PENDETEKSIAN *MUTATING TABLE* PADA *ORACLE* DENGAN METODE PENELUSURAN *DEPENDENCY TRIGGER* PADA TABEL SECARA VISUAL

TUGAS AKHIR

Disusun Oleh :

DIMAS KAHARUDIN INDRA RUPAWAN
5102 100 029

JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA
2006

PENDETEKSIAN *MUTATING TABLE* PADA *ORACLE* DENGAN METODE PENELUSURAN *DEPENDENCY* *TRIGGER* PADA TABEL SECARA VISUAL

TUGAS AKHIR

Diajukan Guna Memenuhi Sebagian Persyaratan
Untuk Memperoleh Gelar Sarjana Komputer
Pada
Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya

Mengetahui / Menyetujui

Dosen Pembimbing I	Dosen Pembimbing II
Irfan Subakti, S.Kom., M.Sc. Eng	Darlis H, S.Kom
NIP. 132 300 412	NIP. 132 306 430

SURABAYA
JANUARI, 2006

ABSTRAK

Visualisasi adalah suatu metode yang populer untuk memodelkan suatu permasalahan. Dengan adanya proses visualisasi, suatu masalah akan lebih mudah untuk dipahami, dianalisis dan dipecahkan. Proses visualisasi untuk memodelkan dan memecahkan suatu masalah ini sudah dilakukan pada banyak hal. Antara lain bisa dilihat dari banyaknya tipe diagram yang ada saat ini, yaitu: Diagram Flowchart, Diagram Use Case, Diagram kelas, Entity Relationship Diagram (ERD)/Diagram ER, Diagram Physical Data Model/PDM, dan lain sebagainya. Dalam suatu database aktif, khusunya Oracle, kasus mutating table adalah kasus yang biasa terjadi. Mutating table adalah kasus dimana ada sebuah statement Data Manipulation Language/DML mencoba mengakses suatu tabel, sedangkan tabel tersebut masih dalam keadaan diubah oleh DML lain. Mutating table sebenarnya bukanlah sebuah kasus yang membahayakan, akan tetapi kasus tersebut terjadi karena adanya perancangan trigger yang kurang sesuai dengan DML yang sering digunakan. Saat ini, kasus mutating table sulit dilacak dengan jalan penelusuran secara manual, yaitu dengan jalan membaca kode-kode dari trigger. Sulitnya penelusuran secara manual tersebut menyebabkan kasus mutating table sering hanya bisa diketahui pada saat runtime saja.

Dalam Tugas Akhir ini, akan dibuat sebuah algoritma dan aplikasi yang bisa digunakan untuk mendeteksi/meramalkan kasus mutating table secara visual, yaitu dengan memvisualisasikan dependency trigger pada tabel untuk kemudian ditelusuri. Visualisasi dilakukan dengan jalan melakukan proses parsing terhadap trigger body, kemudian menampilkannya dalam bentuk diagram. Dependency trigger yang ditampilkan dalam diagram adalah berupa event dan action terhadap tabel.

Dari hasil uji coba terhadap berbagai struktur database, baik yang bersifat fiktif maupun yang bersifat nyata, didapatkan suatu kesimpulan bahwa kasus mutating table lebih mudah untuk diketahui dan dianalisis dengan visualisasi, daripada harus diketahui dengan jalan membaca langsung source code trigger ataupun melalui proses runtime.

Kata Kunci: *teori graph, trigger, mutating table, Oracle, Physical Data Model (PDM)*

KATA PENGANTAR

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

ٱلْحَمْدُ لِلَّهِ رَبِّ ٱلْعَٰلَمِينَ. Segala puji terucap kehadirat لله *Subhanahu wa ta'ala* yang telah melimpahkan karunia-Nya berupa rizki kepada siapa saja yang dikehendaki-Nya dari seluruh makhluk-Nya di alam semesta ini. Sholawat serta salam senantiasa terucapkan kepada seorang manusia terbaik sepanjang zaman, *rasul* terakhir yang diturunkan di bumi ini, yang menjadi panutan bagi orang-orang yang mendapatkan petunjuk, pembawa syariat terakhir bagi manusia, pemberi kabar gembira tentang kehidupan yang kekal di akherat, penerima wahyu dalam bentuk *Al Quran,* yaitu رَسُولُ ٱللَّهِ مُحَمَّدٌ ﷺ

Melalui kata pengantar ini, penyusun ingin mengucapkan banyak ucapan *Jazakumullahu khairan katsiro* (semoga لله membalas anda semua dengan kebaikan yang banyak) kepada pelbagai pihak yang telah membantu penyusun menyelesaikan Tugas Akhir ini, yaitu:

- Bapak Supardi'i serta Ibu Siti Aeniyah selaku ayah dan ibu penyusun. Hj Darosah selaku nenek penyusun. Kepada beliau semuanya, penyusun ucapkan *Jazakumullahu khairan katsiro* karena telah mendidik, mengasuh, membesarkan, menasehati, memotivasi, menegur dan mendoakan penyusun dengan penuh rasa ikhlas, kesabaran, tanpa pamrih, serta penuh rasa kasih sayang.

- Mas Dita, Dik Fahma, Dik Aha selaku saudara penyusun yang telah melalui perjalanan hidup bersama-sama penyusun di kota Kediri tercinta.
- Bapak Irfan Subakti, S.Kom., M.Sc.Eng. dan Bapak Darlis H, S.Kom., selaku dosen-dosen pembimbing yang telah dengan sabar mengarahkan penyusun sehingga bisa menyelesaikan Tugas Akhir ini tepat pada waktunya.
- Bapak Prof. Dr. Ir. Arif Djunaidy, M.Sc., Bapak Arif Bramantoro, ST, M.Sc., dan Ibu Diana Purwitasari, S.Kom., selaku dosen-dosen penguji yang telah memberikan masukan-masukan berharga dalam Tugas Akhir ini.
- Bapak Yudhi Purwananto, S.Kom., M.Kom., selaku Ketua Jurusan yang telah berkenan mengizinkan penyusun mengambil Tugas Akhir lebih cepat setengah semester dari jadwal semestinya.
- Seluruh dosen-dosen di FTIF yang telah membagi ilmunya kepada penyusun.
- Rekan-rekan satu kampus, teman-teman *ForumHidden* (Ain, B@WaH, pak Eko, Ferdic, Hadziq, I$$@, Purna, Mbah Rendi, Rile, Yojana, XuXiLO, dll), juga kepada admin serta penghuni lab IBS yang telah memberikan nuansa menyenangkan dalam pembuatan Tugas Akhir ini.
- Dan kepada pelbagai pihak yang tidak mungkin penyusun sebutkan satu persatu. Penyusun ucapkan *Jazakumullahu khairan katsiro* atas segala bantuannya.

Surabaya, 13 Januari 2006

Penyusun

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I

PENDAHULUAN

BAB 1

PENDAHULUAN

1.1 Latar Belakang

Database, sebagai salah satu hasil dari pengembangan Teknologi Informasi, merupakan sarana penting bagi pengguna Teknologi Informasi untuk mengolah data. Saat ini banyak *vendor* database yang muncul dan memberikan pelbagai fitur serta keunggulan, salah satunya adalah fitur *trigger* pada database aktif.

Database aktif adalah database yang bisa bereaksi secara aktif terhadap *event-event* yang terjadi. Hal ini berbeda dengan konsep database tradisional yang bersifat pasif. Artinya, database pasif membutuhkan keterlibatan aktif pihak administrator database atau pihak aplikasi dalam mengelola *record-record*-nya berdasarkan *event-event* yang terjadi. Untuk database aktif, hal di atas bisa diatasi dengan menggunakan *trigger*. *Trigger* adalah kumpulan perintah atau program yang dieksekusi secara otomatis pada saat terjadi *event-event* dalam database, khususnya pada tabel. Misalnya *event insert*, *update*, atau *delete*.

Database aktif mempunyai peranan yang sangat penting pada komputerisasi pelbagai bidang pada saat ini. Kebutuhan akan database aktif sudah tidak bisa diragukan lagi keberadaannya. Contoh yang paling nyata dari hal ini adalah pesatnya penggunaan *Oracle* dan *MS SQL Server* dalam dunia internet dan bisnis.

Dalam *Oracle*, terdapat dua jenis *trigger*, yaitu *trigger row level* dan *trigger statement level*. Khusus untuk *trigger row level*, terdapat suatu kemungkinan terjadinya kasus *mutating table*. Kasus *mutating table* ini terjadi jika suatu *trigger*

dengan tipe *row level* tereksekusi, sedangkan *trigger* itu sendiri melakukan proses *DML* terhadap tabel dimana *trigger* itu berasal. Apabila proses ini dibiarkan, maka akan dihasilkan suatu ketidak-konsistenan data. Untuk menghindari hal ini, maka *Oracle* menganggap kasus *mutating table* tersebut menjadi sebuah *error*.

Pada kasus *trigger* yang sederhana, pihak *developer PL-SQL* ataupun administrator database bisa dengan mudah memantau *trigger-trigger* apa saja yang telah dibuat, serta bagaimana *dependency*-nya terhadap tabel dan *trigger-trigger* lainnya. Akan tetapi, pada kasus dimana jumlah *trigger* semakin banyak dan semakin kompleks, proses pemantauan akan menjadi semakin sulit. Terutama bila terjadi kasus *mutating table* yang melibatkan lebih dari dua tabel. Pencarian dimana letak pasti penyebab terjadinya *mutating table* akan sulit dilakukan.

Kasus *mutating table* sebenarnya bisa saja dibiarkan oleh administrator database, sebab kasus *mutating table* bukanlah suatu kasus yang membahayakan. Akan tetapi, apabila dalam suatu struktur database sering muncul kasus *mutating table*, maka administrator database harus merombak ulang sebagian/seluruh struktur database yang dikelolanya. Hal ini perlu dilakukan, sebab seringnya kemunculan kasus *mutating table,* yang menyertai *DML* tertentu yang sering digunakan, adalah akibat dari rancangan yang kurang tepat dari struktur suatu database aktif.

1.2 Tujuan

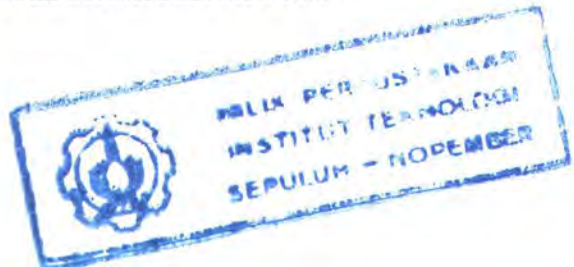
Tujuan dari Tugas Akhir ini adalah membuat algoritma dan aplikasi untuk meramalkan terjadinya *mutating table*, menentukan letak beserta rute penyebabnya dengan jalan memvisualisasikan *Dependency Trigger* pada tabel.

Aplikasi ini nantinya bisa digunakan oleh pihak administrator database untuk menganalisis *trigger-trigger* yang dikelolanya.

1.3 Permasalahan

Permasalahan yang diangkat dalam pembuatan Tugas Akhir ini adalah:

- Bagaimana mengetahui jenis-jenis *trigger*. Apakah itu *trigger statement level, row level* ataupun *event-event* pada saat apa *trigger* tersebut dieksekusi (misal: *insert, update, delete*).
- Bagaimana melakukan proses *parsing* terhadap *trigger body* sehingga bisa diketahui tabel yang diacu dari *trigger* tersebut.
- Bagaimana merancang bentuk visualisasi dari tabel, *integrity constraint* yang berupa relasi, *action*, *event* dan *trigger* menjadi satu diagram terpadu yang sederhana namun informatif. Dengan catatan, diagram baru tadi haruslah kompatibel dengan diagram *Physical Data Model* yang sudah ada.
- Bagaimana membuat struktur data yang merepresentasikan referensi *trigger*-tabel-relasi. Struktur data tersebut haruslah sesederhana mungkin dan mudah diimplementasikan.
- Bagaimana membuat suatu algoritma untuk mencari rute-rute yang mungkin menyebabkan kasus *mutating table*.
- Bagaimana membuat suatu algoritma untuk memverifikasi rute-rute yang telah ditemukan tadi sehingga bisa dipakai untuk meramalkan kasus *mutating table*.
- Bagaimana menampilkan rute-rute *event* dan *action* penyebab kasus *mutating table* tersebut sehingga bisa dipahami dan bermanfaat oleh *user*.



1.4 Batasan Masalah

Batasan masalah dalam Tugas Akhir ini adalah:

- Tugas Akhir ini hanya meramalkan dan memvisualisasikan *trigger* kemudian menunjukkan di mana letak kemungkinan terjadi *mutating table* beserta rute penyebabnya. Tugas Akhir ini bukan suatu *case tools* pembangkit *script trigger* ataupun *query analyzer*.
- Tugas Akhir ini tidak menunjukkan cara pemecahan dari kasus *mutating table* yang terjadi. Pemecahan dari kasus *mutating table* sangatlah melebar dan sangat tergatung dari logika pihak *developer PL-SQL*/administrator database. Dalam hal ini, sulit diketahui jalur logika pihak pengembang dalam membangun *trigger*-nya.
- Proses *parsing* hanya dilakukan pada *trigger body*. Sehingga agar *action trigger* bisa dikenali oleh aplikasi, operasi *DML* haruslah langsung diletakkan pada *trigger body*, bukan pada *stored procedure*.
- Visualisasi *action* pada *trigger* hanya untuk operasi *DML*.
- Visualisasi lebih ditekankan pada *trigger* dengan *event* dan *action* serta *triggering type*. Untuk tabel, visualisasi hanya ditampilkan seperlunya saja (hanya ditampilkan *table name* saja). Hal ini dimaksudkan untuk menambah *visibility* diagram untuk kasus yang kompleks.
- Hasil *output* peramalan *mutating table* hanya bersifat *warning*. Artinya, *output* peramalan *valid* apabila semua *event* dan *action* yang terdapat dalam rute terpenuhi. Tugas Akhir ini tidak menganalisis apakah *query* yang akan

dieksekusi oleh *user* memenuhi semua *event* dan *action* yang disebutkan dalam *output* aplikasi.

- Tugas Akhir ini meramalkan semua *mutating table* hanya berdasarkan *chain reaction action* per-tabel, bukan *action* per-kolom. Sehingga setiap pengaksesan tabel dianggap mengakses *key column*, meskipun pihak pengembang *trigger* sudah mengantisipasi hal di atas dalam *trigger*-nya.
- *Trigger* dibatasi hanya yang bersifat *row level* dan *statement level*. *Trigger* dengan tipe *instead of* tidak dipakai.

1.5 Metodologi Pelaksanaan Tugas Akhir

Metodologi yang digunakan dalam pembuatan Tugas Akhir ini adalah sebagai berikut:

1) **Studi literatur**: Yaitu tahap pencarian bahan-bahan/literatur yang dibutuhkan pada Tugas Akhir ini. Literatur tersebut meliputi teori tentang *trigger* dan *mutating table*. Ditambah sedikit literatur mengenai *java* dan *Object Oriented Programming/OOP* (khususnya *swing*) guna proses pemograman dan visualisasi. Serta teori *graph* dan *tree* guna proses perancangan struktur data.
2) **Perancangan Aplikasi:** Yaitu tahap analisis terhadap permasalahan yang dihadapi untuk mendapatkan kebutuhan aplikasi yang akan dibuat kemudian dimodelkan. Dalam tahap ini juga dibahas tentang perancangan proses-proses yang terjadi pada aplikasi serta algoritma-algoritma apa yang dipakai.
3) **Pembuatan Aplikasi**: Adalah tahap dimana aplikasi utama dikerjakan, yaitu dengan menggunakan bahasa pemograman *Java Standard Edition*. *Integrated Development Environtment/IDE* yang dipakai adalah *NetBeans 4.1*.

4) **Uji Coba dan Evaluasi**: Pada tahap ini dilakukan uji coba aplikasi untuk mengevaluasi aplikasi Tugas Akhir serta menguji fungsionalitasnya.
5) **Pembuatan Dokumentasi Tugas Akhir**: Penyusunan dokumentasi direncanakan akan dikerjakan kurang lebih secara bersamaan dengan proses-proses di atas. Sebab buku Tugas Akhir juga berfungsi sebagai sebuah dokumentasi yang mencatat setiap tahapan-tahapan penting dalam proses-proses sebelumnya. Buku Tugas Akhir juga berguna bagi pihak-pihak lain yang ingin mengetahui penjelasan teknis bagaimana aplikasi ini dibuat.

1.6 Sistematika Penulisan

Dalam penyusunannya, laporan Tugas Akhir ini akan dikelompokkan menjadi enam bab, yaitu bab 1 berisi latar belakang, tujuan, permasalahan, dan batasan masalah. Bab 2 berisi teori dasar dan teori penunjang yang digunakan dalam pembuatan Tugas Akhir ini. Bab 3 berisi metodologi pembuatan aplikasi, analisis untuk mengembangkan aplikasi, algoritma yang dipakai, serta perancangan aplikasi. Bab 4 memuat tentang implementasi dari bab 3. Bab 5 menguraikan tentang lingkungan uji coba, struktur database untuk pelaksanaan uji coba, skenario dan pelaksanaan uji coba pada aplikasi yang telah dibuat. Sedangkan bab terakhir, yaitu bab 6, berisi tentang kesimpulan dan saran yang dapat digunakan untuk proses pengembangan lebih lanjut dari aplikasi yang telah dibuat.

BAB II

TEORI PENUNJANG

BAB 2

TEORI PENUNJANG

Pada bab ini akan dibahas beberapa teori dasar yang menunjang dalam pembuatan Tugas Akhir. Diantaranya adalah *trigger Oracle* [RAC05], database aktif [RAC05], dan teori *graph.*

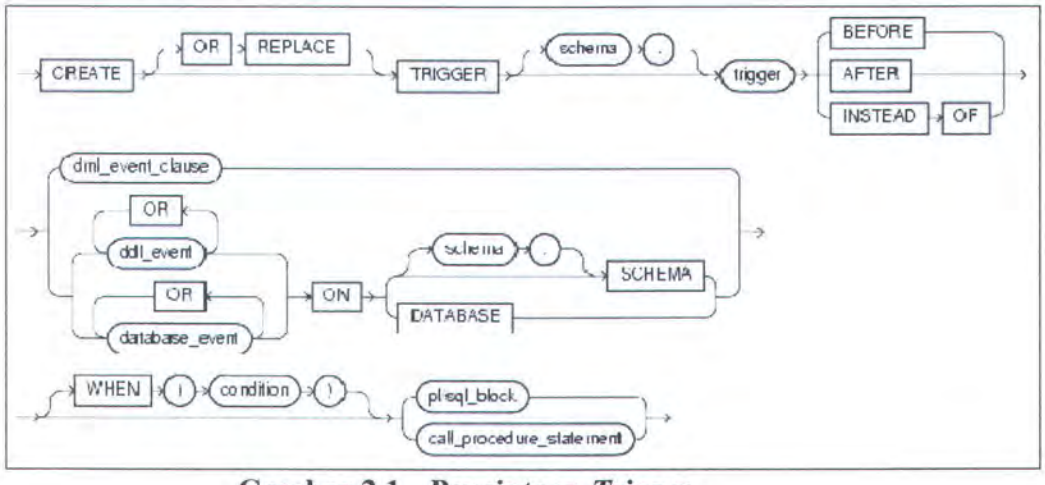
2.1 *Trigger* pada *Oracle*

Trigger adalah sebuah *stored procedure* yang dieksekusi pada saat terjadi *event* yang memodifikasi data pada tabel.

2.1.1 Penciptaan *Trigger*

Sintaks umum dalam penciptaan sebuah *trigger* adalah sebagai berikut:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
    {BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON
<table_name>
    [REFERENCING [NEW AS <new_row_name>] [OLD AS
                <old_row_name>]]
    [FOR EACH ROW [WHEN (<trigger_condition>)]]
    <trigger_body>
```



Gambar 2.1 Penciptaan *Trigger*

- *Trigger_name* adalah nama dari *trigger*.
- *Triggerring_event* adalah *event* yang menyebabkan *trigger* dieksekusi.
- *Trigger_body* adalah kode dari *trigger*.
- *Referencing_clause* adalah nama yang dipakai untuk menunjuk data pada baris yang sedang dimodifikasi.
- *Trigger_condition* adalah kondisi yang bersifat opsional dan akan dievaluasi sebelum penyalaan *trigger*. Bila bernilai *true*, maka *trigger* akan dieksekusi.

Berikut ini adalah beberapa kemungkinan kombinasi tipe *trigger* berdasarkan *event statement, timing, dan level*:

Tabel 2.1 Kombinasi Tipe *Trigger*

Kategori	Nilai	Keterangan
Statement	*insert*, *delete*, atau *update*	Menentukan tipe *statement DML* yang menyebabkan *trigger* dinyalakan.
Timing	*before* atau *after*	Menentukan apakah *trigger* menyala sebelum atau sesudah *statement* dieksekusi.
Level	*row* atau *statement*	Jika *trigger* bertipe *row level*, maka *trigger* akan dinyalakan sekali untuk setiap baris yang dipengaruhi oleh *trigger*. Jika *trigger* bertipe *statement level*, maka *trigger* akan dinyalakan sekali sebelum atau sesudah *statement*. Sebuah *trigger row level* diidentifikasikan oleh klausa *FOR EACH ROW* dalam definisi *trigger*.

Sebuah tabel bisa mempunyai lebih dari satu *trigger* untuk setiap satu tipe *DML*. Berikut ini adalah urutan-urutan pengekseskuisan *trigger* oleh *Oracle*:

1. Mengeksekusi *trigger before statement level*, jika ada.
2. Untuk masing-masing baris yang dipengaruhi oleh pernyataan:

A. Mengeksekusi *trigger before row level*, jika ada.

B. Mengeksekusi pernyataan *DML*.

C. Mengeksekusi *trigger after row level*, jika ada.

3. Mengeksekusi *trigger after statement level*, jika ada.

2.1.2 Korelasi Pengenal dalam *Trigger Row Level*

Trigger row level dijalankan sekali per-baris yang diproses oleh *statement trigger*. Di dalam *trigger* jenis ini, data pada baris yang sedang diproses dapat diakses, yaitu dengan memakai korelasi pengenal ***:old*** dan ***:new***. Korelasi pengenal adalah tipe spesial dari variabel pengikat *PL/SQL*. Titik dua (:) di depan dua korelasi pengenal tersebut menunjukkan bahwa keduanya adalah variabel pengikat yang akan diperlakukan oleh *compiler PL/SQL* sebagai tipe *record*.

```
triggering_table%ROWTYPE
```

`triggering_table` adalah tabel dimana *trigger* didefinisikan.

Korelasi pengenal *:old* dan *:new* dijabarkan sebagai berikut:

Tabel 2.2 Korelasi Pengenal *Trigger Row Level*

Statement *triggering*	**:old**	**:new**
insert	*null*	Nilai baru yang akan di-*insert*-kan
update	Nilai awal	Nilai baru yang akan di-*update*-kan
delete	Nilai awal	*null*

2.1.3 Klausa Kondisi

Klausa kondisi ini berlaku hanya untuk *trigger row level*. kode *trigger* yang terdapat dalam *trigger body* akan dieksekusi hanya pada baris yang sesuai dengan kondisi yang didefinisikan oleh klausa kondisi.

Pendeklarasian *WHEN clause* adalah sebagai berikut:

```
WHEN trigger_condition
```

`trigger_condition` adalah ekspresi *boolean*. Kondisi ini akan dievaluasi untuk tiap barisnya. Sebagai contoh, *body* dari *trigger* `CekUmur` berikut ini hanya akan dieksekusi jika `umur_record` yang diambil lebih dari 20.

```
CREATE OR REPLACE TRIGGER CekUmur
BEFORE INSERT OR UPDATE OF umur_record On penduduk
FOR EACH ROW
WHEN (new.umur_record > 20)
BEGIN
  /*    Body Trigger      */
END;
```

2.2 Database Aktif

Database aktif adalah konsekuensi dari adanya *trigger*. Database aktif ini terdiri dari tiga aturan, yaitu ***Event – Condition – Action*** atau ***ECA Rules***. Aturan dalam model *ECA* memiliki tiga komponen sebagai berikut:

1) *Event* yang memicu suatu *trigger*.
2) *Condition* yang akan menentukan apakah suatu *trigger* dapat dijalankan atau tidak.
3) *Action* menentukan aksi yang dilakukan oleh *trigger*. Aksi dapat berupa *statement SQL* maupun operasi-operasi *Data Manipulation Languange* (*DML*).

2.2.1 *Mutating Table*

Mutating table adalah tabel yang sedang dimodifikasi oleh *statement DML*. Untuk *trigger*, maka tabel tersebut adalah tabel dimana *trigger* tersebut didefinisikan.

Tabel yang terkait suatu relasi *cascade delete* juga *mutating*. Sedangkan *constraining tables* adalah tabel yang mungkin perlu dibaca karena adanya *referential integity constraints*.

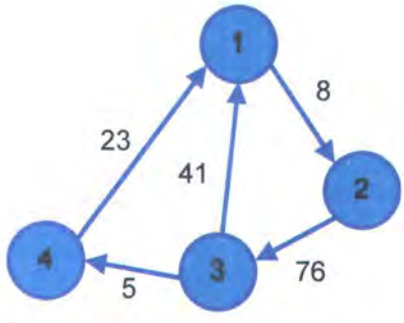
2.3 Teori *Graph*

Graph adalah representasi simbolik dari sebuah *network/route* beserta keterkaitan antara satu sama lain. Teori *graph* adalah salah satu cabang dari ilmu matematika yang memodelkan sebuah *network* dan memecahkan permasalahan-permasalahan di dalamnya [ROD05]:.

Berikut ini adalah istilah-istilah yang ada dalam teori *graph* [ROD05]:

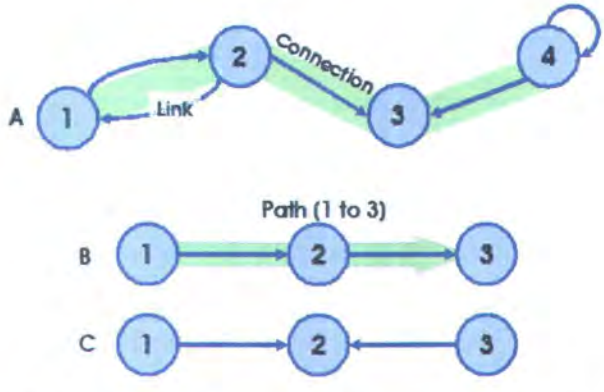
- ***Graph*:** *graph G* adalah himpunan *vertex* (*nodes*) *v* yang terhubung oleh *edge e*. Dilambangkan dengan $G=(v,e)$.
- ***Vertex* (*node*):** adalah *terminal point* atau titik perpotongan dalam sebuah struktur data *graph*. Dalam dunia nyata, *vertex* merupakan pelambangan dari sebuah kota.
- ***Edge* (*link*):** adalah penghubung diantara dua buah *vertex*/*node*. *Edge* (i,j) adalah *edge* yang berawal dari i dan berakhir di j. Dalam dunia nyata, *edge* merupakan pelambangan dari sebuah jalan.
- ***Subgraph*:** adalah *graph* yang terbentuk dari himpunan bagian dari suatu *graph* lainnya. Dilambangkan dengan $G'=(v',c')$.
- ***MultiGraph***: adalah *graph* yang di dalamnya terdapat dua *vertex* yang memiliki lebih dari dua *edge* diantara keduanya.
- ***Directed Graph*:** yaitu *graph* yang setiap *edge* di dalamnya mempunyai arah [CAL05].

- ***Weighted Graph***: yaitu *graph* yang memberikan atribut *weight*/berat pada setiap *edge*-nya.



Gambar 2.2 ***Directed Weighted Graph***

- ***Connection*:** adalah kumpulan dari dua *vertex* yang terhubung antara satu dengan lain minimal oleh sebuah *edge*.
- ***Path*:** adalah urutan dari *edge* yang mengizinkan suatu penelusuran tanpa terputus pada arah yang sama.

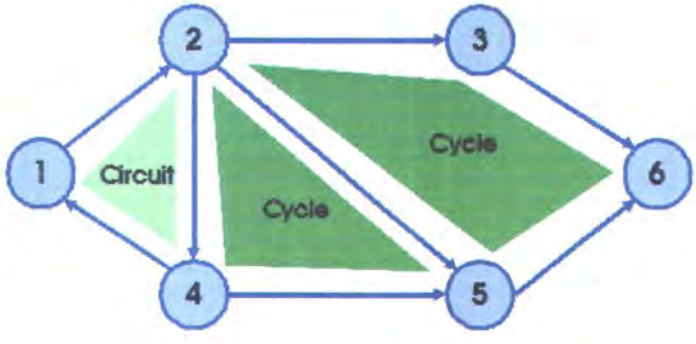


Gambar 2.3 ***Path* dan *Connection* pada *Graph***

- ***Chain*:** adalah urutan dari *edge* yang mempunyai *connection* antara satu dengan yang lain tanpa memperhatikan arah.
- ***Chain Reaction:*** adalah sebuah proses berantai dimana *output* dari sebuah subproses menjadi *input* dari subproses selanjutnya. Bila *ouput* yang

dihasilkan subproses sebelumnya tidak sesuai dengan ketentuan-ketentuan tertentu, maka *output* tersebut tidak akan bisa dipakai sebagai *input* pada subproses berikutnya. Contoh *chain reaction* pada dunia nyata adalah proses reaksi nuklir [CHR06]. Dalam *graph, chain reaction* adalah *path* dengan syarat-syarat tertentu yang harus terpenuhi untuk bisa ditelusuri.

- ***Cycle*:** adalah *chain* yang *vertex* asal dan *vertex* tujuan sama tanpa melewati sebuah *edge* lebih dari satu kali.
- ***Circuit*:** adalah *path* yang *vertex* asal dan *vertex* tujuan sama tanpa melewati sebuah *edge* lebih dari satu kali.



Gambar 2.4 *Cycle* dan *Circuit*

- ***Depth First Search (DFS)*:** adalah sebuah algoritma untuk menelusuri *vertex/edge* pada suatu struktur data *graph/tree*. Penelusuran secara *DFS* ini dimulai dari suatu *vertex*, yang bertindak sebagai titik awal, kemudian dilakukan eksplorasi terhadap satu cabang terlebih dahulu hingga berhenti pada kondisi *depth*/kedalaman tertentu baru kemudian dilanjutkan pada cabang berikutnya [TCR01].

BAB III

ANALISIS DAN PERANCANGAN APLIKASI

BAB 3

ANALISIS DAN PERANCANGAN APLIKASI

Pada bab ini dijelaskan tentang metodologi yang digunakan dalam membangun aplikasi beserta algoritma yang dipakai. Perancangan aplikasi yang akan dijelaskan meliputi perancangan proses dan perancangan kelas.

3.1 Metodologi Pengembangan Aplikasi

Aplikasi yang dibangun dalam Tugas Akhir ini menggunakan metode sekuensial linier (*linear sequential*) atau juga disebut *waterfall model.*

3.1.1 Tahap Analisis

Pada tahap ini akan dilakukan kegiatan sebagai berikut:

- Menentukan *rule-rule* yang dipakai dalam proses *parsing trigger body*.
- Menentukan bentuk diagram yang dipakai untuk memodelkan tabel, relasi, *trigger*, *event*, serta *action*-nya.
- Menentukan struktur data *graph* yang sesuai dengan diagram tadi.
- Menentukan penyebab-penyebab kasus *mutating table* pada *Oracle.*
- Membuat algoritma untuk meramalnya.

3.1.2 Tahap Desain

Proses desain menterjemahkan kebutuhan *user* kedalam representasi aplikasi. Beberapa bentuk desain digunakan untuk menggambarkan aktiftas yang dilakukan oleh *user* serta proses yang dijalankan oleh sistem, antara lain:

- Diagram *use case*

  Diagram *use case* dipakai untuk memodelkan *actor* dan aplikasi serta interaksi apa saja yang terjadi diantara keduanya.

- Diagram Sekuen

  Diagram sekuen dipakai untuk memodelkan interaksi antar *object* dalam aplikasi.

- Diagram Kelas

  Diagram kelas memodelkan struktur kelas-kelas dalam aplikasi serta keterkaitannya antara satu dengan yang lain.

- Diagram *Flowchart*

  Diagram *flowchart* menggambarkan aliran proses/algoritma mulai dari awal suatu hingga ke akhir. Akhir suatu proses tersebut bisa saja menjadi awal dari suatu proses lainnya.

3.1.3 Tahap Implementasi

Adalah tahap untuk mengubah hasil desain menjadi *source code* yang bisa dimengerti oleh komputer.

3.1.4 Tahap Uji Coba

Uji coba difokuskan pada pembuktian hasil peramalan *mutating table* dalam pelbagai kasus. Baik pada struktur database yang bersifat fiktif maupun yang bersifat nyata.

3.2 Analisis Kebutuhan Aplikasi

Sebelum proses visualisasi dilakukan, terlebih dahulu harus diketahui tabel/*view* apa saja yang di-*reference* oleh *trigger*. Untuk mengetahui hal tersebut, maka diperlukan proses *parsing* pada *trigger body*.

3.2.1 Analisis *Rule* untuk Proses *Parsing* Terhadap *Trigger Body*

Trigger body terdapat dalam *dictionary table user_triggers* kolom ketigabelas, yaitu kolom *trigger_body*.

Contoh *query* untuk mendapatkan *trigger_body*:

