```

Kode Sumber 5.10 Cypher *query* 1.

2. *Query* 2

Kode Sumber 5.11 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 2.

```
1.  MATCH(s: Resource)
2.  WHERE s.`http://data.lirmm.fr/ontologies/food#name` =
    ~"Almond.*"
3.  RETURN s.uri
```

Kode Sumber 5.11 Cypher *query* 2.

3. *Query* 3 :

Kode Sumber 5.12 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 3.

```
1. MATCH (s)-
   [:`http://data.lirmm.fr/ontologies/food#containsIngred
   ient`]-
   (:Resource { uri: "http://fr.openfoodfacts.org/ingredi
   ent/tomatoes" })
2. RETURN s.uri
```

Kode Sumber 5.12 Cypher *query* 3.

4. *Query* 4 :

Kode Sumber 5.13 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 4.

```
1. MATCH(s: `http://rdfs.org/ns/void#Dataset`)
2. RETURN s.uri
```

Kode Sumber 5.13 Cypher *query* 4.

5. *Query* 5 :

Kode Sumber 5.14 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 5.

```
1. MATCH(: Resource { uri: "http://world-
   fr.openfoodfacts.org/produit/0052745953028/solid-dark-
   chocolate-fannie-may"})-[p]-
   (:Resource { uri: "http://fr.openfoodfacts.org/ingredi
   ent/vanilla" })
2. RETURN type(p)
```

Kode Sumber 5.14 Cypher *query* 5.

6. *Query* 6 :

Kode Sumber 5.15 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 6.

```
MATCH(s: Resource { uri: "http://world-fr.openfoodfacts.org/produit/0010096752660/gourmet-pizza-sauce-caputo" })
WHERE s.`http://data.lirmm.fr/ontologies/food#containsIngredient` is not null
RETURN s.`http://data.lirmm.fr/ontologies/food#containsIngredient` as o
UNION ALL MATCH(:Resource {  uri: "http://world-fr.openfoodfacts.org/produit/0010096752660/gourmet-pizza-sauce-caputo"})-[:`http://data.lirmm.fr/ontologies/food#containsIngredient`]-(o)
RETURN o.uri as o
```

Kode Sumber 5.15 Cypher *query* 6.

7. *Query* 7 :

Kode Sumber 5.16 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 7.

```
MATCH(s: Resource { uri: "http://world-fr.openfoodfacts.org/produit/0010096752660/gourmet-pizza-sauce-caputo" })
WHERE s.`http://data.lirmm.fr/ontologies/food#containsIngredient` is not null
RETURN s.`http://data.lirmm.fr/ontologies/food#containsIngredient` as o
UNION ALL MATCH(:Resource {  uri: "http://world-fr.openfoodfacts.org/produit/0010096752660/gourmet-pizza-sauce-caputo"})-[:`http://data.lirmm.fr/ontologies/food#containsIngredient`]-(o)
RETURN o.uri as o
```

Kode Sumber 5.16 Cypher *query* 7.

8. *Query* 8 :

Kode Sumber 5.17 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 8.

```
1.  MATCH (x)-
    [:`http://data.lirmm.fr/ontologies/food#containsIngred
    ient`]-(y), (y)-
    [: `http://data.lirmm.fr/ontologies/food#food`]-
    (: Resource { uri: "http://fr.openfoodfacts.org/ingred
    ient/oregano" })
2.  RETURN x.uri as x, y.uri as y
```

Kode Sumber 5.17 Cypher *query* 8.

9. *Query* 9 :

Kode Sumber 5.18 merupakan bentuk Cypher *query* yang akan didapatkan dari mengubah bentuk SPARQL *query* 9.

```
1.  MATCH (x) -
     [: `http://data.lirmm.fr/ontologies/food#containsIngr
    edient`]-(y), (y)-
    [: `http://data.lirmm.fr/ontologies/food#food`]-
    (: Resource { uri: "http://fr.openfoodfacts.org/ingred
    ient/oregano" }), (x:Resource { `http://data.lirmm.fr/
    ontologies/food#proteinsPer100g`: "9.55" })
2.  RETURN x.uri as x, y.uri as y
```

Kode Sumber 5.18 Cypher *query* 9.

Setelah SPARQL *query* dijalankan di masing-masing basis data, hasil yang akan didapatkan berupa :

1. *Running time*.
2. Kecocokan hasil *query*.

5.1.2 Skenario Pengujian *Indexing*

Pada tahap pengujian akan dilakukan pengujian dengan langkah-langkah sebagai berikut :

1. Menyiapkan data RDF pada basis data Neo4j.
2. Melakukan *query* menggunakan *index*.
3. Melakukan *restart* Neo4j, kemudian menghapus *cache* yang sudah ada sebelumnya.
4. Melakukan *query* tanpa menggunakan *index*.

5. Melakukan Apache Jena Fuseki dan Neo4j, kemudian menghapus *cache* yang sudah ada sebelumnya.
6. Melakukan *update* tanpa menggunakan *index*.
7. Melakukan *restart* Jena Fuseki dan Neo4j, kemudian menghapus *cache* yang sudah ada sebelumnya.
8. Melakukan *update* menggunakan *index*.

Pada pengujian ini data yang akan digunakan adalah 6 *file* RDF yang digunakan pada pengujian q*uery* 6. Selain itu pada pengujian ini akan dijalankan 1 bentuk *query* pada Neo4j dan Apache Jena Fuseki. Bentuk *query* Cypher yang akan dijalankan pada Neo4j dapat dilihat pada Kode Sumber 5.19.

```
1. PROFILE MATCH(: Resource { uri: "http://world-
   fr.openfoodfacts.org/produit/0052745953028/solid-dark-
   chocolate-fannie-may" })-[p]-
   (:Resource { uri: "http://fr.openfoodfacts.org/ingredi
   ent/vanilla" })
2. RETURN type(p)
```

Kode Sumber 5.19 Cypher *query* pada pengujian *indexing*.

Pada pengujian ini dalam menguji *update* akan dijalankan 1 bentuk *query update* pada Neo4j. Bentuk *query update* yang akan dijalankan pada Neo4j dapat dilihat pada Kode Sumber 5.21.

```
1. PROFILE MATCH(s: Resource { uri: "http://world-
   fr.openfoodfacts.org/produit/0052745953028/solid-dark-
   chocolate-fannie-may" })
2. SET s.uri = "solid-dark-chocolate-fannie-may"
3. RETURN s.uri
```

Kode Sumber 5.20 Cypher *update query* pada pengujian *indexing*.

5.2 Pengujian *Query*

Pada tahap pengujian, akan dilakukan empat kali pengujian dengan data yang berbeda dari setiap pengujiannya.

Pada tahap ini berisi hasil dari setiap pengujian yang sudah dilakukan.

5.2.1 Pengujian 1

Pada skenario pengujian 1 digunakan data RDF dengan *triples* yang berjumlah **4.530.769**, dan hasil setelah dimasukkan pada Neo4j menghasilkan **1.012.397** *nodes* dengan **1.772.995** *relationship*. Pada pengujian ini, data waktu hasil pengujian terhadap setiap *query* dapat dilihat pada Tabel 5.1.

Tabel 5.1 Tabel Hasil Pengujian *Query* 1

Query	Query Langsung			Menggunakan Program		
	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)
1	12	**19,8**	80	12	**10,2**	600,2
2	267	**34,6**	1273,8	267	**7,4**	1758,2
3	9	**17,4**	77,6	9	**10,6**	612,6
4	1	**20,6**	73,2	1	**9,4**	677,2
5	1	**17,6**	76,6	1	**11**	609,2
6	15	**15,6**	80	15	**12**	598,6
7	1341	**14**	583,2	1341	**7**	1126,6
8	442	**8,8**	519,4	442	**13,4**	1153,4
9	2	**11**	366,4	2	**10**	849,6

Data waktu *query* pada masing-masing basis data merupakan rata-rata yang didapatkan dengan 5 kali *query* yang dijalankan pada masing-masing basis data. Data hasil pengujian *query* 1 secara lengkap ditampilkan pada Lampiran A.1. untuk *query* langsung dan Lampiran A.2. untuk *query* menggunakan program.

5.2.2 Pengujian 2

Pada skenario pengujian 2 digunakan data RDF dengan *triples* yang berjumlah **8.758.450**, dan hasil setelah dimasukkan pada Neo4j menghasilkan **1.957.987** *nodes* dan **3.419.409** *relationship*. Pada pengujian ini, data waktu hasil pengujian terhadap setiap *query* dapat dilihat pada Tabel 5.2.

Tabel 5.2 Tabel Hasil Pengujian *Query* 2

Query	Query Langsung			Menggunakan Program		
	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)
1	18	**20,8**	99,8	18	**8,8**	612,4
2	452	**15,8**	1833,6	452	**9**	2233,2
3	9	**11**	83,8	9	**9,8**	636,8
4	1	**22,4**	83,6	1	**11,6**	652,2
5	1	**17,6**	84	1	**9,2**	650,4
6	15	**14,6**	87,4	15	**9,6**	647,8
7	2777	**14,4**	719,6	2777	**7,8**	1502,4
8	877	**8,6**	858,4	877	**10,8**	1543,2
9	2	**12,8**	614,8	2	**10**	1107,2

Data waktu *query* pada masing-masing basis data merupakan rata-rata yang didapatkan dengan 5 kali *query* yang dijalankan pada masing-masing basis data. Data hasil pengujian *query* 2 secara lengkap ditampilkan pada Lampiran A.3. untuk *query* langsung dan Lampiran A.4. untuk *query* menggunakan program.

