PROGRAM STUDI SARJANA SISTEM INFORMASI

PROPOSAL PENELITIAN TUGAS AKHIR 1 DAN SEMINAR

An Efficient Database Materialized View Maintenance

PENGUSUL

12S15021 PUTRI MORANDA MANURUNG

12S15022 JOE ALLEN BUTARBUTAR

12S15047 CHRISTINE VINCENZA SITORUS

INSTITUT TEKNOLOGI DEL

JANUARI 2018

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

RINGKASAN

Kinerja sistem basis data merupakan isu yang sangat penting seiring berkembangnya teknologi untuk memproses data yang semakin lama akan semakin besar ukuran atau volumenya. Peneliti melihat peningkatan kinerja sistem yang optimal menjadi acuan untuk suatu sistem dapat dipakai dalam jangka panjang. Penggunaan *materialized view* pada basis data akan sangat berpengaruh dalam pengaksesan data maupun pengembangan *query* pada basis data. Dalam melakukan perubahan data pada *base table* akan berpengaruh pada *materialized view* yang telah dibangun. Setiap terjadi perubahan pada *base table*, maka seluruh *materialized view* yang dibangun juga harus dibangun ulang. Hal ini akan membutuhkan *storage cost* yang cukup besar. Apabila perubahan yang terjadi pada *base table* tidak berpengaruh pada suatu *materialized view*, maka tidak dibutuhkan untuk melakukan *rebuild* pada *materialized view* tersebut. Oleh karena itu, dibutuhkan *fungsi* untuk mencegah *rebuilt* secara keseluruhan terhadap *materialized view* untuk menghemat *storage cost.* Perubahan yang terjadi pada *base table* akan dipetakan langsung terhadap *materialized view* yang terpengaruh. Peneliti menerapkan metode *interrupt* pada *materialized view* dengan melakukan peninjauan langsung terhadap arsitektur basis data yang digunakan yaitu PostgreSQL. Agar dapat melakukan pembaharuan terhadap arsitektur, maka dicari titik pada PostgreSQL yang melakukan eksekusi pengecekan basis data.

***Kata kunci**:*
Base table, view, materialized view maintenance, filtering..

BAB 1. PENDAHULUAN

1.1 Latar Belakang

Basis data adalah sekumpulan data yang dapat diolah untuk menghasilkan informasi yang bermanfaat bagi pemilik data [1]. Dalam struktur basis data terdapat objek seperti *table, trigger, index, sequence dan view* [2]. *Table* adalah kumpulan dari *field* dan *record* dimana beberapa *field* memiliki nama yang unik. *Index* adalah *search key* untuk mendefinisikan *sequential order*. *Trigger* adalah *statement* yang dieksekusi untuk memodifikasi *database. Sequence* adalah objek basis data yang melakukan *generate* bilangan yang memiliki nilai *incremental. View* adalah sebuah *virtual table* untuk menyimpan *compiled query* yang telah dieksekusi. Interaksi antara aplikasi dengan *database* dihubungkan menggunakan *query* [1]. Bahasa *query* yang paling umum digunakan untuk mengakses data dalam basis data adalah *Structured Query Language* (SQL) [3].

Pemrosesan *query* dilakukan dalam beberapa tahap yaitu *parsing and translation, optimization,* dan *evaluation engine.* Pada tahap *evaluation engine* terjadi proses *query execution plan* dan *query execution* [1]. Ketika *query* diproses dengan asumsi data tidak berubah (konstan), dibutuhkan *cost* untuk *preparation query* dan *storage cost.* Pada penggunaan *view* akan diturunkan *cost* untuk *preparation query* sedangkan pada *materialized view* akan dihilangkan *cost* untuk *preparation query* dan *storage cost* akan diturunkan. Hal ini berbanding lurus dengan *response time* [1].

Pembuatan *view* yang merupakan *virtual table* pada basis data dapat meminimalisir *response time* dikarenakan saat mengakses data hanya dilakukan eksekusi untuk memanggil *view* yang telah dibangun. Namun, *view* yang telah dibangun tidak disimpan dalam penyimpanan fisik (*cache*). Oleh karenanya, dibutuhkan *materialized view* yang akan menyimpan *query* SQL dan hasil *query* di *storage* [3]. Hasil *query* pada *materialized view* harus selalu di-*rebuilt* secara keseluruhan setiap terjadi perubahan pada *base table*. Proses *rebuild* secara keseluruhan membutuhkan *storage cost* [4]. Namun, ada kemungkinan tidak seluruh *materialized view* terpengaruh terhadap perubahan data pada *base table.* Sehingga, *rebuild materialized view* secara keseluruhan adalah penggunaan *storage cost* yang berlebihan dan sebaiknya dihindari [4]. Hal ini dapat diperjelas dengan contoh di bawah.

Tabel 1 *Order Table*

ID	***CustomerID***	***Date***	***Address***	***Total (Rp)***	***Status***
ODR0001	CST0001	08/08/2018	Balige	53.000	Dalam proses
ODR0002	CST0002	09/09/2018	Laguboti	56.000	Dikirim
ODR…	CST…	……	……	…….	…..
ODR1000	CST1000	09/10/2018	Tambunan	71.000	Dikirim

Tabel 2 *Item Order Table*

ID	***Product***	***Quantity***	***Price***	***Total (Rp)***
ODR0001	Mouse	3	25.000	75.000
ODR0002	Printer	4	70.000	280.000
ODR0003	Flashdisk	5	81.000	405.000
ODR….	….	…	….	….
ODR2000	USB	8	30.000	240.000

Tabel 3 *Customer Table*

ID	***City***	***Gender***	***Age***
CST0001	Balige	Pria	24
CST0002	Laguboti	Pria	26
CST….	….	….	….
CST3000	Tambunan	Wanita	27

Tabel 4 *Product_by_quantity*

ID	***Date***	***Product***	***Quantity***	***Total (Rp)***
ODR0001	08/08/2018	*Mouse*	3	75.000
ODR0002	09/09/2018	*Printer*	4	280.000
ODR0003	03/04/2018	*Flashdisk*	2	162.000
ODR…	…..	…..	…..	…..
ODR2000	09/10/2018	*Flashdisk*	5	405.000

Tabel 5 *Product_by_price*

ID	***Product***	***Price***	***Total (Rp)***
ODR0001	*Mouse*	25.000	75.000
ODR0002	*Printer*	700.000	280.000
ODR0003	*Flashdisk*	81.000	405.000
ODR….	….	….	….
ODR2000	USB	30.000	240.000

Jika diasumsikan setiap *record* dari setiap *table* membutuhkan 4 Kilo Byte (KB), maka setiap *table* membutuhkan ruang penyimpanan yaitu tabel 1 ≈ 4000 KB, tabel 2 ≈ 8000 KB dan tabel 3 ≈ 120000 KB. Ketika dilakukan *cartesian product* yaitu operasi *join* pada tabel 1 dan tabel 2, maka estimasi *cost* untuk mengeksekusi *query* sekitar ≈ 32000 KB pada estimasi waktu t (*seek time*). Apabila tabel ini dibentuk menjadi *view*, maka saat dilakukan pengeksekusian *query* yang sama sebanyak dua kali pada penggunaan operasi *cartesian product*, maka estimasi

cost untuk *view* yang akan dibutuhkan sekitar ≈ 64000 KB pada estimasi waktu t (*seek time*). Sehingga pengeksekusian *view* membutuhkan *cost* yang semakin bertambah. Sedangkan jika menggunakan *materialized view,* estimasi *cost* untuk *materialized view* akan dibutuhkan sekitar ≈ 32000 KB pada estimasi waktu t (*seek time*). Karena hasil eksekusi *cartesian product* akan disimpan dalam *storage* sehingga jika dilakukan eksekusi kembali maka *materialized view* tidak mencari ke *base table.* Sehingga dapat dilihat bahwa *materialized view* memiliki *cost* yang lebih kecil.

Jika terjadi perubahan pada *base table* yaitu dengan *query* sebagai berikut: update tabel Order. Set ID=001, address='Balige';
Maka apabila dilakukan *update* untuk *materialized view, materialized view* 1 dan *materialized view* 2 akan di-*rebuilt,* sehingga penggunaan *storage cost* adalah 32000 KB dalam estimasi waktu t (*seek time*). Dibandingkan jika hanya *materialized view* yang memiliki dampak perubahan pada *base table* saja yang akan dilakukan *rebuild* yaitu *materialized view* 1 akan membutuhkan *storage cost* sebesar 4000 KB dalam estimasi waktu t (*seek time*).

Dalam penelitian ini, peneliti ingin menerapkan konsep *filtering* untuk menghentikan pengecekan terhadap *materialized view* yang tidak berhubungan dengan perubahan yang terjadi pada *base table.* Apabila perubahan pada *base table* tidak mempengaruhi *materialized view,* maka pengecekan akan berhenti pada *filtering.* Namun sebaliknya apabila perubahan pada *base table* mempengaruhi *materialized view,* maka akan terjadi *rebuild* pada *materialized view* yang bersangkutan.

1.2 Tujuan Penelitian

Tujuan dari pengerjaan Tugas Akhir ini adalah untuk menurunkan *storage cost* pada saat melakukan *query processing* dengan menerapkan metode *materialized view maintenance*.

1.3 Rumusan Masalah

Rumusan masalah dari penelitian yang dilakukan adalah:

1. Bagaimana cara meminimalisir *storage cost* dalam penerapan *materialized view and maintenance*?

1.4 Ruang Lingkup

Ruang lingkup dari penelitian adalah basis data yang mendukung penerapan *materialized view* yaitu PostgreSQL.

1.5 Sistematika Penyajian

Dokumen ini selanjutnya disusun sebagai berikut:

Bab 1 adalah pendahuluan yang berisi latar belakang penelitian, tujuan penelitian, rumusan masalah, serta ruang lingkup penelitian.

Bab 2 adalah landasan teori yang berisi teori-teori yang digunakan untuk mendukung penyelesaian masalah yang dibahas dalam bab pendahuluan. Teori-teori ini akan digunakan sebagai acuan dalam melakukan penelitian.

Bab 3 adalah analisis penelitian yang berisi analisis metode penelitian, analisis kasus, analisis penggunaan fungsi dan analisis *cost* yang digunakan untuk *refreshing materialized view* serta analisis terhadap baris dan kolom pada saat memperbaharui *materialized view.*

Bab 4 adalah perancangan penelitian yang berisi perancangan sistem, perancangan proses *filtering* dan perancangan perubahan pada *source code* dalam melakukan penelitian.

BAB 2. LANDASAN TEORI

2.1 *Database Management System*

Suatu *database management system* (DBMS) adalah suatu program yang digunakan untuk mengakses dan mengelola basis data. Tujuan utama dari DBMS adalah untuk menyediakan metode untuk menyimpan dan memperoleh data dari basis data yang sesuai [1].

Dr. Edgar F Codd dalam [5] mengatakan apabila memiliki desain *database management system* yang baik, maka kunci dalam pemrosesan data dan pengambilan keputusan sudah dipegang. Dalam bukunya juga dituliskan bahwa terdapat dua tipe utama basis data yaitu *production oriented database* yang mencerminkan kenyataan dan *exploratory database* yang digunakan untuk merencanakan kegiatan yang mungkin terjadi di masa depan. Dalam kedua kasus tersebut keakuratan, konsistensi dan integritas data sangat penting.

Untuk memenuhi hal tersebut, ditemukan mesin prototipe yang desain lengkapnya didasarkan pada *relational model*, sehingga *Relational Database Management System* (RDBMS) merupakan unsur penting dalam pemodelan basis data [5]. RDBMS memiliki karakteristik yang disebut dengan *ACID Properties* yang berfungsi untuk membuat basis data lebih mudah untuk dipulihkan dan memungkinkan untuk *multi-user* melakukan transaksi bersama-sama namun tetap mempertahankan tampilan kronologis data yang konsisten dalam basis data [1] [6]. *ACID properties* terdiri dari:

1. *Atomicity*

   Atomicity merupakan suatu keadaan dimana seluruh aksi dari suatu transaksi harus dieksekusi secara sempurna, atau apabila terjadi kesalahan maka aksi yang telah terjadi dari setiap transaksi yang tidak sempurna dieksekusi harus dibatalkan

2. *Consistency*

   Basis data yang terdistribusi harus tetap memiliki sifat yang konsisten. Basis data harus bergerak secara serentak menuju keadaan konsisten ketika seluruh aktivitas yang aktif telah diselesaikan atau diperbaharui.

3. *Isolation*

   Isolation merupakan sifat basis data yang memisahkan transaksi dari pengaruh transaksi yang sedang dieksekusi secara bersamaan.

4. *Durability*

   Durability adalah sifat basis data yang mempertahankan transaksi yang telah suskes dieksekusi didalam basis data. Kesalahan yang terjadi dalam sistem tidak akan berdampak pada basis data dan menggantikan transaksi yang telah sukses.

Sebagai contoh untuk *ACID properties* yang terdapat dalam [1], terdapat T_i yang merupakan suatu transaksi pengiriman sebesar $50 dari akun A ke akun B, yang didefinisikan sebagai berikut:

T_i : read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
Write(B).