```
select trigger_body from user_triggers
```

dari *query* di atas, akan didapatkan *trigger_body* milik *user* yang sedang aktif. Berikut ini adalah contoh sebuah *trigger_body*:

```
begin
     insert into bel6 values('testT6');
end;
```

3.2.1.1 *DML Delete*

Untuk nama tabel yang dikenai *action DML delete* bisa diketahui dengan aturan *parsing* sebagai berikut:

- Membaca *string* sesudah kata "**delete from< white_space >**", maka akan diketahui tabel apa yang dikenai *action delete* oleh *trigger*.

  contoh:

  ```
  delete from <nama_table>;
  ```

3.2.1.2 *DML Select*

Untuk nama tabel yang dikenai *action DML Select* bisa diketahui dengan aturan *parsing* sebagai berikut:

- Membaca *string* sesudah kata "`from<white_space>`", maka akan diketahui tabel apa yang dikenai *action select* oleh *trigger*.
- Sebelum kata "`<white_space>from`" bukanlah kata "delete".

3.2.1.3 *DML Update*

Untuk nama tabel yang dikenai *action DML update* bisa diketahui dengan aturan *parsing* sebagai berikut:

- Membaca *string* sesudah kata "`update<white_space>`", maka akan diketahui tabel apa yang dikenai *action update* oleh *trigger*.

  contoh:

  ```
  delete from <nama_table>;
  ```

3.2.1.4 *DML Insert*

Sedangkan untuk nama tabel yang dikenai *action DML insert* bisa diketahui dengan aturan *parsing* sebagai berikut:

- Membaca baca sesudah kata "`insert into<white_space>`", maka akan diketahui tabel apa yang dikenai *action insert* oleh *trigger*

  contoh:

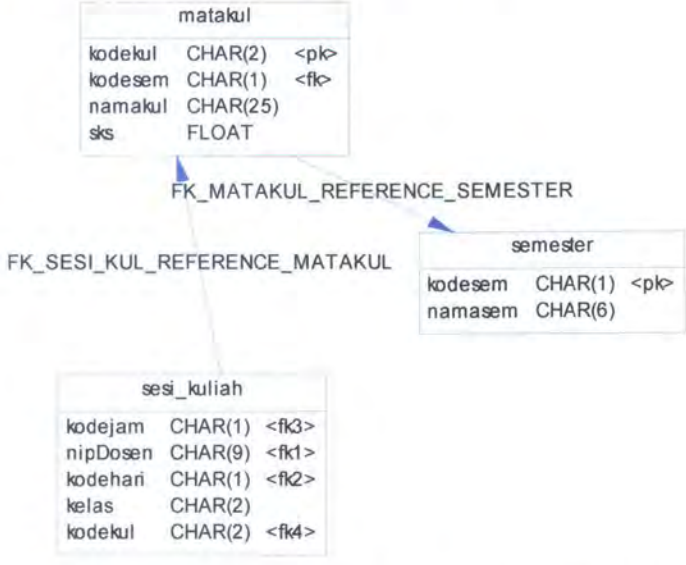
  ```
  insert into <nama_table>;
  ```

3.2.2 Rancangan Bentuk Visualisasi *Trigger*-Tabel (Diagram *Dependency Trigger*).

Pada bagian ini akan dirancang bentuk diagram yang akan dipakai untuk memvisualisasikan keterkaitan antara tabel dan *trigger* berdasarkan relasi serta *event* dan *action*.

Adapun diagram yang akan dirancang haruslah memenuhi ketentuan-ketentuan sebagai berikut:

- Diagram harus memiliki kompatibilitas dengan diagram *physical data model* (*PDM*) yang sudah ada saat ini. Simbol-simbol yang terdapat dalam diagram *PDM* harus masih bisa dipakai dalam diagram ini.



Gambar 3.1 Contoh Diagram *PDM* dari Aplikasi *Power Desaigner 9*

- Diagram harus mudah dibaca secara intuisi, artinya tidak diperlukan penjelasan dan pembelajaran mendetil untuk bisa membaca maksud dari diagram.
- Diagram harus sederhana dan mudah diimplementasikan dalam aplikasi. Artinya, diagram harus sebisa mungkin dihindari adanya hal-hal yang menyulitkan pengimplementasian dalam aplikasi. Selain itu, diagram baru tersebut harus memiliki simbol/pelambang seminimal mungkin.
- Diagram harus mempunyai daya visibilitas yang tinggi. Artinya, diagram dengan kompleksitas yang tinggi harus bisa dimuat dalam *space* yang kecil.

- Visualisasi *trigger* hanya diprioritaskan pada *event* dan *action*-nya saja tanpa mendetilkan proses *event* dan *action* tersebut. Detil tersebut misalnya adalah kolom apa yang diakses oleh *action trigger*. Hal ini diputuskan demikian dengan asumsi bahwa detil *action* bisa dibaca langsung oleh *user* pada *trigger body*. Jika detil *action* ditampilkan dalam diagram, maka akan memperumit diagram sehingga kesederhanaan diagram akan hilang.
- Diagram ini selanjutnya akan diberi nama Diagram *Dependency Trigger*

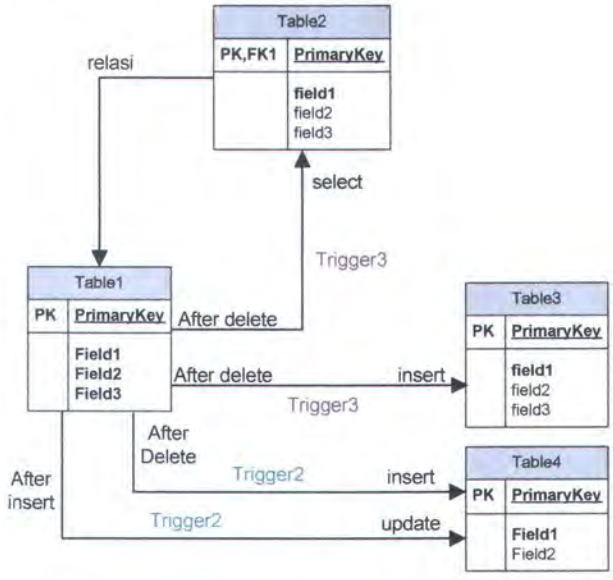
Pada subbab ini akan dijelaskan pula pelbagai bentuk alternatif diagram yang akan dipakai, berserta analisis kelebihan dan kekurangannya. Guna memperjelas proses pembandingan, maka data tabel-*trigger* yang akan didiagramkan tersebut mempunyai struktur yang sama antara satu alternatif diagram dengan alternatif yang lain.

Berikut ini adalah struktur *trigger* dan tabel yang akan dibuat diagramnya:

- Struktur terdiri atas empat buah tabel dan dua buah *trigger*. Tabel yang terdapat dalam struktur antara lain adalah: *Table1, Table2, Table3,* dan *Table4*.
- Sedangkan *trigger* yang terdapat dalam struktur adalah *Trigger2* dan *Trigger3*.
- *Table1* mempunyai hubungan relasional dengan *Table2*.
- *Trigger3* dan *Trigger2* adalah milik *Table1*.
- *Trigger3* adalah *trigger* row level, *Trigger2* adalah *statement level*.
- *Trigger*3 dieksekusi pada *event after delete*.
- *Trigger3* melakukan *action insert* pada *Table3,* serta *select* pada *Table2*.
- *Trigger2* dieksekusi pada *event after delete or insert*.

- *Trigger2* melakukan *action insert* dan *update* pada *Table4.*

3.2.2.1 Bentuk Alternatif Pertama



Gambar 3.2 Alternatif Pertama Diagram *Dependency Trigger*

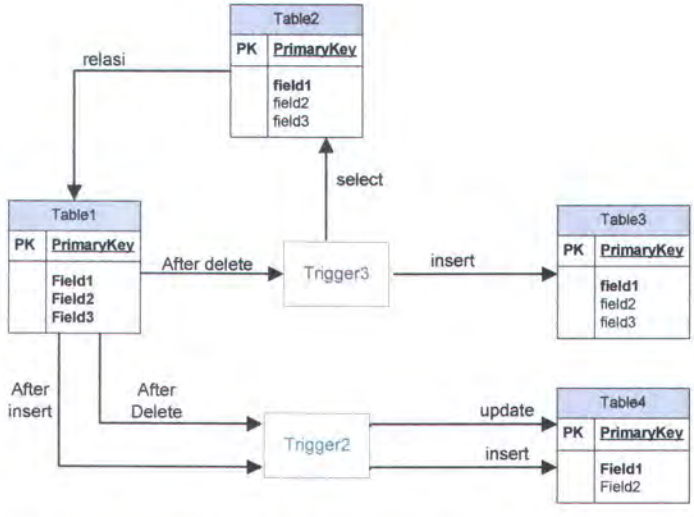
Keterangan diagram:

- Tabel dilambangkan dengan *vertex. Trigger* cukup dilambangkan dengan *edge.*
- Nama *trigger* ditulis sebagai *label* pada tengah-tengah *edge trigger.*
- Untuk *event* dan *action trigger* ditulis sebagai *label* dalam ujung-ujung *edge trigger* itu sendiri.
- Untuk sebuah *trigger,* yang mempunyai dua *action* dan *event* berbeda, digambarkan dengan sebuah *edge* tersendiri.
- Pembedaan *trigger statement level* dengan *trigger row level* adalah dengan pemberian warna merah pada *trigger row level* dan pemberian warna hijau pada *trigger statement level.*

Analisis dari alternatif diagram pertama:

Diagram di atas sudah memenuhi beberapa ketentuan yang telah ditetapkan sebelumnya, yaitu: kompatibel dengan *PDM*, mudah dibaca dan intuitif, daya visibilitas yang tinggi (karena *trigger* dilambangkan hanya dengan *edge*). Akan tetapi diagram di atas mempunyai beberapa kelemahan, yaitu dibutuhkan dua buah *edge* untuk melambangkan satu buah *trigger* (yaitu bila *trigger* tersebut melakukan lebih dari satu proses *action*, baik dalam satu tabel ataupun lebih). Hal ini akan menimbulkan suatu kerancuan dan sedikit membingungkan. Disebut membingungkan sebab keadaan tersebut bisa menimbulkan anggapan bahwa dua *edge* tersebut merupakan dua buah *trigger* yang berbeda satu sama lain, padahal sebenarnya tidak. Kelemahan lainnya dari diagram di atas adalah pengamat tidak bisa menghitung dengan cepat berapa jumlah *trigger* yang ada pada diagram. Sebab patokan jumlah *edge* = jumlah *trigger*, tidak bisa dipakai. Untuk mengatasi permasalahan tersebut, terdapat bentuk diagram alternatif kedua.

3.2.2.2 Bentuk Alternatif Kedua



Gambar 3.3 Alternatif Kedua Diagram *Dependency Trigger*

Keterangan tambahan:

- Tabel dilambangkan dengan *vertex* dan *trigger* kali ini juga dilambangkan dengan *vertex*.
- Nama *trigger* ditulis sebagai *label* pada tengah-tengah *vertex trigger*.
- Untuk sebuah *trigger* yang mempunyai *action* dan *event* lebih dari satu, digambarkan dengan sebuah *edge* tersendiri untuk setiap *action* dan *event*.

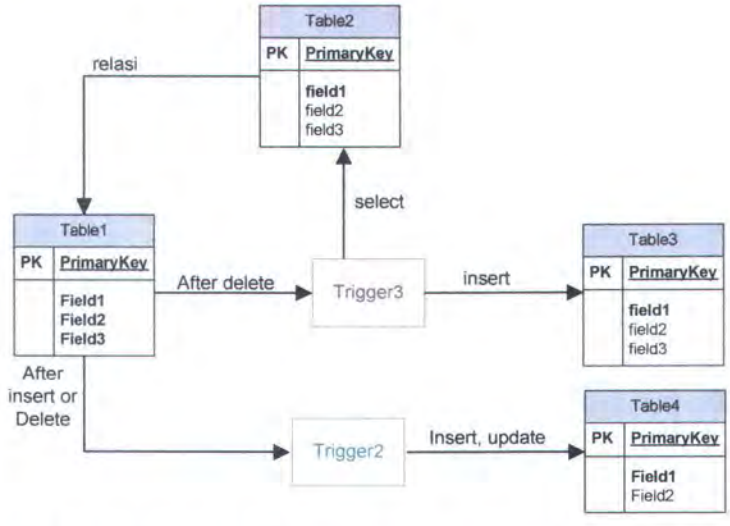
Analisis dari alternatif diagram kedua:

Perbedaan diagram ini dengan alternatif pertama tadi adalah bahwa kali ini *trigger* tidak dilambangkan dengan *edge*, tetapi dilambangkan dengan sebuah *vertex* tersendiri. Hal ini akan menghilangkan kerancuan jumlah *trigger* tanpa banyak menghilangkan karakteristik menguntungkan dari alternatif diagram pertama. Dari bentuk alternatif kedua ini, *user* bisa dengan mudah menghitung berapa jumlah *trigger* yang terdapat dalam diagram, yaitu dengan jalan menghitung *vertex trigger*.

Akan tetapi, dari proses analisis lebih mendalam, akan diketahui bahwa bentuk diagram seperti ini akan menghasilkan kelemahan lain, yaitu diagram ini akan banyak menghasilkan *directed multigraph* (*directed graph* yang terdapat lebih dari satu *edge* antara dua buah *node* dengan arah yang sama). Dalam pembuatan struktur data *graph*, hendaknya bentuk *multigraph* harus sebisa mungkin dihindari. Sebab *multigraph* mempunyai tingkat kerumitan yang lebih tinggi daripada *singlegraph* dan akan menyulitkan proses pembuatan algoritma serta pengimplementasian.

Memang, *multigraph* dalam sebuah *PDM* tidak bisa dihindari, sebab relasi lebih dari satu antara dua buah tabel dalam *PDM* memang diizinkan. Akan tetapi, apabila dalam Diagram *Dependency Trigger* juga memunculkan suatu *directed multigraph,* kerumitan diagram akan bertambah. Oleh karena itu, diusahakan *directed multigraph* hanya muncul pada kasus *PDM* saja tanpa mucul dalam Diagram *Dependency Trigger*.

3.2.2.3 Bentuk Alternatif Ketiga



Gambar 3.4 Alternatif Ketiga Diagram *Dependency Trigger*

Keterangan tambahan:

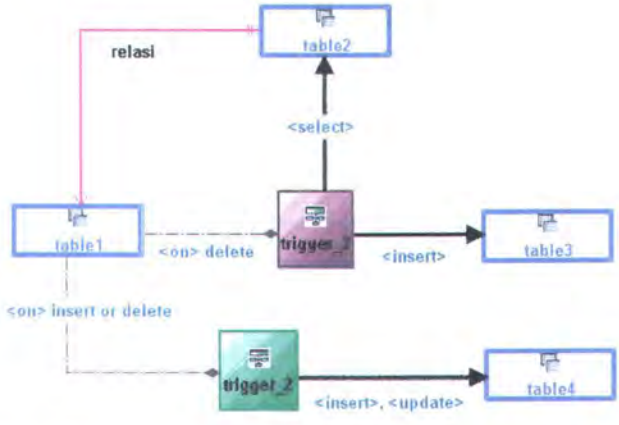
- Sama dengan rancangan diagram kedua. Akan tetapi, setiap *action* untuk satu *trigger* dan satu *table* menjadi hanya sebuah *edge.*

Analisis dari alternatif diagram ketiga:

Bentuk diagram ketiga ini sama dengan diagram kedua namun menghilangkan adanya *multigraph. Action* dan *event* yang berbeda pada *trigger* yang sama dijadikan satu buah *edge*.

Karena Diagram *Dependency Trigger* pada batasan masalah di Tugas Akhir ini diprioritaskan pada visualisasi *trigger* bukan pada Diagram *PDM*, maka visualisasi tabel cukup dengan penampilan nama tabel saja (*field-field* tidak ditampilkan). Hal ini akan juga menambah daya visibilitas Diagram *Dependency Trigger* untuk kasus yang kompleks.

Berikut ini adalah bentuk akhir Diagram *Dependency Trigger* yang diputuskan akan dipakai:



Gambar 3.5 Rancangan Final Diagram *Dependency Trigger*

3.2.3 Analisis Penyebab *Mutating Table* pada *Oracle*

Pada bab sebelumnya telah dibahas mengenai *mutating table* pada *Oracle*. Pada bab ini akan dianalisis lebih mendalam mengenai penyebab kasus *mutating table* pada *Oracle*.

Dalam subbab ini akan dibahas mengenai *Chain reaction*. Adapun *chain reaction* adalah sebuah proses berantai dimana *output* dari sebuah subproses menjadi *input* dari subproses selanjutnya. Bila *ouput* yang dihasilkan subproses sebelumnya tidak sesuai dengan ketentuan-ketentuan tertentu, maka *output*

tersebut tidak akan bisa dipakai sebagai *input* pada subproses berikutnya. Contoh *chain reaction* pada dunia nyata adalah proses reaksi nuklir [CHR06].

Untuk kasus *mutating table*, *chain reaction* bisa diartikan sebagai reaksi berantai antara *action* dari sebuah *trigger* dengan *event* dari *trigger* berikutnya dalam sebuah Diagram *Dependency Trigger*. *Chain reaction* dikatakan berhasil apabila memenuhi ketentuan-ketentuan tertentu. Ketentuan-ketentuan tersebut akan dijelaskan pada subbab ini.

Sesuai dengan batasan masalah pada bab 2, peramalan *mutating table* pada Tugas Akhir ini hanya diprioritaskan pada *event* dan *action* saja. Artinya, Tugas akhir ini mencari semua *mutating table* hanya berdasarkan *chain reaction* antara *event* dan *action* per-tabel, bukan *action* per-kolom. Sehingga setiap pengaksesan tabel dianggap mengakses *key column*, meskipun pihak *developer trigger* sudah mengantisipasi hal di atas dalam *trigger*-nya.

Penyebab kasus *mutating table* pada *Oracle* dibagi menjadi dua, yaitu *mutating table* karena *chain reaction* antara *trigger-trigger* dan *mutating table* karena kasus *chain reaction* antara relasi *cascade delete-trigger*.

3.2.3.1 *Mutating Table* Karena Kasus 1 (*Chain Reaction Trigger – Trigger*)

Adalah *mutating table* yang terjadi karena *chain reaction event* dan *action* dalam suatu rangkaian *trigger* tanpa melibatkan relasi *cascade delete*. Karena *chain reaction* untuk kasus 1 ini tidak melibatkan relasi *cascade delete*, maka kasus 1 ini bisa terjadi pada struktur tabel tanpa relasi sama sekali.

3.2.3.1.1 Analisis Contoh 1 (untuk Kasus 1)

Terdapat enam buah *object*, yaitu tiga tabel dan tiga *trigger*. Keenam *object* dalam contoh kali ini terhubung satu sama lain oleh *chain reaction event* dan *action*. Adapaun *DDL* dari enam buah *object* tersebut adalah sebagai berikut:

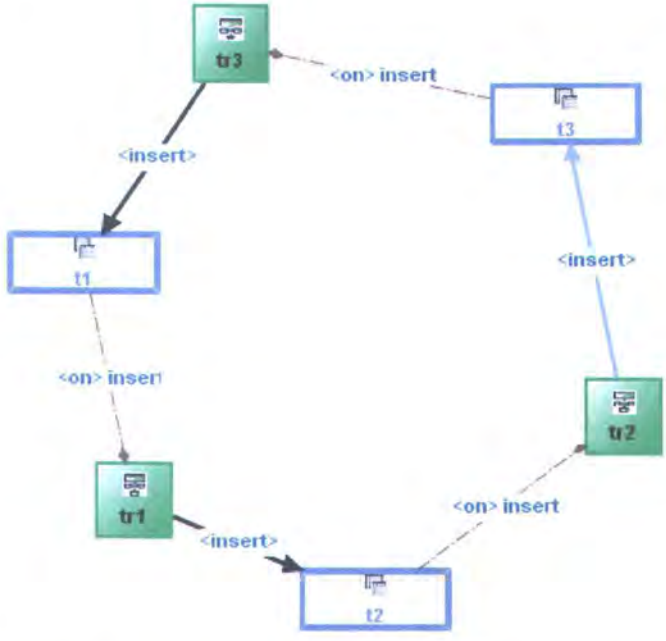
```
create table T1  (
   COL_T1                VARCHAR(50)
)
create table T2  (
   COL_T2                VARCHAR(50)
)
create table T3  (
   COL_T3                VARCHAR(50)
)
create or replace trigger tr1
       after insert on t1
       --for each row
begin
      insert into t2 values('testT2');
end;
go
-------------------------------------------------------
create or replace trigger tr2
       after insert on t2
       --for each row
begin
      insert into t3 values('testT3');
end;
go
-------------------------------------------------------
create or replace trigger tr3
       after insert on t3
       --for each row
begin
      insert into t1 values('testT3');
end;
go
```

Struktur dari *object-object* di atas adalah sebagaimana berikut:

- Terdapat tiga tabel tanpa relasi, yaitu *t1,t2,* dan *t3.*
- Terdapat tiga buah *trigger* pada masing-masing tabel, yaitu *tr1,tr2,* dan *tr3.* Semua *trigger* ini bertipe *statement level.*
- Semua *trigger* di atas mempunyai *event after insert.*

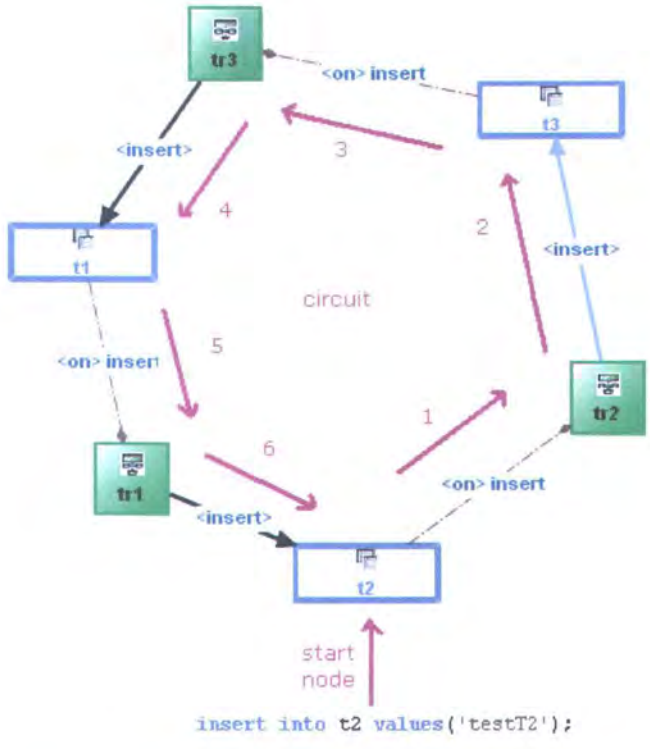
- Setiap *trigger* mempunyai *action* ke arah tabel tetangga terdekatnya, yaitu *tr1* mempunyai *action* ke *t2, tr2* mempunyai *action* ke *t3, tr3* mempunyai *action* ke *t1*.

Apabila *object-object* di atas dibentuk Diagram *Dependency Trigger*-nya, maka akan terbentuk suatu *directed graph* dengan *circuit* tertutup sebagaimana gambar berikut ini:



Gambar 3.6 Diagram *Dependency Trigger* Contoh 1 Kasus 1

Dari analisis Diagram *Dependency Trigger* di atas diketahui bahwa apabila tabel *t2* dikenai operasi *insert*, maka akan terjadi suatu *chain reaction* yang berawal dari tabel *t2*. *Chain reaction* ini membentuk suatu *circuit* tertutup dengan urutan sebagaimana gambar berikut ini:



Gambar 3.7 Analisis *Path* Diagram *Dependency Trigger* Contoh 1 Kasus 1

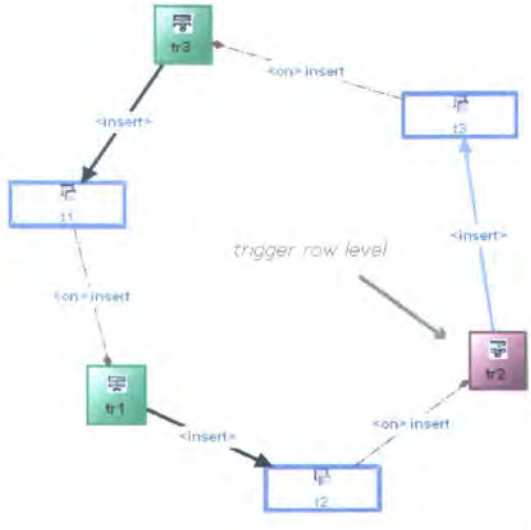
Terlihat bahwa *chain reaction event* dan *action* berhasil menemukan jalan untuk kembali ke tabel asal, yaitu tabel *t2*. Analisis lebih lanjut pada diagram di atas akan diketahui bahwa *chain reaction* tidak akan pernah berhenti (*infinite looping*). Adapun *output* ketika operasi *insert* di atas benar-benar dijalankan pada *Oracle* adalah sebagaimana berikut ini:

```
>[Error] Script lines: 1-1 -------------------------
 ORA-00036: maximum number of recursive SQL levels (50) exceeded
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
```

Ternyata tidak terjadi *mutating table* pada contoh kasus ini, tetapi yang terjadi adalah rekursi. Ini artinya tabel *t2* bukanlah tabel yang *mutating*, meskipun *trigger tr1* mengaksesnya sebagai akibat dari *chain reaction* yang berawal dari tabel *t2* itu sendiri. Jadi, tabel *t2* sudah tidak berada dalam kondisi *mutating* ketika *tr1* mengakses *t2*. Hal ini dikarenakan *tr2* (*vertex* kedua dari *circuit*) dieksekusi dalam kondisi *statement level*, yaitu dieksekusi ketika *DML insert* yang terjadi pada tabel *t2* telah selesai dijalankan seluruhnya.

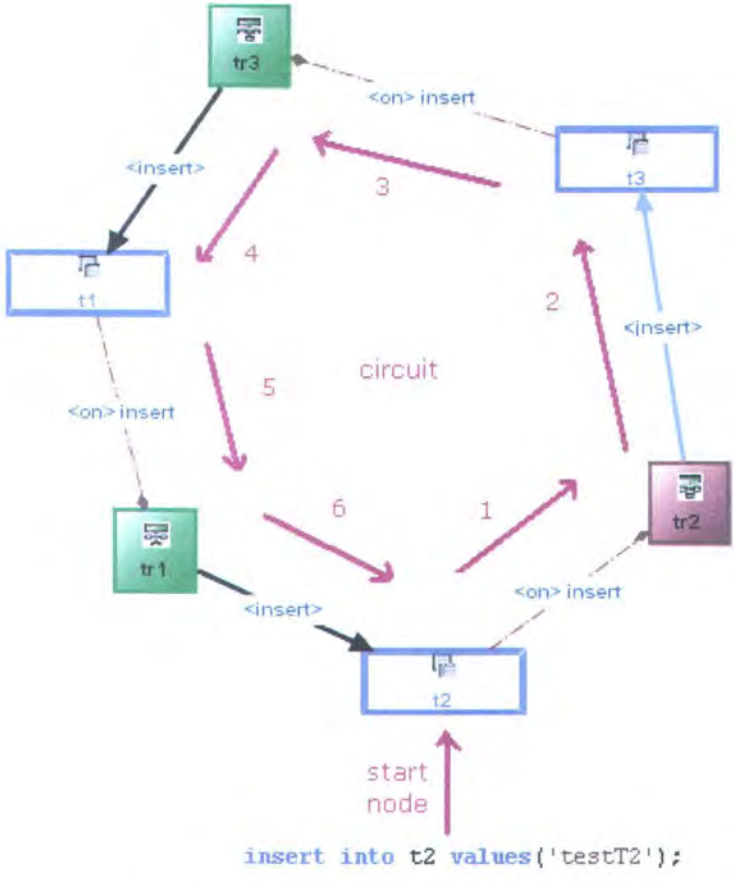
3.2.3.1.2 Analisis Contoh 2 (untuk Kasus 1)

Untuk menghasilkan kasus *mutating table*, maka contoh kasus di atas dimodifikasi sedemikian rupa sehingga bisa menghasilkan suatu kasus *mutating table*. Modifikasi dilakukan pada salah satu *trigger*, yaitu *tr2*. Kali ini *tr2* diubah dari yang sebelumnya berupa *trigger statement level*, maka sekarang menjadi *trigger row level*. Berikut ini adalah Diagram *Dependency Trigger* hasil modifikasi:



Gambar 3.8 Diagram *Dependency Trigger* Contoh 2 Kasus 1

Tampak bahwa *trigger tr2* sekarang berwarna merah menandakan bahwa *tr2* adalah *trigger row level*. Untuk membandingkan struktur Diagram *Dependency Trigger* ini dengan sebelumnya, maka *DML insert* kembali dilakukan pada tabel *t2* sebagaimana contoh 1 sebelumnya.



Gambar 3.9 Analisis *Path* Diagram *Dependency Trigger* Contoh 2 Kasus 1

Output yang muncul adalah:

```
>[Error] Script lines: 1-2 ------------------------
 ORA-04091: table TESTBED.T2 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
```

Ternyata kasus *mutating table* terjadi pada tabel *t2*. Hal ini terjadi sebab tabel *t2* berada dalam keadaan *mutating* ketika *action* milik *tr1* mengaksesnya. Keadaan *mutating table t2* ini disebabkan karena *trigger tr2* yang bersifat *row level*. Sifat *row level* dari *tr2* ini membuat *trigger tr3*, dan *trigger tr1* dieksekusi ketika *trigger tr2* masih dalam keadaan *mutating*. Jadi, *DML insert* belum selesai tereksekusi untuk semua *row* dalam tabel *t2*.

Kasus *mutating* ini tidak hanya terjadi apabila *DML insert* dilakukan pada tabel *t2* saja, akan tetapi, *DML insert* pada semua tabel dalam *circuit* di atas akan menghasilkan kasus *mutating table* pada tabel *t2*.

Contoh:

```
insert into t3 values('testT3')
```

Akan menghasilkan *output*:

```
>[Error] Script lines: 1-2 --------------------------
 ORA-04091: table TESTBED.T2 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
```

Contoh lain:

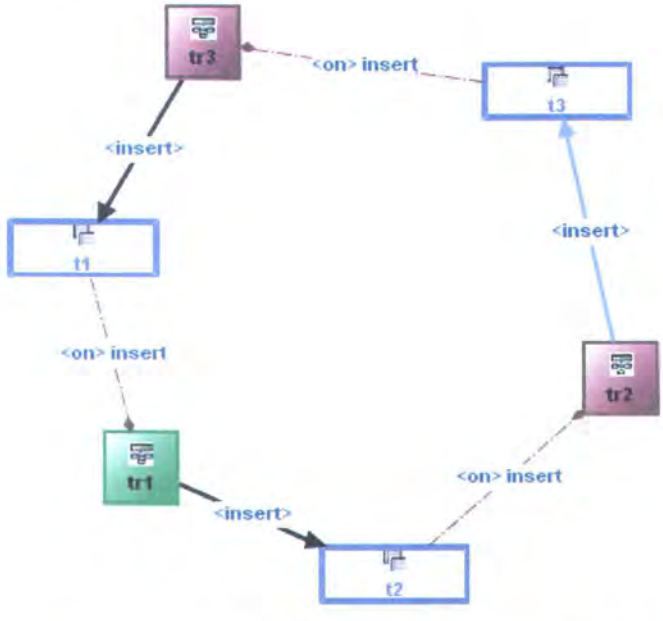
```
insert into t1 values('testT3')
```

Akan menghasilkan *output*:

```
>[Error] Script lines: 1-2 --------------------------
 ORA-04091: table TESTBED.T2 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
```

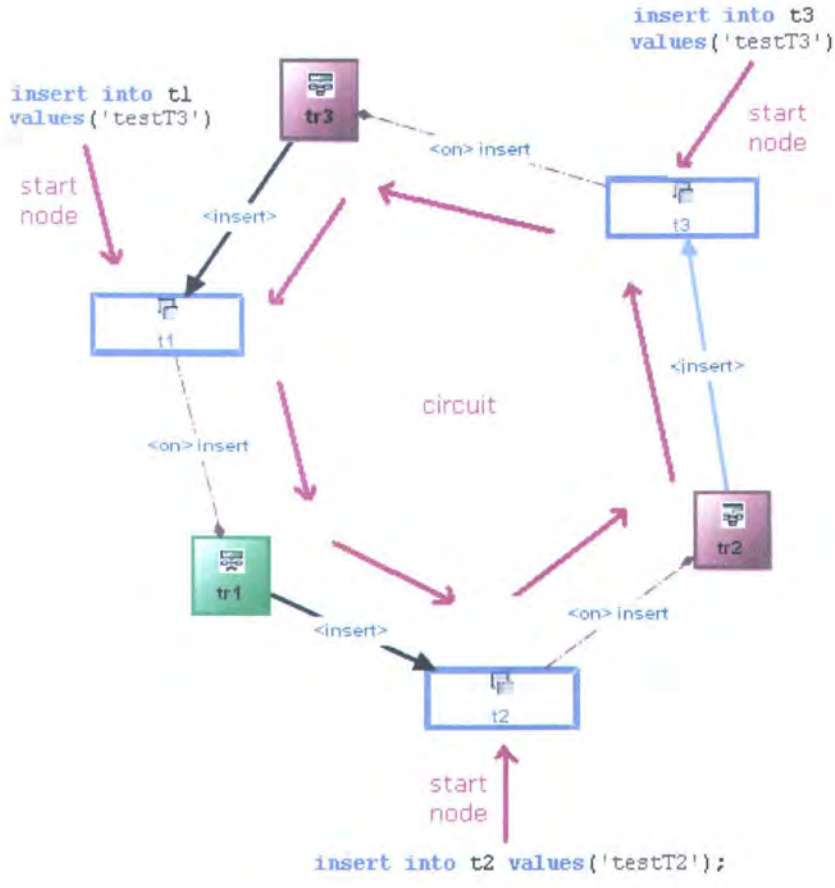
3.2.3.1.3 Analisis Contoh 3 (untuk Kasus 1)

Kasus berikut adalah modifikasi lebih lanjut dari contoh 2. Perubahan yang ada kali ini adalah perubahan *trigger tr3* menjadi *trigger row level*. Hasil perubahan akan tampak pada Diagram *Dependency Trigger* berikut ini:



Gambar 3.10 Diagram *Dependency Trigger* Contoh 3 Kasus 1

Dengan perubahan tipe dari dua buah *trigger row level* pada struktur Diagram *Dependency Trigger* di atas, maka kasus *mutating table* akan juga akan mengalami perubahan. Untuk menganalisis struktur baru ini, *DML insert* kembali dijalankan pada ketiga tabel di atas.



Gambar 3.11 Analisis *Path* Diagram *Dependency Trigger* Contoh 3 Kasus 1

Adapun *output* dari setiap operasi *insert* yang dijalankan pada struktur di atas adalah sebagai berikut:

Output dari *DML insert* pertama:

```
insert into t2 values('test')
```

```
>[Error] Script lines: 1-3 --------------------------
 ORA-04091: table TESTBED.T2 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
```

Output dari *DML insert* kedua:

```
insert into t3 values('test')
```

```
>[Error] Script lines: 1-3 --------------------------
 ORA-04091: table TESTBED.T3 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
```

Output dari *DML insert* ketiga:

```
insert into t1 values('test')
```

```
>[Error] Script lines: 1-3 --------------------------
 ORA-04091: table TESTBED.T2 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
ORA-06512: at "TESTBED.TR3", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR3'
ORA-06512: at "TESTBED.TR2", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR2'
ORA-06512: at "TESTBED.TR1", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR1'
```

Dari hasil eksekusi ketiga *DML insert* di atas, terdapat dua buah tabel yang bersifat *mutating*, yaitu tabel *t2* dan tabel *t3*. Penjelasan mengapa tabel *t2* bersifat *mutating* bisa dilihat pada contoh-contoh sebelumnya. Sedangkan penyebab tabel *t3* sehingga bersifat *mutating* adalah kurang lebih sama dengan penyebab *mutating* dari tabel *t2*, yaitu hal ini terjadi sebab tabel *t3* berada dalam keadaan *mutating* ketika *action* milik *tr2* mengaksesnya. Keadaan *mutating table t3* ini disebabkan karena *trigger tr3* bersifat *row level*. Sifat *row level* dari *tr3* ini membuat *trigger tr1*, dan *trigger tr2* dieksekusi ketika *trigger tr3* masih dalam keadaan belum selesai mengeksekusi semua *row* dalam tabel *t3*.

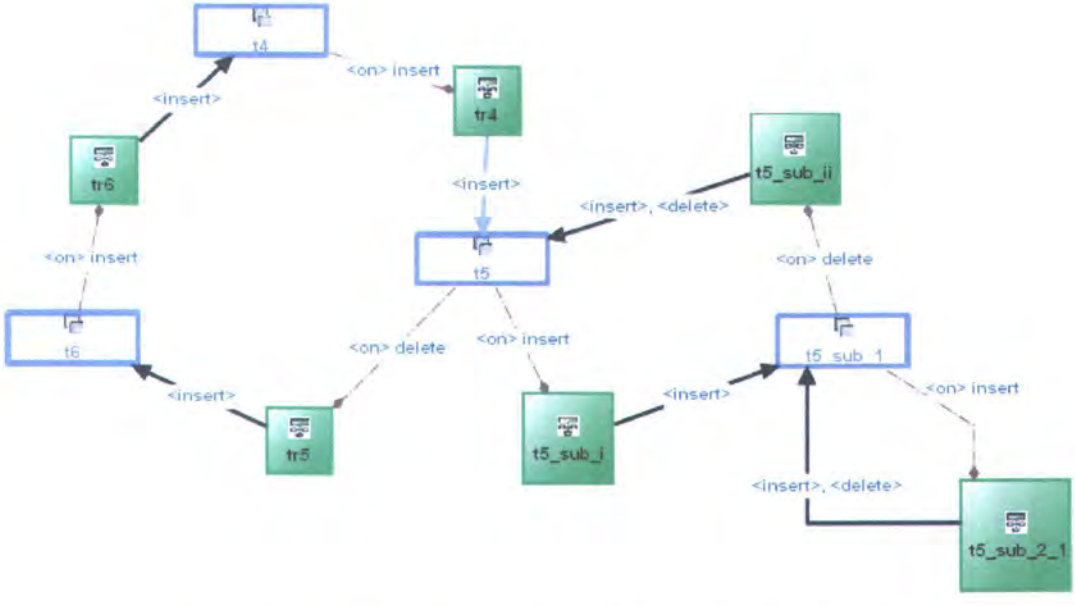
3.2.3.1.4 Analisis Contoh 4 (untuk Kasus 1)

Untuk menganalisis kasus *mutating table* yang lebih rumit, berikut ini adalah contoh struktur diagram yang mempunyai pelbagai percabangan *path* dalam sebuah *circuit*:

```
create table T4  (
   COL_T4                VARCHAR(50)
)
/
create table T5  (
   COL_T5                VARCHAR(50)
)
/
create table T6  (
   COL_T6                VARCHAR(50)
)
create or replace trigger tr4
       after insert on T4
       --for each row
begin
      insert into t5 values('testT4');
end;
go

drop table t5_sub_1 cascade constraint
/
create table t5_sub_1  (
   COL_t5_sub_1                VARCHAR(50)
)
/
create or replace trigger t5_sub_I
       after insert on t5
       --for each row
begin
      insert into t5_sub_1 values('testT5');
end;
go

create or replace trigger t5_sub_2_1
       after insert on t5_sub_1
       --for each row
begin
      insert into t5_sub_1 values('testT5');
      delete from t5_sub_1;
end;
go

create or replace trigger t5_sub_II
       after delete on t5_sub_1
       --for each row
begin
      insert into t5 values('testT5');
      delete from t5;
end;
go

create or replace trigger tr5
       after delete on t5
       --for each row
begin
      insert into t6 values('testT5');
```

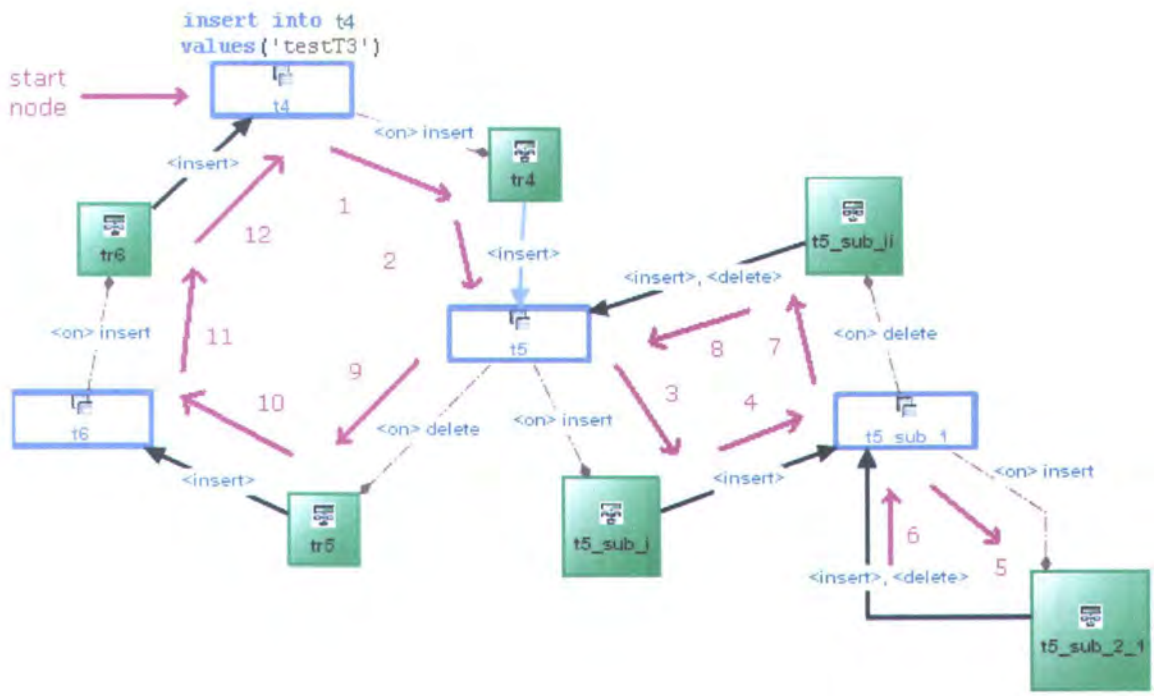
```
end;
go

create or replace trigger tr6
       after insert on t6
       --for each row
begin
      insert into t4 values('testT6');
end;
go
```

Bentuk Diagram *Dependency Trigger* dari struktur tabel-*trigger* di atas adalah sebagaimana berikut ini:



Gambar 3.12 Diagram *Dependency Trigger* Contoh 4 Kasus 1

Pada proses analisis struktur kali ini, tabel *t4* dijadikan *start node*. *DML* yang ditujukan pada tabel *t4* adalah *insert*. Hal ini sengaja supaya *trigger* terdekat dengan *t4* (yaitu *trigger tr4*) bisa tereksekusi.



Gambar 3.13 Analisis *Path* Diagram *Dependency Trigger* Contoh 4 Kasus 1

Output dari *DML insert*:

```
insert into t4 values('test')

>[Error] Script lines: 1-1 --------------------------
ORA-00036: maximum number of recursive SQL levels (50) exceeded
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 2
```

Dari percobaan ini bisa disimpulkan bahwa meskipun terjadi percabangan, *chain reaction* antara *event* dan *action* tetap berlangsung.

Rekursi terjadi pada *circuit t5_sub_1 → 5 → t5_sub_2_1 → 6 → t5_sub_1.*

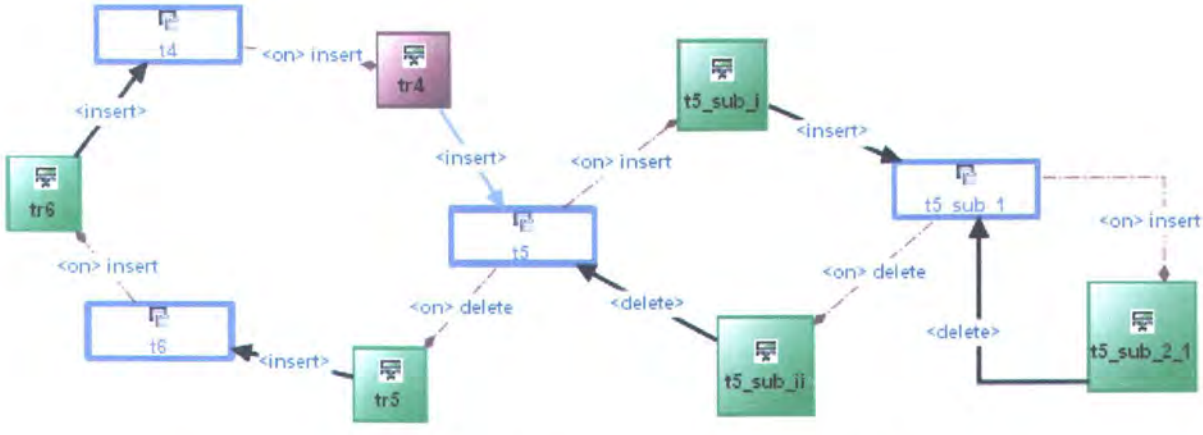
3.2.3.1.5 Analisis Contoh 5 (untuk Kasus 1)

Dari analisis pada contoh-contoh sebelumnya, apabila dari kasus ini ingin dihasilkan suatu *mutating table* (yaitu tabel *t4*), maka hal tersebut bisa terjadi dengan jalan mengubah *trigger tr4* menjadi *row level*. Selain itu, untuk menghindari *recursive SQL* yang mendahului terjadinya *mutating table*, maka *action* untuk beberapa *trigger* harus diubah. Dalam hal ini adalah *trigger t5_sub_ii* dan *t5_sub_2_1* diubah menjadi satu buah *action* saja.

```
create table T4  (
   COL_T4                VARCHAR(50)
)
/
insert into t4 values('testing')
go

create table T5  (
   COL_T5                VARCHAR(50)
)
/
insert into t5 values('testing')
go

create table T6  (
   COL_T6                VARCHAR(50)
)
/
insert into t5 values('testing') --inisialisasi isi
go

create or replace trigger tr4
       after insert on T4
       for each row
begin
      insert into t5 values('testT4');
end;
go

drop table t5_sub_1 cascade constraint
/
create table t5_sub_1  (
   COL_t5_sub_1              VARCHAR(50)
)
/
```

```
insert into t5_sub_1 values('testing') --inisialisasi isi
go

create or replace trigger t5_sub_I
       after insert on t5
       --for each row
begin
      insert into t5_sub_1 values('testT5');
end;
go

create or replace trigger t5_sub_2_1
       after insert on t5_sub_1
       --for each row
begin
      --insert into t5_sub_1 values('testT5');
      delete from t5_sub_1;
end;
go

create or replace trigger t5_sub_II
       after delete on t5_sub_1
       --for each row
begin
      --insert into t5 values('testT5');
      delete from t5;
end;
go

create or replace trigger tr5
       after delete on t5
       --for each row
begin
      insert into t6 values('testT5');
end;
go

create or replace trigger tr6
       after insert on t6
       --for each row
begin
      insert into t4 values('testT6');
end;
go
```

Hasil pembentukan Diagram *Dependency Trigger* dari *DDL* di atas adalah sebagaimana gambar berikut ini:



Gambar 3.14 Diagram *Dependency Trigger* Contoh 5 Kasus 1

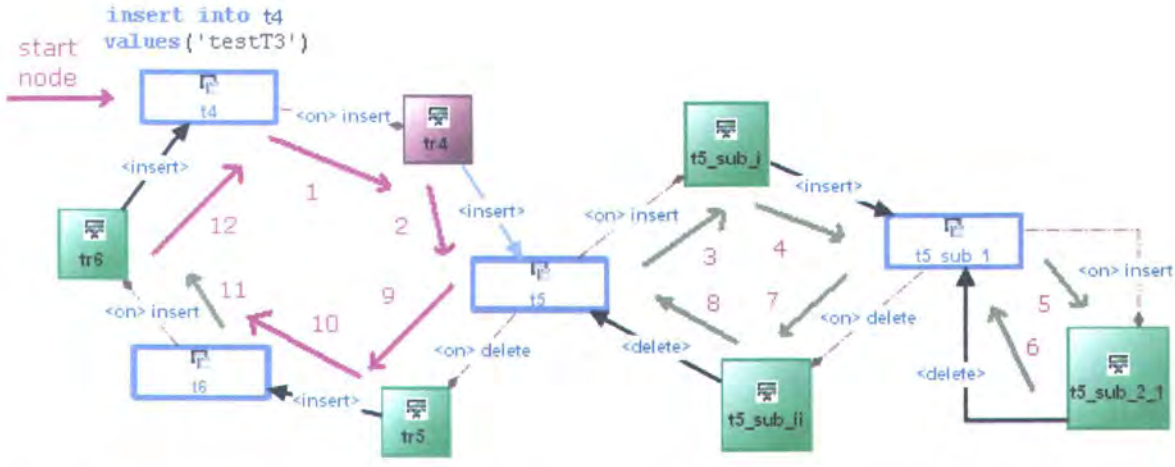
Sekarang apabila *DML insert* dijalankan pada tabel *t4*, diharapkan akan terjadi *mutating table* pada tabel *t4*.

Berikut ini adalah pembuktiannya:

```
insert into t4 values('testT3')
```

```
>[Error] Script lines: 1-1 -------------------------
 ORA-04091: table TESTBED.T4 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED.TR6", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR6'
ORA-06512: at "TESTBED.TR5", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR5'
ORA-06512: at "TESTBED.T5_SUB_II", line 3
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_II'
ORA-06512: at "TESTBED.T5_SUB_2_1", line 3
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_2_1'
ORA-06512: at "TESTBED.T5_SUB_I", line 2
ORA-04088: error during execution of trigger 'TESTBED.T5_SUB_I'
ORA-06512: at "TESTBED.TR4", line 2
ORA-04088: error during execution of trigger 'TESTBED.TR4'
```

Dengan melihat Diagram *Dependency Trigger*, maka rute penyebab ke-*mutating*-an tabel *t4* akan mudah dilihat. Perhatikan rute berikut ini:



Gambar 3.15 Analisis *Path* Diagram *Dependency Trigger* Contoh 5 Kasus 1

Penelusuran rute penyebab kasus *mutating table* bisa dilakukan dengan jalan menelusuri setiap *edge* pada diagram sesuai dengan urutan nomor, yaitu mulai dari *edge* nomor 1 hingga *edge* nomor 12.

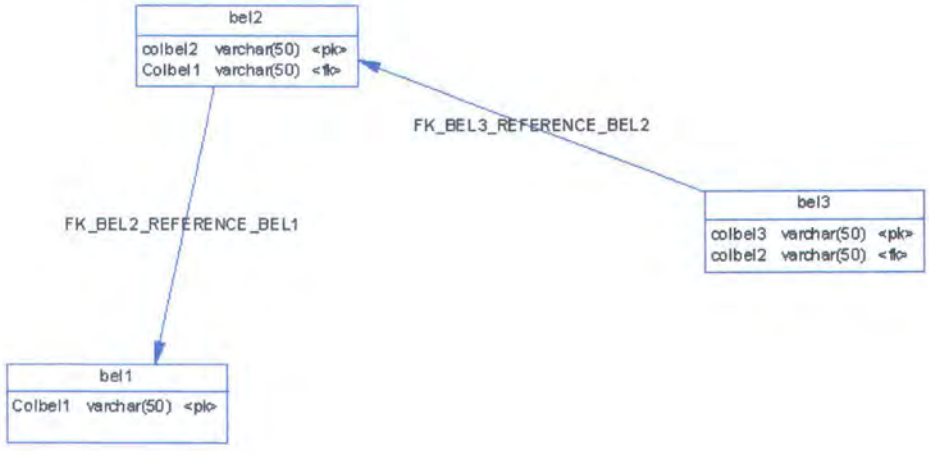
3.2.3.2 *Mutating Table* Karena Kasus 2 (*chain reaction cascade delete-trigger*)

Apabila sebelumnya analisis dilakukan pada kasus tabel tanpa relasi, maka pada bagian ini akan dibahas tentang kasus *mutating table* yang terjadi pada tabel berelasi. Khususnya relasi *cascade delete.*

Adapun pengertian dari *mutating table* kasus 2 disini adalah kasus *mutating table* yang terjadi karena *chain reaction* dari *event* dan *action* ditambah dengan reaksi *cascade delete* dalam suatu rangkaian *trigger* sehingga menyebabkan salah satu *trigger* (dari rangkaian *trigger* yang terekseskusi tersebut) kembali mengakses tabel tempat asalnya berada.

3.2.3.2.1 Analisis Contoh 1 (untuk Kasus 2)

Pada contoh 1, analisis akan dilakukan pada tabel dengan struktur sebagaimana berikut ini:



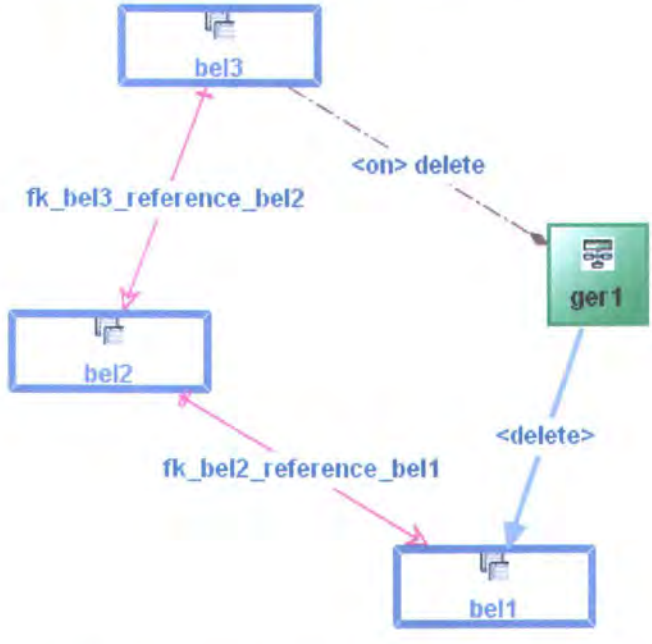
Gambar 3.16 Diagram *PDM* Contoh 1 Kasus 2

Semua relasi dalam struktur Diagram *PDM* di atas adalah relasi *cascade delete*. Adapun *DDL* dari struktur *PDM* di atas adalah sebagaimana berikut:

```
create table BEL1  (
   COLBEL1     varchar(50)         not null,
   constraint PK_BEL1 primary key (COLBEL1)
)
/
create table BEL2  (
   COLBEL2     varchar(50)         not null,
   COLBEL1     varchar(50),
   constraint PK_BEL2 primary key (COLBEL2),
   constraint FK_BEL2_REFERENCE_BEL1 foreign key (COLBEL1)
         references BEL1 (COLBEL1)
         on delete cascade
)
/
create table BEL3  (
   COLBEL3      varchar(50)    not null,
   COLBEL2      varchar(50),
   constraint PK_BEL3 primary key (COLBEL3),
   constraint FK_BEL3_REFERENCE_BEL2 foreign key (COLBEL2)
         references BEL2 (COLBEL2)
        on delete cascade
)
/
CREATE OR REPLACE TRIGGER "TESTBED2"."GER1"
```

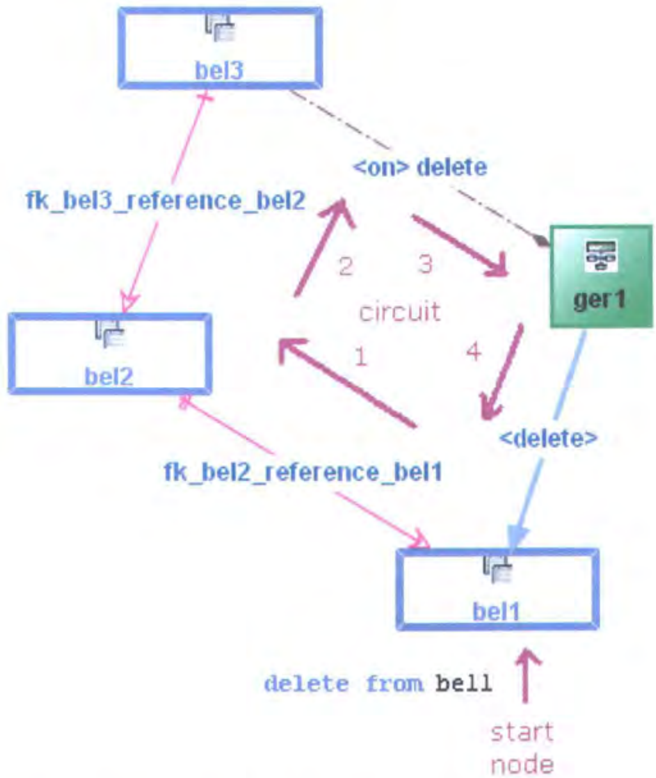
```
        after delete on bel3
        --for each row
begin
      delete from bel1; --into t4 values('testT6');
end;
go
--untuk inisialisasi isi
insert into bel1 values('1')
go
insert into bel2 values('1','1')
go
insert into bel3 values('1','1')
go
```

Diagram *Dependency Trigger*-nya adalah sebagaimana berikut:



Gambar 3.17 Diagram *Dependency Trigger* Contoh 1 Kasus 2

Dari Diagram *Dependency Trigger* di atas, bisa diketahui bahwa terdapat sebuah *circuit* di dalam struktur kali ini. Karena terdapat *circuit* tanpa ada satupun *trigger* yang *row level*, maka kasus rekursi mungkin terjadi. Untuk membuktikannya, dipilih tabel *bel1* sebagai *start node*. *DML* yang dijalankan kali ini adalah *DML delete*. Sebab *DML delete* adalah satu-satunya *DML* yang menyebabkan relasi *cascade delete* bereaksi.



Gambar 3.18 Analisis 1 *Path* Diagram *Dependency Trigger* Contoh 1 Kasus 2

Hasil *output:*

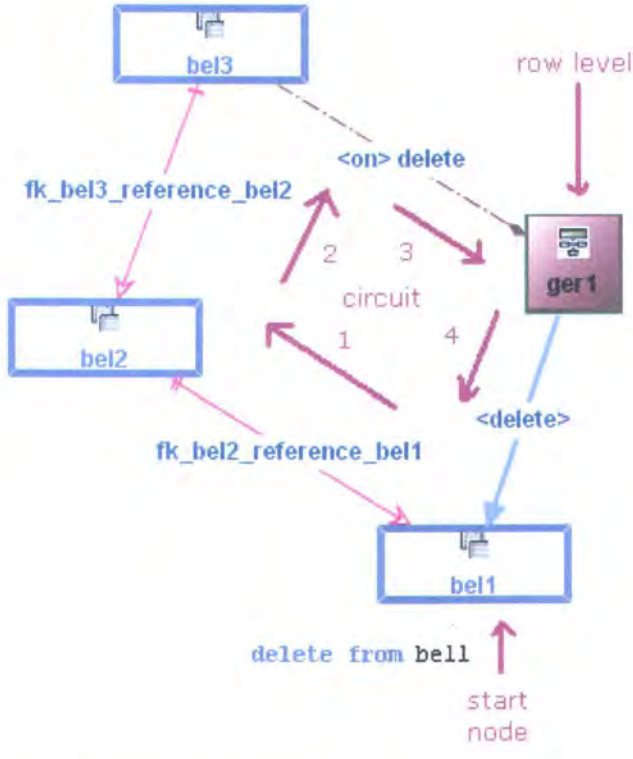
```
delete from bel1

>[Error] Script lines: 1-3 --------------------------
 ORA-00036: maximum number of recursive SQL levels (50) exceeded
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
```

Seperti yang diharapkan, rekursi terjadi pada struktur ini. Hal ini membuktikan bahwa *edge* *FK_BEL2_REFERENCE_BEL1* dan *edge* *FK_BEL3_REFERENCE_BEL2* berhasil meneruskan *chain reaction* dalam *circuit*

di atas. Selanjutnya bisa diketahui bahwa dengan mengubah *trigger ger1* menjadi *row level*, maka kasus *mutating table* diperkirakan akan terjadi pada tabel *bel1*.