5.2.3 Pengujian 3

Pada skenario pengujian 3 digunakan data RDF dengan *triples* yang berjumlah **12.879.619**, dan hasil setelah dimasukkan pada Neo4j menghasilkan **2.916.035** *nodes* dan **5.005.011**

relationship. Pada pengujian ini, data waktu hasil pengujian terhadap setiap *query* dapat dilihat pada Tabel 5.3.

Tabel 5.3 Tabel Hasil Pengujian *Query* 3

Query	Query Langsung			Menggunakan Program		
	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)
1	22	**21,6**	96,8	22	**11**	622,6
2	605	**16,6**	2453,4	605	**9,4**	2843
3	9	**16,2**	76,2	9	**7,8**	593,2
4	1	**21**	79	1	**9**	640,4
5	1	**18**	79,6	1	**8,4**	583,4
6	15	**13**	83,6	15	**9,8**	588,6
7	4128	**15,2**	873	4128	**10,2**	1699,6
8	1279	**8,4**	944,2	1279	**7,8**	1769
9	3	**9,2**	840	3	**11,4**	1294,6

Data waktu *query* pada masing-masing basis data merupakan rata-rata yang didapatkan dengan 5 kali *query* yang dijalankan pada masing-masing basis data. Data hasil pengujian *query* 3 secara lengkap ditampilkan pada Lampiran A.5. untuk *query* langsung dan Lampiran A.6. untuk *query* menggunakan program.

5.2.4 Pengujian 4

Pada skenario pengujian 4 digunakan data RDF dengan *triples* yang berjumlah **17.135.424**, dan hasil setelah dimasukkan pada Neo4j menghasilkan **3.996.689** *nodes* dan **6.718.639** *relationship*. Pada pengujian ini, data waktu hasil pengujian terhadap setiap *query* dapat dilihat pada Tabel 5.4.

Tabel 5.4 Tabel Hasil Pengujian *Query* 4

Query	Query Langsung			Menggunakan Program		
	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)
1	22	**21**	96,8	22	**10,2**	632,8
2	607	**17,4**	3380,6	607	**11,6**	3810,4
3	9	**12,4**	92,4	9	**8,4**	616,6
4	1	**20,8**	81,2	1	**9,6**	667,8
5	1	**15,4**	86,6	1	**10,8**	626,6
6	15	**13**	88,8	15	**7,4**	617,2
7	4144	**16**	939,2	4144	**10,6**	1889,4
8	1295	**10,6**	1014,8	1295	**7,6**	1846,2
9	3	**16,6**	908,6	3	**7,2**	1444,4

Data waktu *query* pada masing-masing basis data merupakan rata-rata yang didapatkan dengan 5 kali *query* yang dijalankan pada masing-masing basis data. Data hasil pengujian *query* 4 secara lengkap ditampilkan pada Lampiran A.7. untuk *query* langsung dan Lampiran A.8. untuk *query* menggunakan program.

5.2.5 Pengujian 5

Pada skenario pengujian 5 digunakan data RDF dengan *triples* yang berjumlah **21.416.071**, dan hasil setelah dimasukkan pada Neo4j menghasilkan **3.996.689** *nodes* dan **8.364.047** *relationship*. Pada pengujian ini, data waktu hasil pengujian terhadap setiap *query* dapat dilihat pada Tabel 5.5.

Tabel 5.5 Tabel Hasil Pengujian *Query* 5

Query	Query Langsung			Menggunakan Program		
	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)
1	22	**24,4**	92	22	**9**	587
2	612	**15,2**	4108,6	612	**9,8**	4434,2
3	9	**14,6**	82	9	**9,4**	630,8
4	1	**20,2**	73	1	**9,6**	630,2
5	1	**14**	71,6	1	**10,2**	603,4
6	15	**15,6**	77,8	15	**7,6**	646,6
7	4171	**13,6**	887,4	4171	**9,6**	1726,2
8	1357	**8,6**	1019,2	1357	**8,4**	1804,8
9	3	**10,8**	972,8	3	**7,6**	1373,6

Data waktu *query* pada masing-masing basis data merupakan rata-rata yang didapatkan dengan 5 kali *query* yang dijalankan pada masing-masing basis data. Data hasil pengujian *query* 5 secara lengkap ditampilkan pada Lampiran A.9. untuk *query* langsung dan Lampiran A.10. untuk *query* menggunakan program.

5.2.6 Pengujian 6

Pada skenario pengujian 6 digunakan data RDF dengan *triples* yang berjumlah **25.555.835**, dan hasil setelah dimasukkan pada Neo4j menghasilkan **6.275.234** *nodes*. Pada pengujiaan ini, data waktu hasil pengujian terhadap setiap *query* dapat dilihat pada Tabel 5.6.

Tabel 5.6 Tabel Hasil Pengujian *Query* 6

Query	Query Langsung			Menggunakan Program		
	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)	Jumlah record	Neo4j (ms)	Apache Jena Fuseki (ms)
1	23	**26,4**	92,8	23	**8,8**	652,4
2	665	**16,2**	4943,6	665	**8**	5549,6
3	9	**30,2**	86,8	9	**8,6**	637
4	1	**22,8**	78,8	1	**9,4**	787,2
5	1	**16,4**	90	1	**11,2**	615,2
6	15	**14,2**	83,2	15	**7,8**	638
7	4282	**15,6**	910,6	4282	**8,8**	1807,6
8	1550	**14,8**	1231,8	1550	**10,4**	2021,2
9	3	**12,8**	1108,6	3	**9,4**	1525,4

Data waktu *query* pada masing-masing basis data merupakan rata-rata yang didapatkan dengan 5 kali *query* yang dijalankan pada masing-masing basis data. Data hasil pengujian *query* 6 secara lengkap ditampilkan pada Lampiran A.11. untuk *query* langsung dan Lampiran A.12. untuk *query* menggunakan program.

5.3 Pengujian *Indexing*

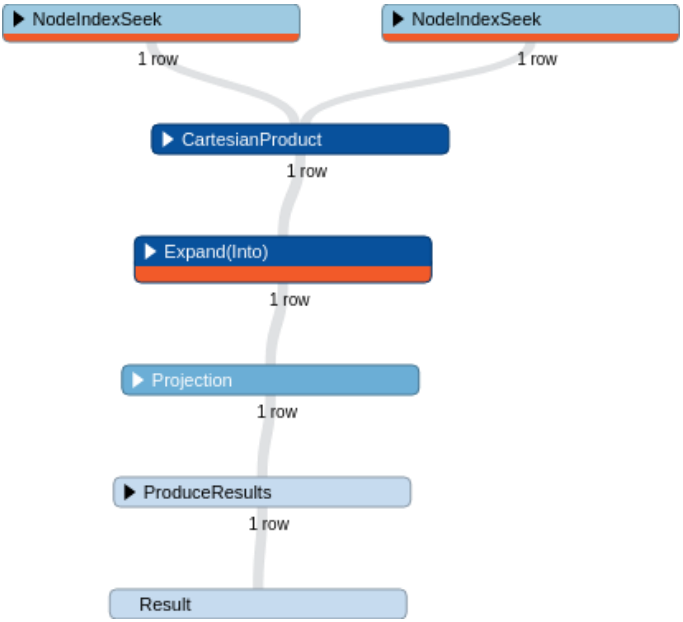
Pada tahap pengujian ini akan dihasilkan 2 data dari masing-masing *query*, yaitu data DB *Hits* yang merupakan data jumlah operasi-operasi dasar dalam basis data yang dijalankan dalam melakukan *query*, dan data waktu *query* tersebut dijalankan. Pada Tabel 5.7 merupakan hasil dari pengujian *indexing* pada *query* maupun *update*.

Tabel 5.7 Tabel Hasil Pengujian *indexing*

Pengujian	DB Hits	Waktu (ms)
Query dengan index	**21**	453,8
Query tanpa index	**12288600**	10479
Update dengan index	**5**	254,8
Update tanpa index	**12288565**	11796,2

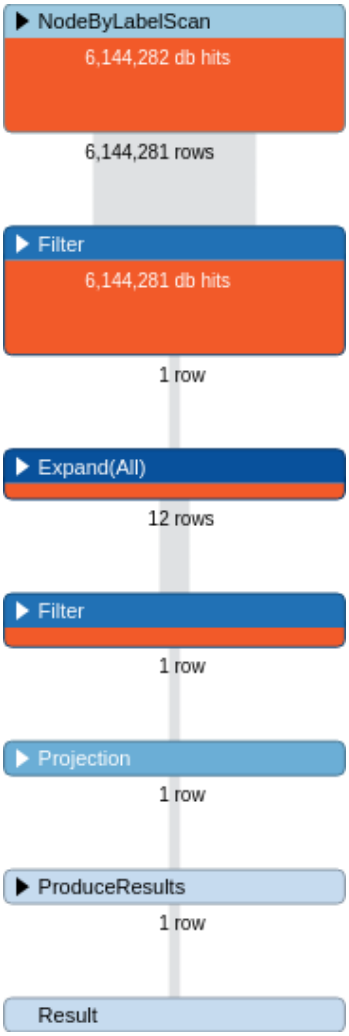
Waktu yang ditampilkan pada Tabel 5.7 merupakan rata-rata dari waktu 5 kali pengujian yang dilakukan pada masing-masing *query*. Hasil pengujian indexing secara lengkap dapat dilihat pada Lampiran B.