Peninjauan dari *ACID properties:*

1. *Atomicity:* Misalkan nominal untuk akun A dan B masing-masing adalah $1000 dan $2000. Lalu terjadi kesalahan setelah write(A) dan sebelum write(B). Hal ini mengakibatkan nominal pada akun A adalah sebesar $950 dan akun B tetap sebesar $2000. Sistem telah menghilangkan sebesar $50 sebagai hasil dari kesalahan yang terjadi, sehingga dapat disimpulkan bahwa jumlah A+B tidak dapat dipertahankan.
2. *Consistency:* Maksud dari konsistensi dalam hal ini adalah jumlah dari A+B tetap dapat dipertahankan setelah proses transaksi telah selesai dilakukan. Jumlah awal kedua akun adalah sebesar $3000 dan setelah prosses transaksi selesai dilakukan jumlah kedua akun tetap $3000.
3. *Isolation:* Pada transaksi pengiriman dana dari akun A ke B terjadi kesalahan. Apabila transaksi selanjutnya melakukan *update* berdasarkan transaksi yang tidak konsisten sebelumnya, maka basis data menjadi tidak konsisten setelah kedua transaksi selesai dilakukan. Maka dalam hal ini, sebaiknya setiap transaksi dipisahkan dari transaksi lainnya agar apabila terjadi kegagaln pada suatu transaksi, tidak mempengaruhi transaksi lainnya.
4. *Durability:* Apabila proses eksekusi transaksi selesai dengan suskses, dan pengguna yang memulai transaksi telah diberitahu bahwa pengiriman dana telah dilakukan dan tidak ada kegagalan sistem yang dapat mengakibatkan hilangnya data terkait dengan pengiriman dana. *Durability properties* menjamin bahwa setelah transaksi selesai dengan sukses, semua pembaharuan yang dilakukan pada basis data tetap ada, bahkan jika ada kegagalan sistem setelah transaksi selesai dieksekusi. Kegagalan sistem computer dapat menyebabkan kerugian data dalam memori utama, tetapi data yang ditulis pada *disk* tidak pernah hilang.

2.2 *Data Storage*

Data storage adalah tempat menyimpan data dan data dapat diakses melalui sistem komputer. *Storage* diklasifikasikan berdasarkan kecepatan untuk akses data, kapasitas penyimpanan data dan *reliability.* Tipe dari *data storage* adalah *cache, main memory, magnetic disks, flash memory, optical disk* dan *magnetic tapes.* Ketika *disk* digunakan, maka *drive* berputar dengan kecepatan yang konstan mencapai 120 putaran per detik. Ukuran kapasitas *disk* yang digunakan untuk menyimpan banyak informasi yaitu *gigabyte* dan *terabyte*. Satu *gigabyte* setara dengan 1024 *megabytes* dan satu *terabytes* setara dengan 1.024.000 *megabytes.*

Hal utama yang diukur dari kinerja dan kualitas *disk* adalah kapasitas, *access time, data-transfer rate,* dan *realibility disk. Seek time* adalah waktu untuk waktu yang dibutuhkan oleh *arm* untuk berpindah ke posisi *track* yang benar, dengan rentang waktu 2 sampai 30 *milisecond*. *Data-transfer rate* adalah pengukuran yang digunakan untuk mengukur kecepatan *disk* untuk mengakses data, dimana *transfer rate* terbesar pada rentang 25 sampai 100 *megabytes* per detik. Sedangkan *mean time to failure* (MTTF) adalah pengukuran untuk mengukur *realibility* dari disk.

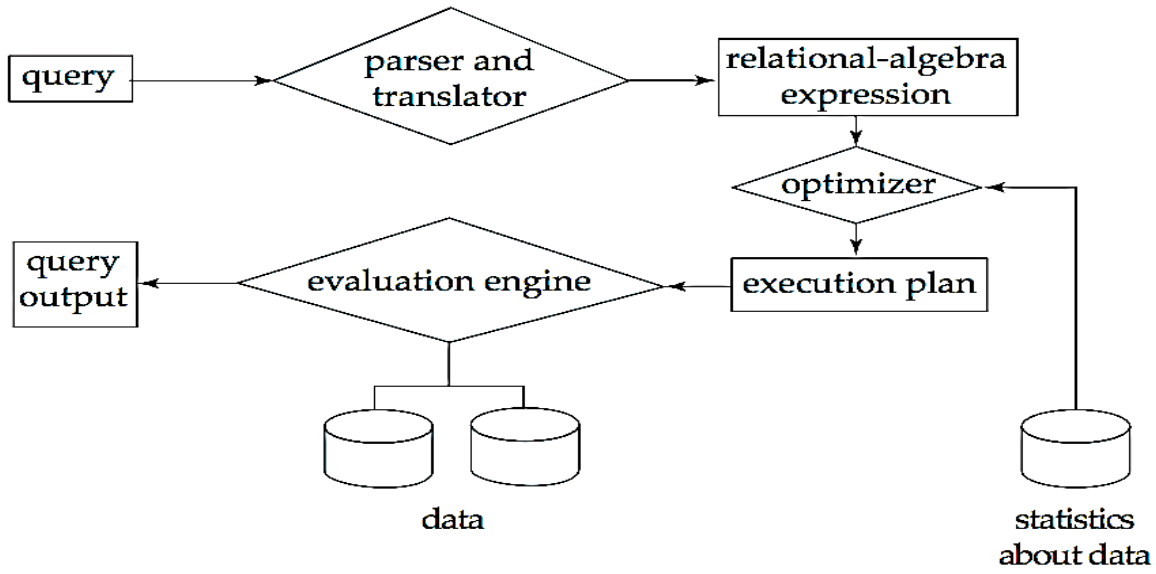
Dalam pemrosesan data dalam *data storage* yang berjumlah besar akan meningkatkan kecepatan *drive* untuk membaca atau menulis data. Sehingga terdapat teknik RAID (*Redundant Arrays of Independent Disk*) untuk pengorganisasian *disk* dan menahan redundansi data untuk meningkatkan kinerja I/O dari disk dan *realibility disk*[1].

2.3 *Query Language*

Proses pengolahan data akan dilakukan pada basis data. Pengguna/aplikasi berinteraksi dengan basis data melalui bahasa *query* yang merupakan bahasa pemrograman dalam basis data. Struktur basis data merupakan faktor yang paling dominan dalam menentukan bahasa *query* yang digunakan untuk mengakses data dalam basis data. *Structured Query Language (SQL)* merupakan salah satu bahasa yang paling umum yang menyediakan sejumlah operasi, seperti membaca, memperbaharui, menambah, menghapus data serta beberapa operasi lainnya pada basis data dan memiliki tiga kategori: *Data Definition Language (DDL), Data Manipulating Language (DML),* dan *Data Control Language (DCL)* [3].

2.4 *Query Processing*

Query Processing adalah aktivitas untuk mengekstraksi data dari basis data seperti menerjemahkan *query* pada bahasa basis data tigkat tinggi menjadi pengungkapan yang dapat digunakan pada *level physical* pada *file system*, transformasi pengoptimalan *query,* dan evaluasi *query* aktual [1].



Gambar 1 Tahapan *Query Processing*

Flow Process dari *Query Execution* dijelaskan sebagai berikut:

1. *Parsing dan Translation*

   Pada tahap ini menjelaskan pengecekan sintaks dan melakukan verifikasi relasi dengan menggunakan *parser*. Kemudian dilakukan *Translating* untuk menerjemahkan *query* ke dalam bentuk internal (*Query Processing Engine*) dan akan diterjemahkan kembali ke dalam bentuk *relational algebra expression* atau dalam bentuk struktur *tree* atau *graph*.

2. *Optimization*

   Pada tahap ini menjelaskan optimalisasi *query* untuk melakukan evaluasi *query* dalam mengukur biaya *resource* sistem yang digunakan oleh *query* tersebut oleh *optimizer* untuk meminimumkan total waktu *query processing* dan mengoptimumkan penyimpanan data.

3. *Evaluation Engine*

   Pada tahap ini menjelaskan proses pengecekan kembali ke *query* agar bisa dilakukan eksekusi dengan melakukan pemilihan *query evaluation plan* terbaik, melakukan eksekusi terhadap *query* tersebut akan menghasilkan jawaban *query* [1].

2.5 Pengukuran *Query Processing Cost*

Dalam [1] untuk melakukan perhitungan terhadap estimasi *cost* pada *query evaluation plan* dibutuhkan jumlah *disk block transfer* dan jumlah *disk seek*. Jika subsistem dari *disk* membutuhkan waktu rata-rata untuk transfer sebuah blok data, dan membutuhkan blok rata-rata dengan *access time*, kemudian *cost* untuk operasi yang melakukan transfer *b* blok dan S *seek* dapat dihitung dengan persamaan sebagai berikut:

$$b_r * t_T + S * t_S \ldots (1)$$

Dimana,

br, jumlah block pada relasi *r*.

t_T, waktu rata-rata untuk transfer blok data

t_S, waktu akses (waktu seek *disk* ditambah rotational latency)

Persamaan yang digunakan untuk menghitung jumlah *block* yang berisi tuple dari relasi r adalah:

$$br = \lceil \frac{n_r}{f_r} \rceil \ldots (2)$$

Dimana,

n_r, jumlah tuple dari relasi *r*.

f_r, jumlah tuple dari relasi *r* yang diasukkan dalam satu blok

Persamaan yang diguanakan untuk melakukan perhitugan terhadap *Nested Loop Join* adalah sebagai berikut:

$$n_r * bs + br \text{ block transfer, plus } n_r + br \text{ seek} \ldots (3)$$

Dimana,

relasi *r* adalah *outer relation* dan relasi *s* adalah *inner relation.*

n_r, jumlah tuple dari relasi *r*

bs, jumlah blok dari relasi *s*

br, jumlah blok dari relasi r

2.6 *Views*

Pada basis data juga dikenal istilah *view* yang merupakan tipe *query* yang digunakan untuk membentuk sebuah *virtual table* yang menampilkan data dari satu atau beberapa tabel. Kolom yang dibutuhkan pada tabel yang sebenarnya diwakilkan, sehingga dapat lebih aman dari segi keamanan karena pengguna diizinkan untuk mengakses bagian-bagian tertentu dari tabel [3].

2.7 Tabel *Index*

Dalam pengaksesan basis data dibutuhkan pengindeksan tabel yang mempercepat pengaksesan data yang diinginkan secara langsung tanpa harus membaca seluruh tabel. Tabel *index* dibangun terpisah dari tabel data. *Index* yang dibangun dari suatu tabel mengandung *key* tabel dan alamat yang saling berhubungan dalam *disk.* Dengan demikian, *query* dengan tabel *index* memiliki waktu akses yang kecil pada *disk* [3].

2.8 *Materialized view*

Untuk lebih mempersingkat waktu pengaksesan, dalam basis data dikenal juga metode *materialized view*. Perbedaan antara *materialized view* dan view adalah ruang penyimpanan fisik yang dapat digunakan. Berbeda dengan *view* yang hanya menyimpan *query* SQL, *materialized view* menyimpan baik hasil *query* maupun *query* SQL pada saat dibangun. Oleh karena itu, hasil *query* dari *materialized view* harus diperbaharui secara periodik [3].

2.9 Mekanisme *Materialized View Selection*

Me-*materialize*kan seluruh *view* tidak direkomendasikan karena keterbatasan *memory space* dan waktu [7]. Dalam melakukan *view selection,* terdapat beberapa faktor yang harus dipertimbangkan seperti *query frequency, query space,* dan *query processing time* [8].

2.10 Mekanisme *View Maintenance*

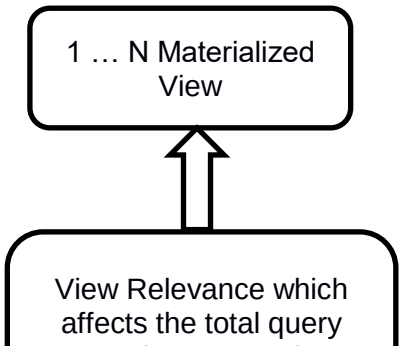
Penelitian [9] [8] [10] [11] mendeskripsikan mekanisme untuk *view maintenance* saat *base table* diperbaharui.

2.10.1 *Insert and Delete Statements*

1. *Insert Tuple Construction*: Untuk menambahkan data ke dalam *base table* yang akan membangun *view,* digunakan relasi *key/foreign key* yang terhubung dengan *view* untuk membaca *tuple* pada *base table* secara berurutan. Selanjutnya, akan dibuat *tuple* pada *view* menggunakan *tuple* yang dibaca sebelumnya pada *base table* dalam *insert statement.*
2. *Delete Construction:* Untuk menghapus sebuah *tuple* pada *view* saat terjadi penghapusan data pada *base table*, digunakan *key* pada *base table* yang disediakan dalam *delete statement.* Untuk menghapus *tuple index* pada *view* dibangun *key* untuk melakukan penghapusan dengan membaca *tuple* pada *view* menggunakan *key* pada *base table* dalam *delete statement*.

2.10.2 *Update Statement*

Tuple Construction: Memperbaharui *view* selama pembaharuan *base table* terjadi membutuhkan *cost* yang besar apabila *key* pada *view* tidak diberi *index* dalam *update statement.* Untuk mempersiapkan pembaharuan *view* dengan efisien, pada skema ditambahkan *index* berdasarkan *workload*.