```
CREATE OR REPLACE TRIGGER "TESTBED2"."GER1"
       after delete on bel3
       for each row
begin
      delete from bel1; --into t4 values('testT6');
end;
go
```

Perubahan Diagram *Dependency Trigger* akan tampak seperti berikut ini:



Gambar 3.19 Analisis 2 *Path* Diagram *Dependency Trigger* Contoh 1 Kasus 2

Sekali lagi, *DML delete* dijalankan pada tabel *bel1*. *Output* dari *DML delete* tersebut adalah:

```
delete from bell

>[Error] Script lines: 1-3 --------------------------
 ORA-04091: table TESTBED2.BEL1 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
```

Seperti perkiraan, terjadi *mutating table* pada tabel *bel1*. Hal ini terjadi karena ketika *bel1* terkena *DML delete*, *constraint FK_BEL2_REFERENCE_BEL1* dan *constraint FK_BEL3_REFERENCE_BEL2* bereaksi sehingga mengeksekusi *trigger ger1*. padahal *ger1* itu sendiri mengakses *bel1* yang sedang *mutating*. Dalam kasus ini bisa dilihat bahwa *constraint* dengan sifat *cascade delete* ternyata mempunyai efek yang sama dengan *trigger row level* dengan *event on delete*. Sebagai pembanding, operasi *delete* juga dilakukan pada tabel-tabel lainnya. *Ouputnya* adalah sebagai berikut:

```
delete from bel2

>[Error] Script lines: 1-2 --------------------------
 ORA-04091: table TESTBED2.BEL2 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'

delete from bel3

>[Error] Script lines: 1-2 --------------------------
 ORA-04091: table TESTBED2.BEL3 is mutating, trigger/function may not see it
ORA-06512: at "TESTBED2.GER1", line 2
ORA-04088: error during execution of trigger 'TESTBED2.GER1'
```

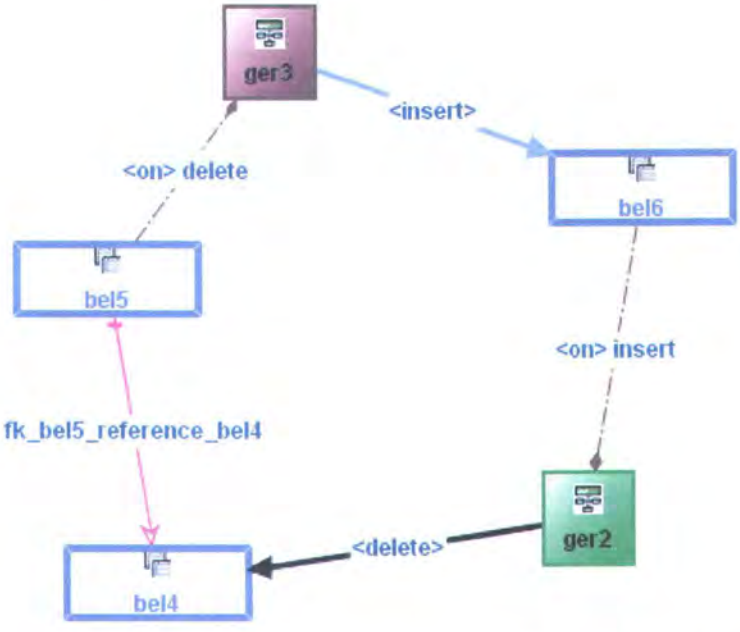
Dimanapun *DML delete* dijalankan, kasus *mutating table* tetap terjadi. Khususnya terjadi pada tabel dimana *DML delete* tersebut dilakukan.

3.2.3.2.2 Analisis Contoh 2 (untuk Kasus 2)

Berikut ini adalah contoh lain untuk kasus 2:

```
create table BEL4  (
   COLBEL4    varchar(50)    not null,
   constraint PK_BEL4 primary key (COLBEL4)
)
/

create table BEL5  (
   COLBEL5     varchar(50)    not null,
   COLBEL4     varchar(50),
   constraint PK_BEL5 primary key (COLBEL5),
   constraint FK_BEL5_REFERENCE_BEL4 foreign key (COLBEL4)
         references BEL4 (COLBEL4)
      on delete cascade
```

```
)
/
create table BEL6  (
   COLBEL6      varchar(50)  not null,
   constraint PK_BEL6 primary key (COLBEL6)
)
/
CREATE OR REPLACE TRIGGER "TESTBED2"."GER2"
       after insert on bel6
       --for each row
begin
      delete from bel4; --into t4 values('testT6');
end;
go

CREATE OR REPLACE TRIGGER "TESTBED2"."GER3"
       after delete on bel5
       for each row
begin
      insert into bel6 values('testT6');
end;
go

--inisialisasi isi
insert into bel4 values('1')
go
insert into bel5 values('1','1')
go
```



Gambar 3.20 Diagram *Dependency Trigger* Contoh 2 Kasus 2

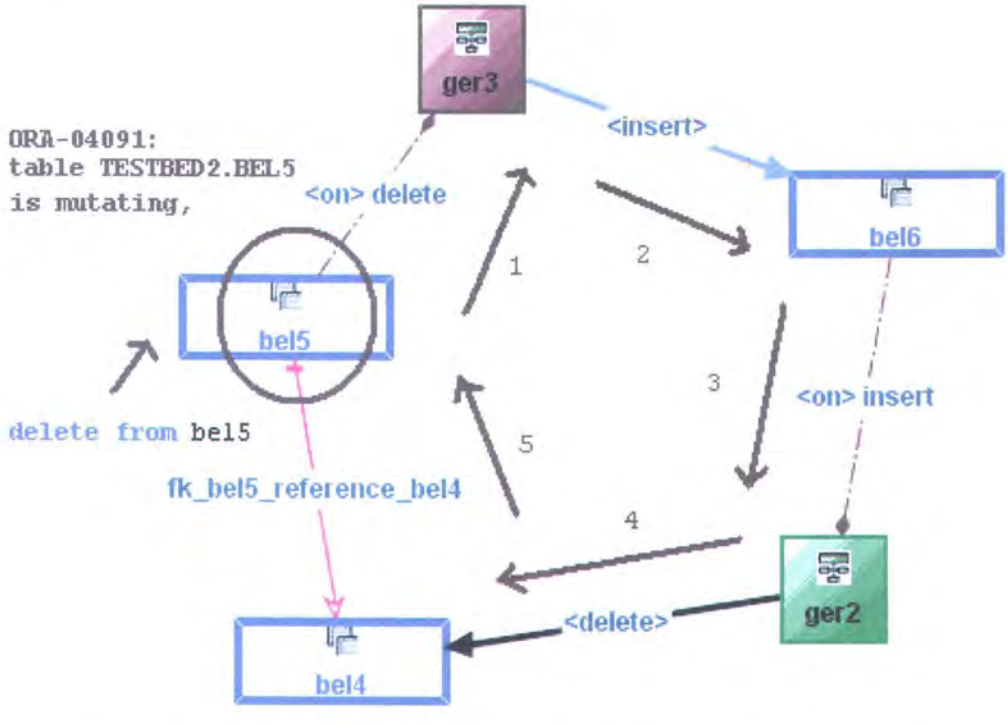
Untuk mengetahui dimana kasus *mutating table* pada Diagram *Dependency Trigger* kali ini, digunakan dua buah *DML*, yaitu:

```
delete from bel5
```

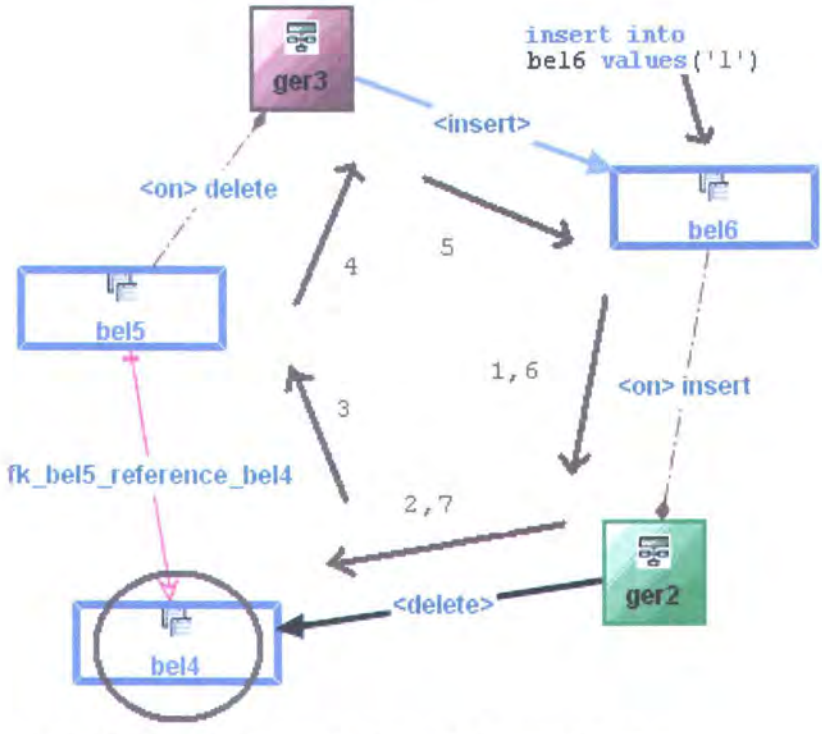
Dan juga:

```
insert into bel6 values('1')
```

Masing-masing dari *DML* di atas akan menghasilkan kasus *mutating table* pada dua tabel yang berbeda, yaitu tabel *bel4* dan juga tabel *bel5*. sedangkan *bel6* bukanlah tabel yang *mutating*.



Gambar 3.21 Analisis 1 *Path* Diagram *Dependency Trigger* Contoh 2 Kasus 2



Gambar 3.22 Analisis 2 *Path* Diagram *Dependency Trigger* Contoh 2 Kasus 2

Dari analisis *path* di atas tampak bahwa *DML insert* pada *bel6* membutuhkan lebih dari satu putaran untuk bisa membuat tabel *bel4* menjadi *mutating.*

Tanpa melakukan percobaan, selanjutnya bisa diketahui pula bahwa operasi