Dari *query* pada Kode Sumber 5.19 yang dijalankan dengan menggunakan *index*, maka akan menghasilkan *query plan* seperti yang dapat dilihat pada Gambar 5.1



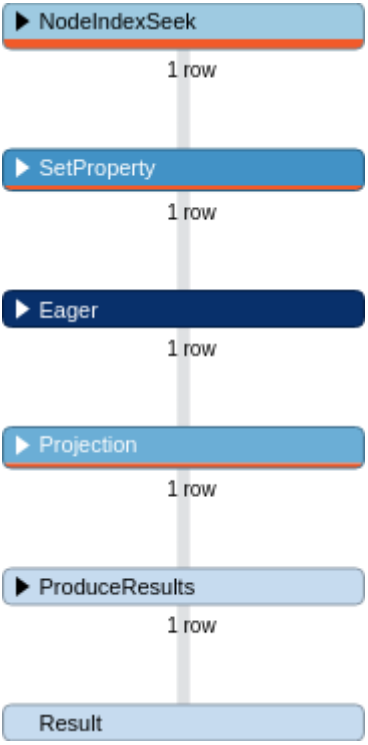
Gambar 5.1 *Query plan* pada *query* dengan *index*

Kemudian dari *query* yang sama pada Kode Sumber 5.19 dijalankan tanpa menggunakan *index*, maka akan menghasilkan *query plan* seperti yang dapat dilihat pada Gambar 5.2



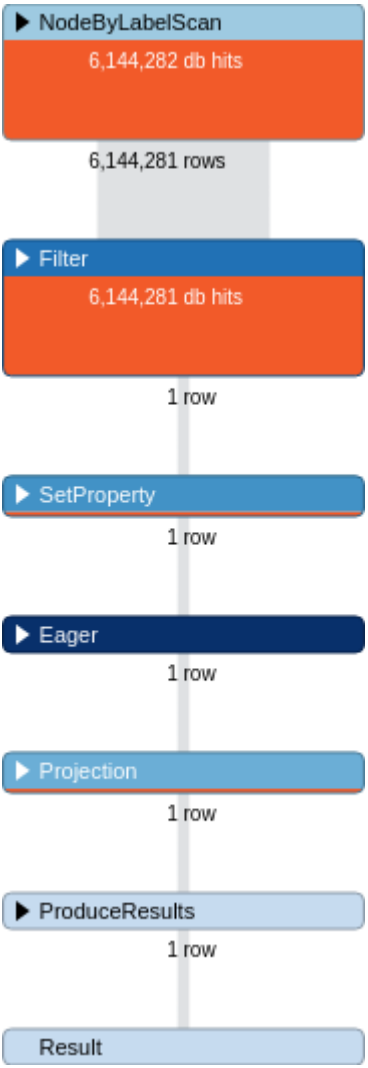
Gambar 5.2 *Query plan* pada *query* tanpa *index*

Dari *update query* pada Kode Sumber 5.20 yang dijalankan dengan menggunakan *index*, maka akan menghasilkan *query plan* seperti yang dapat dilihat pada Gambar 5.3.



Gambar 5.3 *Query plan* pada *update query* dengan *index*

Kemudian dari *query* yang sama pada Kode Sumber 5.20 dijalankan tanpa menggunakan *index*, maka akan menghasilkan *query plan* seperti yang dapat dilihat pada Gambar 5.4.



Gambar 5.4 *Query plan* pada *update query* tanpa *index*

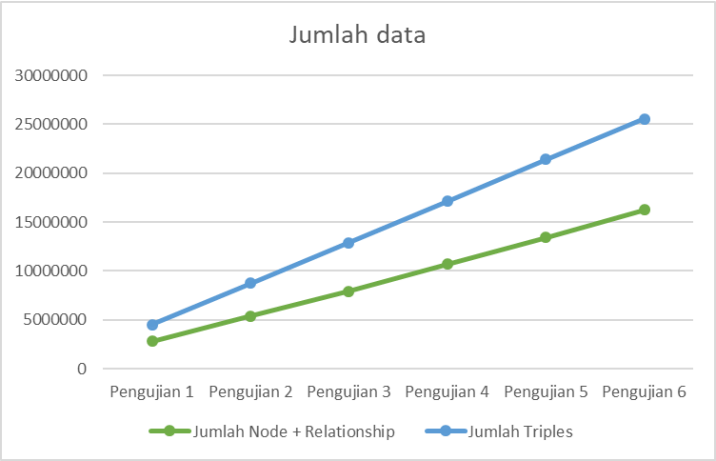
5.4 Analisis hasil pengujian

5.4.1 Hasil Pengujian *Query*

Berdasarkan Pengujian yang sudah dilakukan terbukti bahwa model *Labeled Property Graph* dapat mempercepat SPARQL *query*, hal ini karena model data yang terdapat pada basis data *graph* model *Labeled Property Graph* memangkas *literal object* yang dijadikan *node* dari *triples* untuk dijadikan *value* pada *property* yang terdapat pada sebuah *node*. Selain itu *object* juga dipetakan menjadi *node* bila nilai *object* berupa *URI* sehingga mengurangi jumlah *node* yang terdapat pada data graf. Tabel jumlah data pada Apache Jena Fuseki dan Neo4j pada setiap pengujian dapat dilihat pada Tabel 5.8 Grafik jumlah data dari setiap model data pada setiap pengujian dapat dilihat pada Gambar 5.5.

Tabel 5.8 Tabel Jumlah Data pada Setiap Pengujian

Pengujian	Jumlah *Node* + *Relationship*	Jumlah *Triples*	Jumlah *Node*	Jumlah *Relationship*
Pengujian 1	2785392	4530769	1012397	1772995
Pengujian 2	5377396	8758450	1957987	3419409
Pengujian 3	7921046	12879619	2916035	5005011
Pengujian 4	10715328	17135424	3996689	6718639
Pengujian 5	13441671	21416071	5077624	8364047
Pengujian 6	16234171	25555835	6275234	9958937

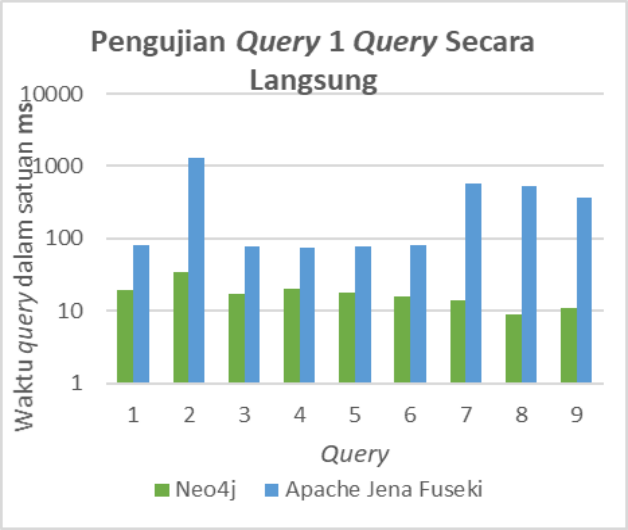


Gambar 5.5 Grafik Jumlah data

Pada pengujian *query* 1, kita dapat mengamati data waktu *running time* yang sudah didapatkan dari pengujian dengan *query* langsung pada masing-masing basis data seperti yang dapat dilihat pada Tabel 5.9. Pada Gambar 5.6 merupakan grafik hasil *running time* dari *query* yang dijalankan langsung pada masing-masing basis data pada pengujian 1, pada grafik ini waktu *running time* menggunakan skala log dengan basis 10.

Tabel 5.9 Tabel *running time* pada pengujian 1

Query	Neo4j	Apache Jena Fuseki
1	**19,8**	80
2	**34,6**	1273,8
3	**17,4**	77,6
4	**20,6**	73,2
5	**17,6**	76,6
6	**15,6**	80
7	**14**	583,2
8	**8,8**	519,4
9	**11**	366,4



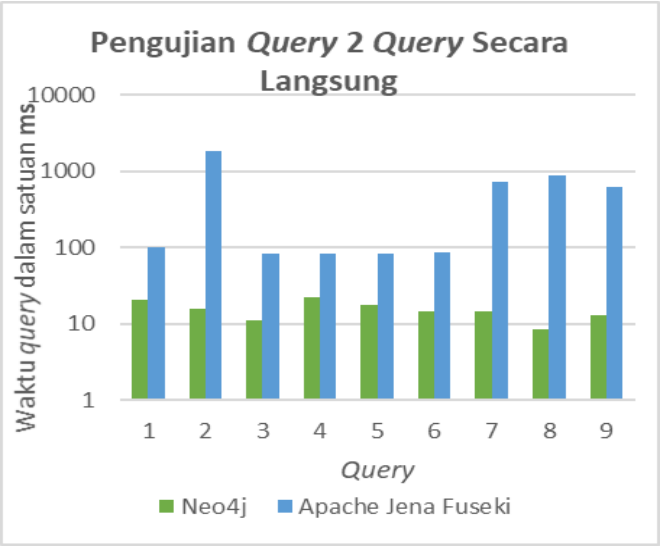
Gambar 5.6 Grafik hasil *running time* dari *query* yang dijalankan langsung pada pengujian 1

Pada pengujian *query* 2, kita dapat mengamati data waktu *running time* yang sudah didapatkan dari pengujian dengan *query*

langsung pada masing-masing basis data seperti yang dapat dilihat pada Tabel 5.10. Pada Gambar 5.7 merupakan grafik hasil *running time* dari *query* yang dijalankan langsung pada masing-masing basis data pada pengujian 2, pada grafik ini waktu *running time* menggunakan skala log dengan basis 10.