Gambar 2 *Materialized View Selection* [12]

Apabila data pada *base table* diperbaharui, maka seluruh *materialized view* yang terkait harus diperbaharui untuk membangun konsistensi dan integritas dalam basis data. Proses pembaharuan suatu *materialized view* saat mengubah data pada *base table* disebut *View Maintenance* yang akan mengakibatkan adanya *View Maintenance Cost* [12]. *View maintenance cost* adalah cost yang digunakan untuk melakukan *refresh* terhadap *materialized view* untuk setiap perubahan yang terjadi pada *base table.* Untuk menghitung *maintenance cost* dapat digunakan rumus dibawah ini [13].

$$\text{Total } (C_m) = \sum_{i=1}^{j} (fu_i * C_m(T_i))$$

Dimana,

f_{ui} = Frekuensi *update* dari *materialized view*

$C_m(T_i)$ = *Cost* untuk *view maintenance*

Dalam [13] dijelaskan bahwa untuk mendapatkan *total cost* dari *materialized view* dapat dilakukan dengan menjumlahkan *query processing cost* dan *maintenance cost*. *Query processing cost* adalah jumlah frekuensi dari *query* dikalikan dengan *cost* dari akses *query* terhadap *materialized view*. Untuk menghitung total dari *query processing cost* dapat digunakan rumus dibawah ini.

$$\text{Total } (C_{qr}) = \sum_{i=1}^{r} (fg_i * C_q(q_i))$$

Dimana,

f_{qi} = Frekuensi Query

$C_q(q_i)$ = *Cost* akses *query* terhadap *materialized view*

Maka, untuk menghitung total *cost view maintenance*, dapat digunakan persamaaan dibawah ini [13].

$$\text{Total } (C_{all}) = \text{Total } (C_{qr}) + \text{Total } (C_m)$$

Dimana,

C_{all} = Total *cost view maintenance*

C_{qr} = Total *query processing cost*

C_m = Total *maintenance cost*

2.11 Perbedaan *Base Table, View* dan *Materialized view*

Pengaksesan *query* secara langsung ke *base table* akan memiliki perbedaan dengan pengaksesan dengan menggunakan *view* dan *materialized view*. Perbedaan tersebut dapat dilihat pada tabel berikut:

Tabel 6 Perbedaan pengaksesan *base table, view* dan *materialized view*

	Base Table	*View*	*Materialized View*
Response Time	*Slow*	*Fast*	*Faster*
Physical Space Occupancy	*Direct Access*	*No Cache*	*Cache*
Cost	*High*	*High*	*Lower*

Pada penelitian [7] dikatakan bahwa *materialized view* membantu dalam melakukan akses data lebih cepat dan oleh karena itu dapat meningkatkan performansi *query*. Hal ini memperkuat pernyataan, bahwa *materialized view* dapat meminimalisir *response time* dari *query*. Pada *study* [3] dikatakan bahwa perbedaan *view* dan *materialized view* terletak pada *physical space occupancy*. Pada *view* data tidak disimpan, namun dibentuk ke dalam *virtual tabel*. Berbeda dengan *materialized view* yang akan menyimpan *SQL query* dan hasil *query* dalam suatu storage (*cache*). Dalam penelitian [3] dilakukan perbandingan antara pengaksesan data langsung pada *base table*, penggunaan *view* dan penggunaan *materialized view*. Dari penelitian tersebut didapat bahwa *cost* untuk mengakses data langsung dari *base table* dan pengembangan *view* adalah sama besar. Namun, dengan menggunakan *materialized view* terlihat bahwa *cost* yang digunakan saat mengakses data sangatlah kecil.

2.12 *Cartesian product*

Cartesian product adalah sebuah operasi yang dinotasikan dengan sebuah *join* dengan mengkombinasikan informasi dari dua relasi yang akan dipindahkan ke dalam sebuah relasi baru. Relasi *Cartesian product* antara dua relasi r_1 dan r_2 merupakan perkalian relasi dari r_1 dan r_2 (r_1 x r_2), dalam hal ini r merupakan jumlah *tuple* dalam sebuah relasi [1].

2.13 *Cost Metrics*

Menurut [14] terdapat beberapa *metrics* yang menjadi tolak ukur penentu besar kecilnya *storage cost*. *Metrics* tersebut adalah:

1. *Query execution frequencies*

   Query dengan frekuensi akses yang rendah membutuhkan *storage space* yang tinggi akan dihapus sedangkan *query* dengan frekuensi akses yang tinggi akan dipilih dan dipertahankan.

2. *Base-relation update frequencies*

   Proses mencerminkan perubahan pada *materialized view* sebagai tanggapan atas perubahan pada *base relation* disebut sebagai *view maintenance* yang akanmengakibatkan terciptanya *maintenance cost.*

3. *Query access cost*

   Pengaksesan seluruh *base table* secara langsung tanpa penerapan *view maintenance* membutuhkan *query processing cost* yang sangat tinggi. Metode materialized view dapat memberikan performa query terbaik dengan menggunakan beberapa *view maintenance* dan terjadi penggunaan *storage cost*.

4. *View maintenance cost*

   Total *cost* dari setiap *view* akan dihitung dan *view* dengan *cost* yang paling minimum selama dilakukan maintenance akan dipilih untuk menjadi *materialized view.*

5. *System's storage space constraints*

   View maintenance cost yang tercipta menjadi suatu alasan tidak digunakannya seluruh *materialized view* dikarenakan keterbatasan *storage space.*

2.14 *Related Work*

Berikut ini adalah beberapa penelitan terdahulu yang terkait dengan topik penelitian ini. Analisi penelitian sebelumya dapat digunakan sebagai dasar dalam mengerjakan Tugas Akhir ini.

1. Penelitian [3] berfokus pada dampak dari *Materialized View* pada *query performance* dan observasi *cost* dari SQL *query* dari *optimized index, view* dan *materialized view*. Penelitian tersebut membandingkan *cost* dari *optimization query* pada model *view* menggunakan *index* dan tidak menggunakan *index,* model tidak menggunakan *view* dengan menggunakan *index* dan menggunakan *index,* dan model *materialized view*. Hasil penelitian menunjukkan dari ketiga model tersebut bahwa *materialized view* merupakan model *optimized query* dengan *cost* yang sedikit pada penggunaan *memory, processor* dan unit *input/output*.

2. Penelitian [8] berfokus pada teknik yang diimplementasikan pada penelitian yang sudah berlalu untuk meningkatkan kecepatan dari *Query processing* pada *Materialized View* dengan perbandingan beberapa algoritma. Algoritma yang dibahas adalah sebagai berikut: Index-Mining algorithm digunakan untuk memilih *materialized view* supaya biaya evaluasi *query* menjadi optimal. Greedy algorithm digunakan untuk pemilihan *materialized view* berdasarkan *maintenance cost* dan *storage cost* sehingga biaya evaluas i *query* menjadi optimal. *Materialized view selection* melibatkan biaya pada *query frecuency, query processing* dan *storage cost* bersamaan dengan biaya *materialized view maintenance.*
3. Penelitian [15] berfokus pada peningkatan dan keefektifan algoritma untuk *materialized view selection*. Peningkatan algoritma menggunakan *Polynomial Greedy Algorithm (PGA)* dengan mempertimbangkan efek dari keseluruhan ruang dan *cost* dengan menambahkan kandidat *materialized view* dan pengurangan kandidat materialized view untuk menemukan *cost* untuk *materialized view selection* yang paling sedikit.
4. Penelitian [10] berfokus pada pendekatan yang digunakan untuk *selection* dan *maintenance view* pada *materialized view*. *Materialized view selection* dibutuhkan untuk meningkatkan *query performance* seperti meminimalkan *response time* dan *maintenance cost* dari *query.* Beberapa pendekatan yang diberikan yaitu dengaan menggunakan klasterisasi *query* untuk meminimalkan eksekusi *query,* menggunakan *framework* untuk mendapat *response time* yang efektif dan pengurangan tabel dengan menggunakan pendekatang *clustering.*
5. Penelitian [11] mengusulkan cara untuk memelihara sekumpulan *maintenance view.* Ketika *terjadi perubahan pada base table* akan dilakukan *rebuild* pada *materialized view* dengan menggunakan *lazy strategy* dan *smart lazy incremental strategy*. *Lazy strategy* adalah penggabungan beberapa *update* dari beberapa transaksi ke suatu tempat operasi *maintenance* dan dilakukan *maintain* setelah waktu *delay.* Pada *Lazy incremental update strategy* dilakukan *maintenance* dengan melihat agregasi dari *materialized view* dan dilakukan pemabaharuan pada agregasi tersebut. Dengan *Smart lazy incremental strategy* akan dilakukan pemeriksaan *where clause* pada setiap baris yang merupakan bagian dari hasil agregasi yang di-*maintained,* tidak pada semua agregasi.
6. Penelitian [16] mengusulkan suatu *framework* untuk memilih *materialized view* terbaik untuk mencapai kombinasi efektif dari *response time query* yang baik, *cost*

pemrosesan yang rendah dan *cost view maintenance* yang rendah dalam suatu batasan *storage space* yang ditentukan. Dalam penelitian ini, ketika terjadi perubahan pada *data source,* akan dikirim notifikasi ke *warehouse* selanjutnya *warehouse* akan mengirim *query* ke *source* yang sesuai dan *source* akan mengirim tanggapan ke *data warehouse*

7. Penelitian [4] berfokus baik tidaknya eksekusi query atau dengan mempertimbangkan berbagai parameter seperti *cost of query, cost of maintenance, net benefit & storage space*. Metodologi yang diusulkan bertujuan untuk memilih kombinasi terbaik dari *view* sehingga dapat meningkatkan performa *query*. Algoritma yang ditemukan lebih efisien dibandingkan *materialized view selection* and *maintenance strategies* lainnya. Total biaya, yang terdiri dari pola dan frekuensi *query* yang berbeda dievaluasi pada 3 *view materialization strategies* yang berbeda, yaitu *all-virtual-vews method,* all *materialized view method* dan *proposed materialized-views method*. Total biaya yang dievaluasi dari penggunaan *materialized view method* yang diusulkan terbukti menjadi yang terkecil diantara 3 stategi tersebut. Percobaan selanjutnya dilakukan untuk mencatat perbedaan waktu eksekusi dari setiap strategi yang diusulkan dalam menghitung *fixed number of queries* dan *maintenance processes*.
8. Penelitian [17] berfokus pada pengurangan waktu untuk *maintenance* dan *response time* untuk *query* pada distribusi *peer to peer architecture*. Pada penelitian ini dilakukan perbandigan terhadap kinerja *Peer Joining Real Time Data Warehouse Algorithm (PJRT)* dengan algoritma v*iew maintenance* yang ada. Pada kedua algoritma tersebut dilakukan eksperimen *single and multiple types of transactions (insert, update and delete)*. Hasil eksperimen menunjukkan algoritma PJRT mengurangi *maintenance time* pada *view maintenane* dan meningkatkan hasil pemrosesan dan perubahan data pada *view maintenance*.
9. Penelitian [18] berfokus pada penyelesaian masalah dari *materialized view maintenance* dalam *platform* analisis data berskala besar. Solusi dalam penelitian ini adalah *relational materialized view* yang disusun khusus pada cara penyimpanan berorientasi kolom. Dua operasi dasar (*delete and insert*) disediakan untuk *view maintenance* dalam lingkungan yang baru. Selain itu, dua model konsistensi diusulkan pada pertukaran konsistensi data untuk efisiensi pemrosesan. Hasil penelitian ini menunjukkan efisiensi dan efektivitas dari metode yang peneliti usulkan.
10. Penelitian [19] berfokus pada metode minimum incremental maintenance dimana incremental maintenance diadopsi tetapi dengan kompleksitas komputasi berbeda pada incremental maintenance yang berbeda. Hal ini untuk memastikan akurasi hasil

pembaharuan *materialized view* ketika sumber data awal diubah. Pada data di dalam *data warehouse*, *minimum maintenance* memengaruhi peningkatan kinerja.

11. Penelitian ini berfokus pada strategi dalam pemilihan materialized view. Ketika SQL *command* dikirim ke DBMS, *optimizer* akan memilih *access plan* dengan mempertimbangkan estimasi *cost* pada setiap modul, *log* dan *statistics* dari setiap *view* yang akan dipilih untuk menjadi materialized view. Setelah SQL optimizer memilih *view* dengan pertimbangan *cost* yang efisien digunakan untuk *optimizer* maka akan menghasilkan sebuah *candidate* materialized view. Kemudian digunakan query *optimizer* menggunakan PostgreSQL untuk menggunakan simulasi konfigurasi dan modul estimasi *cost* untuk menentukan dampak *candidate materialized view* pada *cost* di eksekusi *query* yang akan menghasilkan *materialized view* dengan *cost* yang efisien. Setelah dipilih *materialized view* kemudian dilakukan *maintenance* pada *materialized view* untuk menghitung *cost* ketika terjadi *insert, update* dan *delete* pada *base table* dalam meningkatkan perfomansi pemrosesan *query* [20].
12. Penelitian ini berfokus pada pembangunan program yang dapat meningkatkan *synchronous update* pada *materialized view* dengan otomatis. Apabila terdapat fungsi trigger (insert, update, delete) pada *base table* maka *materialized view* harus tetap dilakukan *update* pada PostgreSQL. Setelah *trigger* dihasilkan, kemudian dilakukan eksekusi *synchronously incremental update* dari *materialized view*. *Synchronous update* adalah sebuah bagian transaksi yang membawa perubahan data pada *base table* di PostgreSQl. Sedangkan *asynchronous update* adalah sebuah *request* untuk melakukan perubahan data yang dikontrol oleh *user*. Adapun tujuan dari pembangunan program adalah peningkatan *synchronous update* yang diintegrasikan ke *source code* pada PostgreSQL yang mungkin lebih optimal. Peningkatan algoritma *update* dapat diintegrasikan ke PostgreSQL pada sebuah modul yang dimodifikasi pada *file matview.c* dan *createas.c*. Dalam penelitian ini dilakukan pengujian dari *trigger* yang dihasilkan program yang efektif dan membandingkan waktu yang dibutuhkan untuk melakukan *update* pada *base table* [21].