```
delete from bel4
```

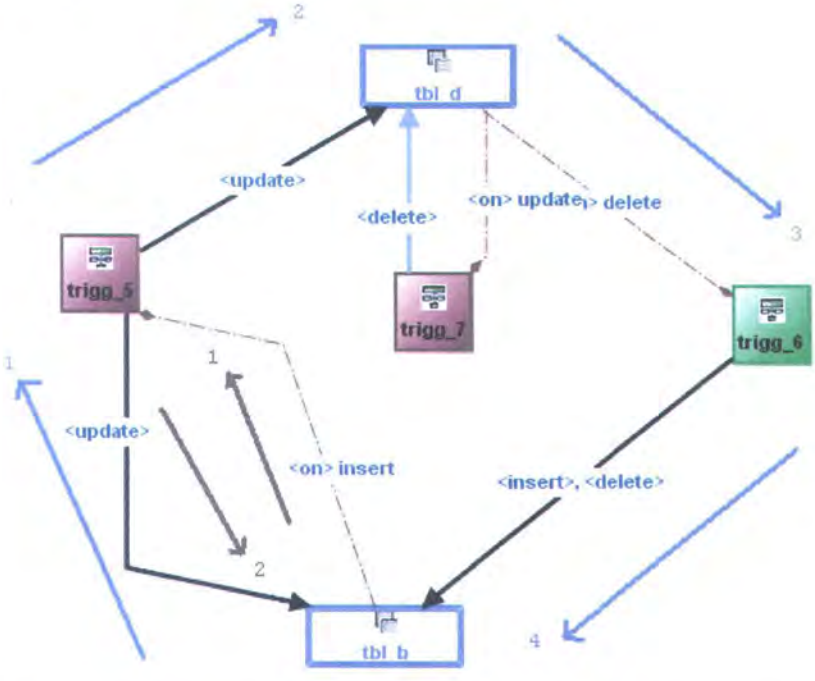
diharapkan akan menghasilkan *mutating table* di tabel *bel4.*

3.2.3.3 Hasil Analisis dari Percobaan Kasus 1 dan Kasus 2

kesimpulan yang diambil dari penyebab *mutating table* pada *Oracle* adalah:

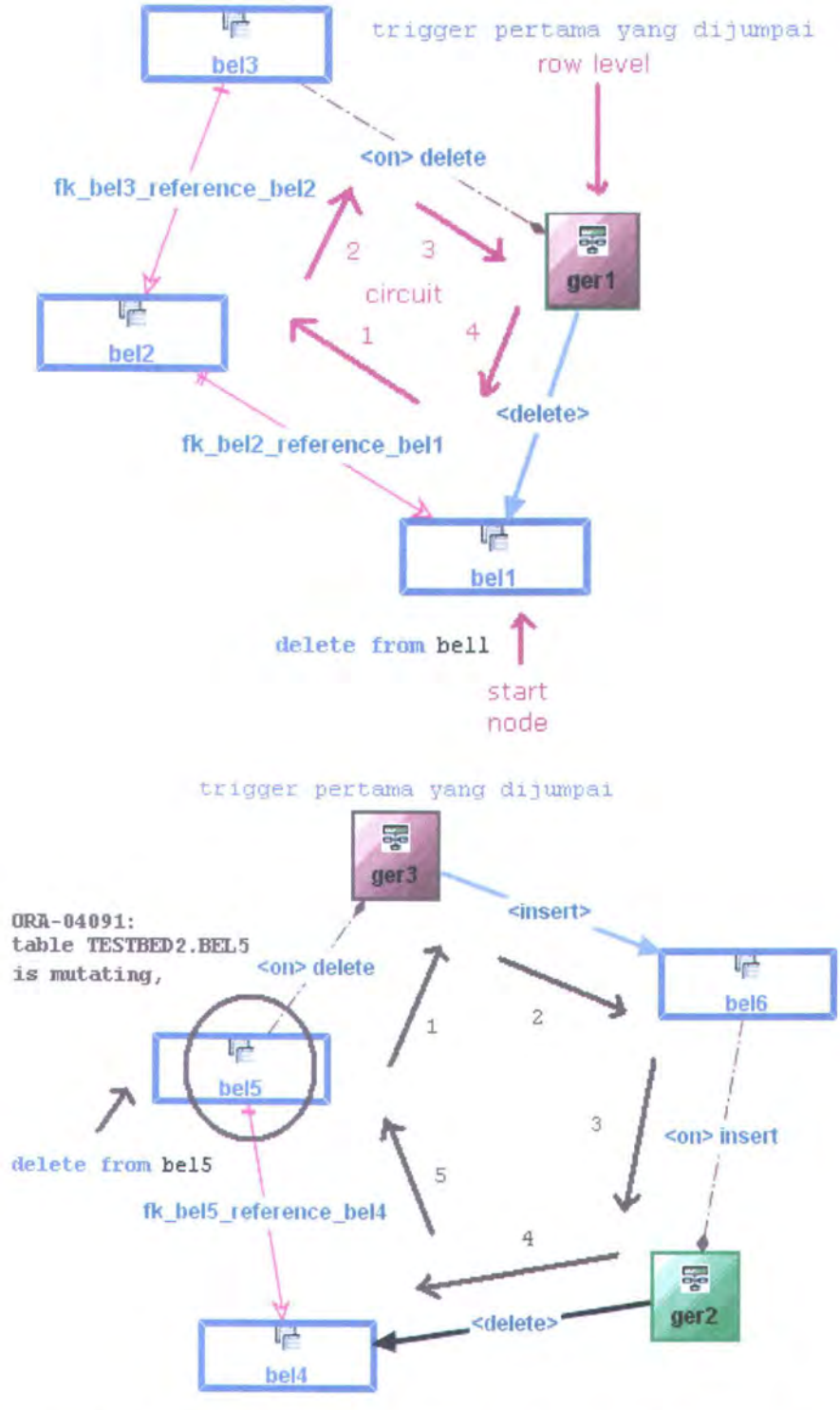
- Pesan kesalahan *ORA-04091* (*mutating table*) terjadi karena ada *DML* yang mengakses tabel yang sedang berada dalam kondisi *mutating*. Baik secara *direct access* ataupun *indirect access. Direct access* maksudnya adalah *trigger*

tersebut mengakses tabel secara langsung tanpa perantara tabel/*trigger* lainnya dalam suatu *circuit*. Sedangkan *indirect access* adalah *trigger* yang tereksekusi dari tabel yang dikenai *event* kemudian men*trigger trigger-trigger* lainnya hingga kembali ke tempat awal (*circuit*)



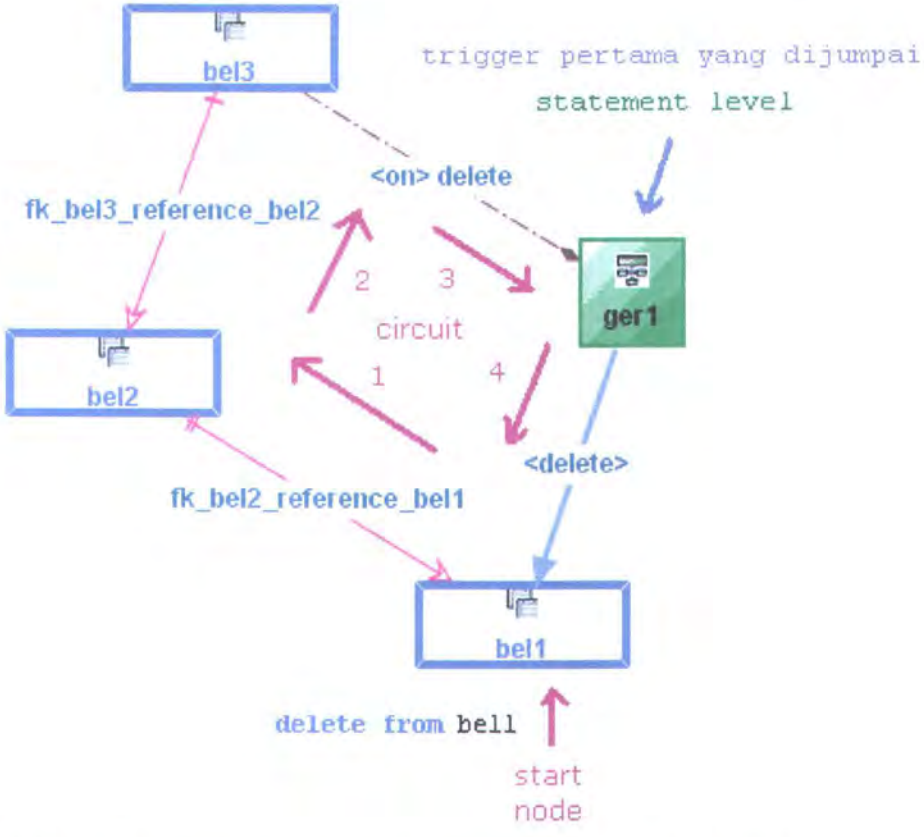
Gambar 3.23 *Direct Acces* (Merah) dan *Indirect Access* (Biru)

- Pengaksesan secara *indirect* akan menghasilkan kasus *mutating table* apabila *vertex trigger* pertama yang dijumpai dalam *circuit* adalah *trigger row level.*



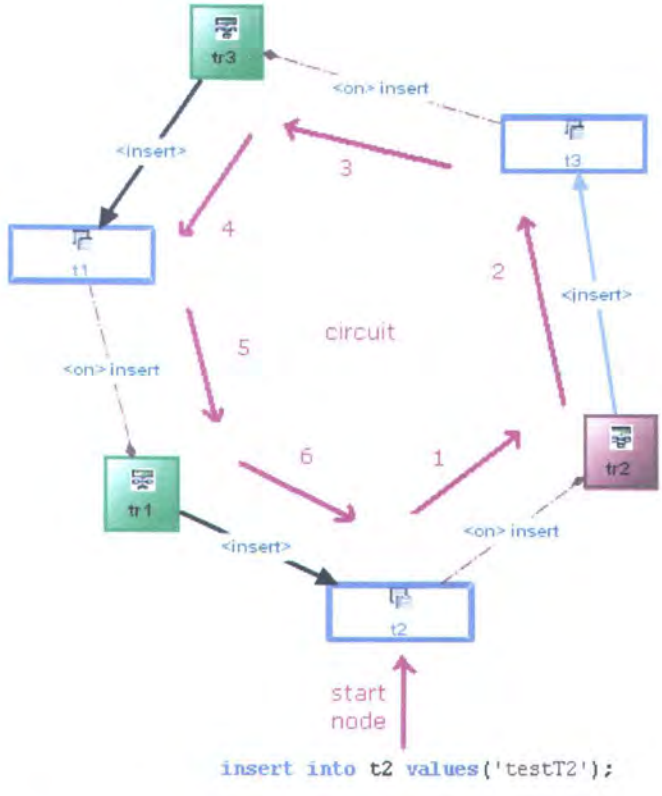
Gambar 3.24 *Trigger* Pertama yang Dijumpai dalam *Indirect Acces* 1

- Dan sebaliknya, *mutating table* tidak akan terjadi apabila pada *path trigger* pertama yang dijumpai dalam *circuit* bukanlah *trigger row level.* Kemungkinan yang terjadi adalah rekursi.

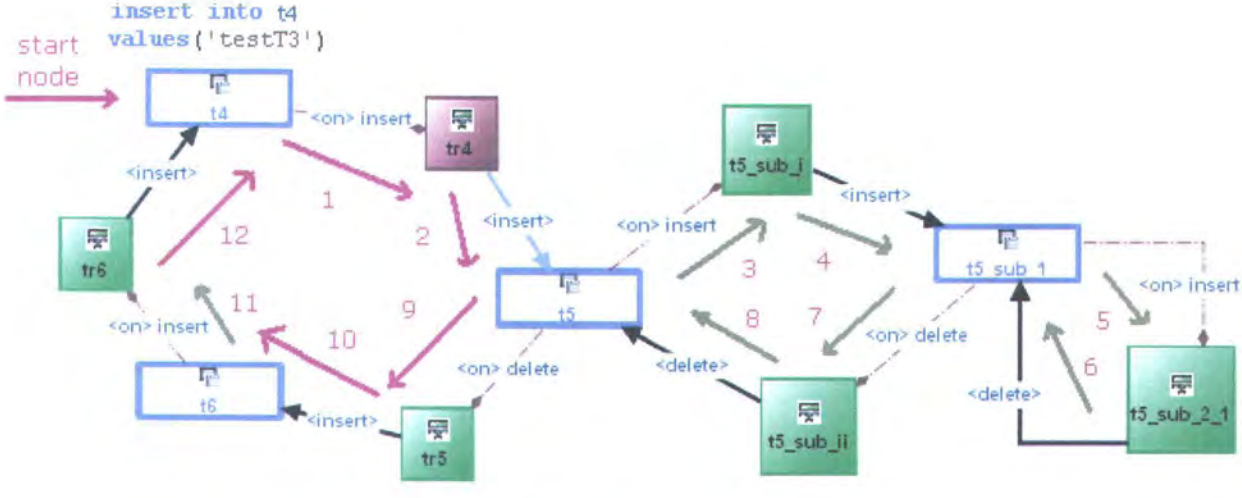


Gambar 3.25 *Trigger* Pertama yang Dijumpai dalam *Indirect Acces* 2

- Pengaksesan secara *indirect* bisa melalui *singlecircuit* ataupun melalui *multicircuit*. Diantara keduanya tidak terdapat perbedaan. Sama-sama bisa menghasilkan kasus *mutating table.*



Gambar 3.26 ***Indirect Acces*** **secara** ***Singlecircuit***



Gambar 3.27 ***Indirect Acces*** **secara** ***Multicircuit***

3.2.4 Analisis dan Perancangan Algoritma Peramalan *Mutating Table*

Dari analisis penyebab *mutating table* pada subbab sebelumnya, bisa diambil dua bagian utama dari algoritma peramalan *mutating table* pada *Oracle*, yaitu:

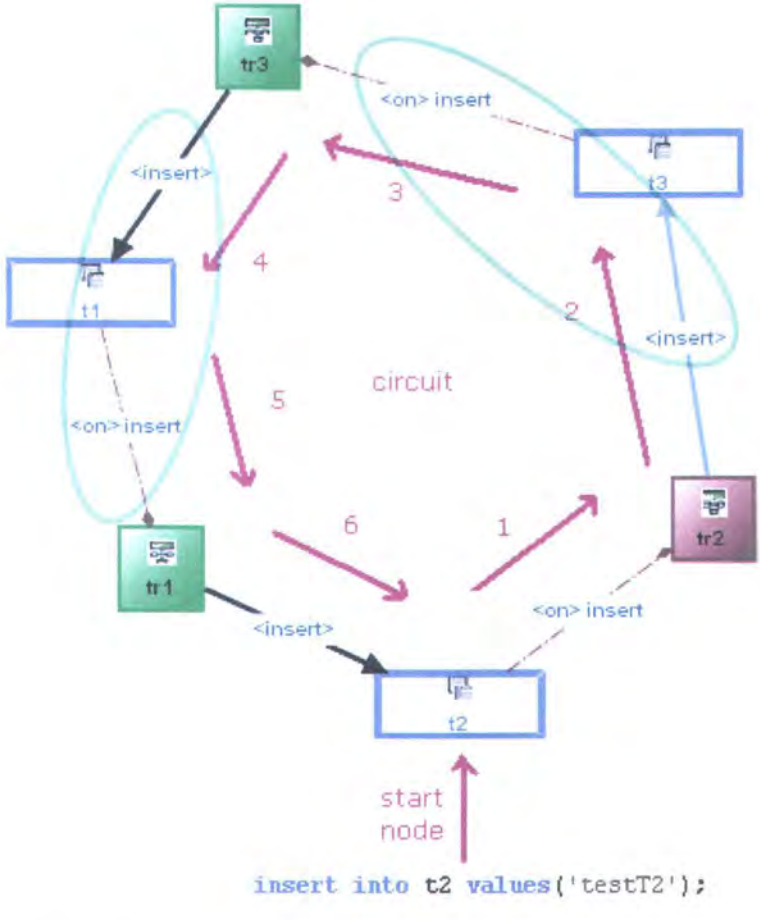
- Algoritma pertama adalah untuk mencari *path circular* itu sendiri. Baik *path circular* yang berupa *singlecircuit* ataupun yang *multicircuit.*
- Algoritma kedua berfungsi untuk memverifikasi ke-*valid*-an *chain reaction.* Bila *valid*, maka *path* tersebut kemungkinan akan menyebabkan *mutating table*.

Adapun pembagian algoritma peramal *mutating table* ini menjadi dua bagian utama adalah bertujuan untuk menyederhakan permasalahan sehingga lebih memudahkan dalam proses implementasi secara *Object Oriented Programming* (*OOP*). Dengan kata lain, proses penyederhanaan dilakukan dengan jalan memisahkan bagian kode/kelas yang mempunyai fungsi umum (contoh: pencari *circuit*) dengan bagian kode/kelas yang berfungsi hanya untuk meramalkan *mutating table* (contoh: verifikasi *chain reaction* pada *path circular*).

3.2.4.1 Algoritma *Chain Reaction Verificator* (*CRV*) pada *Path Circular*

Supaya suatu *path circular* bisa menghasilkan suatu kasus *mutating table*, maka setiap *action trigger* haruslah bersesuaian dengan *event* yang diperlukan untuk men-*trigger trigger* berikutnya. Apabila hal ini tidak dipenuhi, maka *chain reaction* tidak akan terjadi dalam *path circular* tersebut. Tanpa ada *chain reaction*, kasus *mutating table* tentu tidak akan terjadi.

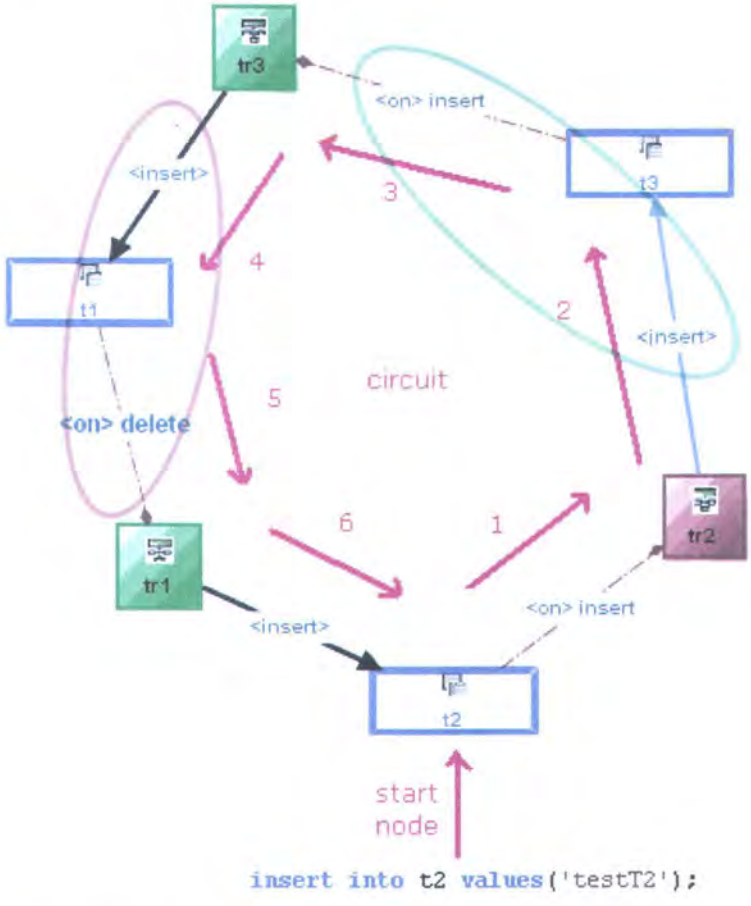
Berikut ini adalah Diagram *Dependency Trigger* dari salah satu contoh kasus *mutating table* yang pernah dibahas pada bab sebelumnya. Guna memperjelas, terdapat tambahan keterangan gambar mengenai kesamaan antara *action trigger* dengan *event* yang dibutuhkan untuk men-*trigger trigger* berikutnya.



Gambar 3.28 Salah Satu Contoh *Circuit* dengan Kasus *Mutating Table*

Pada gambar di atas, kesamaan antara *action trigger* dengan *event* yang dibutuhkan untuk men-*trigger trigger* berikutnya terlihat pada elips berwarna hijau. Karena dalam *singlecircuit* di atas semua *action* dan *event* bersesuaian, maka *chain reaction* terjadi dan kasus *mutating table* muncul pada tabel *t2*.

Berikut ini adalah contoh kasus Diagram *Dependency Trigger* yang di dalamnya terdapat *path circular* berupa *singlecircuit*, namun tidak memunculkan kasus *mutating table*.



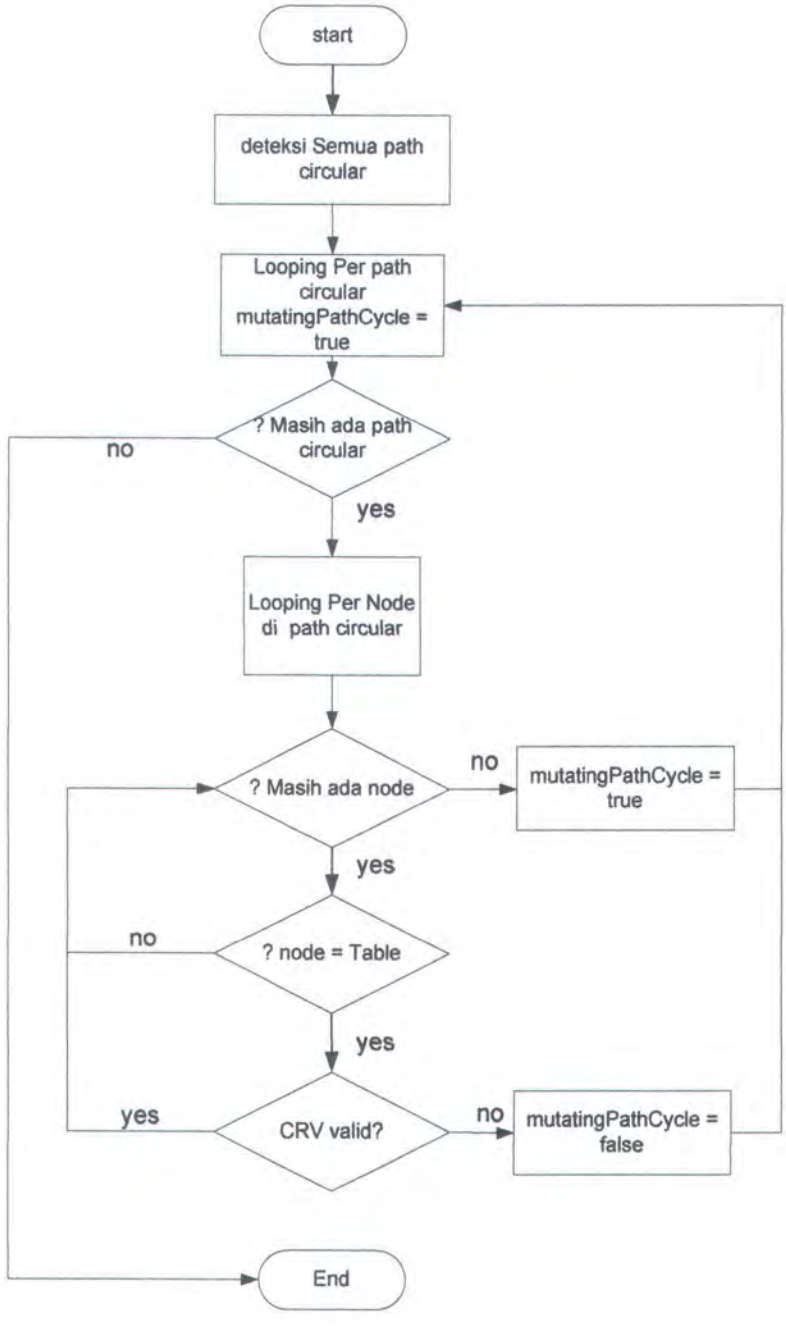
Gambar 3.29 Contoh Kasus *Singlecircuit* Tanpa *Mutating Table*

Alasan mengapa dalam *singlecircuit* ini tidak menghasilkan kasus *mutating table* adalah karena *chain reaction* tidak bisa berlangsung. Hal ini terlihat pada elips warna merah pada diagram di atas. Terdapat perbedaan antara *action trigger* dengan *event* yang dibutuhkan untuk men*trigger trigger* berikutnya. *Trigger tr3* mempunyai *action* berupa *insert* kepada tabel *t1*, sedangkan *tr2* membutuhkan

event jenis *delete* untuk bisa tereksekusi. Karena tidak sama, maka *chain reaction* terhenti sampai pada tabel *t1* dan *mutating table* gagal terjadi.

Untuk mengetahui apakah suatu *circuit* berhasil memunculkan kasus *mutating table* atau tidak, dibutuhkan sebuah algoritma untuk memverifikasinya. Algoritma ini adalah *Chain Reaction Verificator* (selanjutnya disingkat dengan *CRV*). Berikut ini adalah langkah-langkah penggunaan algoritma *CRV* dan *Brute Force* murni untuk meramalkan ke-*mutating*-an suatu tabel:

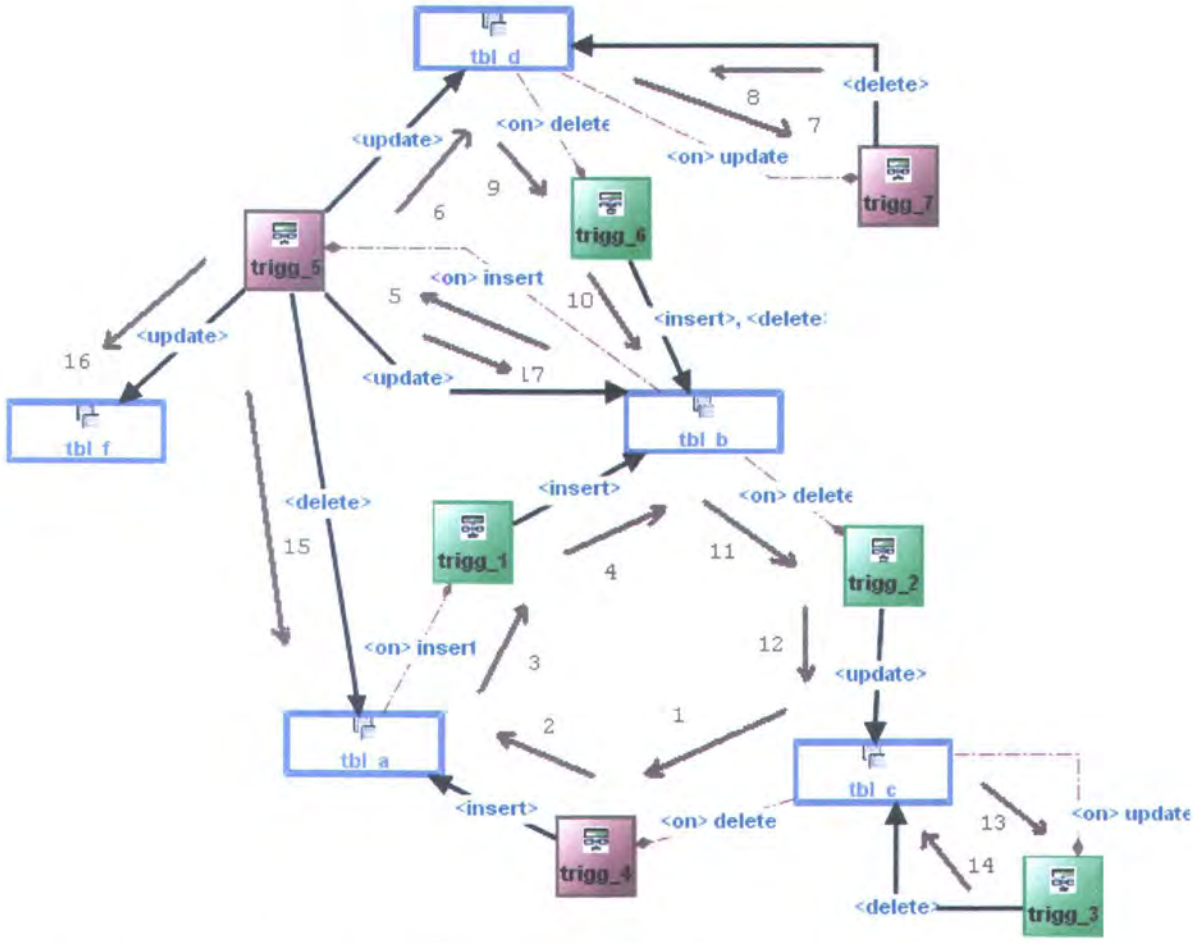
1. Cari semua *path circular* yang dimiliki oleh tabel yang akan dicek ke-*mutating*-annya.
2. Masukkan semua *path circular* tersebut ke dalam *array.* Setiap satu buah *path circular* dalam array tersebut terdiri atas tabel, *trigger* dan *edge.*
3. Cek setiap array *path circular* tadi (misal: *path circular A*).
4. Setiap satu buah *path circular* dicek. Dengan awal mulai dari tabel yang akan dicek ke-*mutating*-annya (misal: tabel *X*).
5. *Path circular* tersebut ditelusuri. Dimulai dari tabel *X* hingga kembali tabel *X.*
6. Bila selama perjalanan menemui *edge* <...> dan <on>... berurutan yang beda isinya 100%, maka *path circular A* gagal meneruskan *chain reaction* (Tabel *X* gagal *mutating*).
7. Bila tidak, maka *path circular A* berhasil meneruskan *chain reaction.* Tabel *X mutating* melalui *path circular A.*
8. Iterasi hingga habis semua *path circular* dalam Array.



Gambar 3.30 *Flowchart* Peramalan *Mutating Table* dengan Algoritma *CRV* dan *Brute Force* Murni

3.2.4.2 Algoritma Pencari Rute Penyebab Kasus *Mutating Table.*

Bagian kedua dari algoritma peramalan *mutating table* adalah bagaimana mencari rute mulai dari tabel yang ter-*trigger* hingga kembali ke tabel tersebut.



Gambar 3.31 Diagram *Dependency Trigger* untuk Studi Kasus

Algoritma yang dibutuhkan adalah algoritma yang bisa mencari semua rute *path* yang mungkin menghasilkan *mutating table*. Sesuai hasil analisis contoh kasus 1 dan 2, diketahui bahwa rute ini bisa merupakan *singlecircuit* ataupun gabungan antara dua/lebih *singlecircuit* (*multicircuit*).

Dari contoh Diagram *Dependency Trigger* di atas, rute *path circular* (yaitu *path* yang dimulai dari *vertex* x dan berakhir di *vertex* x) yang dimiliki oleh tabel *tbl_c* antara lain:

1. 1→2→3→4→11→12 (*singlecircuit path*)
2. 1→2→3→4→5→17 (*multicircuit path*)

3. 1→2→3→4→5→6→7→8→9→10→11→12 (*multicircuit path*)
4. 1→2→3→4→5→15→3→4→11→12 (*irregular cycle path*)
5. 1→2→3→4→11→12→13→14 (*multicircuit path*)
6. 1→2→3→4→5→6→7→8→9→10→11→12→13→14 (*multicircuit path*): ini adalah satu-satunya *path* yang berhasil meneruskan *chain reaction* sehingga menghasilkan kasus *mutating table* pada tabel *tbl_c*.
7. 13→14 (*singlecircuit path*)
8. 13→14→1→2→3→4→11→12 (*multicircuit path*)
9. 13→14→1→2→3→4→5→17 (*multicircuit path*)
10. 13→14→1→2→3→4→5→6→7→8→9→10→11→12 (*multicircuit path*)
11. 13→14→1→2→3→4→5→15→3→4→11→12 (*irregular cycle path*)
12. dan seterusnya

Yang perlu diperhatikan adalah bahwa *path* penghasil *mutating table* tersebut tidak mungkin berupa *irregular cycle path* (yaitu *path circular* di luar jenis *singlecircuit* ataupun *multicircuit).*

3.2.4.2.1 Pendekatan *Brute Force* Murni

Algoritma *Brute Force* murni bisa digunakan untuk mencari semua *path* yang tersebut di atas. Disebut *Brute Force* murni karena algoritma tersebut mencoba-coba semua kemungkinan *path* yang ada. *Brute Force* murni ini bisa dibangun dengan menggunakan algoritma rekursi, dan penelusurannya digunakan metode *Depth First Search (DFS).*

Langkah-langkah penggunaan algoritma *Brute Force* murni dalam hal peramalan kasus *mutating table* pada Diagram *Dependency Trigger* adalah sebagaimana berikut (misal: mengecek ke-*mutating*-an tabel *X*):

1. Cari **semua** *path circular* yang mungkin dengan menggunakan algoritma *Brute Force*.
2. Cek setiap *path circular* yang dihasilkan pada langkah 1 dengan menggunakan algoritma *CRV*.
3. Bila *path circular* yang sedang diverifikasi sepenuhnya *valid*, maka *path circular* tersebut berhasil menghasilkan *mutating table* (*Path* yang sudah *valid* ini selanjutnya disebut sebagai *mutating route).*
4. Bila gagal, maka ulangi iterasi langkah 1-3 pada *path circular* lainnya yang dimiliki oleh tabel *X*.
5. Simpan semua *mutating route* yang ditemukan ke dalam sebuah struktur data.

Adapun kelemahan dari algoritma *Brute Force* murni ini adalah:

- Kompleks dan sulit diimplementasikan. Sebab saat pencarian *path* dengan penelusuran *DFS*, algoritma tersebut harus membuat *flag-flag* guna menandai setiap percabangan. Baik yang berupa percabangan *multicircuit path,* ataupun percabangan yang berupa *irregular cycle path* supaya tidak terjadi *infinite looping*.
- Kurang efektif. Sebab *path* yang secara teoritis tidak mungkin menghasilkan kasus *mutating table* juga ikut dicek oleh algoritma *CRV*. Selain itu semua kemungkinan *path* dicari satu-persatu untuk kemudian ditelusuri satu-persatu

pula dengan algoritma *CRV* tanpa ada kecuali. Contoh: rute *path* 8-10, serta rute *path* yang berupa *irregular cycle path.*

3.2.4.2.2 Optimasi dengan Cara Ekspansi

Algoritma ini dikembangkan untuk menghindari kompleksitas pengimplementasian dan kekurangefektifan algoritma *Brute Force* murni untuk meramalkan kasus *mutating table*. Sama seperti algortima *Brute Force* di atas, Algoritma ini tetap dikembangkan dari proses rekursi dan penelusuran secara *DFS.*

Adapun prinsip utama yang membedakan antara algoritma ini dengan *Brute Force* adalah eksplorasi *DFS* yang dipakai di algoritma ini adalah untuk mencari *singlecircuit* saja, bukan untuk mencari *path circular*. Hal ini akan sedikit mengurangi jumlah kedalaman/*depth* yang harus ditelusuri oleh penelusuran *DFS* bila dibandingkan dengan penelusuran *DFS Brute Force* murni. Sedangkan untuk rute *multicircuit* pada algoritma ini tetap bisa dihasilkan, yaitu melalui proses ekspansi dari *vertex* ke *circuit*. Ekspansi ini dilakukan dengan jalan mensubtitusikan antara sebuah *vertex* dalam *path* milik *singlecircuit* dengan sebuah *singlecircuit* lainnya. Subtitusi ini bisa berlangsung satu kali atau berkali-kali, tergantung kebutuhan. Untuk lebih memudahkan pengacuan, algoritma ekspansi ini selanjutnya disebut sebagai algoritma *VtCX (Vertex to Circuit Expansion)*.

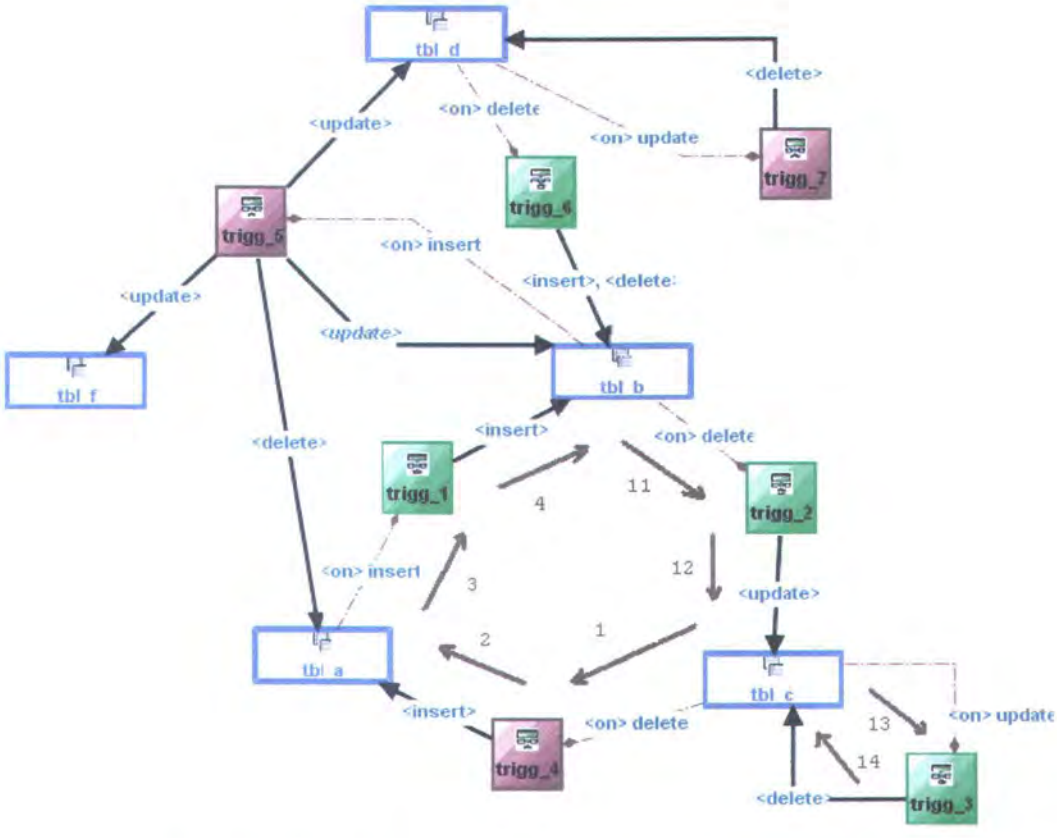
Langkah-langkah penggunaan algoritma *VtCX* dalam hal peramalan kasus *mutating table* pada tabel tertentu (misal tabel *X*) adalah:

1. Cari **semua** *path circular* yang **hanya** berupa *singlecircuit* pada tabel *X*.

2. Cek setiap *singlecircuit* yang dihasilkan pada langkah 1 dengan menggunakan algoritma *CRV*.
3. Bila *singlecircuit* yang sedang dicek sepenuhnya valid, maka *singlecircuit* tersebut berhasil menghasilkan kasus *mutating table*.
4. Bila algoritma *CRV* menemukan adanya kegagalan *chain reaction* pada tabel tertentu (misal tabel *Y*), maka tabel *Y* diekspansi sehingga menghasilkan *singlecircuit* baru dengan *start node* pada tabel *Y*. Gabungan *singlecircuit* tabel *Y* dengan *singlecircuit* tabel *X* menghasilkan *multicircuit*.
5. Cek *circuit* hasil ekspansi tadi dengan algoritma *CRV*.
6. Bila gagal, ulangi langkah 4 dan 5. bila masih gagal coba ekspansi tabel *Y* dengan *singlecircuit* lainnya yang masih dipunyai oleh tabel *Y*.
7. Bila tetap gagal, maka *path circular* tersebut tidak bisa menghasilkan kasus *mutating table* pada tabel *X*.
8. Simpan semua *mutating route* yang ditemukan ke dalam sebuah struktur data.

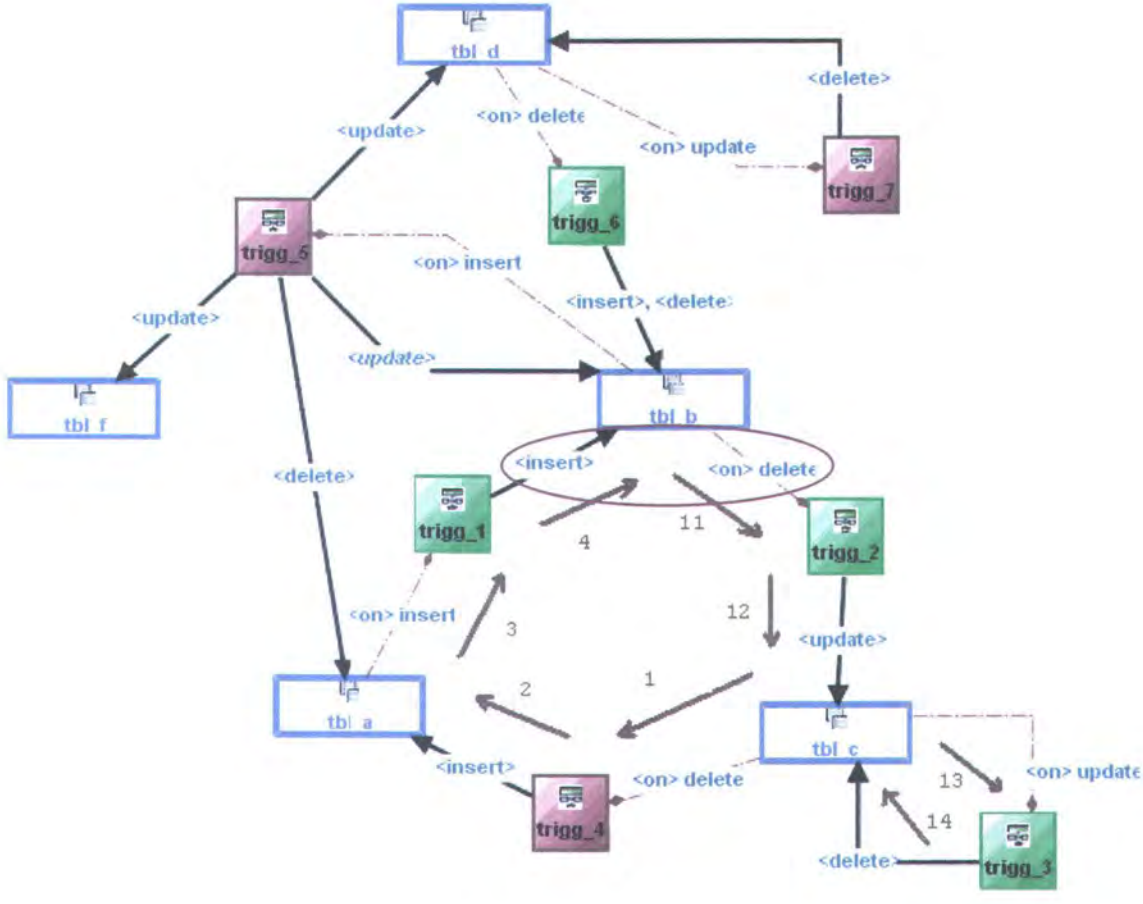
Untuk lebih memperjelas langkah-langkah di atas. Berikut ini adalah sebuah contoh nyata penggunaan algoritma *VtCX* untuk meramalkan ke-*mutating*-an tabel *tbl_c*.

1. Cari semua *path circular* yang hanya berupa *singlecircuit* pada tabel *tbl_c*. Hasilnya *output*-nya adalah *circuit* 1→2→3→4→11→12 dan juga *circuit* 13→14 (*circuit* 13→14 diabaikan karena *trigger statement level* tidak mungkin menghasilkan kasus *mutating table* pada *tbl_c*).



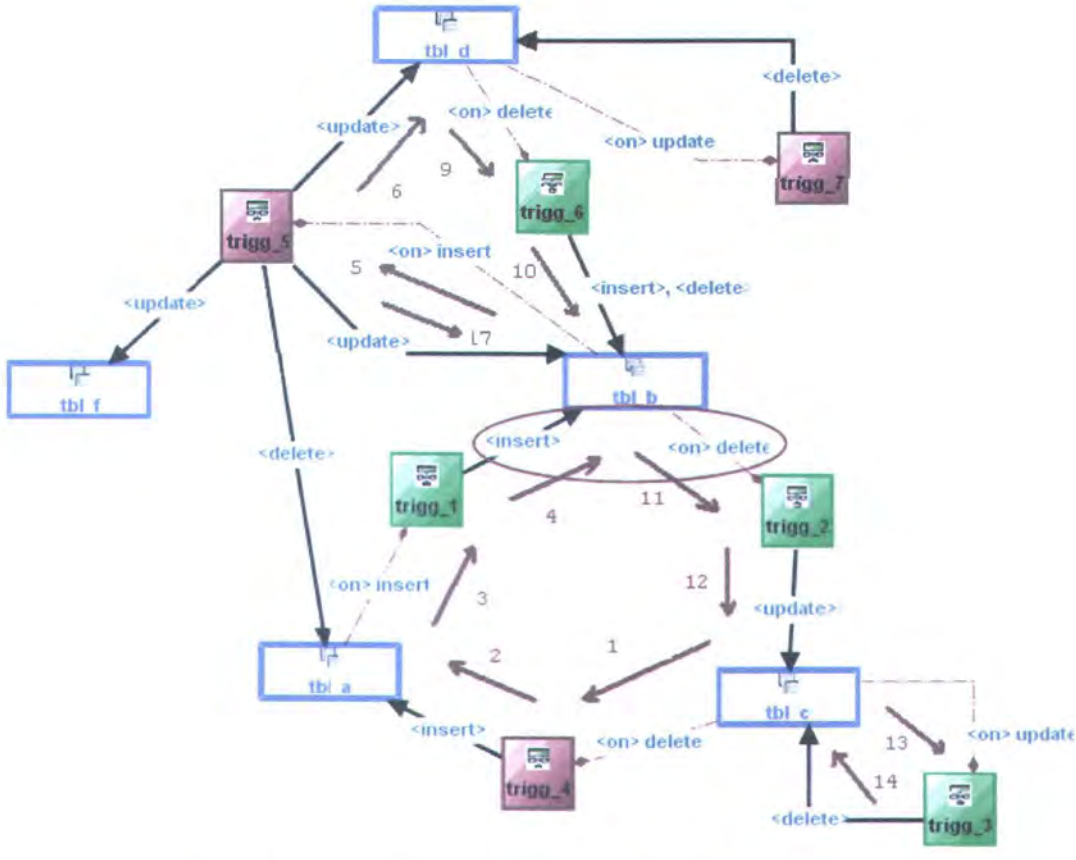
Gambar 3.32 Dua Buah *Singlecircuit* yang Dimiliki oleh *tbl_c*

2. Algoritma *CRV* menemukan adanya kegagalan *chain reaction,* yaitu terjadi pada tabel *tbl_b* (ditunjukkan pada elips berwarna merah. *Action insert* tidak bisa mengeksekusi *trigger* dengan *event delete*).



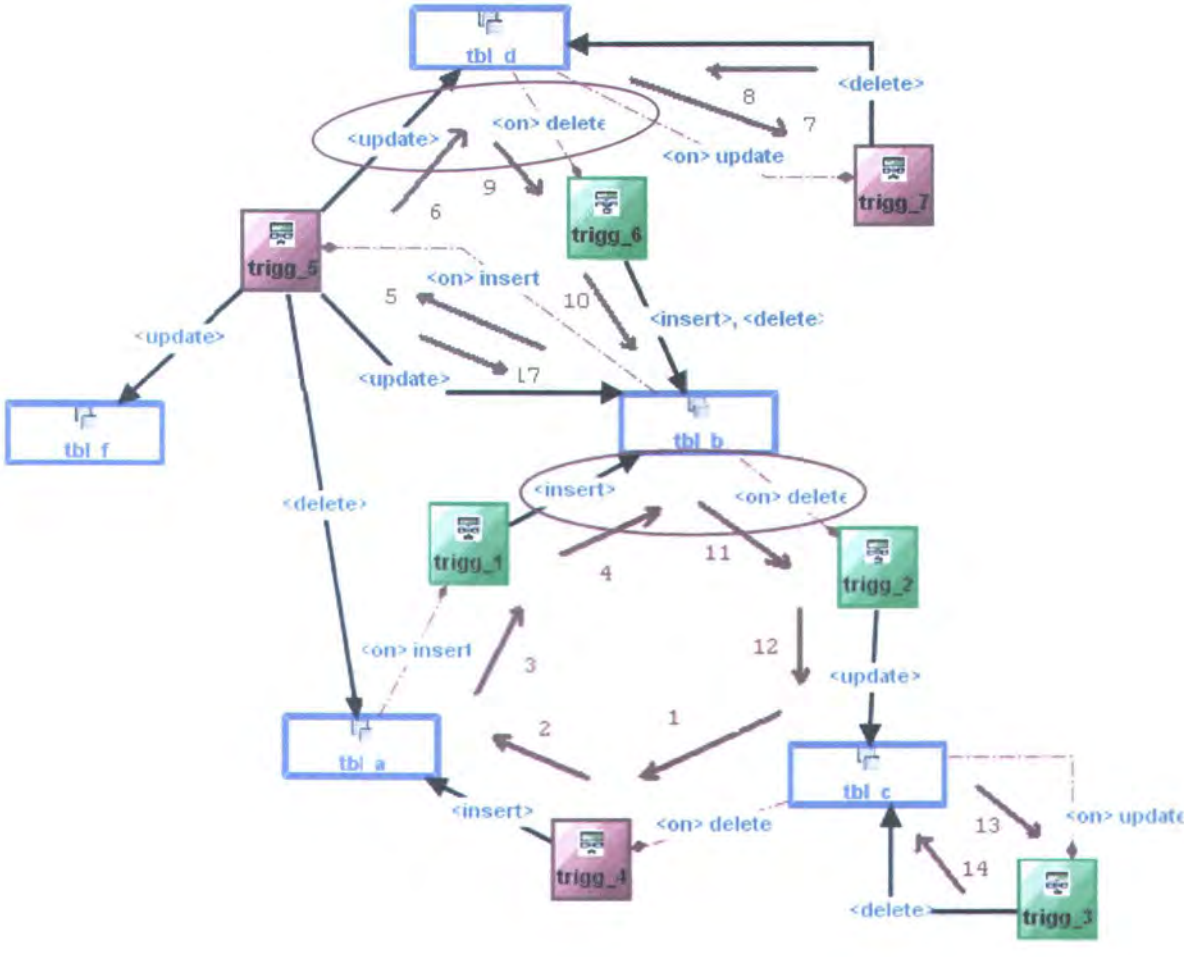
Gambar 3.33 *tbl_b* Gagal Meneruskan *Chain Reaction*

3. Karena tabel *tbl_b* gagal meneruskan *chain reaction*, maka *tbl_b* diekspansi sehingga menghasilkan *singlecircuit* baru dengan *start node* pada tabel *tbl_b*. Adapun *singlecircuit* milik *tbl_b* ini yang *valid* antara lain adalah 5→17 dan 5→6→9→10.



Gambar 3.34 Dua Buah *Singlecircuit* Hasil Ekspansi Milik *tbl_b*

4. Cek *circuit* hasil ekspansi tadi (*singlecircuit* 5→17 dan 5→6→9→10) dengan algoritma *CRV*. Ternyata *circuit* 5→17 gagal (*action update* pada *edge* 17 gagal men-*trigger event delete* pada *edge* 11). *Circuit* ini tidak memungkinkan untuk diekspansi lagi. Sedangkan *circuit* 5→6→9→10 ternyata juga gagal pada tabel *tbl_d*, yaitu *action update* milik *trigg_5* tidak bisa meneruskan *event delete* milik *trigg_6*.
5. Karena *tbl_d* masih memungkinkan untuk diekspansi, maka *tbl_d* diekspansi. *Singlecircuit* hasil ekspansi ini adalah *singlecircuit* 8→7.



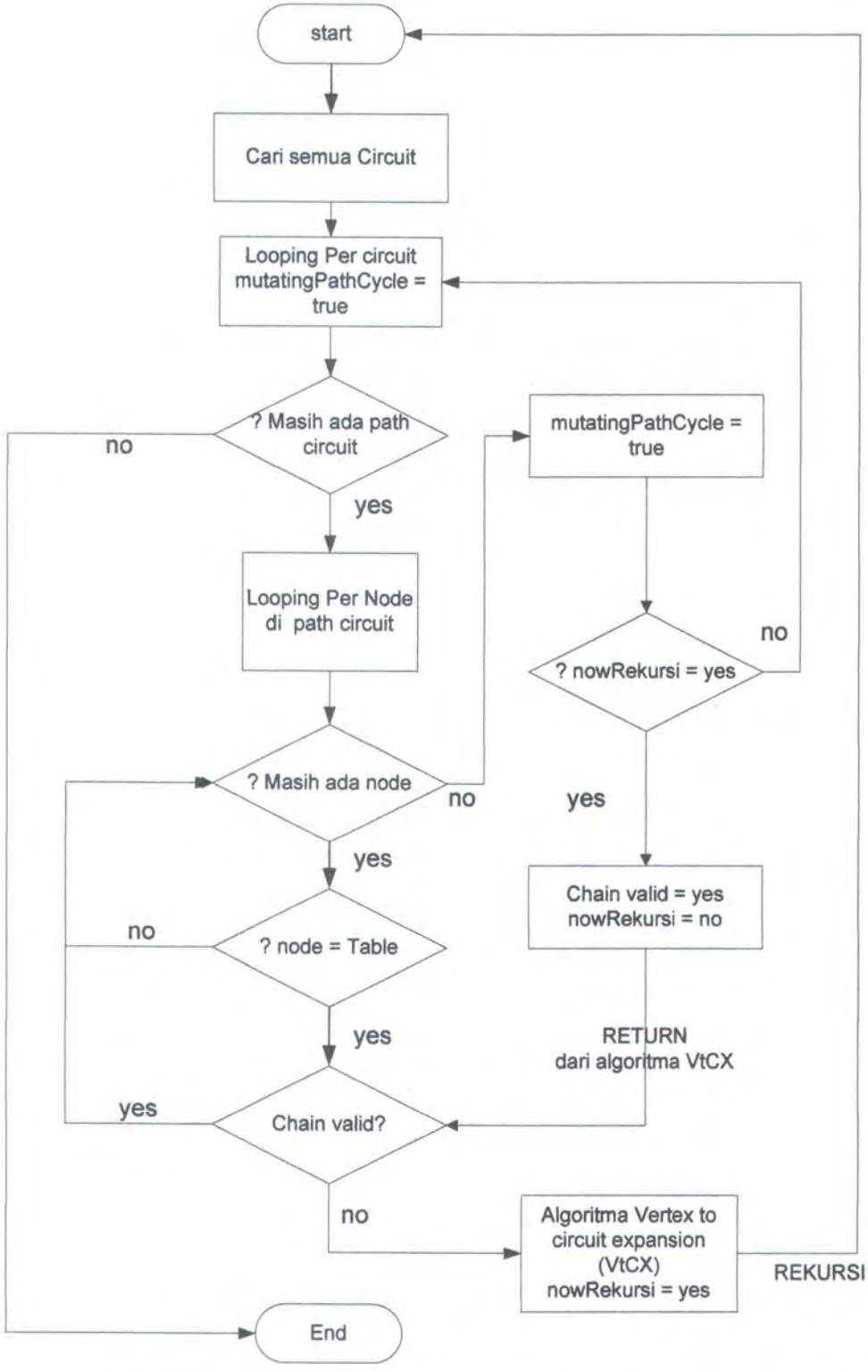
Gambar 3.35 Sebuah *Singlecircuit* Hasil Ekspansi Milik *tbl_d*

6. Hasil ekspansi ini ternyata berhasil meneruskan *event delete* milik *trigg_6.* padahal sebelumnya *event delete* ini mengalami kegagalan apabila diakses langsung oleh *action update* milik *trigg_5.*
7. Untuk selanjutnya, semua rangkaian *action event* berhasil sampai ke tabel awal, yaitu tabel *tbl_c.* Dengan demikian maka *tbl_c* mengalami *mutating.*
8. Adapun *mutating route*-nya adalah 1→2→3→4→ (5→6→ (7→8→) 9→) 10→11→12→13→14. *Path* yang ditulis dalam tanda kurung berarti adalah *path* hasil ekspansi.

Hasil yang didapat dari algoritma ini bila dibandingkan dengan metode *Brute Force* adalah sebagaimana berikut:

- Sederhana. Sebab algoritma rekursi yang hanya bertugas mencari *singlecircuit* saja membutuhkan lebih sedikit *flag* jika dibandingkan dengan algoritma yang mencari semua kemungkinan *path cicular* yang ada. Hal ini akan mempermudah pengimplementasian dalam bentuk *source code*.
- Lebih efektif. Sebab *path* yang dicek oleh algoritma *CRV* dari algoritma *VtCX* ini hanyalah *path* yang mempuyai kemungkinan menghasilkan kasus *mutating table* saja (*singlecircuit* ataupun *multicircuit*). Sedangkan *path* yang tidak mungkin menghasilkan kasus *mutating table* tidak akan ditelusuri oleh algoritma *CRV*. Adapun *Path* yang dicari oleh algoritma ini pada dasarnya hanyalah yang berupa *singlecircuit*. Untuk *path* yang *multicircuit* bisa didapatkan hanya melalui proses ekspansi (tanpa pencarian langsung). Lagipula, proses ekspansi itu sendiri tidak dilakukan setiap waktu. Itu artinya proses ekspansi hanya dilakukan bila perlu saja, yaitu bila algoritma *CRV* menemui adanya kegagalan *chain reaction* pada sebuah *vertex* tertentu, dan *vertex* tempat di mana kegagalan itu terjadi memungkinkan untuk diekspansi. Proses ekspansi ini akan menghemat jumlah kedalaman/*depth* yang akan ditelusuri dalam penelusuran *DFS* dan juga menghemat jumlah *circuit* yang dicek oleh algoritma *CRV*.

Berikut ini adalah *flowchart* dari proses peramalan *mutating table* yang telah dimodifikasi. Dari yang sebelumnya menggunakan *Brute Force*, sekarang menjadi Algoritma *VtCX*:



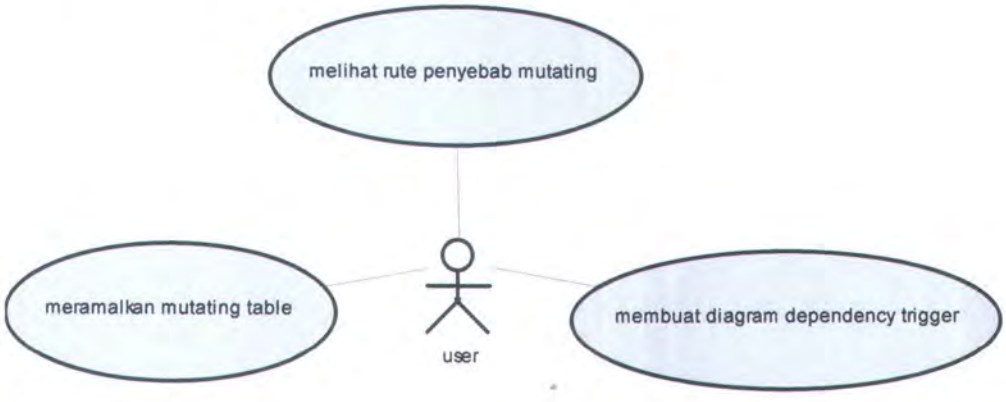
Gambar 3.36 *Flowchart* Peramalan *Mutating Table* dengan Menggunakan Algoritma *CRV* dan *VtCX*

3.3 Perancangan Aplikasi

Proses perancangan ini ditujukan untuk mengetahui bagiamana aplikasi nantinya berjalan dan bagaimana membangun aplikasi.

3.3.1 Perancangan Proses dalam Aplikasi

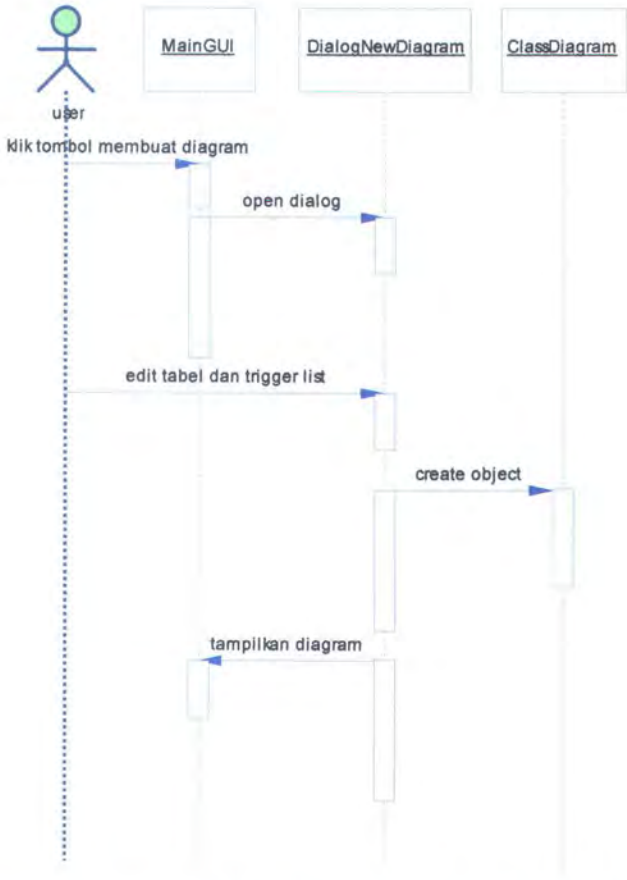
Terdapat beberapa proses utama yang bisa dilakukan oleh user dalam aplikasi ini. Proses utama tersebut digambarkan dengan Diagram *Use Case* sebagaimana berikut:



Gambar 3.37 Diagram *Use Case* Utama dari Aplikasi

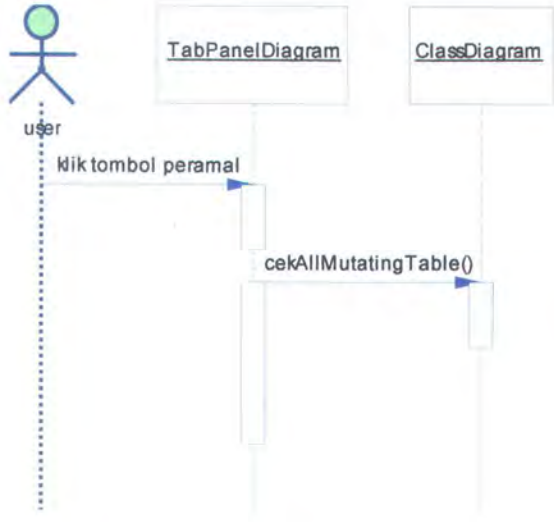
Selanjutnya adalah membuat Diagram Sekuen dari Diagram *Use Case* di atas.

- Diagram Sekuen dari "membuat Diagram *Dependency Trigger*"



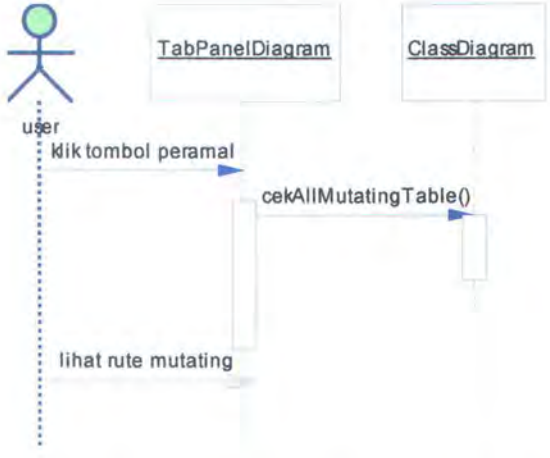
Gambar 3.38 Diagram Sekuen "Membuat Diagram *Dependency Trigger*"

- Diagram Sekuen dari "meramalkan *mutating table*"



Gambar 3.39 Diagram Sekuen "Meramalkan *Mutating Table*"

- Diagram Sekuen dari "melihat rute penyebab kasus *mutating*"



Gambar 3.40 Diagram Sekuen "Meramalkan *Mutating Table*"

3.3.2 Perancangan *Object-Object* yang Ada Dalam Aplikasi

Pada bab ini akan dijelaskan tetang perancangan *object-object* apa saja yang akan ada dalam aplikasi nantinya. *Object-object* yang disebutkan dalam subbab ini hanya mencakup *object* yang bersifat primer saja, *object-object* yang bersifat sekunder tidak disebutkan

3.3.2.1 *Object* Utama Diagram *Dependency Trigger* (`ClassDiagram`)

Ketika diimplementasikan, Diagram *Dependency Trigger* ini nantinya akan direperesentasikan menjadi sebuah kelas tersendiri. *Instance* dari Kelas ini dinamakan *object* `ClassDiagram`.

Untuk menyederhakan kompleksitas dari `ClassDiagram`, maka setiap proses yang bisa berdiri sendiri akan dipecah menjadi *object* tersendiri. *Object-object* ini adalah *object* penyimpan struktur data, *object-object vertex* dan *edge*, *object* pencari *circuit*, serta *object* penampil *graph*.

3.3.2.1.1 *Object* Penyimpan Struktur Data *Graph*

Struktur data yang akan dipakai dalam implementasi Diagram *Dependency Trigger* adalah struktur data *graph*.

Dari pelbagai macam jenis *graph* yang ada dalam teorema *graph*, jenis yang dipakai dalam implementasi Diagram *Dependency Trigger* antara lain:

- *Graph* berarah (*directed graph*). *Graph* jenis ini dipakai karena *edge* relasi, *edge event* dan *edge action* membutuhkan *edge* yang mempunyai arah. Ada *source* dan *target*-nya, tidak bisa dibalik-balik. Oleh karenanya, dibutuhkan *directed graph*.
- *Weighted graph*, yaitu *graph* yang mempunyai "berat" pada setiap edgenya. Berat ini digunakan sebagai *flag* penanda pada *edge*. Sehingga melalui *flag* ini bisa diketahui apakah *edge* itu merupakan *edge event, action*, relasi biasa ataupun relasi *cascade delete*.

Untuk mempermudah proses implementasi, maka Tugas Akhir ini menggunakan sebuah librari struktur data *graph* yang sudah ada. Librari ini bernama `JGraphT` (www.jgrapht.sourceforge.net). Sedangkan kelas yang dipakai dari librari ini adalah `ListenableDirectedWeightedGraph`.

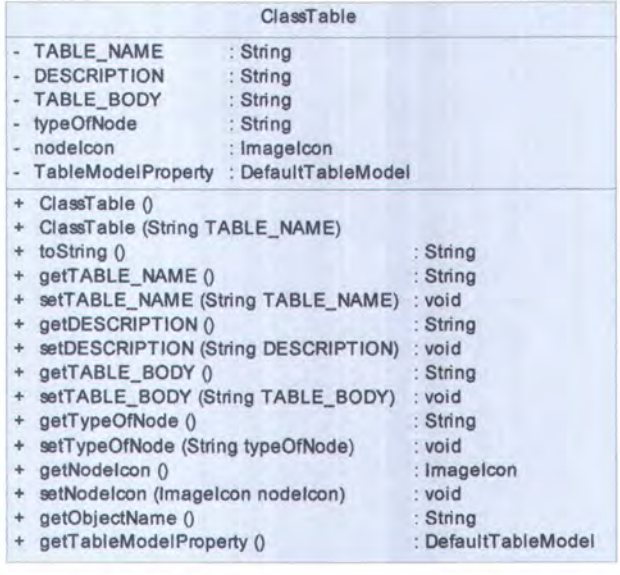
3.3.2.1.2 *Object-Object* Yang Disimpan Dalam Struktur Data *Graph*

Struktur data `ListenableDirectedWeightedGraph` mempunyai kemampuan untuk menyimpan data *vertex* dalam bentuk `java.lang.Object`. Sedangkan data *Edge* disimpan dalam bentuk *object* yang mengimplementasi *interface* `org._3pq.jgrapht.edge.DefaultEdge`.

Adapun jenis-jenis *object* yang akan disimpan dalam kelas `ListenableDirectedWeightedGraph` adalah:

3.3.2.1.2.1 `ClassTable`

Merupakan representasi dari *object table* dalam aplikasi. *Object* ini nantinya akan menyimpan pelbagai *property* dan *method* sebagai berikut ini:.



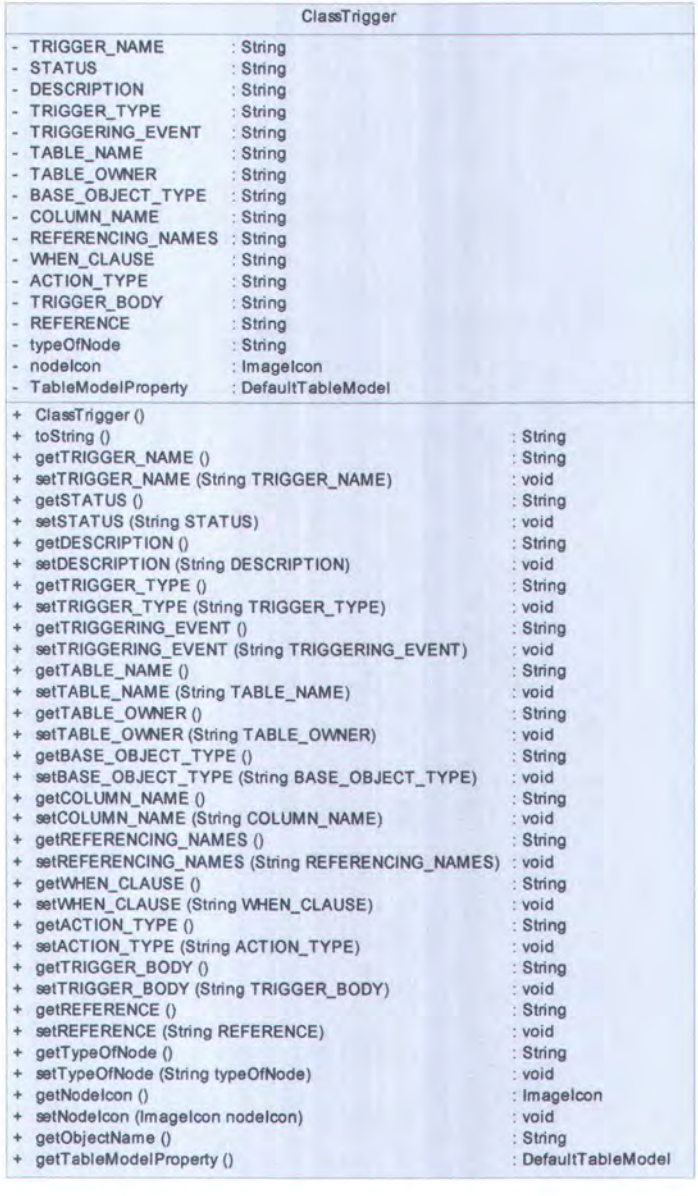
Gambar 3.41 Diagram Kelas untuk `ClassTable`

Field yang dicetak huruf kapital merupakan *field* yang didapatkan dari hasil *query* terhadap tabel `user_tables` di *Oracle*.

Sedangkan `typeOfNode`, `nodeIcon`, dan `TableModelProperty` merupakan *field* yang harus ada sebagai akibat dari implementasi *interface* `ClassNode`. Pengertian tetang *interface* ini akan dijelaskan kemudian.

3.3.2.1.2.2 `ClassTrigger`

Merupakan representasi dari *object trigger* dalam aplikasi. *Object* ini nantinya akan menyimpan pelbagai *property* dan *method* sebagai berikut ini



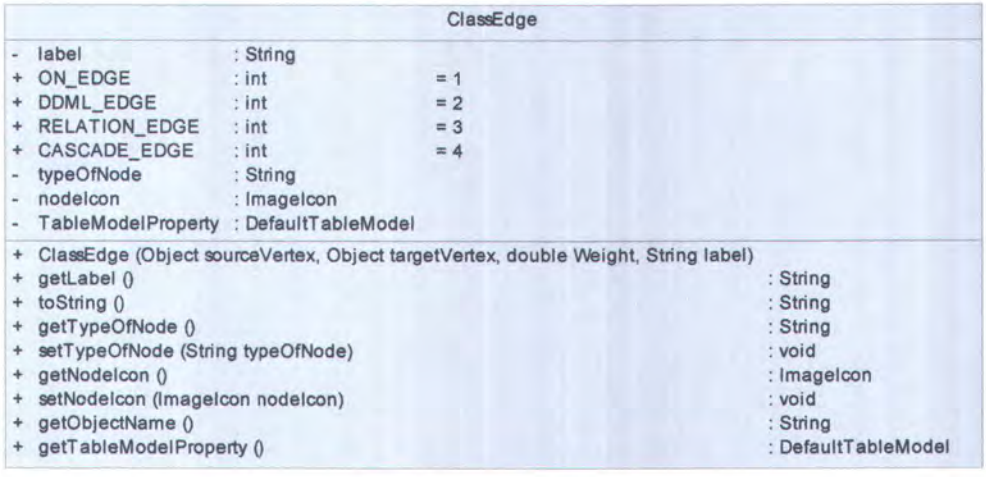
Gambar 3.42 Diagram Kelas untuk `ClassTrigger`

Field yang dicetak huruf kapital (kecuali *field* `REFERENCE`) merupakan *field* yang didapatkan dari hasil *query* terhadap tabel `user_triggers` di *Oracle*.

Field `REFERENCE` merupakan *field* yang menyimpan nama nama tabel yang diacu oleh *trigger* beserta *action*-nya. *Field* ini merupakan *output* dari proses *parsing* terhadap *trigger_body*.

3.3.2.1.2.3 `ClassEdge`

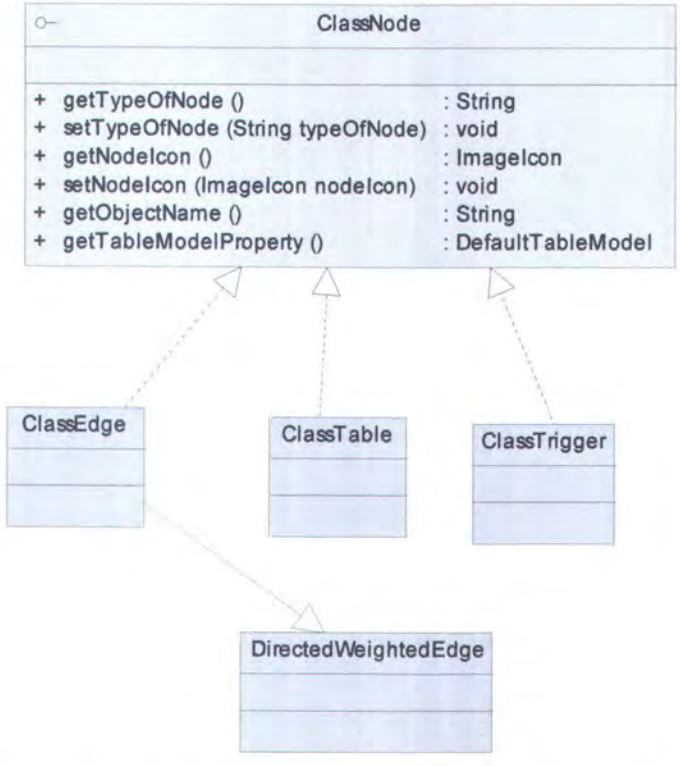
Merupakan representasi dari relasi, *action*, dan *event* di Diagram *Dependency Trigger* dalam *source code*. *Object* ini nantinya akan menyimpan pelbagai *property* dan *method* sebagai berikut:



Gambar 3.43 Diagram Kelas untuk `ClassTrigger`

3.3.2.1.2.4 Diagram penurunan kelas

Ketiga kelas yang akan disimpan dalam struktur data *graph* dan telah disebutkan di atas (yaitu `ClassTable`, `ClassTrigger`, `ClassEdge`) merupakan implementasi dari sebuah *interface, yaitu interface* `ClassNode`. Penyamaan ini bertujuan supaya ketika ketiga *object* yang berbeda di atas masuk ke dalam struktur data *graph*, ketiganya masih bisa dibedakan dengan jalan meng-*casting* ke *interface* `ClassNode` tadi. Berikut ini adalah diagram penurunan kelas untuk ketiga *object* tersebut:



Gambar 3.44 Diagram Penurunan Kelas untuk `ClassTable`, `ClassTrigger`, `ClassEdge`

3.3.2.1.3 *Object* Pencari *Circuit*

Untuk mencari *Circuit* yang terdapat dalam *graph*, dibutuhkan *object* tersendiri untuk melakukan hal ini. Pemisahan ini sengaja dilakukan sebab fungsi pencarian *circuit* bisa berlaku umum, tidak khusus dibutuhkan oleh `ClassDiagram` saja. Kelas dari *object* ini akan dinama `CycleDetectorku`.

Kelas yang mereprentasikan *object* ini merupakan pengembangan dari kelas yang sudah ada dalam librari `JGraphT`. Kelas ini bernama `CycleDetector`. Dengan kata lain `CycleDetectorku` merupakan turunan langsung dari `CycleDetector`. Pengembangan ini dilakukan sebab dalam kelas `CycleDetector`

belum terdapat *method* yang berfungsi untuk mencari *circuit*. Yang ada adalah *method* untuk mendeteksi adanya *cycle* [KAM03] atau tidak.

3.3.2.1.4 *Object* Penampil *Graph*

Object yang terakhir adalah *object* penampil *graph* itu sendiri. Proses ini akan ditangani sebuah librari penampil struktur data *graph*. Librari ini bernama `JGraph` [ALD05].

Object `JGraph` menerapkan prinsip *Model View Controller* (*MVC*) [ALD03]. *Model* yang diperesentasikan oleh *object* ini adalah kelas penyimpan struktur data *graph* yang telah disebutkan sebelumnya. *Object* ini adalah *instance* dari kelas `ListenableDirectedWeightedGraph`.

3.3.2.2 *Method-method* yang Ada dalam `ClassDiagram`

Terdapat dua macam *method* yang bisa dipanggil dari luar *object* untuk melakukan peramalan *mutating table*. *Method* yang pertama adalah digunakan untuk mengecek semua *mutating table* pada diagram, sedangkan *method* yang kedua adalah digunakan untuk mengecek *mutating table* per-tabel yang dipilih oleh *user*. Karena *method* ini akan dipanggil dari luar *object*, maka *modifier* dari *method* ini adalah *public*.

- `cekAllMutatingTable` → *Method* yang dipakai untuk melakukan peramalan ke-*mutating*-an semua tabel yang terdapat dalam Diagram *Dependency Trigger*. *Return value* dari *method* ini adalah `Map` berisi semua *mutating table* yang ditemukan.

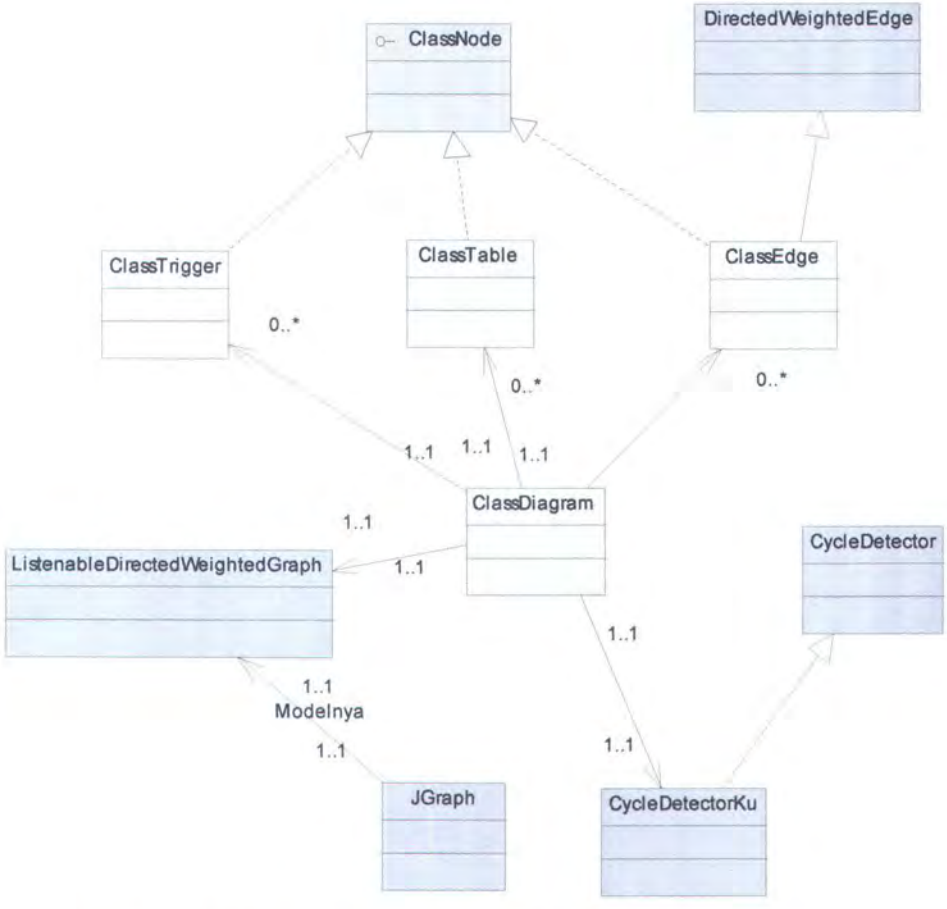
- `cekOneMutatingTable` → *Method* untuk melakukan peramalan *mutating table* pada satu buah tabel saja. Nama tabel yang akan dicek ke-*mutating*-annya dijadikan parameter.

Dua *method* yang bersifat *public* di atas membutuhkan beberapa *method* pendukung yang bersifat *private*. *Method-method* pendukung ini antara lain adalah:

- `cekEdgeInOutPakai`*CRV*`Algorithm` → *Method* implementasi dari algoritma *CRV*. Parameter dari *method* ini adalah dua buah *edge* yang akan diverifikasi (`edgeIn` dan `edgeOut`). *Return value* dari *method* ini adalah *boolean* hasil verifikasi `edgeIn` dan `edgeOut`.
- `cekVertexPakaiVtCXAlgorithm` → *Method* implementasi dari algoritma *VtCX*. Parameter dari *method* ini adalah *vertex* yang akan dicek apakah perlu diekspansi atau tidak, *edge* masuk dan *edge* keluar milik *vertex*, serta rute yang berhasil didapatkan oleh algoritma ini. *Return value*nya adalah *boolean* yang menyatakan apakah tabel ini bisa meneruskan *chain reaction* atau tidak. Sebagai catatan: *method* ini memanggil *method* implementasi dari Algoritma *CRV*.

3.3.2.3 Diagram kelas dari `ClassDiagram`

Kesemua *object-object* yang disebutkan di atas berasosiasi satu dengan yang lainnya seperti tampak pada diagram *UML* berikut ini:



Gambar 3.45 Diagram kelas untuk `ClassDiagram`

3.3.3 Perancangan Proses Utama Pembentukan Diagram *Dependency Trigger*

Aplikasi yang akan dibuat nantinya akan bisa men-*generate* Diagram *Dependency Trigger* yang berasal dari dua macam inputan. Inputan ini antara lain adalah melalui *file* text dan melalui database.

3.3.3.1 Input *File* Teks

File teks yang dipakai dalam inputan ini berisi data-data yang dibutuhkan oleh aplikasi untuk men-*generate* Diagram *Dependency Trigger*. Inputan dari file

teks tidak bisa membuat Diagram *Dependency Trigger* yang di dalamnya terdapat tabel berelasi.

Data yang disimpan dalam file teks itu antara lain adalah:

- Nama-nama *trigger* yang akan di-*generate* Diagram *Dependency Trigger*-nya.
- Nama tabel pemilik dari *trigger-trigger* di atas.
- *Triggering_event* dari setiap *trigger* yang terdapat pada poin sebelumnya.
- *Trigger_type* milik masing-masing *trigger*.
- *Action* dan tabel yang diacu oleh *trigger*.

Adapun tujuan dari disediakannya inputan melalui file teks adalah untuk:

- Melakukan pengujian terhadap aplikasi. Sebab kasus-kasus yang rumit lebih mudah dibuat melalui file teks daripada harus dibuat secara nyata.
- Melakukan perancangan struktur *event action trigger*. Hal ini dibutuhkan oleh para administrator database yang ingin merancang Diagram *Dependency Trigger* mereka. Dengan adanya file teks, maka struktur Diagram *Dependency Trigger* bisa dirancang dan dianalisis terlebih dahulu sebelum diimplementasikan dalam struktur database sesungguhnya.

3.3.3.2 Input dari Database

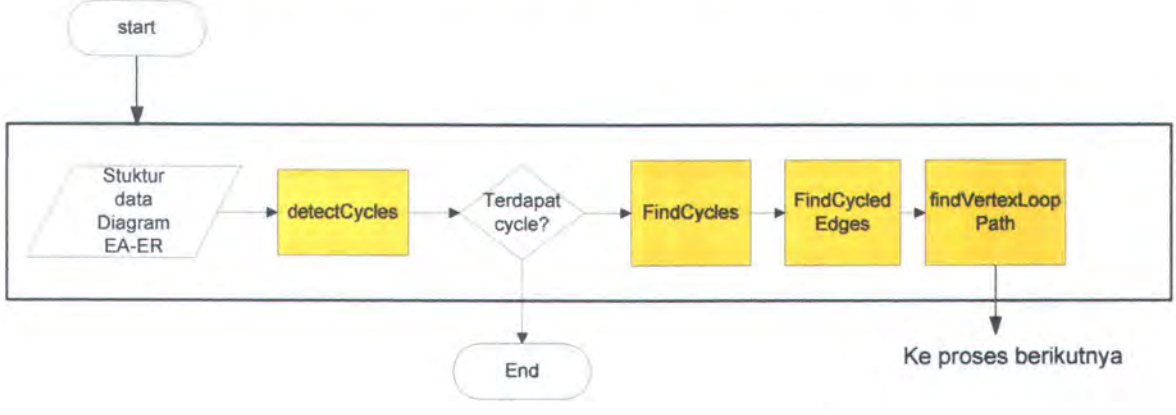
Selain file teks, Diagram *Dependency Trigger* bisa dibuat dengan jalan membaca langsung *script trigger* dan *table* dari database.

Sebelum bisa di-*generate* oleh *constructor* dari `ClassDiagram`, data mentah hasil pembacaan dari database (*trigger body)* harus terlebih dahulu diolah oleh sebuah *parser*. *Parser* ini mengambil informasi-informasi yang ada pada *trigger body* untuk kemudian dijadikan parameter dari *constructor* kelas `ClassDiagram`.

3.3.4 Perancangan Proses Peramalan *Mutating Table*

Seperti yang telah disebutkan pada bab-bab sebelumnya. Proses peramalan *mutating table* itu sendiri sebenarnya terdiri atas pelbagai proses terpisah, yaitu: proses pencari *circuit*, proses algoritma *CRV*, serta proses untuk algoritma *VtCX*. Beberapa bagian dari Proses-proses ini yang tidak berhubungan erat dengan peramalan kasus *mutating table* akan dipecah menjadi *object* tersendiri. Sedangkan proses sisanya akan langsung diintegrasikan dalam `ClassDiagram`. Alasan pemisahan ini bisa dilihat pada keterangan subbab 3.3.2.1.3 dengan judul *Object Pencari Circuit.*

Berikut ini adalah bagian *flowchart* yang lebih mendetil tentang langkah kedua *flowchart* (langkah "*Cari Semua Circuit*") dari proses peramalan *mutating table* dalam `ClassDiagram` yang telah dijelaskan pada sebelumnya (bisa dilihat pada subbab 3.2.5.4 dengan judul *Algoritma Vertex to Circuit Expansion*).



Gambar 3.46 *Flowchart* proses "*Cari Semua Circuit*"

Bagian *Flowchart* yang berwarna kuning merupakan proses yang dilakukan di *object* terpisah dari *object* utama (`ClassDiagram`). Proses tersebut dilakukan di *object* `CycleDetectorku`.

Proses `detectCycle` dan `FindCycles` sudah ada secara *default* dalam librari `JGraphT`. Sedangkan fungsi `FindCycledEdges` dan `findVertexLoopPath` adalah *method* tambahan yang terdapat dalam *child class* `CycleDetectorku`.

Method `findVertexLoopPath` berfungsi untuk mencari *circuit* pada sebuah *vertex* parameter. *Method* ini nantinya akan memanggil *method* lain secara rekursi. *Method* ini adalah `LoopTracerAlgorithm()`.

3.3.5 Perancangan *Output*

Output dari peramalan *mutating table* ini adalah rute-rute penyebab *mutating table*. Supaya lebih informatif, *output* yang ditampilkan haruslah merepresentasikan algoritma yang dipakai. Dari *output* ini haruslah diketahui *path* mana yang merupakan hasil pencarian *circuit* secara langsung, dan *path* mana yang merupakan hasil ekspansi. *Output* ini nantinya akan ditampilkan dalam bentuk *tree*. Bentuk *tree* ini lebih memudahkan *user* untuk memahami maksud *path* yang ditemukan. Contoh: *ouput* hasil analisis pada bab 3.2.4.2.2 adalah 1→2→3→4→(5→6→ (7→8→) 9→) 10→11→12→13→14. *path* tersebut akan ditampilkan dalam bentuk *tree* sebagai berikut ini:

```
Tbl_c:
  |__ Route #1
        |__ TRIGG_4
        |__ TBL_A
        |__ TRIGG_1
        |_+ TBL_B
        |      |__ TRIGG_5
        |      |_+ TBL_D
        |      |      |__ TRIGG_7
        |      |__ TRIGG_6
        |__ TRIGG_2
```

Dari bentuk *tree* seperti di atas akan mudah diketahui bahwa *circuit* tersebut terdiri atas dua buah proses ekspansi. Proses ekspansi ini terjadi pada dua buah *vertex*, yaitu *vertex tbl_b* dan *vertex tbl_d*.

BAB IV

PEMBUATAN APLIKASI

BAB 4

PEMBUATAN APLIKASI

Setelah melalui proses perancangan, dilakukan proses pembuatan aplikasi yang merupakan implementasi dari tahap perancancangan *object,* input, proses dan *output* pada bab 3.

4.1 Lingkungan Aplikasi

Seperti yang disebutkan dalam bab pendahuluan, aplikasi ini diimplementasikan dengan menggunakan sebuah *Integrated Development Environment/IDE opensource* yang *freeware* milik *sun microsystem*, yaitu *NetBeans 4.1.*

Komponen-komponen tambahan dari pihak *third-party* yang digunakan selama proses implementasi ini antara lain adalah librari `JGraph`, `JGraphT`, serta *driver JDBC* dari *Oracle.*

4.2 Implementasi *Object-object* Utama

Berikut ini akan dijelaskan hasil implementasi dari *object-object* utama dalam aplikasi:

4.2.1 *Object-object Vertex* dan *Edge.*

Object `ClassTable`, `ClassTrigger`, dan `ClassEdge` nantinya akan dimasukkan ke dalam *object* untama (`ClassDiagram`). Implementasi dari masing-masing *object* ini adalah sebagai berikut:

4.2.1.1 *interface* `ClassNode`.

Penjelasan mengenai *interface* ini bisa dilihat pada subbab 3.3.2.1.2.4. Berikut ini adalah *source code* untuk *interface* `ClassNode`:

```
21  public interface ClassNode {
22
23 ⊞     /**...*/
27      public String getTypeOfNode();
28
29 ⊞     /**...*/
33      public void setTypeOfNode(String typeOfNode);
34
35 ⊞     /**...*/
39      public ImageIcon getNodeIcon();
40
41 ⊞     /**...*/
45      public void setNodeIcon(ImageIcon nodeIcon);
46
47 ⊞     /**...*/
50      public String getObjectName();
51
52      public DefaultTableModel getTableModelProperty();
53  }
```

Dengan memanggil *method* `getTypeOfNode()` (baris 27), maka *class* yang mengimplementasikan *interface* ini bisa diketahui tipenya. Apakah itu tipe `TABLE`, `TRIGGER`, ataupun `EDGE`.

Sedangkan *method* `getObjectName()` pada baris 50 akan me-*return* nama dari *object* yang mengimplementasikan *interface* ini. Misalnya untuk *object* dengan tipe `ClassTable` akan me-*return*-kan *field* `TABLE_NAME` bila *method* ini dipanggil.

4.2.1.2 *Object-object* yang Mengimplementasikan *Interface* `ClassNode`.

Semua *object* representasi dari *object* database mengimplementasikan *interface* `ClassNode`. Implementasi dari *object-object* representasi *object* database

tersebut pada dasarnya hanya terdiri atas *field* serta *getter* dan *setter*. Hal ini berlaku pada `ClassTable`, `ClassTrigger` dan `ClassEdge`.

Berikut ini adalah pelbagai macam konstruktor dari *object* yang mengimplementasi `ClassNode`:

Konstruktor `ClassTable` adalah sebagai berikut:

```
23     // <editor-fold defaultstate="collapsed" desc=" CONSTRUCTOR ">
24     /**...*/
27     public ClassTable() {
28         setNodeIcon(new javax.swing.ImageIcon(getClass().
29                 getResource("/img/dbindex2.GIF")));
30         setTypeOfNode("TABLE");
31     }
32
33     /**...*/
36     public ClassTable(String TABLE_NAME) {
37         setNodeIcon(new javax.swing.ImageIcon(getClass().
38                 getResource("/img/dbindex2.GIF")));
39         this.TableModelProperty
40                 = new DefaultTableModel();
41         this.TableModelProperty.addColumn("key");
42         this.TableModelProperty.addColumn("value");
43
44         this.TABLE_NAME = TABLE_NAME;
45         this.TableModelProperty.
46                 addRow(new Object[]{"TABLE_NAME",TABLE_NAME});
47         setTypeOfNode("TABLE");
48     }
```

Konstruktor `ClassTrigger` adalah sebagai berikut:

```
28      public ClassTrigger() {
29          setNodeIcon(new javax.swing.ImageIcon(getClass()
30                  getResource("/img/LOGGOFF.PNG")));
31          setTypeOfNode("TRIGGER");
32      }
33
34      /**...*/
37      public ClassTrigger(String TRIGGER_NAME,
38                          String TRIGGER_TYPE,
39                          String TRIGGERING_EVENT,
40                          String TABLE_OWNER,
41                          String BASE_OBJECT_TYPE,
42                          String TABLE_NAME,
43                          String COLUMN_NAME,
44                          String REFERENCING_NAMES,
45                          String WHEN_CLAUSE,
46                          String STATUS,
47                          String DESCRIPTION,
48                          String ACTION_TYPE,
49                          String TRIGGER_BODY) {...}
82
83      // </editor-fold>
```

Konstruktor `ClassEdge` adalah sebagai berikut:

```
31      /** Creates a new instance of ClassEdge */
32      public ClassEdge(Object sourceVertex,
33              Object targetVertex,
34              double Weight,
35              String label) {
36          super(sourceVertex,targetVertex,Weight);
37          this.label
38                  = new String(label);
39          this.TableModelProperty
40                  = new DefaultTableModel();
41          this.TableModelProperty.addColumn("key");
42          this.TableModelProperty.addColumn("value");
43
44          this.TableModelProperty.
45                  addRow(new Object[]{"LABEL",this.label});
46
47          setTypeOfNode("EDGE");
48      }
```

4.2.2 *Object* `ClassDiagram`.

Seperti yang dijelaskan pada bab 3. `ClassDiagram` merupakan kelas utama dalam aplikasi ini. Di dalamnya terdapat pelbagai macam *method* dan *field* yang berhubungan dengan pembuatan Diagram *Dependency Trigger*.

4.2.2.1 *Field-field* Utama dari `ClassDiagram`.

Berikut ini adalah *field-field* utama yang dipakai dalam `ClassDiagram`:

```
1254    protected ListenableDirectedWeightedGraph graphUtama
1255            = new ListenableDirectedWeightedGraph();
1256    protected DirectedWeightedSubgraph cycledSubgraph;
1257    protected JGraphModelAdapter JGraphModel
1258            = new JGraphModelAdapter(graphUtama);
1259    protected JGraph jgraph
1260            = new JGraph( JGraphModel );  //JGraph Visualitation
```

Struktur data utama *graph* disimpan dalam *field* `graphUtama`. *Field* ini bertipe `ListenableDirectedWeightedGraph`. *Class* ini terdapat dalam librari `JGraphT`. `ListenableDirectedWeightedGraph` bersifat *listenable*, artinya perubahan-perubahan yang terjadi dalam struktur *graph* akan menimbulkan *event-event* yang bisa ditangkap. Hal ini penting ketika `JGraphT` diintegrasikan dengan `JGraph`. Integrasi ini dijembatani oleh *Object* `JGraphModelAdapter`.

Object `cycledSubgraph` (baris 1256) merupakan *subgraph* dari `graphUtama`. `cycledSubgraph` ini dibuat ketika dalam `graphUtama` ditemukan adanya *cycle*. Untuk memudahkan analisis dan algoritma, *graph* yang terlibat *cycle* ini dipisahkan dari `graphUtama`. Oleh karena itulah *object* ini diperlukan.

4.2.2.2 Implementasi dari Algoritma *CRV*.

Inti dari *method* ini adalah mengecek apakah *edge* masuk bisa men-*trigger edge* keluar untuk tabel tertentu. Pengecekan ini dilakukan dengan jalan membandingkan `ClassEdge` berdasarkan *weight* dan kesamaan *label*-nya. Untuk lebih jelasnya. Berikut ini adalah *source code* untuk *method* implementasi dari algoritma *CRV*. Parameter dari *method* ini adalah dua buah `ClassEdge` yang akan diverifikasi.

Terdapat empat macam kondisi dalam *method* ini, yaitu:

1. Kondisi ketika `ClassEdge` yang dibedakan sama-sama berupa *event* dan *action* (baris 136-153).
2. Kondisi bila `ClassEdge edgeIn` berupa relasi *cascade delete* dan `ClassEdge edgeOut` berupa *event* (baris 155-161).
3. Kodisi bila `ClassEdge edgeIn` berupa *action* dan `ClassEdge edgeOut` berupa relasi *cascade delete* (baris 163-169).
4. Terakhir bila `ClassEdge edgeIn` dan `edgeOut` berupa relasi *cascade delete* (baris 171-173).

Source code lengkap dari *method* implementasi dari algoritma *CRV* adalah:

```
133     private boolean cekEdgeInOutPakaiCRVAlgorithm (ClassEdge edgeIn,
134             ClassEdge edgeOut){
135         //kombinasi trigger-trigger
136         if(edgeIn.getWeight()<3&&edgeOut.getWeight()<3){
137             //<select>,<update>,<delete>,<insert>
138             String stringIn = edgeIn.getLabel();
139             String stringOut =
140                     ((ClassTrigger) edgeOut.
141                     getTarget()).
142                     getTRIGGERING_EVENT(); //update or delete or insert
143
144             String triggered;
145             StringTokenizer tk = new StringTokenizer(stringOut," ");
146             while(tk.hasMoreElements()){
147                 triggered = tk.nextToken();
148                 if(!triggered.equals("or")){
149                     if(stringIn.contains(triggered))
150                         return true;
151                 }
152             }
153         }
154         //apakah cascade bisa mentrigger on delete
155         else if(edgeIn.getWeight()==4&&edgeOut.getWeight()<3){
156             if(edgeOut.getLabel().
157                     toLowerCase().
158                     contains("delete")){
159                 return true;
160             }
161         }
162         //apakah action delete bisa mentrigger relasi cascade
163         else if(edgeIn.getWeight()<3&&edgeOut.getWeight()==4){
164             if(edgeIn.getLabel().
165                     toLowerCase().
166                     contains("delete")){
167                 return true;
168             }
169         }
170         //sama-sama cascade delete. OK deh
171         else if(edgeIn.getWeight()==4&&edgeOut.getWeight()==4){
172             return true;
173         }
174         return false;
175     }
```

4.2.2.3 Implementasi dari Algoritma *VtCX*.

Setiap `ClassTable` dalam *circuit*, dari tabel yang akan dicek ke-*mutating*-annya hingga akhir *path* dalam *circuit*, harus melewati *method* ini untuk bisa ditentukan apakah `ClassTable` tadi bisa meneruskan *chain reaction* atau tidak. Baik itu secara langsung ataupun harus melalui proses ekspansi. *Method* pengecek *vertex* ini bernama `cekVertexPakaiVtCXAlgorithm()`.

Parameter dari *method* ini bisa dilihat pada *source code* berikut:

```
183     private boolean cekVertexPakaiVtCXAlgorithm(ClassTable tableYangDicek,
184                                                 ClassEdge edgeMasuk,
185                                                 ClassEdge edgeKeluar,
186                                                 DefaultMutableTreeNode pathMutating)
```

- `tableYangDicek` merupakan *vertex* yang akan dicek oleh algoritma *VtCX*.
- `edgeMasuk` dan `edgeKeluar` adalah dua buah *edge* yang akan diverifikasi oleh `cekEdgeInOutPakai`*CRV*`Algorithm()` dalam *method* ini. Apabila `cekEdgeInOutPakai`*CRV*`Algorithm()` me*return false*, maka ekspansi akan dilakukan.
- `pathMutating`. Adalah rute yang nantinya akan diisi secara otomatis oleh *method* ini.

Langkah detil dari `cekVertexPakaiVtCXAlgorithm()` adalah mula-mula *method* ini akan memanggil *method* `cekEdgeInOutPakai`*CRV*`Algorithm()` dengan parameter `edgeIn` dan `edgeOut`. Selanjutnya, proses ekspansi akan dilakukan apabila *method* `cekEdgeInOutPakaiCRVAlgorithm()` tadi me*return* nilai *false*.

Ekspansi dilakukan dengan cara mencari semua *circuit* baru untuk *vertex* yang sedang dicek dengan menggunakan *method* `findAllVertexLoopPath()`.

Method tersebut terdapat di *object* `CycleDetectorku`. *Circuit* hasil ekspansi tadi akan dicek satu-persatu lagi oleh *method* `cekVertexPakaiVtCXAlgorithm()`. Pemanggilan *method* ini adalah pemanggilan secara rekursi.

4.2.3 *Object* `CycleDetectorku`.

Adalah *object* yang digunakan untuk mencari *singlecircuit* pada *vertex* terntentu. Penjelasan mengenai *object* ini bisa dilihat pada bab 3.3.2.1.3. Berikut ini potongan kode untuk konstruktor dari `CycleDetectorku`.

```
46      public CycleDetectorku(DirectedGraph graph) {
47          super(graph);
48          this.graphku = graph;
49      }
```

`DirectedGraph` yang dijadikan parameter *object* ini adalah *graph* dimana *cycle* akan dideteksi dan dicari *path*-nya.

Implementasi dari *method* pencari *circuit* adalah sebagai berikut:

```
267     public LinkedList findVertexLoopPath(Object vertexToSearch,
268                                          Graph graphYangDicariLoopnya){
269         LinkedList retur = new LinkedList();
270         LoopTracerAlgorithm(vertexToSearch,
271                             vertexToSearch,
272                             graphYangDicariLoopnya,
273                             new LinkedHashSet(30),
274                             retur,
275                             new LinkedHashSet(30));
276         return retur;
277     }
```

Method di atas memanggil *method* `LoopTracerAlgorithm()`.

```
163 private int LoopTracerAlgorithm(Object StartVertex,
164                                Object VertexNow,
165                                Graph subGraphnya,
166                                LinkedHashSet VertexSequence,
167                                LinkedList loopPathFixedForThisStartVertex,
168                                LinkedHashSet VertexHistoryThisVertex){
```

Method ini adalah *method* dengan *modifier* `private` yang mempunyai sifat rekursi untuk mencari semua *circuit* berdasarkan prinsip *DFS*.

Keterangan parameter:

1. `StartVertex` merupakan *vertex* penanda *start* dari *circuit* yang akan dicari.
2. `VertexNow` adalah *path* sekarang.
3. `subGraphnya` adalah *graph* dimana pencarian *circuit* berlangsung.
4. `VertexSequence` adalah *path* sementara yang telah berhasil ditemukan. Struktur data yang dipakai adalah `LinkedHashSet` supaya tidak ada nilai yang kembar.
5. `loopPathFixedForThisStartVertex` merupakan `LinkedList` penyimpan tetap dari *circuit* yang telah berhasil ditemukan.
6. `VertexHistoryThisVertex` adalah kumpulan *vertex* yang pernah ditelusuri. *Variable* ini diperlukan supaya tidak terjadi redundansi pengecekan.

Langkah-langkah dari proses pencarian *circuit*:

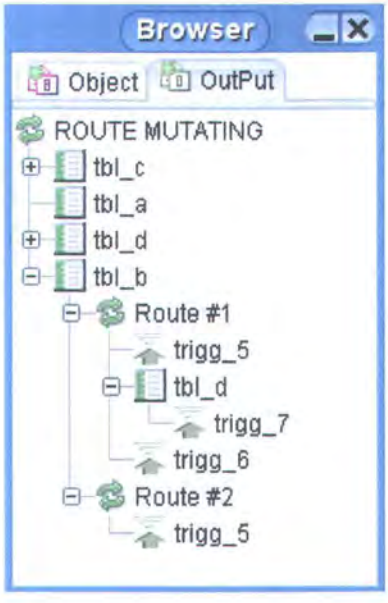
1. Interasi satu-persatu *vertex* yang ada dalam *graph*. Dicari apakah ada *edge* penghubung dengan *vertex* atau tidak.
2. Apabila ditemukan, cek apakah hubungan ini sudah ada dalam `VertexHistoryThisVertex` atau belum.
3. Apabila belum, cek apakah *vertex* tersebut merupakan akhir dari *circuit* atau tidak.
4. Bila iya, maka tambahkan *circuit* yang telah ditemukan ke dalam `loopPathFixedForThisStartVertex`.

5. Bila tidak, maka rekursi lagi dengan mengubah parameter `VertexNow` menjadi nilai *vertex* yang sedang dalam proses iterasi saat itu.

```
163 private int LoopTracerAlgorithm(Object StartVertex,
164                                 Object VertexNow,
165                                 Graph subGraphnya,
166                                 LinkedHashSet VertexSequence,
167                                 LinkedList loopPathFixedForThisStartVertex,
168                                 LinkedHashSet VertexHistoryThisVertex){
169     HashSet temp = new HashSet(subGraphnya.vertexSet());
170     temp.remove(VertexNow);                 //dihilangkan dirinya sendiri b
171     Object[] allVertex = temp.toArray();    //dipindah ke Object saja biar
172     int totalVertex = allVertex.length;
173     for(int i=0;i<=totalVertex-1;i++){      //looping ini adalah untuk meng
174         if(subGraphnya.containsEdge(VertexNow,allVertex[i])){       //diexp
175             ClassEdge EdgeNow
176                     = (ClassEdge) subGraphnya.
177                         getEdge(VertexNow, allVertex[i]);
178             if(!EdgeNow.getTarget().equals(StartVertex)){       //artinya
179                 if(!VertexHistoryThisVertex.contains(allVertex[i])){
180                     VertexSequence.add(VertexNow);                  //ditamb
181                     VertexHistoryThisVertex.add(VertexNow);         //dan ke
182                     Object NewVertexNow =  new Object();        //dinew biar
183                     NewVertexNow =  allVertex[i];       //dimulai rekursi l
184                     LinkedHashSet NewVertexHistoryThisVertex
185                             = new LinkedHashSet(VertexHistoryThisVertex);
186                     LoopTracerAlgorithm(StartVertex,
187                                         NewVertexNow,
188                                         subGraphnya,
189                                         new LinkedHashSet(VertexSequence),
190                                         loopPathFixedForThisStartVertex,
191                                         NewVertexHistoryThisVertex);
192                 } else
193                     VertexSequence.remove(VertexNow);
194             } else {
195                 VertexSequence.add(VertexNow);                  //ditambahka
196                 LinkedList temp2 = new LinkedList(VertexSequence);
197                 temp2.add(StartVertex);
198                 loopPathFixedForThisStartVertex.add(temp2);             //
199                 VertexSequence.remove(VertexNow);
200                 VertexHistoryThisVertex.remove(VertexNow);
201             }
202         }
203     }
204     return 1;
205 }
```

4.3 Implementasi *Output.*

Sesuai dengan analisis pada subbab 3.3.5, *output* akhir dari aplikasi ini nantinya adalah *circuit-circuit* penyebab *mutating table*. *Circuit* ini ditampilkan dalam bentuk *tree* supaya bisa diketahui dengan jelas mana yang merupakan hasil ekspansi dan mana yang bukan. Berikut ini adalah contoh *output*-nya:



Gambar 4.1 ***Window Ouput***

Dari *window output* di atas, akan mudah diketahui bahwa terdapat tiga buah tabel yang diramalkan akan terjadi *mutating*, yaitu tabel *tbl_c*, *tbl_d*, dan *tbl_b*. Untuk *tbl_b*, terdapat dua buah rute penyebab *mutating*. Rute ini adalah *Route #1* dan *#2*. Untuk *Route #1*, terdapat 4 buah *vertex*, yaitu trigg_5, *tbl_d*, trigg_7, dan trigg_6. Proses ekspansi terlihat pada *vertex tbl_d*.

BAB V

UJI COBA DAN EVALUASI

BAB 5

UJI COBA DAN EVALUASI

Bab ini menjelaskan tentang lingkungan uji coba, skenario serta *database* untuk uji coba dan pelaksanaan uji coba aplikasi yang telah dibuat.

5.1 Lingkungan Uji Coba

Uji coba aplikasi dilakukan dengan menggunakan spesifikasi sebagai berikut:

- Spesifikasi Perangkat Keras
  - Prosesor *Intel Pentium 4* berkecepatan 2.8 *GHz*
  - *Memory* 1024 *MB*.
- Spesifikasi Sistem
  - *Windows 2000 Professional*
  - *NetBeans 4.1*
  - *Oracle* 9i dan *Oracle* 10g *Release* 2
  - *Aqua Data Studio* 4.5.2
  - *Power Desaigner 9*

5.2 Skenario Uji Coba

Pada uji coba ini diberikan beberapa skenario yang ditujukan untuk mengetahui fungsionalitas aplikasi yang dibuat. Terdapat dua skenario yang diujikan pada aplikasi ini. Masing- masing skenario yang diujikan menggunakan kasus yang berbeda-beda. Garis besar skenario uji coba aplikasi dijelaskan pada subbab berikut.

5.2.1 Garis Besar Skenario 1

Pengujian Skenario 1 adalah pengujian aplikasi dengan menggunakan kasus-kasus fiktif seperti yang terdapat dalam bab analisis (bab 3). Skenario 1 ini digunakan untuk menguji apakah aplikasi telah mampu melakukan proses-proses seperti pada analisis bab 3.

5.2.1.1 Garis Besar Skenario 1.1

Skenario 1.1 diambil dari contoh kasus pada subbab 3.2.3.1.3, yaitu contoh 3 kasus 1. Penjelasan lebih lengkap mengenai contoh kasus ini bisa dilihat langsung pada subbab yang dikutip. Alasan mengapa skenario ini dijadikan salah satu kasus penguji adalah karena skenario ini merupkan bentuk dasar dari kasus *mutating table* yang berupa *singlecircuit*.

5.2.1.1 Garis Besar Skenario 1.2

Sama seperti skenario 1.1. Kasus ini diambil dari bab analisis, yaitu bab 3.2.3.1.5. Skenario ini digunakan untuk menguji aplikasi untuk kasus *multicircuit*.

5.2.1.1 Garis Besar Skenario 1.3

Kasus kali ini diambil dari dua kasus dalam bab analisis, yaitu bab 3.2.3.2.1. dan 3.2.3.2.2. Skenario 1.3 ini mewakili kasus *mutating* karena *cascade delete*.

5.2.1.1 Garis Besar Skenario 1.4

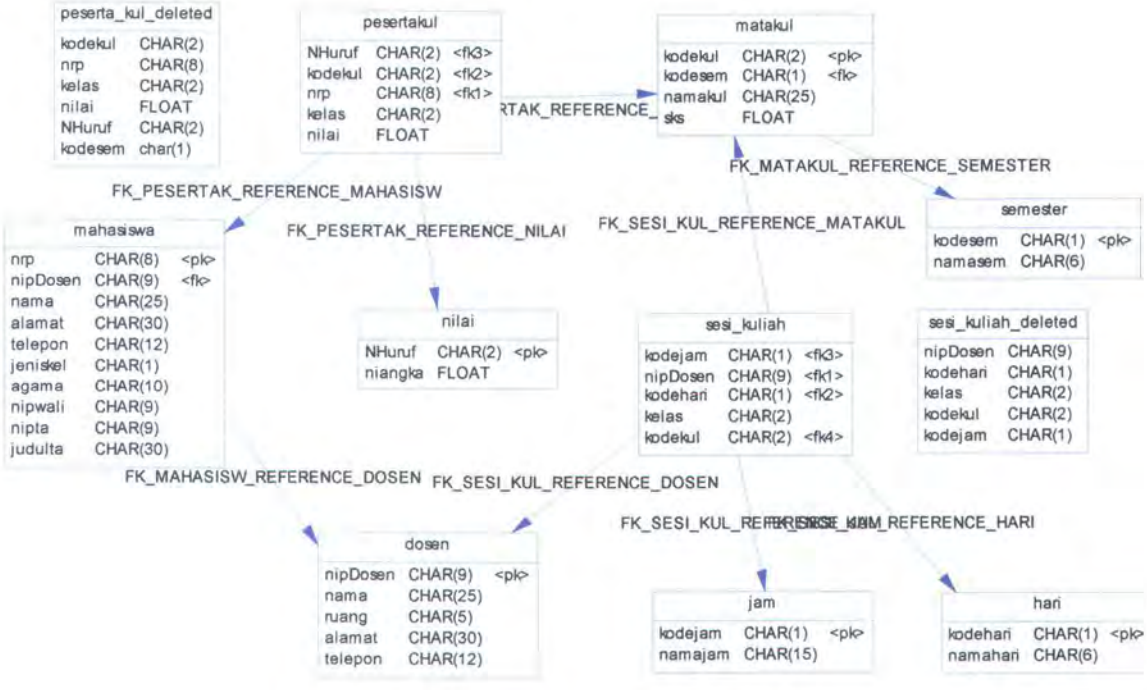
Skenario 1.4 ini diambil dari bab 3.2.4.2. Skenario ini dipilih untuk mewakili kasus *mutating* dengan *path* berupa *multicircuit* dengan jumlah tabel

yang *mutating* lebih dari satu buah. Skenario 1.4 ini dibaca dari *file* teks, bukan dari database seperti skenario-skenario sebelumnya.

5.2.2 Garis Besar Skenario 2

Skenario 2 adalah bertujuan untuk menguji aplikasi pada struktur database nyata, yaitu database SIM Akademik.

Berikut ini adalah struktur *PDM* dari sistem database SIM Akademik yang dibuat dengan aplikasi *Power Desaigner 9*:



Gambar 5.1 PDM untuk Skenario 2

Semua relasi di atas merupakan relasi *cascade delete*. Kecuali relasi dari tabel *mahasiswa* ke tabel *dosen*, serta relasi dari tabel *nilai* ke tabel *pesertakul*.

5.3 Pelaksanaan Uji Coba

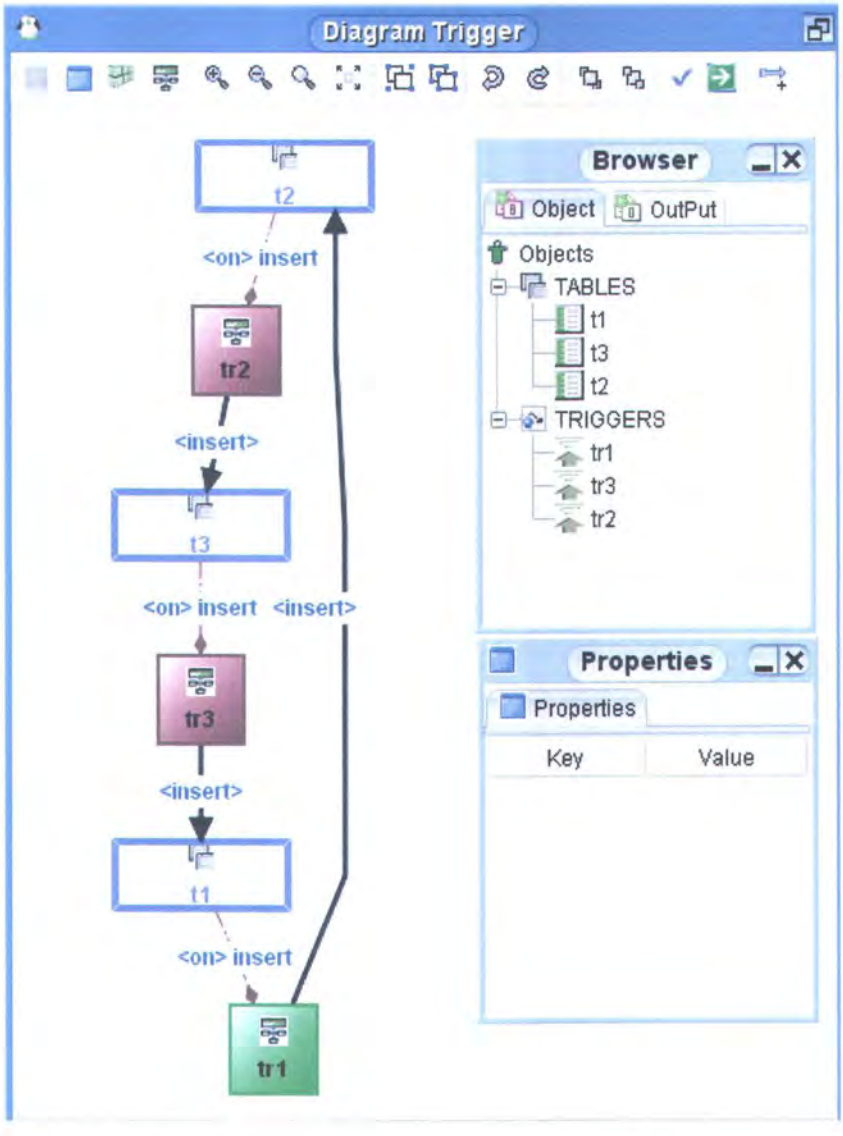
Uji coba dilaksanakan dengan jalan men-*generate* Diagram *Dependency Trigger*, lalu meramalkan dimana saja terjadi kasus *mutating table*. Ramalan ini kemudian dibuktikan dengan jalan mengekseskusi perintah-perintah *query* yang menyebabkan *mutating table* tersebut. Namun pembuktian ini hanya bisa dilakukan untuk Diagram *Dependency Trigger* yang di*generate* dari database, bukan dari *file* teks. Beberapa perintah-perintah *query* pada proses pelaksanaan uji coba ini dilakukan dengan menggunakan sebuah *tool* untuk *query analyzer*, yaitu *Aqua Data Studio 4.5.2* (www.aquafold.com)

5.3.1 Pelaksanaan Skenario 1

Skenario 1 diuji coba dengan jalan men-*generate* Diagram *Dependency Trigger* yang berasal dari database, meramalkan dimana saja letak *mutating table*, kemudian membuktikan ramalan tersebut dengan perintah-perintah *DML*. Khusus untuk skenario 1.4, pembuktian secara *query* tidak bisa dilakukan. Pembuktian yang ada hanyalah dengan jalan menganalisis *path* yang berhasil ditemukan oleh aplikasi secara manual.

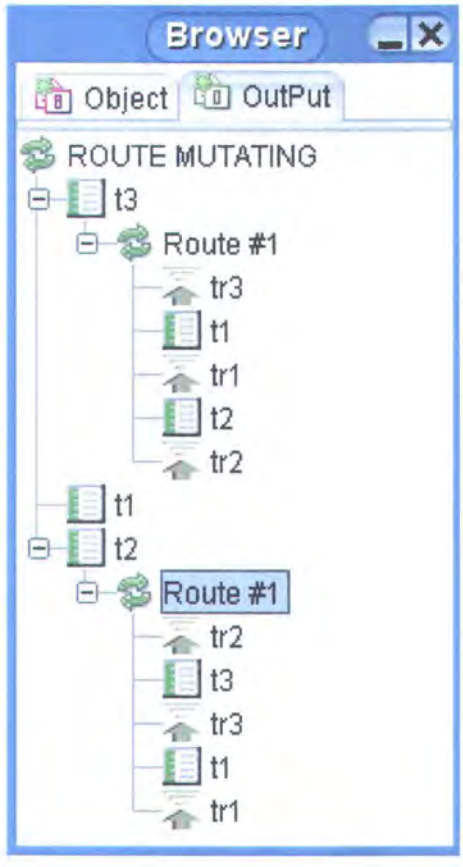
5.3.1.1 Pelaksanaan Skenario 1.1

Berikut ini adalah Diagram *Dependency Trigger* yang dibentuk oleh aplikasi dari *script DDL* pada skenario 1.1



Gambar 5.2 Diagram *Dependency Trigger* Skenario 1.1

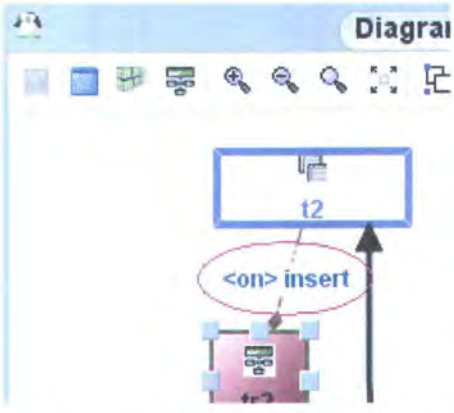
Dari Kasus ini, aplikasi berhasil menemukan dua buah tabel yang diramalkan akan terjadi kasus *mutating table*, yaitu tabel *t3* dan tabel *t2*. Selain menampilkan nama tabel yang dicurigai terjadi *mutating*, aplikasi juga akan menampilkan rute yang menyebabkan *mutating* tersebut. Rute ini tampak pada *tab output* di *window browser*.



Gambar 5.3 ***Window Browser*** **untuk menampilkan** ***output***

Dengan melihat *tree* ini, *user* akan mengetahui bahwa terdapat dua buah tabel yang diramalkan *mutating*, yaitu tabel *t3* dan *t2*. Tabel *t2* dan tabel *t3* masing-masing mempunyai satu buah rute penyebab *mutating*, yaitu *Route #1*. Untuk tabel *t2*, *Route #1* selama perjalanannya akan melewati empat buah *vertex*, yaitu *trigger tr2*, tabel *t3*, *trigger tr3*, tabel *t1*, dan *trigger tr1*.

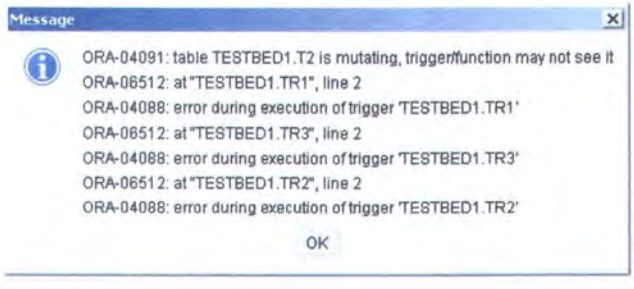
Untuk mengecek apakah peramalan ini benar atau tidak, maka perintah *DML* dijalankan pada tabel yang bersangkutan. Perintah *DML* ini haruslah sesuai dengan *edge* pertama dalam Diagram *Dependency Trigger*. Contohnya pada tabel *t2*. Karena *edge* pertama dalam *Route #1* adalah mempunyai label *<on> insert*, maka perintah *DML* untuk mengeceknya adalah *DML insert*.



Gambar 5.4 *Event* Pertama dalam Rute Penyebab *Mutating*

```
insert into t2 values('test')
```

Hasil yang didapat adalah keluarnya pesan kesalahan dari *Oracle* sebagaimana berikut ini:



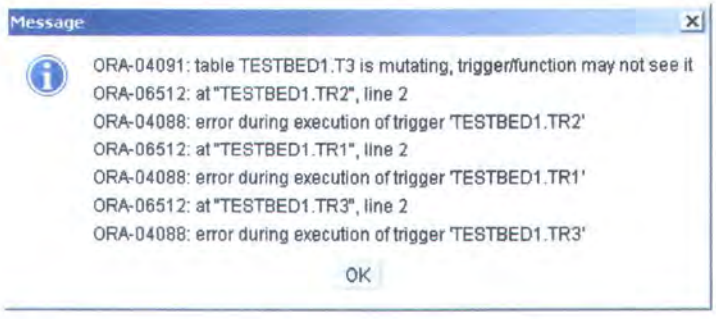
Gambar 5.5 Pesan Kesalahan

Peramalan ke-*mutating*-an tabel *t2* oleh aplikasi telah terbukti.

Berikutnya adalah membuktikan ke-*mutating*-an tabel yang kedua, yaitu tabel *t3*. Proses pembuktian ini juga dilakukan dengan jalan mengeksekusi perintah *DML insert* pada tabel *t3*. Karena *edge* pertama dalam rute penyebab ke-*mutating*-an tabel *t3* adalah *<on> insert*.