Tabel 5.10 Tabel *running time* pada pengujian 2

Query	Neo4j	Apache Jena Fuseki
1	**20,8**	99,8
2	**15,8**	1833,6
3	**11**	83,8
4	**22,4**	83,6
5	**17,6**	84
6	**14,6**	87,4
7	**14,4**	719,6
8	**8,6**	858,4
9	**12,8**	614,8

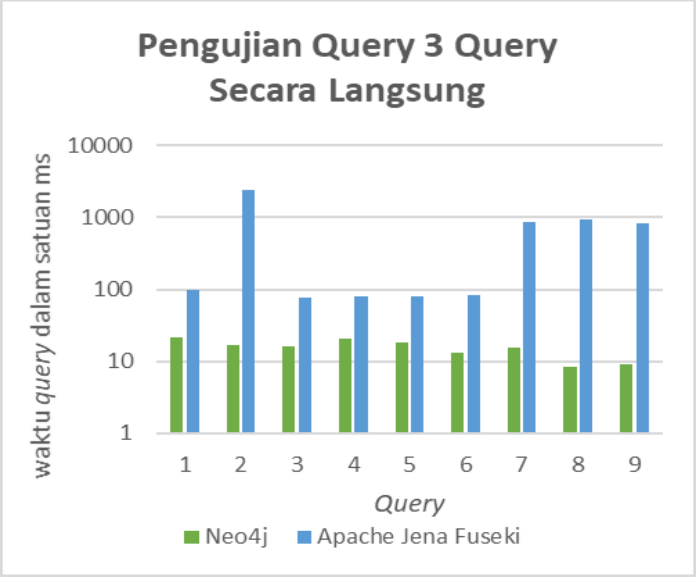


Gambar 5.7 Grafik hasil *running time* dari *query* yang dijalankan pada basis data pengujian 2

Pada pengujian *query* 3, kita dapat mengamati data waktu *running time* yang sudah didapatkan dari pengujian dengan *query* langsung pada masing-masing basis data seperti yang dapat dilihat pada Tabel 5.11. Pada Gambar 5.8 merupakan grafik hasil *running time* dari *query* yang dijalankan langsung pada masing-masing basis data pada pengujian 3, pada grafik ini waktu *running time* menggunakan skala log dengan basis 10.

Tabel 5.11 Tabel *running time* pada pengujian 3

Query	Neo4j	Apache Jena Fuseki
1	**21,6**	96,8
2	**16,6**	2453,4
3	**16,2**	76,2
4	**21**	79
5	**18**	79,6
6	**13**	83,6
7	**15,2**	873
8	**8,4**	944,2
9	**9,2**	840

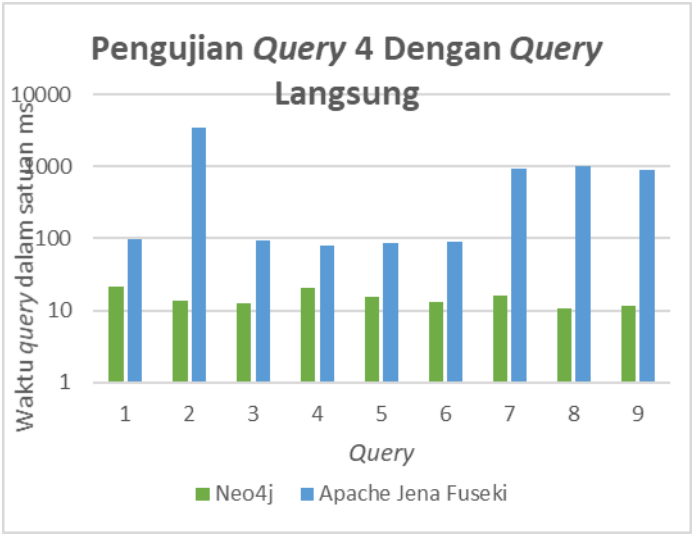


Gambar 5.8 Grafik hasil *running time* dari *query* yang dijalankan pada basis data pengujian 3

Pada pengujian *query* 4, kita dapat mengamati data waktu *running time* yang sudah didapatkan dari pengujian dengan *query* langsung pada masing-masing basis data seperti yang dapat dilihat pada Tabel 5.12. Pada Gambar 5.9 merupakan grafik hasil *running time* dari *query* yang dijalankan langsung pada masing-masing basis data pada pengujian 4, pada grafik ini waktu *running time* menggunakan skala log dengan basis 10.

Tabel 5.12 Tabel *running time* pada pengujian 4

Query	Neo4j	Apache Jena Fuseki
1	**21**	96,8
2	**13,8**	3445,2
3	**12,4**	92,4
4	**20,8**	81,2
5	**15,4**	86,6
6	**13**	88,8
7	**16**	939,2
8	**10,6**	1014,8
9	**11,8**	903

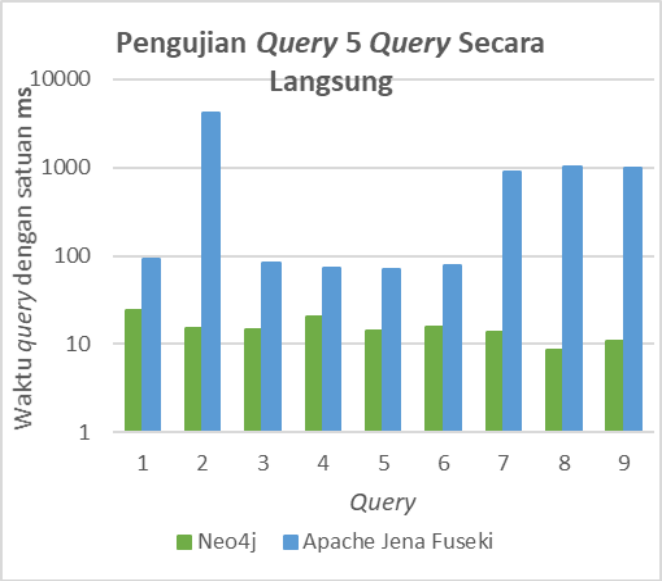


Gambar 5.9 Grafik hasil *running time* dari *query* yang dijalankan pada basis data pengujian 4

Pada pengujian *query* 5, kita dapat mengamati data waktu *running time* yang sudah didapatkan dari pengujian dengan *query* langsung pada masing-masing basis data seperti yang dapat dilihat pada Tabel 5.13. Pada Gambar 5.10 merupakan grafik hasil *running time* dari *query* yang dijalankan langsung pada masing-masing basis data pada pengujian 5, pada grafik ini waktu *running time* menggunakan skala log dengan basis 10.

Tabel 5.13 Tabel *running time* pada pengujian 5

Query	Neo4j	Apache Jena Fuseki
1	**24,4**	92
2	**15,2**	4108,6
3	**14,6**	82
4	**20,2**	73
5	**14**	71,6
6	**15,6**	77,8
7	**13,6**	887,4
8	**8,6**	1019,2
9	**10,8**	972,8

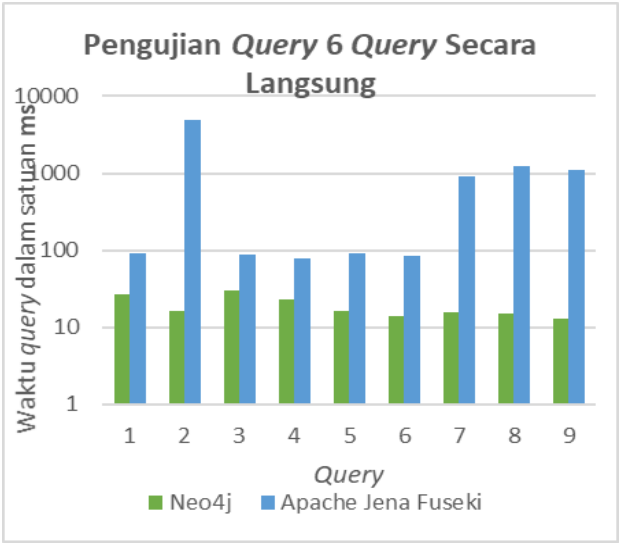


Gambar 5.10 Grafik hasil *running time* dari *query* yang dijalankan pada basis data pengujian 5

Pada pengujian *query* 6, kita dapat mengamati data waktu *running time* yang sudah didapatkan dari pengujian dengan *query* langsung pada masing-masing basis data seperti yang dapat dilihat pada Tabel 5.14. Pada Gambar 5.11 merupakan grafik hasil *running time* dari *query* yang dijalankan langsung pada masing-masing basis data pada pengujian 6, pada grafik ini waktu *running time* menggunakan skala log dengan basis 10.

Tabel 5.14 Tabel *running time* pada pengujian 6

Query	Neo4j	Apache Jena Fuseki
1	**26,4**	92,8
2	**16,2**	4943,6
3	**30,2**	86,8
4	**22,8**	78,8
5	**16,4**	90
6	**14,2**	83,2
7	**15,6**	910,6
8	**14,8**	1231,8
9	**12,8**	1108,6



Gambar 5.11 Grafik hasil *running time* dari *query* yang dijalankan pada basis data percobaan 6

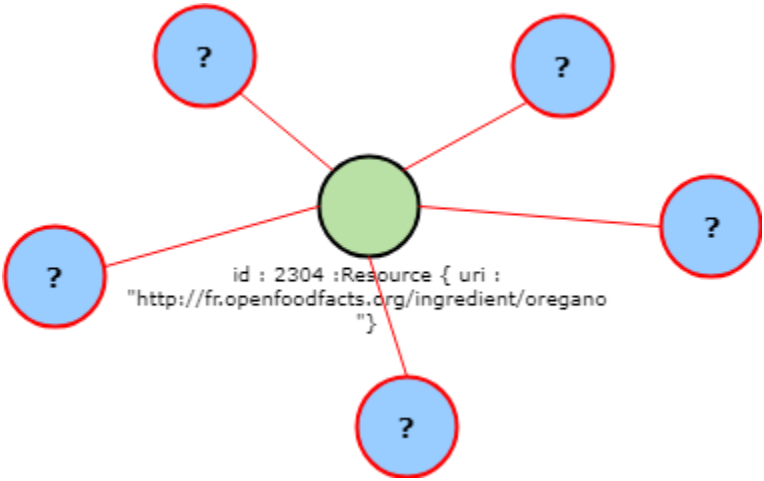
Pada Gambar 5.7, Gambar 5.8, Gambar 5.9, Gambar 5.10, Gambar 5.11 dapat dilihat bahwa hasil *running time* dari *query* yang dijalankan pada model *Labeled Property Graph* Neo4j memiliki perbedaan yang sangat signifikan bila dibandingkan dengan model *Triple Store* Apache Jena Fuseki. Perbedaan yang sangat signifikan terjadi pada *query* 2, 7, 8, dan 9.