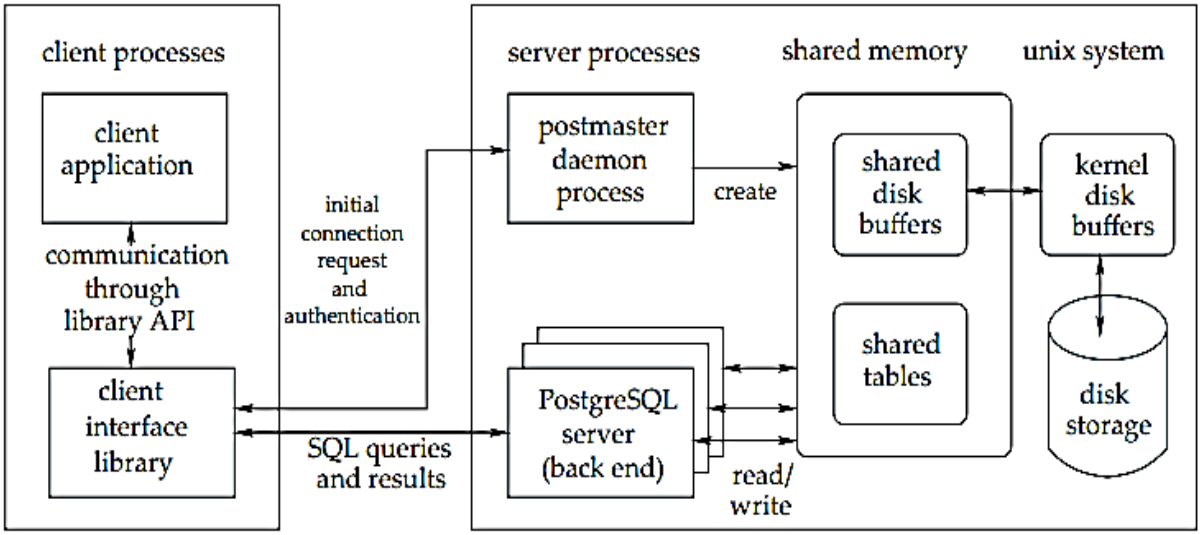
Dari beberapa *related work* yang telah diulas, dapat dilihat bahwa sudah dilakukan beberapa penelitian untuk mencari solusi dalam mengurangi *cost* terhadap *materialized view maintenance*. Namun, belum ditemukan penelitian yang melakukan pengurangan *cost* terhadap *materialized view maintenance* dengan meggunakan konsep *filtering* secara otomatis yang dilakukan langsung oleh sistem PostgreSQL.

2.15 PostgreSQL

PostgreSQL merupakan salah satu *Object Relational Database Management System* (ORDBMS) yang dapat diakses secara gratis. PostgreSQL mendukung banyak aspek dari SQL mendukung banyak aspek dari SQL dan menawarkan fitur seperti *query* yang kompleks, *foreign key, triggers, views, transactional integrity, full-text searching* dan replikasi data terbatas. PostgreSQL mendukung berbagai macam bahasa pemrograman (termasuk C, C++, Java, Perl, dan Python) juga sebagai antarmuka basis data JDBC dan ODBC [1].

2.15.1 Arsitektur PostgreSQL

PostgreSQL memiliki arsitektur berorientasi objek yang dibagai menjadi tiga subsistem besar. Bentuk arsitektur sistem yang digunakan PostgreSQL adalah sebagai berikut:



Gambar 3 Arsitektur Keseluruhan dari PostgreSQL [1]

1. *Client Interface Layer*

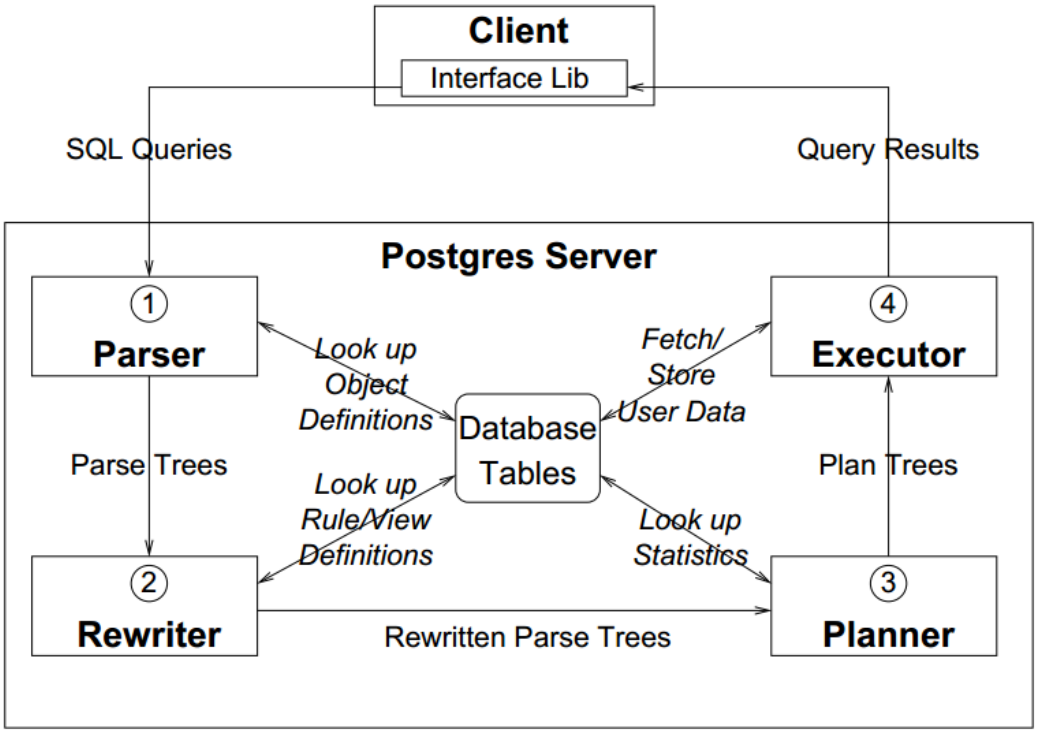
Client Interface Layer terdiri dari dua komponen utama: *client application* dan *client interface library. Client interface library* merupakan cara aplikasi agar dapat berkomunikasi ke server dengan menghubungkan koneksi dan autentikasi keke *postmaster daemon process*, dimana *client application* dapat dihubungkan ke PostgreSQL server dan dan mengirim *query* dari satu dari banyak basis data program aplikasi yang didukung oleh PostgreSQL (libpq, JDBC, ODBC, Perl DBD) yang disediakan pada *client-side* [22].

2. *Server Processes*

Server process terdiri dari dua komponen utama: *Postmaster* dan *Postgres Server* (*back-end*). Postmaster adalah sebuah thread daemon yang responsif terhadap penanganan

koneksi awal *client* dengan mengetahui *port*, kemudian postmaster mebuat sebuah proses server *back-end* baru untuk menangani sebuah *client*. Kemudian proses *back-end server* responsif untuk pengeksekusian *query* yang dikirimkan oleh *client* dengan menampilkan langkah eksekusi *query* mencakup *parsing*, *optimization* dan *execution*. Posgres Server berfungsi untuk meneruskan sebuah query SQL untuk kemudian diterjemahkan menjadi data hasil [22].

Adapun arsitektur umum *Postgres Server* adalah sebagai berikut:



Gambar 4 Arsitektur Konseptual dari Postgre Server [23]

Arsitektur *Postgre Server* dijelaskan sebagai berikut:

1. *Parser*

   Parser menerima *query* dalam bentuk teks ASCII. Setelah *query* diterima, pada tahap *parser* akan dilakukan pemotongan (*tokenized*), kemudian dipastikan bahwa SQL *query* tersebut memiliki sintaks yang *valid* [22].

2. *Rewriter*

   Rewriter merupakan tempat persinggahan sementara dari *query* sebelum menuju ke *planner/optimizer* untuk melakukan penulisan ulang kembali *query*. *Rewriter* harus responsif terhadap *rule system* pada PostgreSQL [1].

3. *Planner/Optimizer*

Query yang sudah di *rewrite* dilakukan perencanaan *estimation cost* yang minimum. Kemudian dioptimasi berdasarkan struktur *query plan* [1].

4. *Executor*

   Executor berfungsi untuk menerima rencana dari *planner/optimizer* dan dilakukan eksekusi *query plan* atau eksekusi *tree*, dan pengembalian *output* ke *client* [1].

3. *Database Store Layer*

Pada *data store layer* digunakan untuk mengelola ruang di dalam *disk* ketika data disimpan dalam basis data [22].

2.15.2 Penerapan *Materialized View* pada PostgreSQL

Penggunaan *materialized view* dalam PostgreSQL dimuali dengan membangun *materialized view* terlebih dahulu. Pembangunan *materialized view* dapat dilakukan dengan membangun *query* berikut.

```
CREATE MATERIALIZED VIEW table_name
        [ (column_name [, …] ) ]
        [ WITH (storage_parameter [= value] [. …] ) ]
        [ TABLESPACE tablespace_name ]
        AS query
        [ WITH [ NO ] DATA ];
```

Materialized view tidak dapat diperbaharui secara langsung, data baru pada *materialized view* dapat dihasilkan dengan:

1. Melakukan *Refresh* Terhadap Seluruh *Materialized View*

   Melakukan *refresh* terhadap seluruh *materialized view* bertujuan agar konsistensi data pada tabel materialized view tetap terjaga. Dalam penerapannya, melakukan *refresh* untuk seluruh *materialized view* akan membutuhkan waktu yang cukup lama untuk dijalankan. Teknik ini dapat diterapkan dengan *query* berikut [24].

```
SELECT string_agg (
            'REFRESH MATERIALIZED VIEW " ' || schemaname || ' "," ' || mvname || ' " ;
',
            E' \n' ORDER BY refresh_order) AS script
FROM mat_view_refresh_order \gset

\echo :script

:script
```

2. Melakukan *Refresh Materialized View* Terhadap Satu *Materialized View*

Teknik ini dilakukan saat hanya terdapat satu materialized view yang akan terpengaruh dikarenakan terjadi perubahan *pada base table.* Teknik ini dapat diterapkan dengan menggunakan *query* berikut [25].

```
REFRESH MATERIALIZED VIEW tablename;
```

2.14.2 *Trigger* pada Refresh *Materialized View*

Terdapat beberapa teknik yang dapat digunakan untuk meningkatkan kinerja *materialized view*, antara lain [26]:

1. *Snapshot*

Teknik ini baik digunakan saat terjadi perubahan yang cukup banyak pada *base table*, namun tidak baik untuk perubahan yang sedikit pada *base table.* Teknik ini dilakukan secara manual sesuai dengan kebutuhan pengguna.

2. *Eager*

Teknik ini memugkinkan *materialized view* diperbaharui segera setelah terjadi perubahan pada *base table*, namun dengan menggunakan ini akan terjadi kemungkinan terdapat data yang tidak *valid* dikarenakan penggunaan mutable function now (). Untuk menggunakan teknik ini, harus dibuatkan fungsi *trigger* untuk setiap tabel *materialized view*

3. *Lazy*

Perubahan data pada *materialized view* tidak langsung dilakukan apabila terjadi perubahan pada *base table.* Dengan menggunakan teknik ini, seluruh perubahan akan dikumpulkan terlebih dahulu dalam jangka waktu tertentu, dan akan akan dilakukan perubahan terhadap *materialized view* dalam sekali aksi.

4. *Very Lazy*

Kinerja teknik *very lazy materialized view* menyerupai teknik *snapshot* namun dengan pembaharuan yang lebih ringan dan cepat serta membutuhkan lebih sedikit sumber daya. Dengan teknik ini, baris yang perlu diperbarui akan di-*record.* Data akan menjadi tidak

sinkron segera setelah data berubah karena pembaharuan dilakukan sekaligus dan tidak *up to date.*

BAB 3. ANALISIS

Pada bab analisis dijelaskan analisis yang dilakukan terhadap masalah yang terjadi dalam menentukan bagian *source code* yang akan dimodifikasi untuk menempatkan algoritma yang tepat sehingga dapat diimplementasikan dalam PostgreSQL.