Dari hasil 6 pengujian di atas, hal yang paling mencolok adalah perbedaan waktu pada *query* 2,7,8,9 pada setiap pengujian.

Pada *Query* 2, perbedaan *running time* yang signifikan dikarenakan *subject* dan *object* tidak diketahui pada SPARQL *query*, sehingga basis data dengan model *Triple Store* akan iterasi pada semua *triples* yang memiliki *predicate* tertentu, kemudian semua hasil *string object* nya diperiksa dengan *FILTER regex*. Pemeriksaan dengan *regex* ini memakan waktu yang sangat lama

karena pemeriksaan *string* terhadap *regular expression*. Pada model *Labeled Property Graph* akan berbeda karena *string object* yang diperiksa menggunakan sintaks *WHERE* dilakukan secara *on the fly* pada *node* yang memiliki *proerty* tertentu, bukan pada seluruh hasil yang didapatkan sebelumnya.

Pada *Query* 7 terjadi perbedaan *running time* yang singnifikan dikarenakan pada SPARQL *query subject* dan *predicate* tidak diketahui maka model *Triple Store* akan melakukan iterasi pemeriksaan seluruh data. Namun hal ini berbeda dengan model data *Labeled Property Graph* karena *string object* sudah diketahui dan berupa *URI* dari pada memeriksa seluruh *node* yang ada pada model ini, proses pemeriksaan data hanya dilakukan berdasarkan *node* yang memiliki *uri* berupa *string object* tersebut dan memeriksa *node* apa saja yang memiliki *relationship* dengan *node* tersebut. Seperti yang dapat dilihat pada Gambar 5.12, dimana pemeriksaan berawal dari *node* dengan warna hijau kemudian menampilkan *relationship* dan *node* yang dimiliki *node* tersesebut.

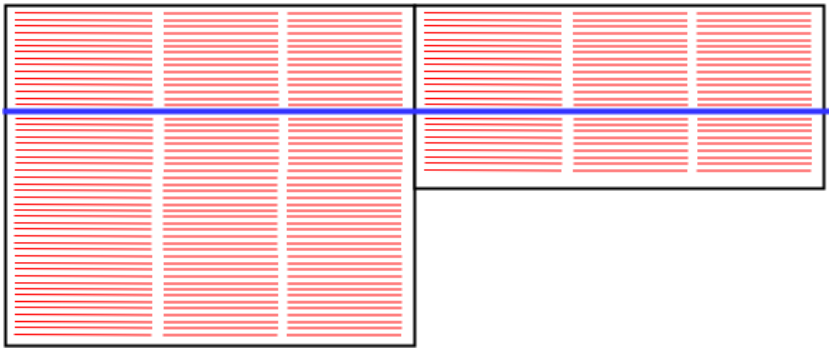


Gambar 5.12 Gambaran *traverse query* 7 pada Neo4j.

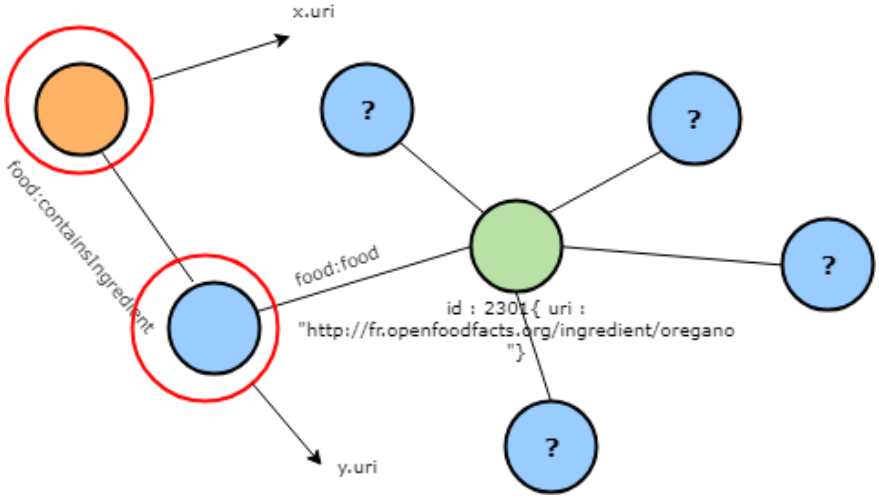
Gambar 5.13 Bentuk *traverse query* 8 yang diinginkan.

Query 8 memiliki bentuk *traverse* yang diinginkan seperti yang dapat dilihat pada Gambar 5.13. Pada *Query* 8 hal ini disebabkan karena pada SPARQL *query* ingin dicari *triples* yang memiliki *predicate* "*food*:*containsIngredient*" tanpa mengetahui *subject* dan *object* dari *predicate* tersebut. Hal ini menyebabkan model *Triple Store* akan melakukan pemeriksaan data dengan iterasi pada seluruh data yang ada. Selain itu pada data *Open Food Facts*, semua *resource* berupa data makanan yang mengandung *property* "*food*:*containsIngredient*". Selain itu, pada konidisi kedua akan dilakukan pemeriksaan data dengan memeriksa hasil pemeriksaan pada kondisi sebelumnya dengan kondisi " ?y *food*:*food* <http://fr.open*food*facts.org/ingredient/oregano>" sehingga akan dilakukan pemeriksaan data dengan iterasi seluruh *triples* pada hasil kondisi sebelumnya. Operasi ini seperti melakukan join yang digambarkan pada Gambar 5.14. Sedangkan pada Cypher *query* pemeriksaan dilakukan dengan memulai pencarian *node* yang memiliki *property uri* dengan nilai "http://fr.open*food*facts.org/ingredient/oregano", setelah *node* tersebut didapatkan baru akan dilakukan *traverse* ke *node* berikutnya yang memiliki *relationship* "*food*:*food*", bila *node* ditemukan maka *node* tersebut akan dijadikan hasil y, kemudian akan dilakukan *traverse* kembali pada *node* yang terhubung pada *node* y tersebut. Bila terdapat *node* dengan *relationship* "*food*:*containsIngredient*" maka *node* tersebut akan menjadi hasil dari variabel x. Hal ini seperti yang digambarkan pada Gambar

5.15, dimana pemeriksaan akan dimulai pada *node* dengan warna hijau.



Gambar 5.14 Join hasil 2 kondisi pada SPARQL *query* dari *query* 8 pada Apache Jena Fuseki.



Gambar 5.15 Bentuk *traverse query* Cypher dari *query* 8 pada Neo4j.

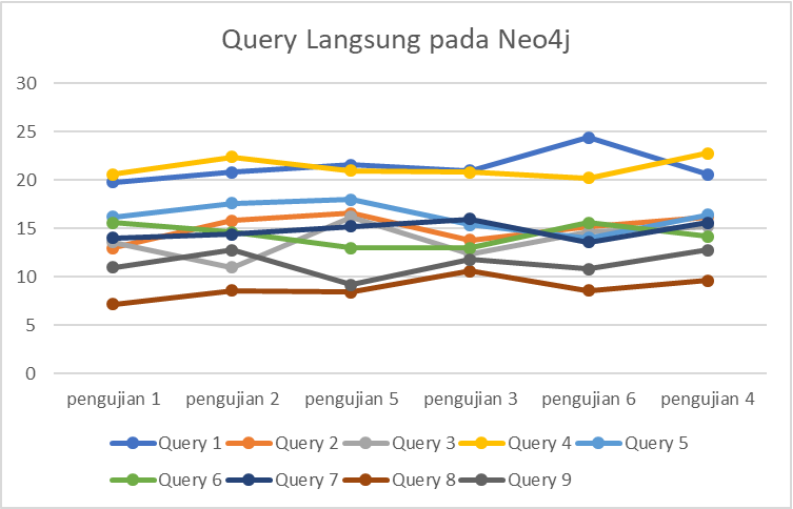
Pada *Query* 9 penyebab perbedaan waktu yang signifikan disebabkan karena 2 kondisi *traverse* pada SPARQL *query* masih sama seperti *Query* 8, namun yang membedakan pada *query* ini

terdapat kondisi tambahan sehingga dapat mengurangi jumlah *triples* yang diperiksa pada kondisi pertama dan kedua.

Data waktu *query* terhadap setiap pengujian dengan *query* yang dilakukan secara langsung pada Neo4j dapat dibuat menjadi tabel yang dapat dilihat pada Tabel 5.15. Dari tabel ini kita dapat melihat perkembangan waktu yang dibutuhkan untuk menjalankan setiap *query* apabila jumlah data pada Neo4j semakin bertambah. Grafik perkembangan waktu untuk menjalankan *query* berdasarkan setiap pengujian dapat dilihat pada Gambar 5.16.

Tabel 5.15 Tabel waktu *query* pada setiap pengujian dengan *query* secara langsung pada Neo4j.

Data pengujian	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
pengujian 1	19,8	13	13,6	20,6	16,2	15,6	14	7,2	11
pengujian 2	20,8	15,8	11	22,4	17,6	14,6	14,4	8,6	12,8
pengujian 5	21,6	16,6	16,2	21	18	13	15,2	8,4	9,2
pengujian 3	21	13,8	12,4	20,8	15,4	13	16	10,6	11,8
pengujian 6	24,4	15,2	14,6	20,2	14	15,6	13,6	8,6	10,8
pengujian 4	20,6	16,2	15,2	22,8	16,4	14,2	15,6	9,6	12,8

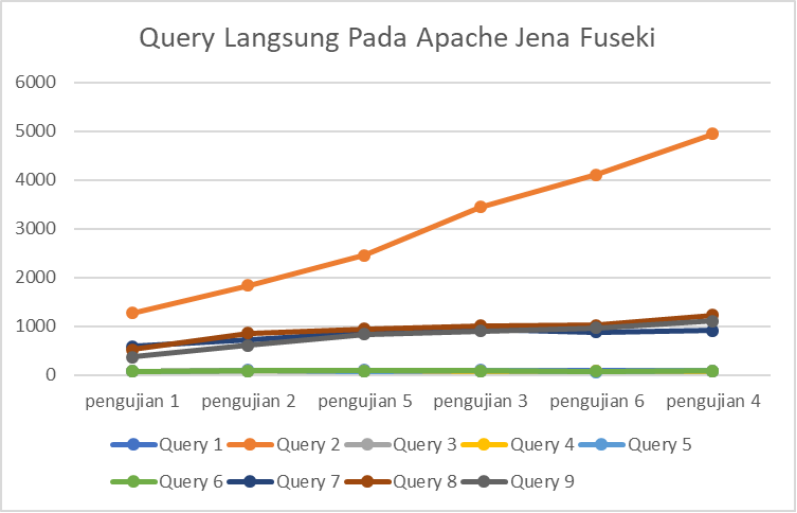


Gambar 5.16 Grafik hasil perkembangan waktu *query* dari setiap pengujian pada Neo4j.

Data waktu *query* terhadap setiap pengujian dengan *query* yang dilakukan secara langsung pada Apache Jena Fuseki dapat dibuat menjadi tabel yang dapat dilihat pada Tabel 5.16. Dari tabel ini kita dapat melihat perkembangan waktu yang dibutuhkan untuk menjalankan setiap *query* apabila jumlah data pada Apache Jena Fuseki semakin bertambah. Grafik perkembangan waktu untuk menjalankan *query* berdasarkan setiap pengujian dapat dilihat pada Gambar 5.17.

Tabel 5.16 Tabel waktu *query* pada setiap pengujian dengan *query* secara langsung pada Apache Jena Fuseki.

Data pengujian	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
pengujian 1	80	1273,8	77,6	73,2	76,6	80	583,2	519,4	366,4
pengujian 2	99,8	1833,6	83,8	83,6	84	87,4	719,6	858,4	614,8
pengujian 5	96,8	2453,4	76,2	79	79,6	83,6	873	944,2	840
pengujian 3	96,8	3445,2	92,4	81,2	86,6	88,8	939,2	1014,8	903
pengujian 6	92	4108,6	82	73	71,6	77,8	887,4	1019,2	972,8
pengujian 4	92,8	4943,6	86,2	78,8	90	83,2	910,6	1231,8	1108,6



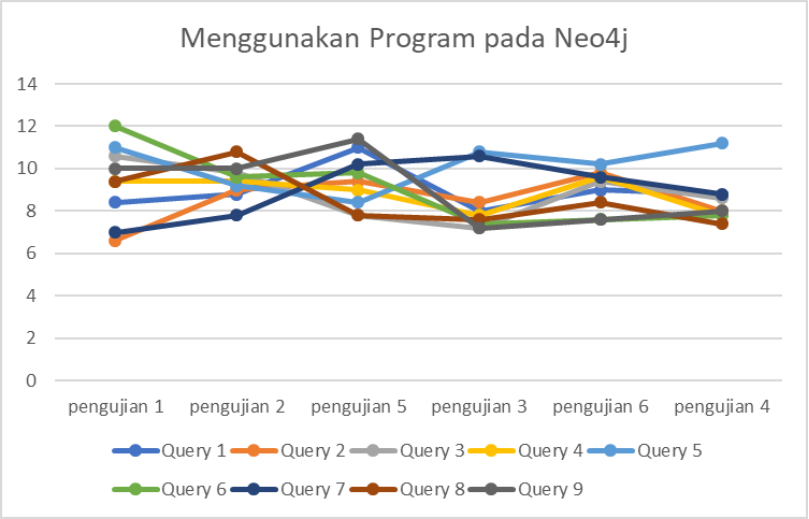
Gambar 5.17 Grafik hasil perkembangan waktu *query* dari setiap pengujian pada Apacje Jena Fuseki.

Data waktu *query* terhadap setiap pengujian dengan *query* yang dilakukan menggunakan program pada Neo4j dapat dibuat menjadi tabel yang dapat dilihat pada Tabel 5.17. Dari tabel ini kita dapat melihat perkembangan waktu yang dibutuhkan untuk menjalankan setiap *query* apabila jumlah data pada Neo4j

semakin bertambah. Grafik perkembangan waktu untuk menjalankan *query* berdasarkan setiap pengujian dapat dilihat pada Gambar 5.18.

Tabel 5.17 Tabel waktu *query* pada setiap pengujian dengan *query* menggunakan program pada Neo4j.

Data pengujian	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
pengujian 1	8,4	6,6	10,6	9,4	11	12	7	9,4	10
pengujian 2	8,8	9	9,8	9,4	9,2	9,6	7,8	10,8	10
pengujian 5	11	9,4	7,8	9	8,4	9,8	10,2	7,8	11,4
pengujian 3	8	8,4	7,2	7,8	10,8	7,4	10,6	7,6	7,2
pengujian 6	9	9,8	9,4	9,6	10,2	7,6	9,6	8,4	7,6
pengujian 4	8,8	8	8,6	7,8	11,2	7,8	8,8	7,4	8

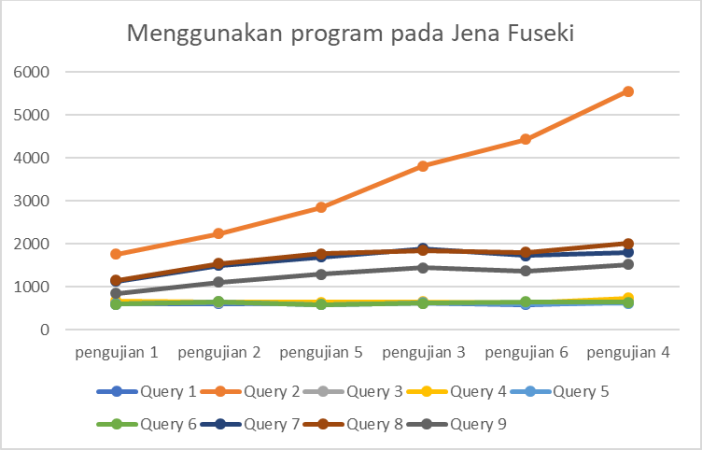


Gambar 5.18 Grafik hasil perkembangan waktu *query* dari setiap pengujian pada Neo4j menggunakan program.

Data waktu *query* terhadap setiap pengujian dengan *query* yang dilakukan menggunakan program pada Apache Jena Fuseki dapat dibuat menjadi tabel yang dapat dilihat pada Tabel 5.18. Dari tabel ini kita dapat melihat perkembangan waktu yang dibutuhkan untuk menjalankan setiap *query* apabila jumlah data pada Apache Jena Fuseki semakin bertambah. Grafik perkembangan waktu untuk menjalankan *query* berdasarkan setiap pengujian dapat dilihat pada Gambar 5.19.

Tabel 5.18 Tabel waktu *query* pada setiap pengujian dengan menggunakan program pada Apache Jena Fuseki.

Data pengujian	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
pengujian 1	600,2	1758,2	612,6	677,2	609,2	598,6	1126,6	1153,4	849,6
pengujian 2	612,4	2233,2	636,8	652,2	650,4	647,8	1502,4	1543,2	1107,2
pengujian 5	622,6	2843	593,2	640,4	583,4	588,6	1699,6	1769	1294,6
pengujian 3	626,6	3810,4	616,6	648,2	626,6	617,2	1889,4	1846,2	1444,4
pengujian 6	587	4434,2	630,8	630,2	603,4	646,6	1726,2	1804,8	1373,6
pengujian 4	652,4	5549,6	637	736	615,2	638	1807,6	2014,6	1518,2



Gambar 5.19 Grafik hasil perkembangan waktu *query* dari setiap pengujian pada Jena Fuseki menggunakan program.

Dari Gambar 5.17 dan Gambar 5.19 dapat dilihat bahwa pada setiap pengujian untuk *query* 2, *query* 7, *query* 8, dan *query* 9 pada Apache Jena Fuseki memiliki tren yang cenderung meningkat ketika jumlah data semakin bertambah, tren penikngkatan pada *query* 7, *query* 8, dan *query* 9 meningkat secara linier tidak terlalu signifikan sedang kan pada *query* 2 pengingkatan secara linier sangat signifikan. Hal ini menandakan secara normal tanpa optimasi maka waktu *query* yang akan dihasilkan akan cenderung meningkat seiring dengan meningkatnya jumlah data.