3.1 Metode Penelitian

Tahapan-tahapan yang dilaksanakan dalam penelitian ini adalah sebagai berikut.

1. Perumusan Masalah

   Pada tahap ini dilakukan perumusan malasah pada penelitian yang akan dilakukan. Hal yang akan melatarbelakangi penyebab sebuah masalah dapat dilihat berdasarkan hasil penelitian sebelumnya dan teori yang sudah ada yang telah dibahas pada bab landasan teori.
2. Pengumpulan Referensi Penelitian

   Pada tahap ini dilakukan pencarian serta pengumpulan referensi yang akan memperkuat latar belakang penelitian serta mendukung landasan teori. Referensi penelitian dapat berupa buku, paper serta *website* resmi basis data yang akan digunakan.
3. Pendefenisian Solusi

   Merujuk pada berbagai referensi yang diperoleh, peneliti akan melakukan analisis persolan lebih mendalam untuk menemukan solusi. Menentukan solusi dilakukan dengan memilah solusi yang ditawarkan oleh referensi-referensi yang diulas. Solusi-solusi yang dianggap menjadi solusi yang terbaik akan dianalisis lebih mendalam untuk menemukan satu solusi terbaik untuk diimplemetasikan.
4. Implementasi

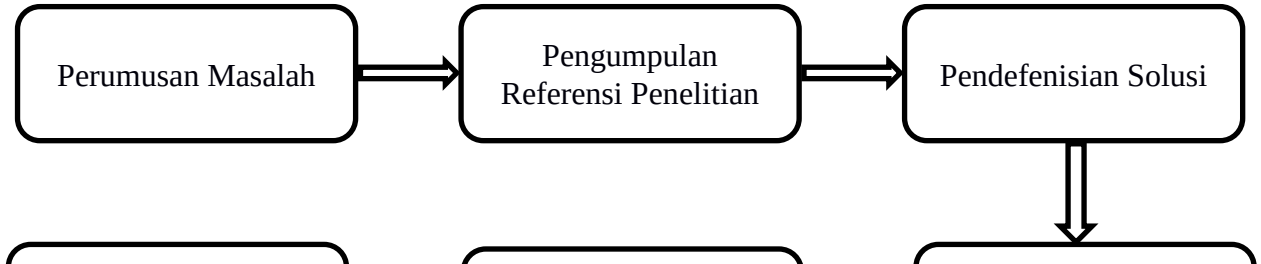
   Pada tahap ini dilakukan pengembangan solusi yang telah ditentukan sebelumnya. Pengimplementasian solusi dilakukan dengan pemrograman *tools* yang akan dikembangkan sebagai solusi dari permasalahan yang telah didefenisikan.
5. Komparatif

   Setelah dilakukan pengimplementasian solusi, akan didapat hasil yang akan digunakan sebagai pembanding. *Tools* solusi yang telah selesai dikembangkan akan dibandingkan dengan *tools* sebelum dilakukan penelitian.

6. Hasil

   Setelah didapat hasil dari perbandingan antara *tools* yang lama dengan *tools* yang telah dikembangkan, selanjutnya dapat dibuat kesimpulan apakah *tools* yang telah dimodifikasi lebih baik atau tidak dari pada *tools* sebelumnya. Pada tahap ini juga dapat dilakukan pemberian saran untuk pegembangan selanjutnya.

Tahapan-tahapan yang telah dijelaskan, dapat digambarkan kedalam diagram pada gambar 5 berikut ini.



Gambar 5 Metodologi Penelitian

3.2 Analisis *Cost*

Pada bab 2 telah dijelaskan konsep dari *base table*, *view*, dan *materialized view*. Dalam penggunaan basis data berukuran kecil, pengaksesan langsung terhadap *base table* merupakan hal yang lumrah untuk dilakukan. Namun pengaksesan langsung terhadap *base table* untuk basis data yang berukuran besar akan memerlukan *response time* yang cukup tinggi. Apabila ditinjau dari sisi *memory resources*, untuk melakukan pengaksesan langsung pada *base table* dibutuhkan *cost* untuk menjalankan seluruh proses pada *query processing* yaitu *parsing and translation*, *optimization* dan *evaluation engine.* Untuk melalui ketiga tahapan dalam *query processing* dibutuhkan *cost* yang cukup besar karena harus melalui seluruh proses secara keseluruhan.

Untuk mengurangi proses yang harus dilalui dalam pengaksesan data pada *database*, dapat digunakan teknik *view*. Dengan menggunakan *view*, *query* yang disimpan dalam *virtual table* dapat langsung dieksekusi tanpa harus melalui proses *parsing and translation* dan juga proses *optimization.* Hal ini akan mengurangi *cost* untuk pemrosesan *query* karena proses yang dijalankan adalah proses eksekusi *query.*

Pada teknik *view*, *query* yang disimpan pada *virtual table* tidak memiliki bentuk fisik, sehingga akan lebih optimal untuk menggunakan teknik *materialized view*. Dengan menggunakan *materialized view*, baik *query* dan hasilnya akan disimpan dalam suatu penyimpanan. Hal ini akan berdampak terhadap *cost* yang dibutuhkan dalam pengaksesan data karena dalam mengakses data, data yang dibutuhkan akan langsung diakses dari penyimpanan (*cache*) yang berisikan *materialized view.*

Untuk pembuktian pernyataan yang telah dipaparkan sebelumnya, dapat dilihat dari kasus berikut:

Terdapat suatu basis data “dvd rental” yang memiliki 10 tabel dengan asumsi ukuran 1 *tuple* pada tabel yang memiliki 8-10 kolom adalah 1024 bytes = 1 KB dan tabel yang memiliki 1-7 kolom adalah 512 *bytes* serta 1 blok = 4 KB.

Category Table

Tabel 7 *Category Table*

category_id	*Name*	*last_update*
1	*Action*	*2006-02-15 09:46:27*
…	*…*	*…*
30000	*Animation*	*2006-02-15 09:46:27*

Ukuran tabel: 512 *bytes* * 30000 = 15.360.000 byte = 15.000 KB

Maka, jumlah blok = 3.750 blok

City Table

Tabel 8 *City Table*

city_id	***city***	***country_id***	***last_update***
1	*Abha*	*87*	*2006-02-15 09:45:25*
…	*…*	*…*	*…*
20000	*Zalantun*	*82*	*2006-02-15 09:45:25*

Ukuran Tabel: 512 *bytes* * 20.000 = 10.240.000 bytes = 10.000 KB

Maka, jumlah blok = 2.500 blok

Country Table

Tabel 9 *Country Table*

country_id	***Country***	***last_update***
1	*Afghaninstan*	*2006-02-15 09:44:00*
…	*…*	*…*
200	*Zambia*	*2006-02-15 09:44:00*

Ukuran Tabel: 512 *bytes* * 200 = 102.400 bytes = 100 KB

Maka, jumlah blok = 25 blok

Address Table

Tabel 10 *Addess Table*

Address_id	*Address*	*district*	*city_id*	*postal_code*	*phone*	*last_update*
1	*47 MySakila Drive*	*Alberta*	*300*	*35200*	*14033335568*	*2006-02-15 09:45:30*
…	*…*	*…*	*…*	*….*	*…*	*…*
400000	*28 Workhaven Lane*	*QLD*	*576*	*17886*	*61722235667*	*2006-02-15 09:45:30*

Ukuran tabel: 512 *bytes* * 400.000 = 204.800.000 bytes = 200.000 KB

Maka, jumlah blok: 50.000 blok

Customer Table

Tabel 11 *Customer Table*

customer_ id	***store_ id***	***first_ name***	***last_ name***	***Email***	***address_ id***	***active_ bool***	***create_ date***	***last_update***	***active***
1	*1*	*Jared*	*Ely*	*jared.ely@sakil acustomer.org*	*5*	*True*	*2006-02-14*	*2013-05-26 14:49:45*	*1*
...	*...*	*...*	*...*	*...*	*...*	*...*	*...*	*...*	*...*
1000000	*2*	*Mary*	*Smith*	*mary.smit@saki lacustomer.org*	*7*	*True*	*2006-2-14*	*2013-05-26 14:49:45*	*1*

Ukuran tabel: 1 KB * 1.000.000 = 1.000.000 KB

Maka, jumlah blok: 250.000 blok

Film Table

Tabel 12 *Film Table*

film_id	***tittle***	***description***	***release _year***	***rental _duration***	***replacement _cost***	***last_update***	***special_feature***
1	*Academy Dinosaur*	*A Fateful Reflection of a Moose And a Husband who must Overcome a Monkey in Nigeria*	*2006*	*7*	*14.99*	*2013-05-26 14:50:58.951*	*{Trailers}*
...	*...*	*...*	*...*	*...*	*...*	*...*	*...*
100000	*Zorro Ark*	*A Intrepid Panorama of a Mad Scientist And a Boy who must Redeem a Boy in A Monastery*	*2006*	*3*	*18.99*	*2013-05-26 14:50:58.951*	*{Trailers,Commentaries,"Behind the Scenes"}*

Ukuran tabel: 1 KB * 100.000 = 100.000 KB

Maka, jumlah blok: 25.000 blok

Film category table

Tabel 13 Film *Category Table*

film_id	***category_id***	***last_update***
1	1	2006-02-15 10:02:19
.....	…	…
100000	30000	2006-02-15 10:02:19

Ukuran tabel: 512 *bytes* * 100.000 = 51.200.000 bytes = 50.000 KB

Maka, jumlah blok: 12.500 blok

Inventory Table

Tabel 14 Inventory Table

inventory_id	***film_id***	***Jumlah***	***last_update***

1	1	23	2006-02-15 10:09:17
…	…	…	…….
100000	100000	34	2006-02-15 10:09:17

Ukuran tabel: 512 *bytes* * 100.000 = 51.200.000 bytes = 50.000 KB

Maka, jumlah blok: 12.500 blok

Payment Table

Tabel 15 *Payment Table*

payment_id	***customer_id***	***rental_id***	***amount***	***payment_date***
1	341	1520	7.99	2007-02-15 22:25:46
…	…	…	……..	……..
1200000	264	14243	5.99	2007-02-15 19:36:27

Ukuran tabel: 512 *bytes* * 1.200.000 = 614.400.000 bytes = 600.000 KB

Maka, jumlah blok: 150.000 blok

Rental Table

Tabel 16 *Rental Table*

rental_id	***rental_date***	***inventory_id***	***customer_id***	***return_date***	***last_update***
1	2005-05-24 22:54:33	1525	459	2005-05-28 22:54:33	2005-06-08 21:51:33
…	……..	…	…	…	…….
1100000	2005-05-28 19:40:33	2666	393	2005-05-30 19:40:33	2005-06-07 11:31:22

Ukuran tabel: 512 *bytes* * 1.100.000 = 563.200.000 bytes = 550.000 KB

Maka, jumlah blok: 137.500 blok

Dari 10 tabel diatas, diperoleh hasil sebagai berikut:

Tabel 17 Tabel Hasil Perhitungan

Nama Tabel	Jumlah *Tuple*	Size/*Tuple (bytes)*	Ukuran Tabel (KB)	Jumlah Blok
Category	30000	512	15.000	3.750
City	300000	512	10.000	2.500
Country	200	512	100	25
Address	400000	512	200.000	50.000
Customer	1000000	1024	1.000.000	250.000
Film	100000	1024	100.000	25.000
Film category	100000	512	50.000	12.500
Inventory	100000	512	50.000	12.500
Payment	1200000	512	600.000	150.000
Rental	1200000	512	550.000	137.500

1. Menghitung *Cost* Terhadap *Base Table*

Cost untuk *query evaluation* dapat diukur beberapa terminologi dengan sumber daya yang digunakan yaitu *disk access*, waktu CPU untuk eksekusi *query* dan *cost* pada sistem basis data yang terdistribusi atau paralel. Pada sistem basis data yang besar, *cost* untuk mengakses data dari *disk* adalah *cost* yang paling penting untuk mengukur *query evaluation plan*.

Secara konseptual dalam perhitungan *cost* dari *query evaluation plan* adalah sebagai berikut:

Sebagai contoh, dengan meggunakan tabel *payment* pada basis data dvd rental, untuk memilih *tuple* dari semua pembayaran rental. Operasi *Select* dapat ditunjukkan pada *query* sebagai berikut:

```
SELECT * FROM Payment
```

Sebuah tabel *Payment* memiliki 1200000 tuple. Waktu rata-rata dari t_T untuk transfer sebuah block data ≈ 0.1 *miliseconds*, membutuhkan t_S waktu rata-rata *block access* ≈ 4 *miliseconds*, dan membutuhkan 3 *disk seek*. Untuk mencari blok transfer dapat menggunakan persamaan (2) yang terdapat dalam subbab 2.5 :

Total *block* dari tabel *Payment* adalah 150.000 blok

150.000/2 = 75.000 blok rata-rata.