Selain itu berdasarkan Gambar 5.16 dan Gambar 5.17 dapat dilihat bahwa Cypher *query* yang dihasilkan dari proses convert memiliki data tren yang tidak cenderung meningkat, terdapat peningkatan dan juga penurunan waktu walaupun jumlah datanya semakin meningkat. Hal ini menandakan bahwa bentuk *traverse* graf yang dihasilkan oleh *converter* dapat menjawab optimasi SPARQL *query* kedalam bentuk Cypher *query*.

5.4.2 Hasil Pengujian *Indexing*

Berdasarkan Pengujian yang sudah dilakukan terbukti bahwa *indexing* yang dibuat dapat meningkatkan performa basis data Neo4j dalam melakukan *query* dan *update*. Dapat dilihat pada Gambar 5.1 *query plan* pada *query* yang menggunakan *index* akan memulai operasi nya dengan *NodeIndexSeek* yaitu dengan mencari data yang diinginkan pada *index* yang sudah ada. Karena bentuk *query* yang dijalankan terdapat label :*Resource* dan kondisinya pada *property uri* maka memenuhi *index* yang sudah dibuat sebelumnya. Tanpa harus melakukan join pada seluruh *node* yang ada, operasi yang dilakukan hanya mencari *node* dengan *uri* "http://world-fr.open*food*facts.org/produit/0052745953028/solid-dark-chocolate-fannie-may" pada *index*, kemudian mencari *node* dengan *uri* "http://fr.open*food*facts.org/ingredient/vanilla" pada *index* baru dilakukan perbandingan dari kondisi yang diberikan.

Hal ini akan mengoptimalkan performa dengan hanya melakukan operasi sebanyak 21 DB *Hits*. Hal tersebut akan berbeda bila *query* dilakukan tanpa menggunakan *index*, Neo4j akan memeriksa seluruh *node* yang ada hingga kondisi terpenuhi dan akan dilakukan *filtering* dengan *node* lainnya kurang lebih sebanyak *node* yang ada pada basis data.

Hal ini juga akan terjadi pada proses *update* karena pada dasarnya sebelum melakukan *update*, harus dilakukan pencarian *node* yang ingin untuk diupdate terlebuh dahulu.

BAB VI
KESIMPULAN DAN SARAN

Bab ini membahas kesimpulan yang dapat diambil dari hasil uji coba dan perancangan perangkat lunak sebagai jawaban dari rumusan masalah yang telah dikemukakan dan saran yang berisi pengembangan yang dapat dilakukan lebih lanjut untuk menyempurnakan perangkat lunak.

6.1 Kesimpulan

Berikut merupakan kesimpulan yang dapat diambil dari hasil implementasi dan hasil uji coba.

1. Dengan memetakan bentuk data *triples* berdasarkan aturan yang sudah diajukan pada [6], data *triples* pada model *Triple Store* dapat diubah menjadi *node* dan *relationship* pada model *Labeled Property Graph* sehingga memungkinkan untuk menyimpan *file* RDF ke dalam basis data graf dengan model *Triple Store* maupun *Labeled Property Graph.*
2. Dengan menerapkan *indexing* pada Neo4j dapat meningkatkan performa basis data dengan cara mengurangi operasi dasar yang dilakukan pada basis data sehingga waktu yang dibutuhkan basis data untuk menyelesaikan *query* akan semakin cepat. Hal ini dapat dilihat pada subbab 5.3. yang menjelaskan hasil pengujian *indexing*.
3. Dengan menggunakan aturan untuk memetakan *triples* kita dapat memetakan SPARQL *query* menjadi bentuk Cypher *query* agar Neo4j dapat menghasilkan hasil *query* yang sesuai dengan SPARQL *query* dengan menggunakan program *converter* yang sudah dibuat.
4. Melakukan *wildcard query* dapat dijawab dengan baik oleh Neo4j karena pada neo4j sudah tersedia *filter* pada sintaks *WHERE* untuk menjawab *wildcard query*. Pada Apache Jena Fuseki *wildcard query* dapat dijawab menggunakan sintaks

FILTER regex dengan pemeriksaan string terhadap *regular expression*.

5. Dengan merubah struktur data yang digunakan dalam penyimpanan *file* RDF dan melakukan perubahan bentuk SPARQL *query* menjadi bentuk Cypher *query* untuk mendapatkan data nya, penelitian ini berhasil mengoptimalkan waktu *query* sehingga waktu *query* yang dijalankan menjadi stabil walaupun jumlah data yang digunakan dalam pengujian semakin bertambah.

6.2 Saran

Berikut ini merupakan saran terhadap pengembangan lebih lanjut yang dapat dilakukan untuk menyempurnakan sistem yang sudah dibangun.

1. Perlu dibuat analisis kembali dalam membuat program yang dapat menjawab SPARQL *query* yang sintaksnya tidak dapat dijawab oleh Cypher *query*.
2. *Query* yang digunakan dalam pengujian dibuat lebih beragam lagi agar dapat mengetahui batasan–batasan lain pada setiap model basis data.

DAFTAR PUSTAKA

[1] D. J. Abadi, A. Marcus, S. R. Madden and K. Hollenbach, "vertical partitioning," in *Scalable semantic web data management using vertical partitioning*, Vienna, VLDB, 2007, pp. 411-422.

[2] T. Neumann and G. Weikum, RDF-3X: a risc-style engine for RDF, VLDB, 2008, pp. 647-659.

[3] C. Weiss, P. Karras and A. Bernstein, "sextuple indexing," in *Hexastore: sextuple indexing for semantic web data management*, PVLDB, 2008, pp. 1008-1009.

[4] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma and Y. Pan, Efficient Indices Using Graph Partitioning in RDF Triple Stores, IEEE, 2009.

[5] D. J. Abadi, A. Marcus, S. R. Madden and K. Hollenbach, "SW-Store," in *SW-Store: a vertically partitioned DBMS for Semantic Web data management*, VLDB J., 2009, pp. 385-406.

[6] J. BARRASA, 7 Juni 2016. [Online]. Available: https://jbarrasa.com/2016/06/07/importing-rdf-data-into-neo4j/. [Accessed 4 Juni 2018].

[7] H. Thakkar, D. Punjani, Y. Keswani, J. Lehmann and S. Auer, "A Stitch in Time Saves Nine – SPARQL Querying of Property Graphs using Gremlin Traversals," no. arXiv, p. 24, 2018.

[8] T. H., P. D., L. J. and A. S., "Killing Two Birds with One Stone – Querying Property Graphs using SPARQL via GREMLINATOR," no. arXiv, p. 4, 2018.

[9] T. Berners-Lee, J. Hendler and O. Lassila, The Semantic Web, Scientific American, 2001.

[10] "Resource Description Framework (RDF)," The World Wide Web Consortium (W3C), [Online]. Available:

https://www.w3.org/TR/WD-rdf-syntax-971002/. [Accessed 03 Mei 2018].

[11] T. Segaran, C. Evans and J. Taylor, Programming the Semantic Web : Build Flexible Applications with Graph Data, O'Reilly Media, Inc., 2009.

[12] L. Zou, J. Mo, L. Chen, M. Tamer Özsu and D. Zhao, "VS-Tree," in *gStore: Answering SPARQL Queries via Subgraph Matching*, VLDB, 2011.

[13] I. Robinson, J. Webber and E. Eifrem, Graph Databases : New Oppotunities For Connected Data 2nd Edition, O'Reilly Media, Inc., 2015.

[14] Neo4J, Inc., "Neo4j Graph Database," Neo4j, Inc., [Online]. Available: https://neo4j.com/developer/graph-database/. [Accessed 4 Juni 2018].

[15] "Apache Jena Fuseki," [Online]. Available: https://jena.apache.org/documentation/fuseki2/. [Accessed 3 Januari 2018].

[16] "Open Food Facts," Open Food Facts, [Online]. Available: https://world.openfoodfacts.org/. [Accessed Juli 2018].

[17] T. B. Lee, J. Hendler and O. Lassila, "Scientific American," NATURE AMERICA, INC., Mei 2001. [Online]. Available: https://www.scientificamerican.com/magazine/sa/2001/05-01/#article-the-semantic-web. [Accessed 4 Juni 2018].

[18] J. Barrasa, "RDF Triple Stores vs. Labeled Property Graphs: What's the Difference?," Neo4j, Inc., Agustus 2017. [Online]. Available: https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/. [Accessed 4 Juni 2018].

LAMPIRAN

A. Hasil Pengujian *Query*

Pada baigan ini akan dilampirkan tabel data hasil pengujian *query* yang sudah dilakukan. Data hasil pengujian berisikan data jumlah baris data yang dihasilkan, dan waktu *running time query* pada setiap model basis data graf. Setiap *query* akan dijalankan sebanyak 5 kali, kolom dengan nama "ke-1" menunjukkan waktu dari *query* yang dijalankan pertama kali nya.

A.1. Hasil Pengujian *Query* 1 Dengan *Query Langsung*

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	12	20	22	22	21	14	12	88	91	70	76	75
2	267	5	14	13	19	14	267	1231	1249	1369	1319	1201
3	9	16	13	12	11	16	9	75	83	76	80	74
4	1	23	21	19	17	23	1	66	71	80	76	73
5	1	16	15	17	17	16	1	79	71	75	87	71
6	15	19	14	14	15	16	15	83	86	75	82	74
7	1341	14	13	18	14	11	1341	532	786	401	596	601
8	442	7	6	6	10	7	442	474	428	604	478	613
9	2	8	12	10	12	13	2	391	358	350	361	372

A.2. Hasil Pengujian *Query* 1 Dengan Menggunakan Program

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	12	10	8	7	6	11	12	532	601	583	660	625
2	267	8	6	7	7	5	267	1741	1716	1897	1733	1704
3	9	16	15	7	5	10	9	591	619	636	589	628
4	1	12	8	10	10	7	1	624	694	728	660	680
5	1	17	7	6	14	11	1	617	604	606	602	617
6	15	17	17	11	5	10	15	539	598	597	630	629
7	1341	6	7	7	7	8	1341	1111	1115	1179	1108	1120
8	442	16	10	11	5	5	442	1300	1107	1096	1108	1156
9	2	10	10	6	10	14	2	862	832	843	845	866

A.3. Hasil Pengujian *Query* 2 Dengan *Query* Langsung

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	18	15	24	26	21	18	18	99	89	107	96	108
2	452	17	16	17	15	14	452	1730	1893	1924	1847	1774
3	9	13	12	9	12	9	9	83	82	85	92	77
4	1	20	21	25	21	25	1	78	79	100	85	76
5	1	16	22	18	18	14	1	73	92	87	85	83
6	15	10	16	14	17	16	15	95	80	88	83	91
7	2777	10	12	15	13	22	2777	731	839	667	696	665
8	877	10	8	6	8	11	877	779	738	901	1084	790
9	2	6	14	15	13	16	2	606	605	619	623	621

A.4. Hasil Pengujian *Query* 2 Dengan Menggunakan Program

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	18	13	8	6	11	6	18	625	627	622	608	580
2	452	7	11	12	9	6	452	2213	2222	2300	2195	2236
3	9	11	6	9	19	4	9	615	615	732	625	597
4	1	15	6	6	11	9	1	661	686	659	632	623
5	1	7	6	14	13	6	1	770	619	649	597	617
6	15	6	11	19	6	6	15	613	705	636	602	683
7	2777	7	8	10	8	6	2777	1467	1480	1554	1482	1529
8	877	6	9	11	13	15	877	1667	1547	1495	1488	1519
9	2	7	7	12	7	17	2	1092	1086	1167	1078	1113

A.5. Hasil Penguian *Query* 3 Dengan *Query* Langsung

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	22	21	19	25	22	21	22	80	102	121	98	83
2	605	17	17	17	17	15	605	2394	2546	2484	2450	2393
3	9	14	13	20	18	16	9	70	79	76	81	75
4	1	20	20	20	22	23	1	71	72	85	94	73
5	1	16	17	15	25	17	1	97	79	71	76	75
6	15	13	12	14	12	14	15	73	82	80	98	85
7	4128	20	14	12	15	15	4128	783	1152	807	835	788
8	1279	7	8	8	9	10	1279	954	1011	898	907	951
9	3	11	11	9	7	8	3	833	846	843	853	825

A.6. Hasil Pengujian *Query* 3 Dengan Menggunakan Program

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	22	17	11	7	5	15	22	562	598	755	604	594
2	605	7	12	8	14	6	605	2922	2874	2747	2799	2873
3	9	8	10	8	6	7	9	578	604	589	608	587
4	1	6	6	8	14	11	1	591	661	626	689	635
5	1	11	7	14	4	6	1	591	589	547	599	591
6	15	8	15	8	11	7	15	575	593	587	577	611
7	4128	12	16	10	7	6	4128	1672	1694	1679	1644	1809
8	1279	7	8	9	7	8	1279	1841	1744	1742	1760	1758
9	3	14	7	18	9	9	3	1280	1298	1313	1294	1288

A.7. Hasil Pengujian *Query* 4 Dengan Quey Langsung

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	22	18	28	20	20	19	22	93	104	88	98	101
2	607	14	17	14	13	11	607	3313	3476	3506	3282	3649
3	9	8	11	15	12	16	9	88	91	92	94	97
4	1	24	18	18	21	23	1	76	69	76	94	91
5	1	15	18	13	14	17	1	78	93	90	95	77
6	15	9	13	12	14	17	15	80	100	84	94	86
7	4144	16	14	13	16	21	4144	815	867	1103	1019	892
8	1295	14	6	10	13	10	1295	979	1011	997	1064	1023
9	3	9	8	11	16	15	3	916	869	889	935	906

A.8. Hasil Pengujian *Query* 4 Dengan Menggunakan Program

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	22	6	9	6	13	6	22	593	622	565	704	649
2	607	10	9	8	9	6	607	3921	3767	3692	3823	3849
3	9	9	7	7	6	7	9	649	542	673	600	619
4	1	8	9	7	4	11	1	649	713	618	665	596
5	1	10	14	9	9	12	1	715	584	597	595	642
6	15	9	6	6	9	7	15	605	537	625	696	623
7	4144	9	6	6	15	17	4144	1798	1999	1860	2044	1746
8	1295	8	8	6	9	7	1295	1941	1789	1748	1863	1890
9	3	7	8	5	9	7	3	1363	1393	1322	1353	1791

A.9. Hasil Pengujian *Query* 5 Dengan Quey Langsung

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	22	26	23	21	26	26	22	78	88	109	88	97
2	612	17	16	9	21	13	612	4179	4345	4184	3900	3935
3	9	6	17	18	19	13	9	89	78	84	75	84
4	1	22	20	21	20	18	1	70	78	71	77	69
5	1	11	15	19	12	13	1	67	68	69	73	81
6	15	10	17	11	24	16	15	73	83	81	74	78
7	4171	10	15	13	16	14	4171	1254	762	803	776	842
8	1357	5	10	8	13	7	1357	991	1084	998	1032	991
9	3	5	7	14	13	15	3	903	924	1142	935	960

A.10. Hasil Pengujian *Query* 5 Dengan Menggunakan Program

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	22	18	6	7	7	7	22	599	580	577	579	600
2	612	10	9	7	7	16	612	4730	4292	4378	4337	4434
3	9	15	6	7	11	8	9	607	607	655	621	664
4	1	7	13	6	16	6	1	630	663	576	668	614
5	1	11	7	6	13	14	1	579	692	585	590	571
6	15	8	7	8	6	9	15	584	597	676	794	582
7	4171	9	6	16	9	8	4171	1601	1711	1660	1770	1889
8	1357	5	6	6	7	18	1357	1821	1792	1777	1876	1758
9	3	7	6	6	13	6	3	1431	1385	1376	1335	1341

A.11. Hasil Pengujian *Query* 6 Dengan Quey Langsung

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	23	24	21	19	22	17	23	86	90	94	92	102
2	665	16	17	19	15	14	665	5004	4724	4909	5004	5077
3	9	17	17	16	16	10	9	82	99	76	95	79
4	1	21	24	24	23	22	1	77	87	74	82	74
5	1	17	17	16	18	14	1	83	99	94	83	91
6	15	12	14	15	15	15	15	85	79	85	78	89
7	4282	13	16	17	17	15	4282	837	881	934	893	1008
8	1550	12	8	8	10	10	1550	1134	1259	1246	1277	1243
9	3	8	15	12	13	16	3	1097	1064	1228	1118	1036

A.12. Hasil Pengujian *Query* 6 Dengan Menggunakan Program

Query	Neo4j						Apache Jena Fuseki					
	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5	Jumlah record	ke-1	ke-2	ke-3	ke-4	ke-5
1	23	7	7	8	9	13	23	685	652	625	635	665
2	665	5	12	6	12	5	665	6027	5052	5998	5362	5309
3	9	10	7	6	11	9	9	621	626	643	694	601
4	1	9	6	10	5	9	1	749	663	595	720	953
5	1	6	18	12	14	6	1	526	645	661	653	591
6	15	9	6	12	6	6	15	717	588	632	656	597
7	4282	11	7	7	9	10	4282	1772	1735	1917	1660	1954
8	1550	6	7	6	11	7	1550	1936	2024	1972	2049	2092
9	3	9	8	10	7	6	3	1529	1520	1492	1521	1529

B. Hasil pengujian *indexing* dengan 5 kali percobaan

Hasil Pengujian *indexing* yang dilakukan sebanyak 5 kali pada *query* dan *update* yang dijalankan dengan keterangan DB *Hits* merupakan banyaknya operasi dasar yang dilakukan oleh basis data, dan rata-rata adalah rata-rata dari 5 kali percobaan.

Pengujian	DB Hits	Ke-1	Ke-2	Ke-3	Ke-4	Ke-5	Rata-rata
Query dengan index	**21**	427	452	407	450	533	**453,8**
Query tanpa index	**12288600**	10388	10599	10451	10251	10706	**10479**
Update dengan index	**5**	387	326	131	306	124	**254,8**
Update tanpa index	**12288565**	9807	10255	14290	11223	13406	**11796,2**

[Halaman ini sengaja dikosongkan]

BIODATA PENULIS

NAFIAR RAHMANSYAH, lahir di Jakarta, pada tanggal 13 Mei 1997. Dalam menempuh Pendidikan penulis pernah bersekolah di TK MI Al – Fallahiyyah Jakarta (2001 - 2002), melanjutkan sekolah dasar pada MI Al – Fallahiyyah Jakarta (2002 - 2004), kemudian pindah ke kota Tangerang dengan melanjutkan sekolah dasar pada SDI Gunung Jati sampai lulus (2004 - 2008), pendidikan menengah pertama di SMP Negeri 19 Tangerang (2008 - 2011), pendidikan menengah atas di SMA Negeri 8 Tangerang (2011 - 2014) dan pendidikan tinggi di S1 Teknik Informatika ITS (2014 - 2018).

Selama kuliah di Teknik Informatika ITS, penulis mendalami bidang minat Manajemen Informasi (MI). penulis sendiri pernah menjadi asisten dosen pada matakuliah bidang Manajemen Informasi sebagai asisten matakuliah Sistem Basis Data. Selain itu penulis juga pernah berpartisipasi dalam penelitian di bidang Manajemen Informasi. Penulis dapat dihubungi melalui surel: **nafiar21@gmail.com**.