Maka dengan menggunakan persamaan (1) pada bab 2.5. Didapat *cost* untuk *query plan* adalah sebagai berikut: 75.000*0.1+3*4 = 7.512 t

2. Menghitung *cost* untuk kasus pengaksesan terhadap *Nested Loop Join* pada *Base Table*

Untuk menampilkan daftar *customer* dan *city* dari Indonesia dapat dilakukan dengan menggunakan *query* berikut.

```
SELECT c.first_name AS customer, ci.city AS  city
        FROM (((country co
          JOIN city ci ON ((co.country_id = ci.country_id)))
          JOIN address a ON ((ci.city_id = a.city_id)))
          JOIN customer c ON ((a.address_id = c.address_id)))
        WHERE co.Country = 'Indonesia';
```

Dari *query* diatas, dapat dilihat bahwa terdapat empat tabel yang digabungkan (*join*). Dimana tabel *country* merupakan relasi *outer (r)* dan tabe*l city*, *address* dan *customer* merupakan relasi *inner (s).*

1. *Join* Tabel *customer* dan *address*

   Untuk menghitung total *tuple* yang di-*scan* dapat digunakan rumus: $n_r * n_s$

   Sehingga didapat total *tuple* = 400.000 * 1.000.000 = $4 * 10^{11}$ *tuple*

   Untuk mencari *cost* proses *nested join* dapat digunakan persamaan (3)

   Maka, total *cost* = 400.000 * 250.000 + 50.000 plus 400.000 + 50.000 *seek*

   = 100.000.050.000 t plus 450.000 *seek*

2. *Join* Table (*customer join address*) dan *city*

   Untuk menghitung total *tuple* yang di-*scan* dapat digunakan rumus: $n_r * n_s$

   Sehingga didapat total *tuple* = 300.000 * 1.000.000 = $3 * 10^{11}$ *tuple*

   Untuk mencari *cost proses nested join* dapat digunakan rumus $n_r * b_s + b_{r,}$

   Maka, total *cost* = 300.000 * 250.000 + 2.500 plus 300.000 + 2.500 *seek*

= 75.000.002.500 t plus 302.500 *seek*

3. *Join* Tabel (*customer join address join city*) dan *country*

   Untuk menghitung total *tuple* yang di-*scan* dapat digunakan rumus: $n_r * n_s$

   Sehingga didapat total *tuple* = 200 * 1.000.000 = 2 * 10^8 *tuple*

   Untuk mencari *cost proses nested join* dapat digunakan rumus $n_r * b_s + b_{r,}$

   Maka, total *cost* = 200 * 250.000 + 25 plus 200 + 25 *seek*

   = 50.000.025 plus 225 *seek*

Maka, total *cost* untuk menampilkan daftar *customer* dan *city* yang berasal dari Indonesia adalah = *Cost join* tabel *customer* dan *address* + *Cost join* tabel (*customer join address*) dan *city* + *cost join* tabel (*customer join address join city*) dan *country.*

Sehingga didapat hasil = 100.000.050.000 t *plus* 450.000 *seek* + 75.000.002.500 t *plus* 302.500 *seek* + 50.000.025 *plus* 225 *seek* = 175.050.052.525 t plus 752.725 *seek.*

3. Menghitung *Cost* Terhadap *View*

Berdasarkan salah satu penelitian yang terdapat dalam bab 2, dikatakan bahwa *query cost* yang dibutuhkan pada *view* sama dengan *query cost* yang dibutuhkan pada saat pengaksesan data secara langsung pada *base table.* Hal ini dikarenakan dalam penggunaan *view, query* disimpan dalam *virtual table* tanpa bentuk fisik sehingga masih harus dilakukan proses eksekusi *query* untuk mengakses data.

4. Analisis *Cost* Terhadap *Materialized View*

Dari tabel yang telah tersedia, dapat dilakukan create materialized view dengan mengeksekusi *query* dibawah ini.

```
CREATE MATERIALIZED VIEW customer_from_indonesia
AS
 SELECT c.first_name AS cutomer,
    ci.city AS city
   FROM (((country co
     JOIN city ci ON ((co.country_id = ci.country_id)))
     JOIN address a ON ((ci.city_id = a.city_id)))
     JOIN customer c ON ((a.address_id = c.address_id)))
WHERE co.Country = 'Indonesia'
WITH NO DATA;
```

Dari *query* diatas, didapat hasil:

Tabel 18 Tabel *Materialized View customer_from_indonesia*

customer	***city***
Norman	Cianjur
…	…
Leslie	Pontianak

```
CREATE MATERIALIZED VIEW rental_by_charles
AS
 SELECT c.first_name AS customer,
    f.title AS film_rental
   FROM (((film f
     JOIN inventory i ON ((f.film_id = i.film_id)))
     JOIN rental r ON ((i.inventory_id = r.inventory_id)))
     JOIN customer c ON ((r.customer_id = c.customer_id)))
 WHERE c.first_name = 'Charles'
 WITH NO DATA;
```

Dari *query* diatas, didapat hasil:

Tabel 19 Tabel Materialized View rental_by_charles

customer	***film_rental***
Charles	Hamlet Wisdom
…	…
Charles	Circus Youth

```
CREATE MATERIALIZED VIEW rental_by_category
AS
 SELECT c.name AS category,
    sum(p.amount) AS total_sales
   FROM (((((payment p
     JOIN rental r ON ((p.rental_id = r.rental_id)))
     JOIN inventory i ON ((r.inventory_id = i.inventory_id)))
     JOIN film f ON ((i.film_id = f.film_id)))
     JOIN film_category fc ON ((f.film_id = fc.film_id)))
     JOIN category c ON ((fc.category_id = c.category_id)))
  GROUP BY c.name
  ORDER BY sum(p.amount) DESC
 WITH NO DATA;
```

Dari query diatas, didapat hasil:

Tabel 20 Tabel Materialized View rental_by_category

category	***total_sales ($)***
Sports	4892.19
…	…
Animation	4245.31

Dari *query* yang telah dieksekusi dihasilkan tiga *materialized view* yaitu customer_from_indonesia, rental_by_category, dan rental_by_charles.

Apabila akan ditampilkan daftar *customer* dan *city* dari customer yang berada di Indonesia dan data diakses dari *materialized view*, maka daftar tersebut akan langsung diakses dari tabel

materialized view. Hal ini akan berdampak pada *cost* yang dibutuhkan. Saat mengakses data pada *materialized view*, tidak diperlukan proses *join* dari beberapa tabel yang akan meyebabkan *cost* pengaksesan jauh lebih kecil dibanding harus mengakses langsung pada *base table.* Hal ini dapat dilihat dari perhitungan berikut.

Dibutuhkan data *customer* dan *city* asalnya dari negara Indonesia. Data ini diakses dari tabel *materialized view*. Apabila dimisalkan waktu rata-rata dari t_T untuk transfer sebuah block data ≈ 0.1 *miliseconds*, membutuhkan t_S waktu rata-rata *block access* ≈ 4 *miliseconds*, dan membutuhkan 3 *disk seek.* Diketahui juga total *customer* yang berasal dari Indonesia adalah 100 *tuple*, dan kolom didalam tabel *materialized view customer_from_indonesia* adalah 2 kolom yaitu *customer* dan *city*. Diasumsikan bahwa ukuran 1 *tuple* adalah 512 *bytes*. Maka, untuk mencari blok transfer dapat menggunakan persamaan (2) yang terdapat dalam bab 2:

Total *block* dari tabel *customer_from_indonesia* adalah 100 * 512 bytes = 51.200 / 1.024 = 50 KB / 4 KB = 13 blok.

13/2 = 7 blok rata-rata,

Maka, dengan menggunakan persamaan (1) didapat total *cost* yaitu sebagai berikut: 7*0.1+3*4 = 12.7 *milisecond*

Maka, dari perhitunngan yang telah dilakukan didapat hasil:

Pengaksesan terhadap *base table:* 175.050.052.525 t plus 752.725 *seek*

Pengaksesan dengann *view*: 175.050.052.525 t plus 752.725 *seek*

Pengaksesan terhadap *materialized view*: 12.7 *mililsecond*

Maka, dari hasil yang didapat, dapat disimpulkan bahwa *cost* saat pengaksesan data dengan menggunakan *materialized view* jauh lebih rendah dari pada pengaksesan data langsung pada *base table.*

3.3 Analisis *Refresh Materialized View*

1. *Insertion*

Dilakukan *insertion* data kedalam tabel *inventory* pada base table dengan *query*:

```
INSERT INTO inventory (inventory_id, film_id, jumlah) VALUES (1001, 1000, 32);
```

Hal ini menyebabkan perubahan pada *base table* yang akan mempengaruhi *materialized view* apabila dilakukan *update* pada *materialized view.*

2. *Deletion*

Dilakukan *deletion* data dari tabel rental pada *base table* dengan *query*:

```
DELETE from rental WHEN rental_id = 8;
```

Proses ini akan menyebabkan perubahan pada *base table* yang akan mempengaruhi *materialized view* apabila dilakukan *update* pada *materialized view***.**

3. *Updating*

Dilakukan *update* data pada tabel *customer* dengan *query:*

```
UPDATE customer
SET last_name = 'Munthe'
WHERE customer_id = 2
RETURNING customer_id,
last_name;
```

Proses ini akan menyebabkan perubahan pada *base table* yang akan mempengaruhi *materialized view* apabila dilakukan *update* pada *materialized view***.**

Apabila setelah terjadi perubahan pada *base tabel, materialized view* tidak *di-update* maka data pada *materialized view* menjadi tidak konsisten. Oleh karena itu, *materialized view* harus selalu di-*update* setelah terjadi perubahan pada *materialized view.*

Insertion yang terjadi pada tabel *inventory* akan berpengaruh juga terhadap tabel rental karena *inventory_id* merupakan *foreign key* pada tabel rental. Tabel *inventory* dan rental akan berpengaruh terhadap *materialized view rental_by_charles* dan *materialized view rental_by_category.* Namun pada saat dilakukan *update* pada *materialized view* akan dilakukan pembaharuan untuk seluruh *materialized view.*

Deletion yang terjadi pada tabel rental akan berpengaruh terhadap tabel rental sendiri dan juga tabel *payment* karena *rental_id* yang merupakan *primary key* pada tabel rental menjadi *foreign key* pada tabel *payment.* Perubahan ini juga akan berpengaruh terhadap *materialized view* yang ada yaitu pada *materialized view rental_by_charles* dan *rental_by_category.* Namun sama halnya dalam *insertion,* saat *materialized view* di-*update,* maka bukan hanya dua *materialized view* yang di-*update* namun seluruh *materialized view* yang ada.

Updating yang terjadi pada tabel *customer* akan berpengaruh terhadap tabel *customer* saja karena yang diperbaharui berada pada kolom *last_name* yang hanya digunakan pada tabel *customer* saja. Maka, saat dilakukan *update* terhadap *materialized view* hanya akan berpengaruh terhadap *materialized view customer_from_indonesia.* Pada *materialized view rental_by_charles* juga digunakan tabel *customer* namun *value* yang digunakan hanya *customer* bernama Charles, sementara *value* yang berubah berada

pada *customer_id = 2* dan mengubah *value* pada kolom *last_name* menjadi "Munthe". Dalam hal ini terlihat bahwa perubahan tidak berpengaruh terhadap *materialized view rental_by_charles* sehingga tidak dibutuhkan pembaruann untuk *materialized view* tersebut. Namun, karena perubahan terjadi pada *base table*, seluruh materialized view akan tetap di-*rebuilt* ulang.

Dari ketiga kasus diatas, mungkin saja dilakukan *update* satu persatu *materialized view* dikarenakan dalam kasus ini hanya terdapat tiga *materialized view.* Namun, pada umumnya teknik *materialized view* digunakan untuk data yang cukup besar sehingga *materialized view* akan di-*update* secara keseluruhan. Dilakukannya *update* satu per satu secara manual oleh *user* terhadap seluruh *materialaized view* untuk data yang besar dan memiliki banyak *materialize view* bukanlah pilihan yang tepat karena tidak efisien.
Apabila ditinjau dari penggunaan *cost, update* terhadap *materialized view* secara keseluruhan akan membutuhkan *cost* yang besar karena materialized view yang tidak perlu untuk di-*update* ikut serta di-*update.* Namun, apabila melakukan *update* satu-per satu materialized view secara manual akan membutuhkan waktu yang cukup lama karena harus dianalisis terlebih dahulu *base table* yang terjadi perubahan, hubungan antar *base table* yang mengalami perubahan dan menganalisis *materialized view* yang terkena dampak dari perubahan yang terjadi pada *base table.*

Untuk menangani hal ini, ditemukan solusi yaitu dengan menggunakan *trigger*. Dengan menggunakan *trigger materialized view* akan diperbaharui secara otomatis sesuai dengan ketentuan *trigger* yang digunakan, namun untuk mengggunakan *trigger*, seluruh *materialized view* harus ditandai dengan *trigger* terlebih dahulu. Misalkan sebuah basis data terdiri dari 100 *materialized view*, maka harus dilakukan pendefenisian trigger untuk 100 *materialized view*. Teknik *trigger* ini akan menyulitkan pengguna basis data untuk awal pemakaian. *Trigger* yang digunakan adalah *snapshot materialized view, very lazy materialized view, lazy materialized view,* dan juga *eager materialized view*. Namun, penggunaan *trigger* ini sering terkendala dalam masalah waktu yang akhirnya menyebabnya data sering menjadi tidak valid.

Maka, dapat disimpulkan bahwa kontrol dalam melakukan refresh materialized view berada pada kepputusan *user.* Apakah *user* akan melakukan *refresh materialized view* secara manual atau melakukan *refresh materialized view* secara otomatis dengan menggunakan *trigger*.

3.4 Analisis Waktu *Refresh Materialized View*

Terdapat dua penerapan waktu untuk melakukan *refresh* pada *materialized view*, yaitu:

1. Segera setelah terjadi perubahan pada *base table* (*Immediate*). Dalam hal ini, *materialized view* langsung diperbaharui setelah perubahan terjadi pada *base tabel.* Penerapan waktu ini baik untuk menjamin konsistensi data setiap waktu data diakses dari *materialized view*.

2. Menentukan waktu untuk melakukan *refresh materialized view* (*Deferred*). Dalam hal ini, perubahan terhadap *materialized view* dilakukan apabila terjadi permintaan untuk data yang *update* pada *materialized view.* Ataupun apabila telah ditentukan waktu untuk melakukan pembaharuan terhadap *materialized view.* Penerapan waktu ini sangat rentan terhadap konsistensi data pada *materialized view* dikarenakan terdapat kemungkinan saat data diakses dari *materialized view,* belum dilakukan pembaharuan terhadap *materialized view.* Namun, dengan penerapan waktu ini dapat meminimalisir *cost* karena frekuensi pembaharuan terhadap *materialized view* lebih kecil.

   Dari penerapan kedua waktu tersebut, akan lebih baik untuk melakukan pembaharuan *materialized view* segera setelah perubahan pada *base tabel base tabel* terjadi. Hal ini dilakukan agar konsistensi data pada *materialized view* tetap terjaga sepanjang waktu. Namun, menetukan waktu untuk melakukan *refresh* pada *materialized view* juga baik digunakan untuk kebutuhan data yang tidak terlalu sensitif terhadap data yang selalu data. Sehingga dapat disimoulkan bahwa penggunaan kedua waktu ini tergantung pada kebutuhan *user.*

3.5 Analisis Baris dan Kolom saat Memperbaharui *Materialized View*

Saat menerapkan perubahan yang terjadi pada *base table* terhadap *materialized view,* harus dilakukan identifikasi baris dan kolom pada *materialized view* yang berhubungan pada baris dan kolom yang mengalami perubahan pada *base table.* Sebagai contoh dapat dilihat pada kasus dibawah ini.

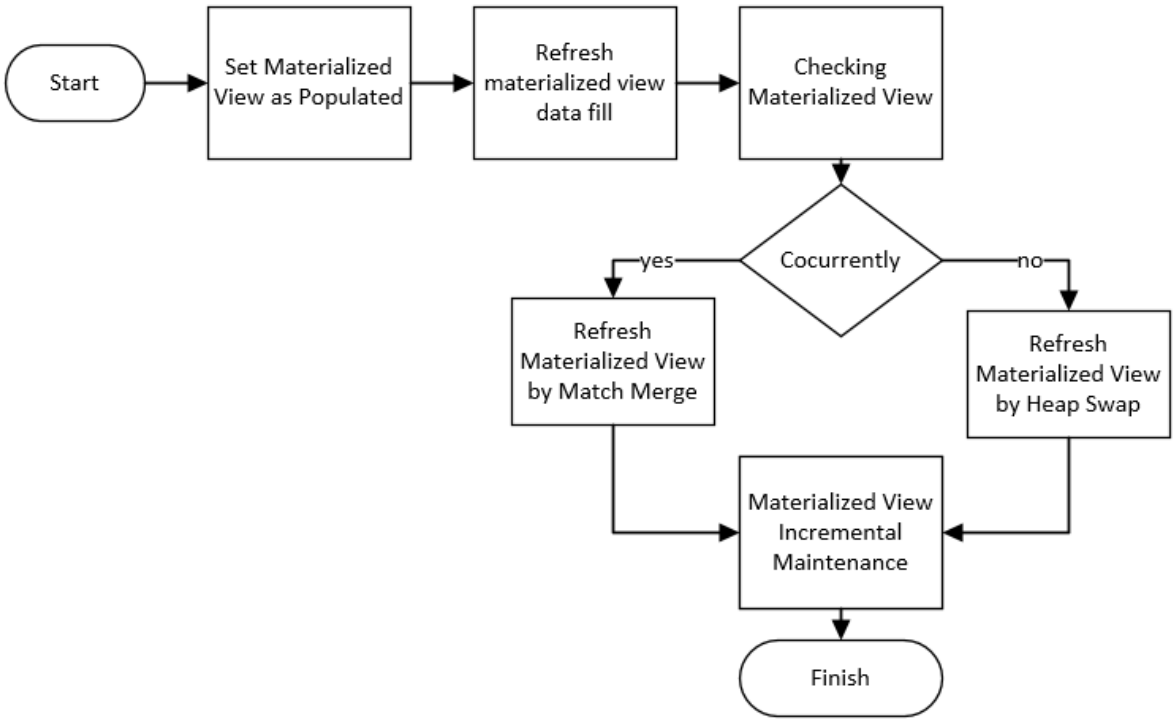
Terdapat *materialized view customer_from_indonesia* yang berisikan jumlah penjualan dari setiap kategori film. Apabila dilakukan *update* untuk jumlah penjualan salah satu *film* pada kategori *animation,* akan sulit untuk memperbaharui *materialized view customer_from_indonesia* karena dalam *materialized view* ini, seluruh film telah digabungkan kedalam satu kategori sehingga sulit untuk menemukan dalam kategori mana film yang di-*update* berada.

Tabel 21 Tabel *Materialized View customer_from_indonesia*

category	***total_sales ($)***
Sports	4892.19
…	…
Animation	4245.31

3.6 Analisis Tahapan *Refresh Materialized View* dalam *PostgreSQL*

Refresh materialized view adalah proses untuk melakukan pembaharuan terhadap *materialized view*. Fungsi untuk *refresh materialized view* pada PostgreSQL versi 11.1 yang terdapat pada *file* matview.c yang dengan lokasi *file* yaitu postgresql-11.1/src/backend/commands/matview.c. Proses melakukan eksekusi perintah REFRESH MATERIALIZED VIEW menggunakan fungsi ExecRefreshMatView() dengan flow proses sebagai berikut.



Gambar 6 Tahapan *Refresh Materialized View*

Penjelasan proses *refresh materialized view* adalah sebagai berikut.

1. *Set Materialized View as Populated*

   Berikut adalah proses menandai *materialized view* yang ada pada relasi apakah *populated* atau tidak. Sebelum dilakukan pengeksekusian *refresh* dipastikan *materialized view* tersebut dapat dilakukan *generate* atau tidak, dengan mendapatkan nama relasi dari *materialized view* tersebut.

2. *Refresh Materialized View Data Fill*

Proses ini memastikan bahwa perencanaan terhadap *query* yang telah mengubah isi basis data saat dilakukan *refresh* akan dikirim ke dalam argumen DestReceiver sebagai target tabel materialized view akan disimpan pada tempat persinggahan sementara atau *transient*.

3. *Checking Materialized View*

Pada proses ini menjelaskan tentang pengecekan yang terjadi pada materialized view saat dilakukan eksekusi *refresh*. Pengecekan yang dilakukan adalah memastikan bahwa *query refreshing* untuk target *materialized view* tersebut merupakan sebuah *materialized view*, kemudian memastikan *refresh* yang dilakukan *concurrent* adalah *materialized view* yang *populated* untuk dapat dilakukan *refresh materialized view*. Jika *materialized view concurrent* maka dilakukan *refresh materialized view by match merge*, jika tidak maka dilakukan *refresh materialized view by heap swap*.

4. *Refresh Materialized View by Match Merge*

Pada proses ini, fungsi refresh_by_match_merge adalah *refresh materialized view* dengan menggunakan *statement concurrently* karena memiliki Exclusivelock untuk materialized view yang terlebih dahulu dilakukan *refresh+*. Kemudian menemukan *unique index* di beberapa tabel *materialized view* dengan menggunakan *full outer join* untuk melakukan *refresh materialized view*. Jika *index* ditemukan kemudian di beberapa tabel tidak ada data *duplicate* dan *null* maka akan di-*insert* ke tabel *temporary* kemudian dilakukan *refresh materialized view.*

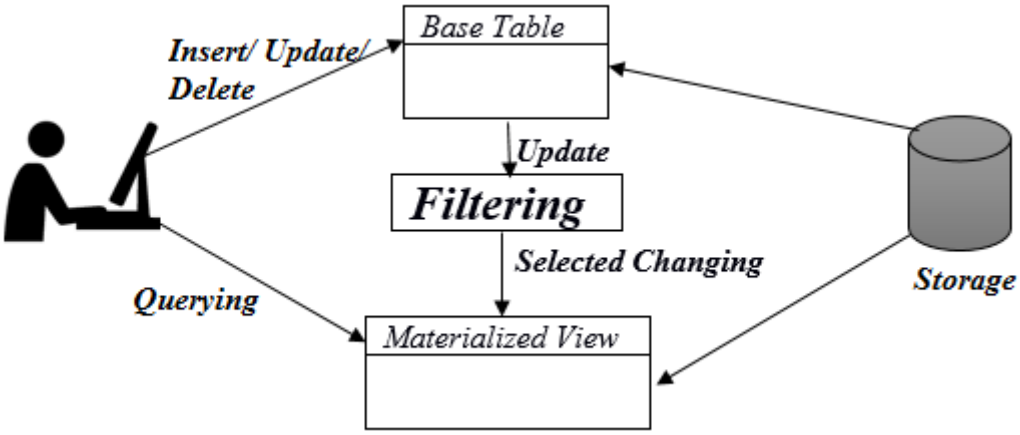
5. *Refresh Materialized View by Heap Swap*

Pada proses ini, fungsi refresh_by_heap_swap adalah *refreh materialized view* dengan menukar target tabel dengan tabel sementara dengan *index* yang terdapat pada *target table*, kemudian tabel sementara dihapus.

4. DESAIN

Pada bab ini dijelaskan proses perancangan yang dilakukan untuk pembangunan algoritma *filtering update* pada *materialized view*.

4.1 Desain Arsitektur Sistem



Gambar 7 Desain Arsitektur Sistem

Dari gambar 7 yang merupakan desain arsitektur sistem, dapat dilihat bagian-bagian penting penyusun sistem diantaranya:

1. *User*

   User merupakan pengguna dari basis data yang melakukan manajemen pada data dalam basis data.

2. *Storage*

   Storage merupakan tempat penyimpanan data dan didalamnya terdapat tabel-tabel dan juga materialized view yang akan diolah untuk kepentingan pengguna.

3. *Base Table*

   Base table merupakan tabel-tabel yang saling berhubungan satu dengan tabel lainnya dalam basis data. Data yang terdapat pada basis data dikelompokkan ke dalam tabel-tabel yang terdapat dalam basis data.

4. *Materialized View*

   Materialized view merupakan tabel yang berisikan *query* dan hasil dari *query* yang telah dibangun sebelumnya. Data yang berada dalam tabel *materialized view* berasal dari base table. Konsistensi data pada *materialized view* harus tetap selalu dijaga agar selalu sinkron dengan data yang berada pada *base table.*

5. *Filtering*

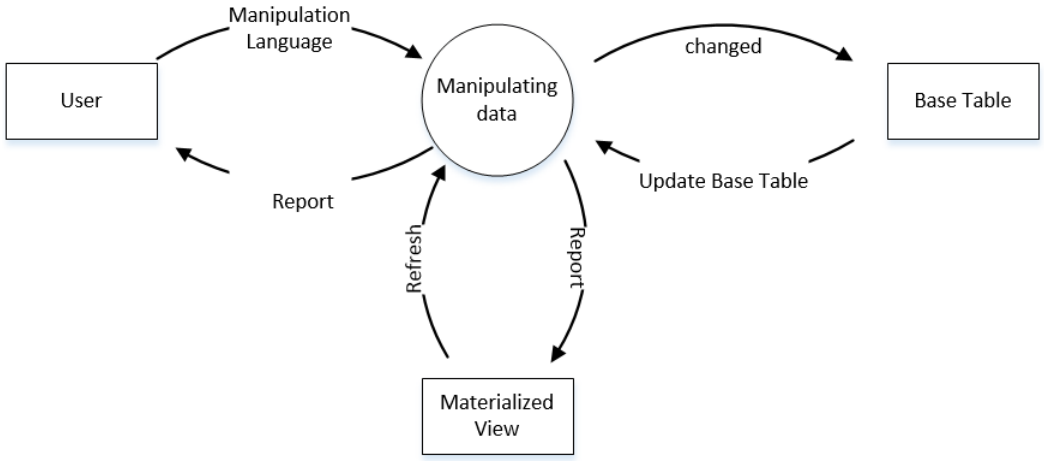
   Filtering merupakan fungsi yang akan ditambahkan dalam tahapan *refresh materialized view*. Dengan fungsi *filtering* ini, akan dipisahkan *base table* yang berpengaruh dan yang tidak berpengaruh terhadap *materialized view*. Tabel yang berpengaruh akan diteruskan ke *materialized view* secara otomatis oleh sistem tanpa menunggu perintah

refresh dari *admin user* dan data dalam *materialized view* akan digantikan dengan data yang baru yang berasal dari *base table.*

4.2 Desain Proses *Filtering*

Fungsi *filtering* adalah fungsi yang akan meyeleksi perubahan yang terjadi pada *base table* yang akan berpengaruh terhadap pembaharuan *materialized view*. Proses *filtering* ini dapat dilihat pada gambar berikut.

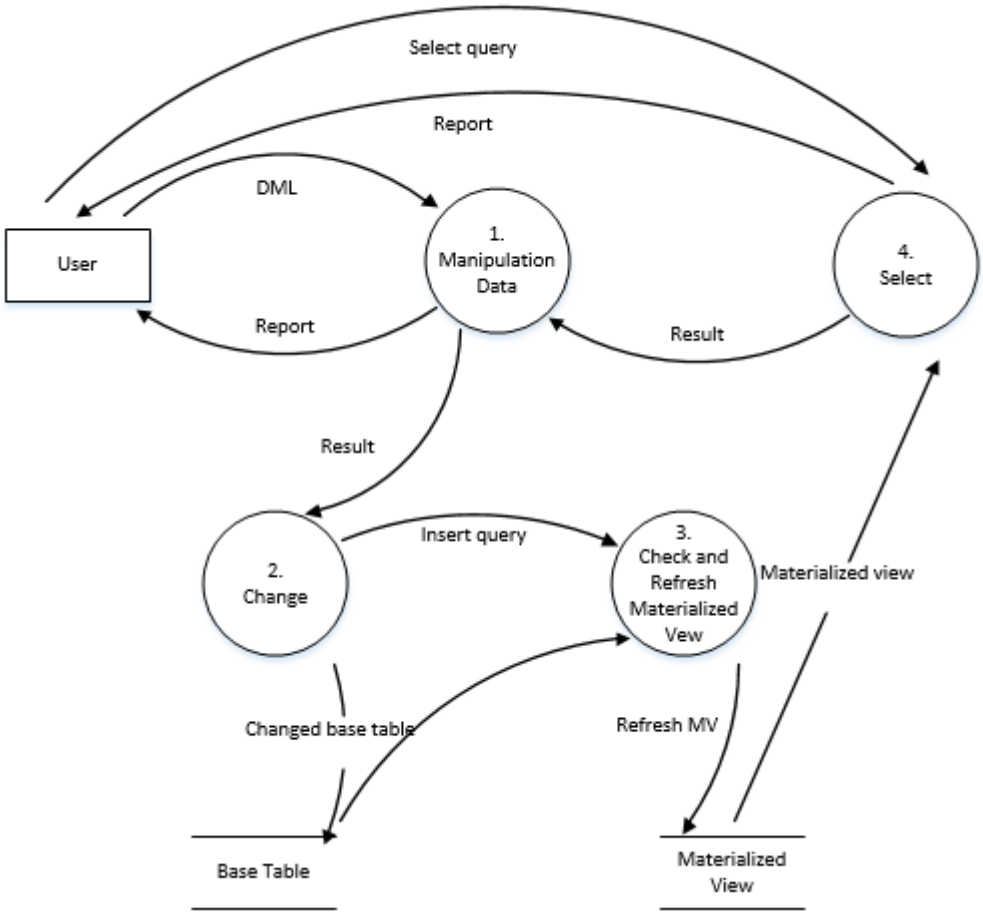
4.2.1 *Context Diagram* Proses *Filtering*



Gambar 8 *Context Diagram* Proses *Filtering*

User melakukan manipulasi data untuk merubah data pada *base table*. Dimana *user* akan mengirimkan *query* ke sistem untuk menambahkan, menghapus ataupun memperbaharui data yang terdapat pada *base table.* Apabila *query* yang dibuat oleh *user* benar, maka sistem akan menampilkan notifikasi bahwa *query* sukses dieksekusi. Selanjutnya sistem akan memperbaharui data yang terdapat pada *base table.* Kemudian sistem akan memiliki *base table* dengan data yang telah diperbaharui oleh *user*. Setelah sistem menerima *base table* yang telah mengalami perubahan, maka sistem akan langsung memperbaharui *materialized view* yang berhubungan dengan *base table* yang sudah terjadi perubahan. Selanjutnya, sistem akan menerima tabel *materialized view* yang telah memiliki data yang konsisten dan sesuai dengan data yang terdapat pada *base table.* Apabila materialized view telah mengalami perubahan, maka sistem akan menampilkan notifikasi bahwa *materialized view* sukses dilakukan perubahan.

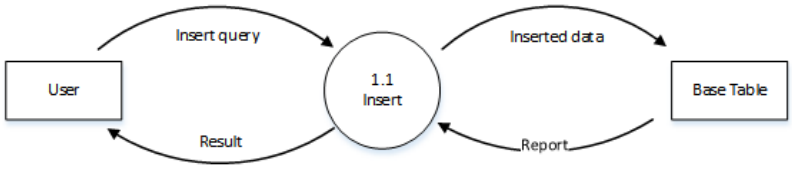
4.2.2 DFD Level 1 Proses *Filtering*



Gambar 9 DFD Level 1 Proses *Filtering*

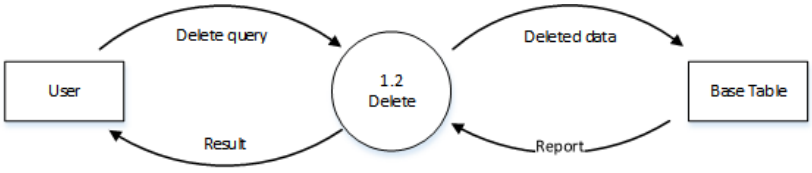
User melakukan manipulasi terhadap data yang terdapat pada *base table.* Dimana manipulasi yang dilakukan adalah menambahkan, menghapus ataupun memperbaharui data yang terdapat pada *base table.* Kemudian *base table* yang sudah dilakukan manipulasi akan dilakukan pemeriksaan terhadap *materialized view* yang memiliki dampak perubahan pada *base table* saja yang akan dilakukan *refresh materialized view.* Setelah itu *user* dapat mengakses *materialized view* yang memiliki data yang konsisten dan sesuai dengan data yang terdapat pada *base table*.

4.2.3 DFD Level 2 Proses *Querying*



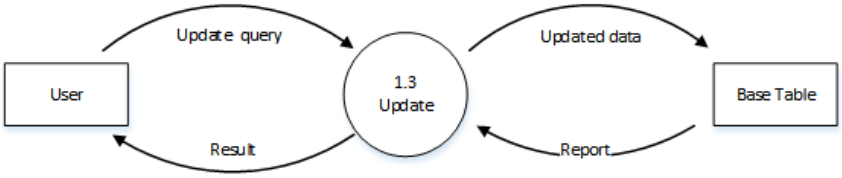
Gambar 10 DFD Level 2 Proses *Insert*

Saat *user* melakukan *query insert* ke dalam *base table*, data pada *base table* akan bertambah sesuai dengan data yang di-*insert* oleh *user.*



Gambar 11 DFD Level 2 Proses *Delete*

Saat *user* melakukan *query delete* terhadap data pada *base table*, data pada *base table* akan berkurang sesuai dengan data yang dihapus oleh *user.*



Gambar 12 DFD Level 2 Proses *Update*

Saat *user* melakukan *query update* terhadap data pada *base table,* data pada *base table* akan berubah sesuai dengan data yang diperbaharui oleh *user.*

4.3 Perubahan Pada *Source Code*

Dalam menerapkan algoritma *filtering* pada struktur PostgreSQL, dibutuhkan penguasaan terhadap arsitektur PostgreSQL itu sendiri. Pada bab 2.15.1 telah digambarkan arsitektur secara keseluruhan dari PostgreSQL.

Dari studi literatur yang telah dilakukan dan dengan dilakukan pengecekan langsung terhadap *source code* PostgreSQL, maka ditentukan penempatan fungsi *filtering* ini akan dilakukan sebelum proses eksekusi dan sesudah proses optimisasi pada *backend* PostgreSQL, tepatnya pada *file createas.c* dan *matview.c.* Dari kedua *file* ini akan ditelusuri *file* yang berhubungan dengan perubahan yang akan dilakukan.

Daftar Pustaka

[1] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 6th ed. New York: McGraw-Hill, 2011.

[2] S. Vijayprasath and S. P. Rajan, "Design of a Simple Graphical User Interface to the Relational Database Management System," p. 6, 2015.

[3] Y. Z. Ayık and F. Kahveci, "Materialized View Effect on Query Performance," vol. 11, no. 9, p. 4, 2017.

[4] V. M. Thakare and P. P. Karde, "An Efficient Materialized View Selection Approach for Query Processing in Database Management," presented at the International Journal of Computer Science and Network Security, 2010, vol. 10, pp. 1–3.

[5] E. F. Codd, *The relational model for database management: version 2*. Reading, Mass: Addison-Wesley, 1990.

[6] L. Frank, "Countermeasures Against Consistency Anomalies in Distributed Integrated Databases with Relaxed ACID Properties," in *2011 International Conference on Innovations in Information Technology*, Abu Dhabi, United Arab Emirates, 2011, pp. 266–270.

[7] Alka and A. Gosain, "A Comparative Study of Materialised View Selection in Data Warehouse Environment," in *2013 5th International Conference on Computational Intelligence and Communication Networks*, Mathura, India, 2013, pp. 455–459.

[8] A. P. Mohod and M. S. Chaudhari, "Improve Query Performance Using Effective Materialized View Selection and Maintenance: A Survey," no. 4, p. 6, 2013.

[9] A. Tapdiya, Y. Xue, and D. Fabbri, "A Comparative Analysis of Materialized Views Selection and Concurrency Control Mechanisms in NoSQL Databases," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Honolulu, HI, USA, 2017, pp. 384–388.

[10] S. Kurzadkar and A. Bajpayee, "Anatomization of Miscellaneous Approaches for Selection and Maintenance of Materialized View," in *2015 IEEE 9th International Conference on Intelligent Systems and Control (ISCO)*, Coimbatore, India, 2015, pp. 1–5.

[11] S. Muller, L. Butzmann, K. Howelmeyer, S. Klauck, and H. Plattner, "Efficient View Maintenance for Enterprise Applications in Columnar In-Memory Databases," in *2013 17th IEEE International Enterprise Distributed Object Computing Conference*, Vancouver, BC, Canada, 2013, pp. 249–258.

[12] S. R. Valluri, S. Vadapalli, and K. Karlapalem, "View Relevance Driven Materialized View Selection in Data Warehousing Environment," p. 10.

[13] P. P. Karde, D. V. M. Thakare, and S. P. Deshpande, "An Effective Cost Approach Technique Using Materialized View for Query Evaluation," vol. 4, no. 1, p. 6, 2011.

[14] B. Ashadevi and D. R. Balasubramanian, "Cost Effective Approach for Materialized Views Selection in Data Warehousing Environment," p. 7, 2008.

[15] D. Yao, A. Abulizi, and R. Hou, "An Improved Algorithm of Materialized View Selection within the Confinement of Space," in *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*, Dalian, China, 2015, pp. 310–313.

[16] R. N. Jogekar and A. Mohod, "Design and Implementation of Algorithms for Materialized View Selection and Maintenance in Data Warehousing Environment," vol. 3, no. 9, p. 7, 2013.

[17]R. M. Ismail, “Maintenance of materialized views over peer-to-peer data warehouse architecture,” in *The 2011 International Conference on Computer Engineering & Systems*, Cairo, Egypt, 2011, pp. 312–318.
[18]C. Xu, M. Zhou, and W. Qian, “Materialized View Maintenance in Columnar Storage for Massive Data Analysis,” in *2010 4th International Universal Communication Symposium*, Beijing, China, 2010, pp. 69–76.
[19]Lijuan Zhou, Qian Shi, and Haijun Geng, “The Minimum Incremental Maintenance of Materialized Views in Data Warehouse,” in *2010 2nd International Asia Conference on Informatics in Control, Automation and Robotics (CAR 2010)*, Wuhan, China, 2010, pp. 220–223.
[20]J. M. Monteiro, S. Lifschitz, and Â. Brayner, “Automated Selection of Materialized Views,” p. 14.
[21]N. T. Q. Vinh, “Synchronous Incremental Update of Materialized Views for PostgreSQL,” vol. 42, pp. 307–315, 2016.
[22]A. Hindle, Z. M. Jiang, W. Koleilat, M. W. Godfrey, and R. C. Holt, “YARN: Animating Software Evolution,” in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Banff, AB, Canada, 2007, pp. 129–136.
[23]T. Lane, “A Tour of PostgreSQL Internals,” p. 25, 2000.
[24]“Refresh All Materialized Views - PostgreSQL wiki.” [Online]. Available: https://wiki.postgresql.org/wiki/Refresh_All_Materialized_Views. [Accessed: 12-Jan-2019].
[25]“PostgreSQL: Documentation: 10: 40.3. Materialized Views.” [Online]. Available: https://www.postgresql.org/docs/10/rules-materializedviews.html. [Accessed: 19-Nov-2018].
[26]“PostgreSQL/Materialized Views - Jonathan Gardner’s Tech Wiki.” [Online]. Available: https://tech.jonathangardner.net/wiki/PostgreSQL/Materialized_Views. [Accessed: 17-Jan-2019].
[27]S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener, “Incremental Maintenance for Materialized Views over Semistructured Data_,” p. 12.