```
insert into t3 values('test')
```

Pesan kesalahan yang muncul adalah:



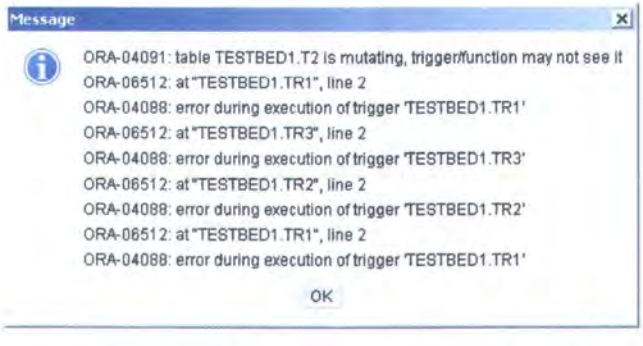
Gambar 5.6 Pesan Kesalahan yang Menunjukkan Bahwa Tabel *T3* *Mutating*

Peramalan tabel *t3* ini juga telah berhasil.

Untuk percobaan lebih lanjut, operasi *insert* juga dijalankan pada tabel yang tidak diramalkan akan terjadi *mutating*, yaitu tabel *t1*.

```
insert into t1 values('test')
```

Output yang didapatkan adalah:



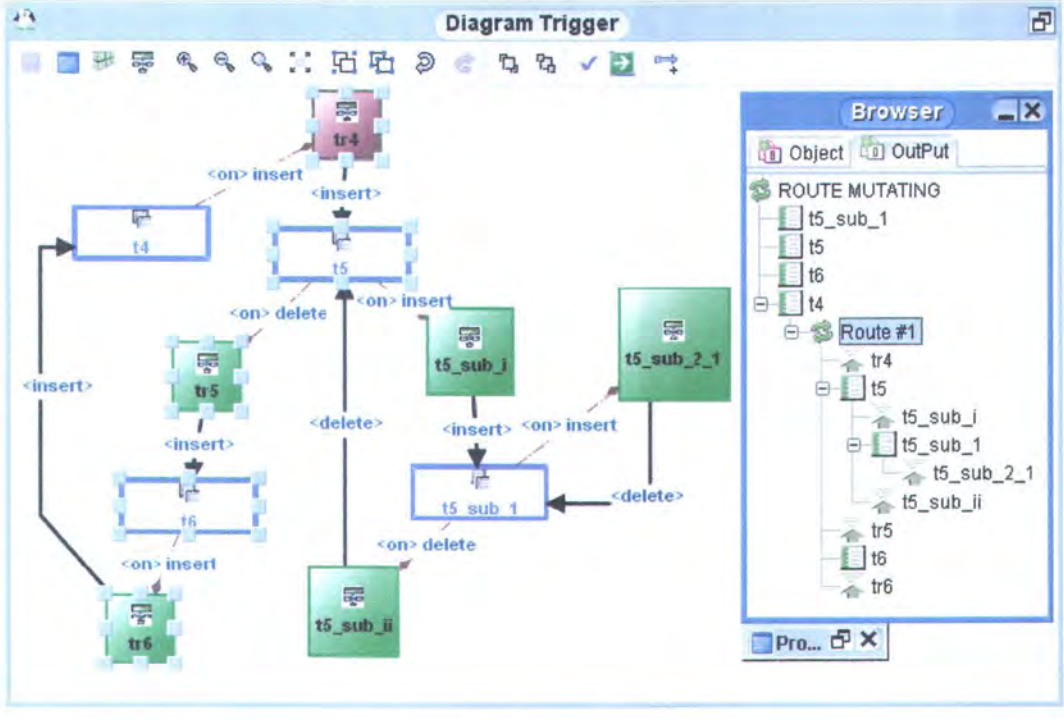
Gambar 5.7 Pesan Kesalahan yang Menunjukkan Bahwa Tabel *T2* *Mutating*

Ternyata tetap terjadi kasus *mutating table* pada tabel *t2*. hal ini membuktikan bahwa tabel *t1* tidak *mutating*. Ke-*mutating*-an tabel *t2* ini karena *DML insert* pada tabel *t1* memicu *trigger tr1* melakukan *action insert* pada tabel *t2*. Padahal tabel *t2* itu sendiri akan terjadi *mutating* ketika terkena *action insert*.

5.3.1.2 Pelaksanaan Skenario 1.2

Tujuan dari skenario 1.2 ini adalah menguji coba hasil implementasi dari algoritma *VtCX*.

Aplikasi meramalkan bahwa akan terjadi *mutating table* pada tabel *t4* dengan rute sebagaimana berikut:



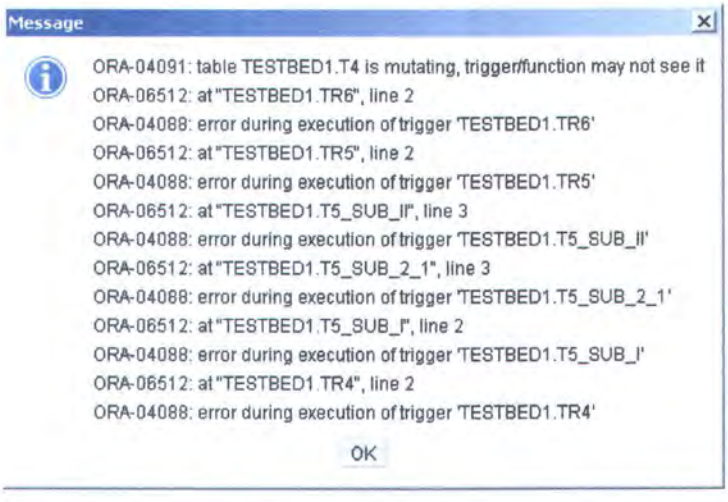
Gambar 5.8 Penandaan Rute "*Route #1*"

Untuk kasus kali ini rute yang ditemukan adalah *multicircuit*. Dari *window browser* bisa diketahui bahwa proses ekspansi dari algoritma *VtCX* berlangsung selama dua kali. Ekspansi ini terjadi pada tabel *t5* dan *t5_sub_1*.

Berikutnya adalah pembuktian peramalan di atas. Perintah *DML* yang dipakai untuk pembuktian ini adalah:

```
insert into t4 values('test')
```

Output dari perintah *SQL* di atas adalah tampak pada pesan kesalahan sebagaimana berikut ini:

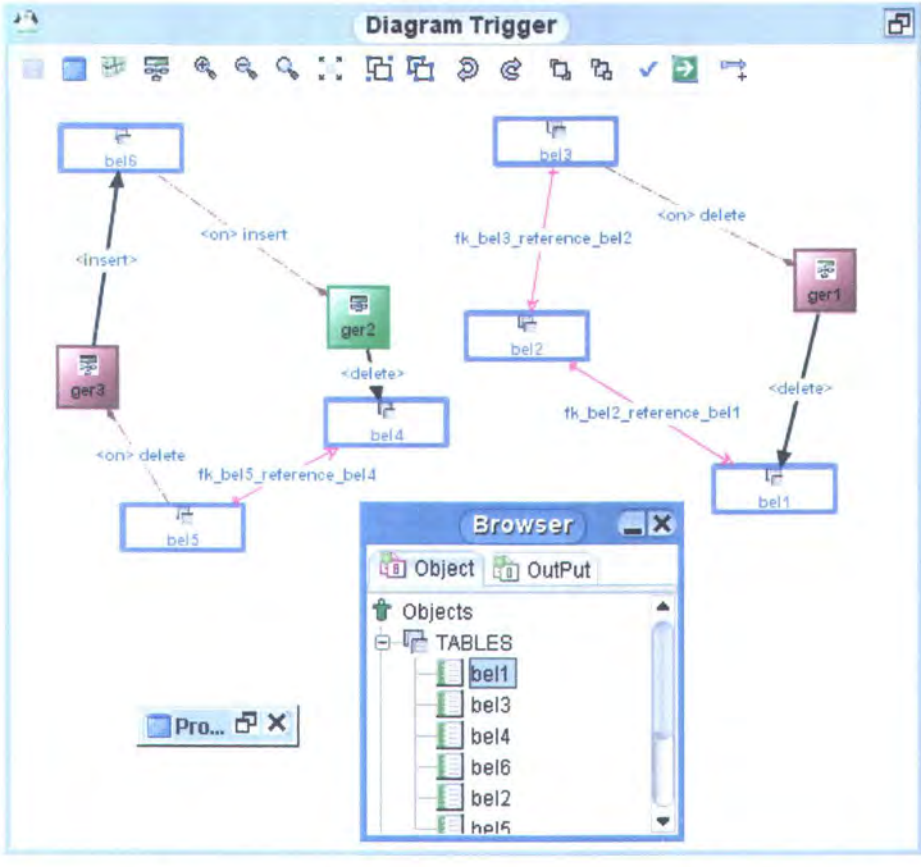


Gambar 5.9 Pesan Kesalahan yang Menunjukkan Bahwa Tabel *t4 Mutating*

Peramalan pada skenario 1.2 ini telah terbukti.

5.3.1.3 Pelaksanaan Skenario 1.3

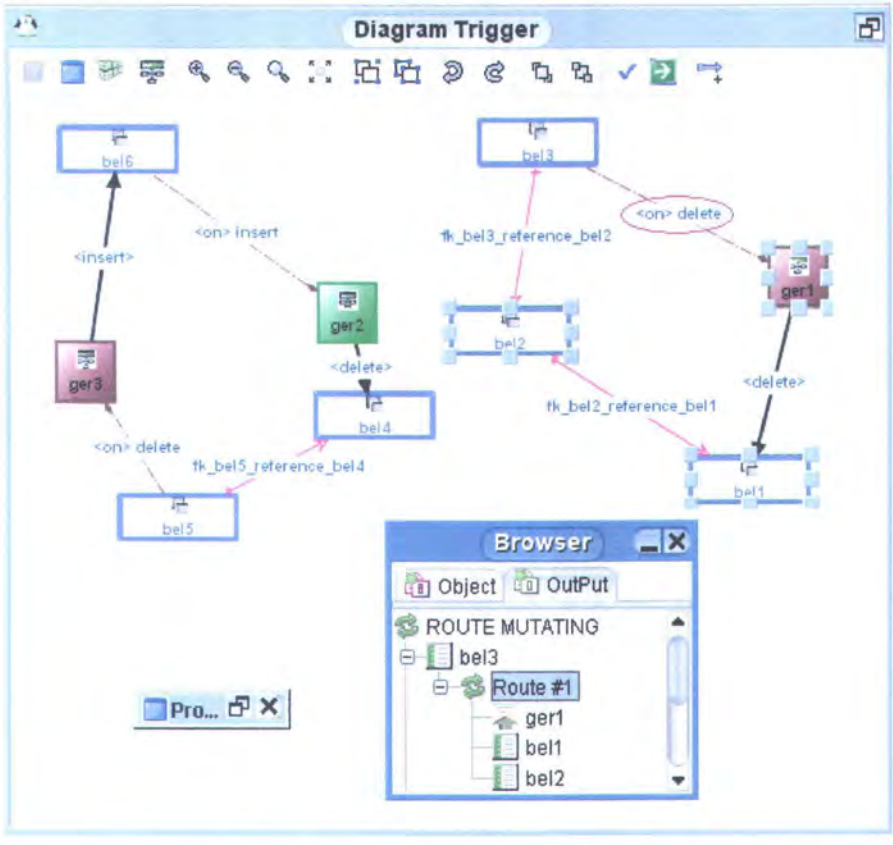
Kasus ini adalah untuk menguji aplikasi dalam meramalkan *mutating table* karena *cascade delete*. Diagram yang terbentuk adalah sebagaimana berikut ini:



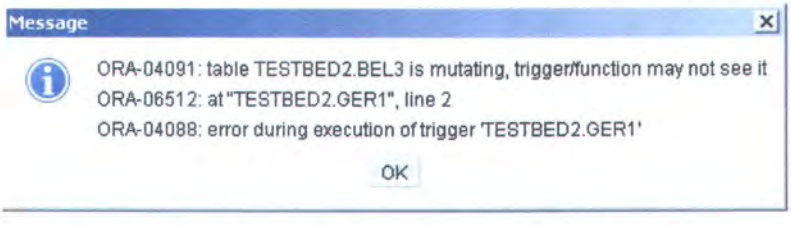
Gambar 5.10 Diagram *Dependency Trigger* untuk Schema *testbed2*

Adapun tabel yang diramalkan terkena kasus *mutating table* adalah semua tabel yang ada dalam diagram, kecuali tabel *bel6*.

Untuk pembuktian pertama, dilakukan *DML delete* pada tabel *bel3* (`delete from bel3`). hal ini karena *delete* adalah *edge* pertama dari *path* penyebab *mutating*.



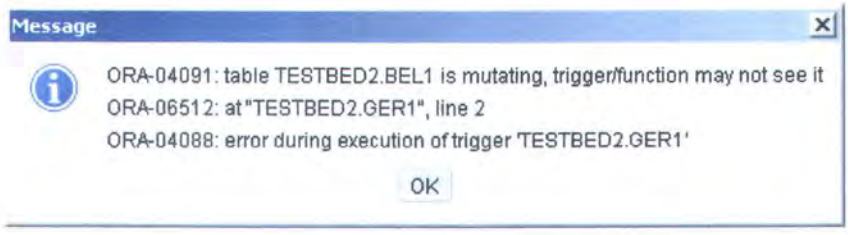
Gambar 5.11 *Event* Pertama dalam Rute "*Route* #1"



Gambar 5.11 Pesan Bahwa Tabel *bel3* adalah *Mutating*

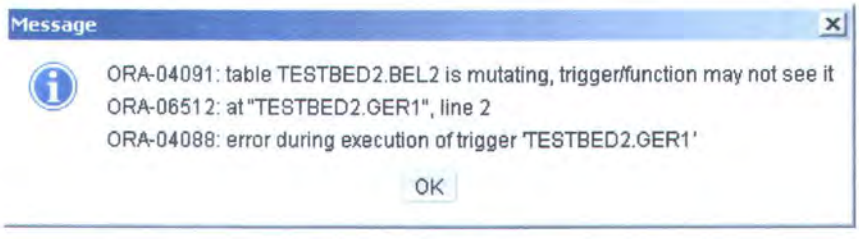
Begitu juga untuk dua tabel yang masih menjadi satu *circuit* dengan *bel3*, yaitu *bel2* dan *bel1*. Kedua tabel tersebut dibuktikan ke-*mutating*-annya dengan mengeksekusi *DML delete*.

Hasil yang didapat dari *DML delete* terhadap tabel *bel1* dan *bel2* adalah sebagaimana berikut ini:



Gambar 5.12 Pesan Bahwa Tabel *bel2* adalah *Mutating*

```
delete from bel1
```



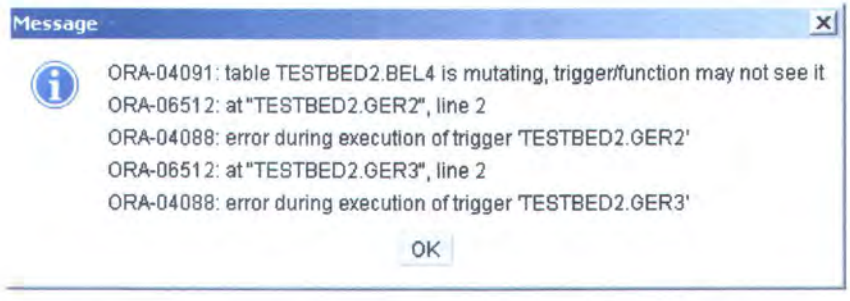
Gambar 5.13 Pesan Bahwa Tabel *bel2* adalah *Mutating*

```
delete from bel2
```

Peramalan *mutating table* oleh aplikasi terbukti benar adanya.

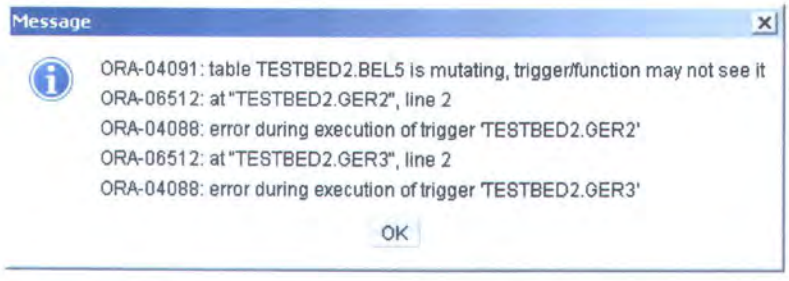
Selanjutnya adalah membuktikan ramalan ke-*mutating*-an dari tabel *bel4* dan bel5. Sama seperti sebelumnya, *DML* yang dipakai untuk proses pembuktian adalah *delete.*

```
delete from bel4
```



Gambar 5.14 Pesan Bahwa Tabel *bel4* adalah *Mutating*

```
delete from bel5
```

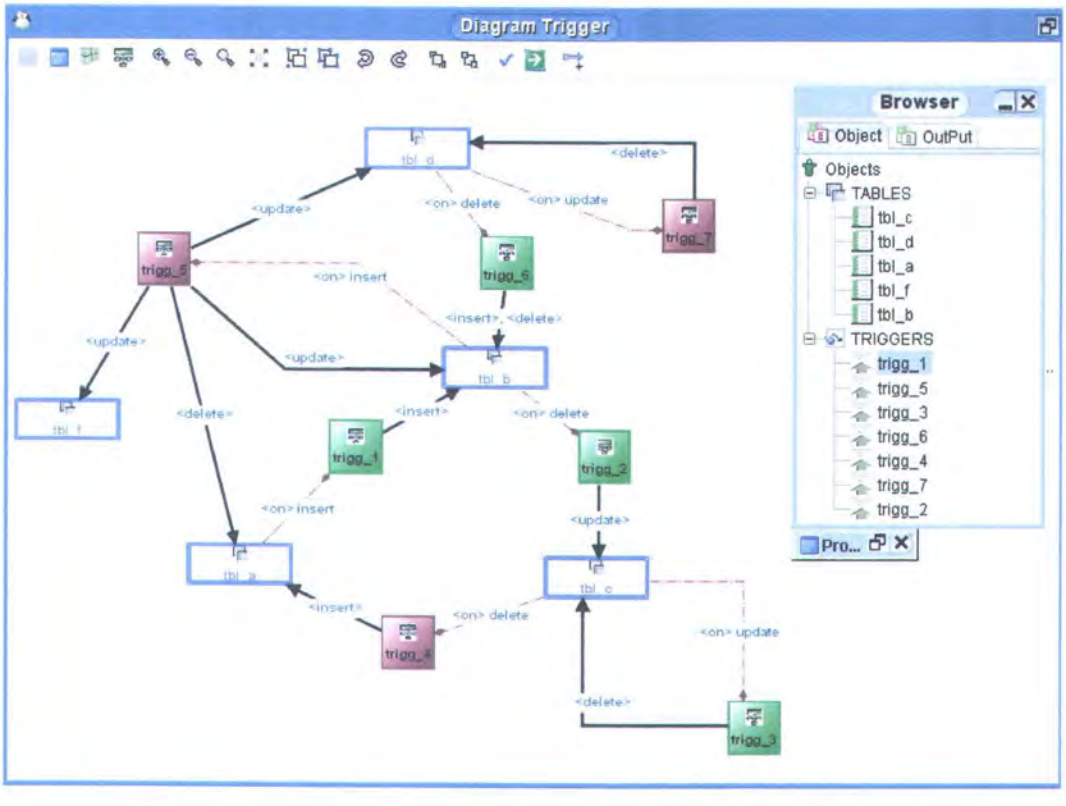


Gambar 5.15 Pesan Bahwa Tabel *bel5* adalah *Mutating*

Ramalan ke-*mutating*-an untuk tabel *bel4* dan *bel5* oleh aplikasi juga terbukti.

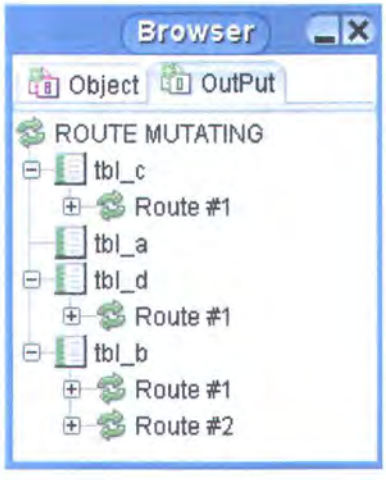
5.3.1.4 Pelaksanaan Skenario 1.4

Karena untuk skenario ini hanya membaca dari *file* teks, maka koneksi ke database tidak dibutuhkan.



Gambar 5.16 Diagram *Dependency Trigger* dari *File* Teks

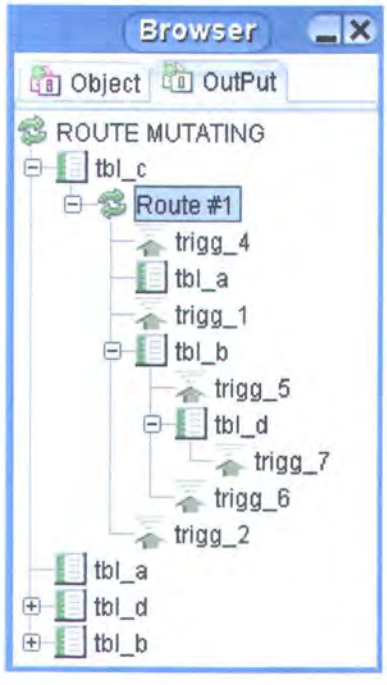
Berikutnya adalah meramalkan semua *mutating table* yang ada dalam Diagram *Dependency Trigger* di atas. Hasil peramalannya bisa dilihat pada *window output* berikut ini:



Gambar 5.17 *Window Ouput* Penampil Rute Penyebab *Mutating*

Apabila pembuktian peramalan *mutating table* pada Diagram *Dependency Trigger,* yang dibuat melalui database, adalah dengan menjalankan perintah-perintah, maka untuk kali ini pembuktiannya adalah dengan cara manual, yaitu dengan jalan menelusuri satu-persatu *path* yang ditemukan oleh aplikasi. Penelusuran ini untuk menegetahui apakah ramalan tersebut benar atau tidak.

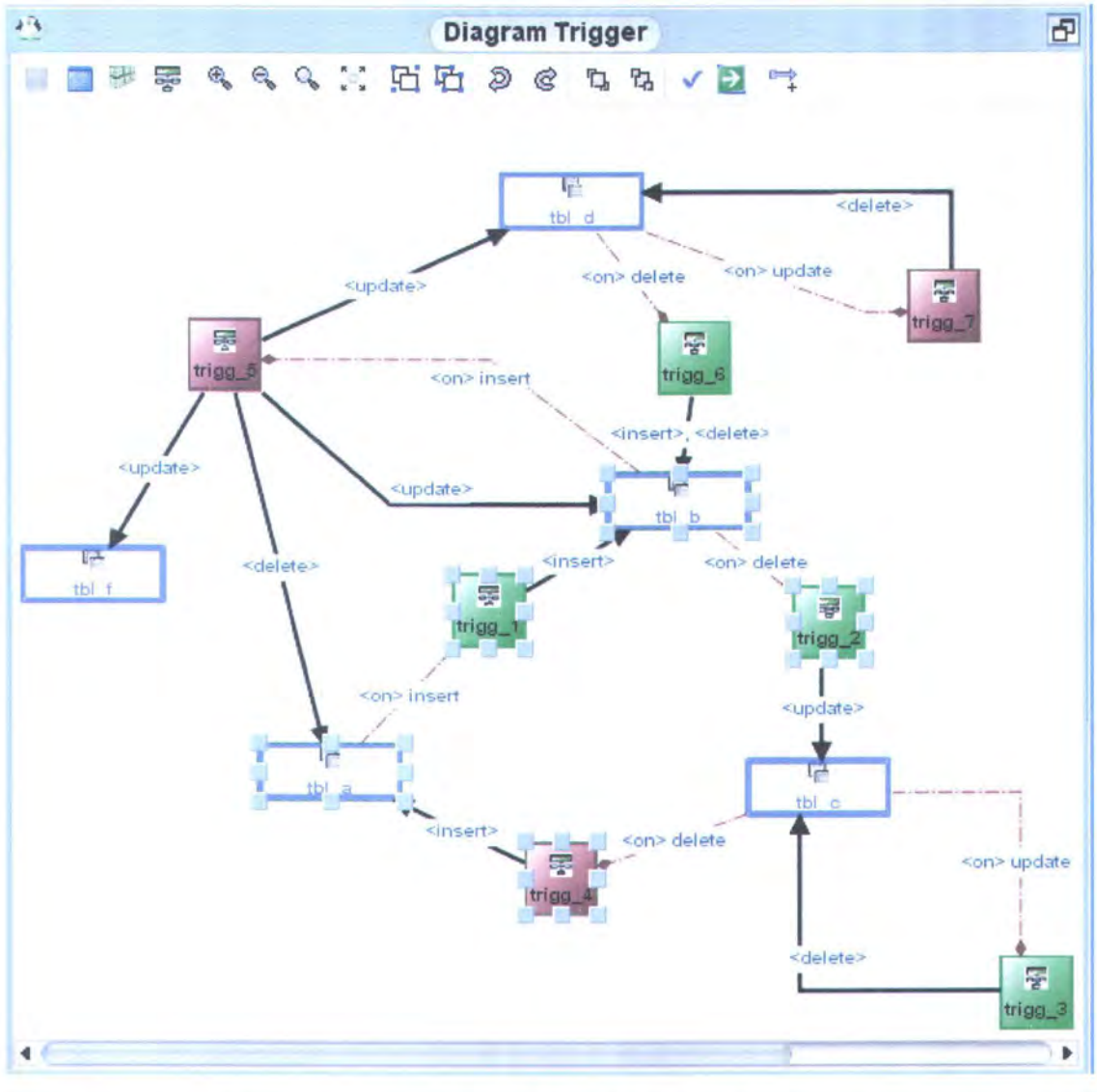
Pembuktian pertama dilakukan pada rute *mutating* untuk tabel *tbl_c*. Rute *mutating* yang dimiliki oleh *tbl_c* bisa dilihat pada *screen shot* berikut ini:



Gambar 5.18 Rute *Mutating* Untuk Tabel *tbl_c* dalam *Window Output*

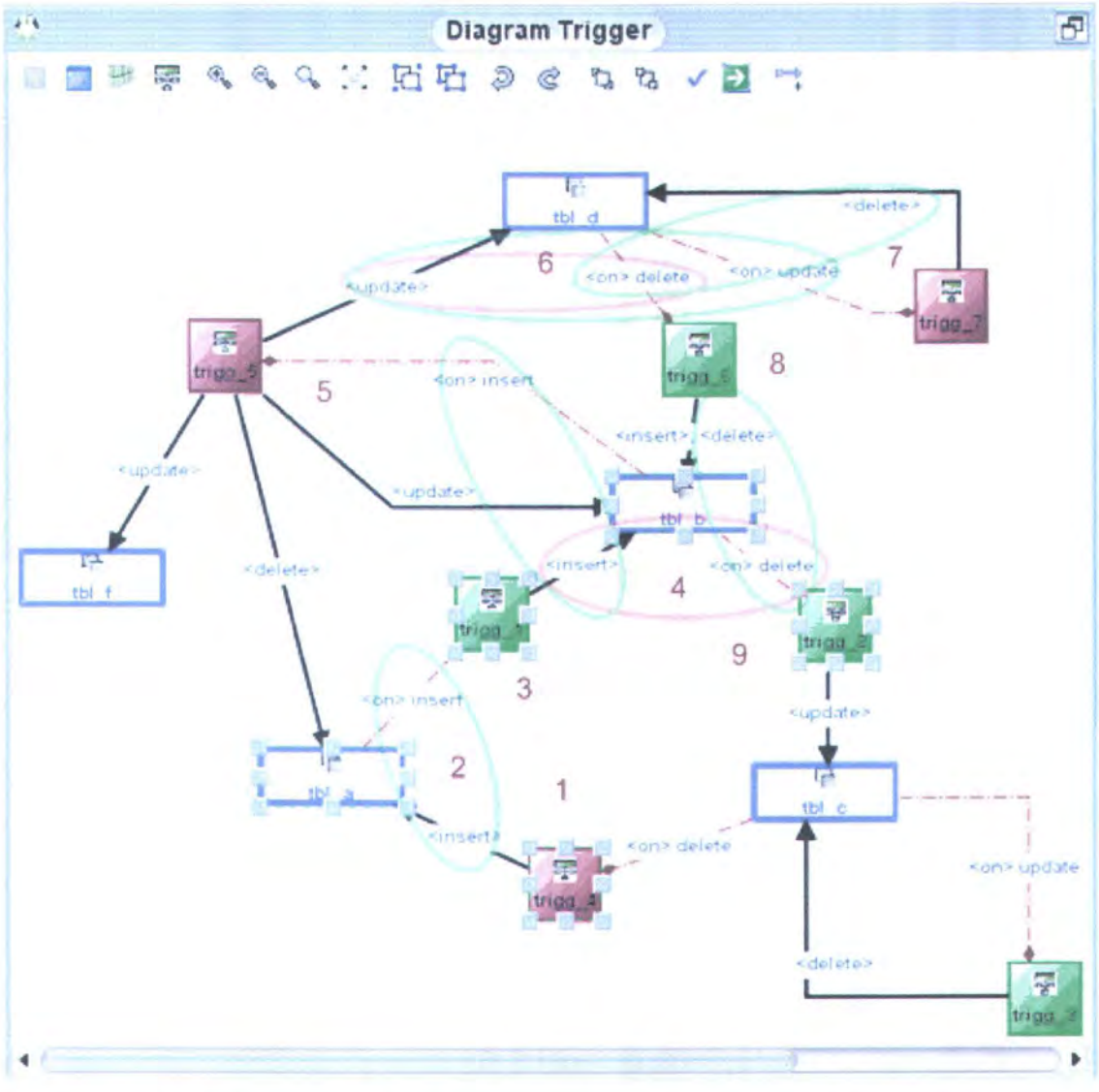
Dalam *window* di atas, terlihat bahwa rute *mutating* untuk tabel *tbl_c* mengalami dua kali proses ekspansi. Ekspansi ini terjadi pada tabel *tbl_b* dan *tbl_d*.

Berikut ini adalah rute *root* yang telah diberi tanda oleh aplikasi.



Gambar 5.19 Rute *Mutating* dari Tabel *tbl_c* di *Window* Diagram *Dependency Trigger*

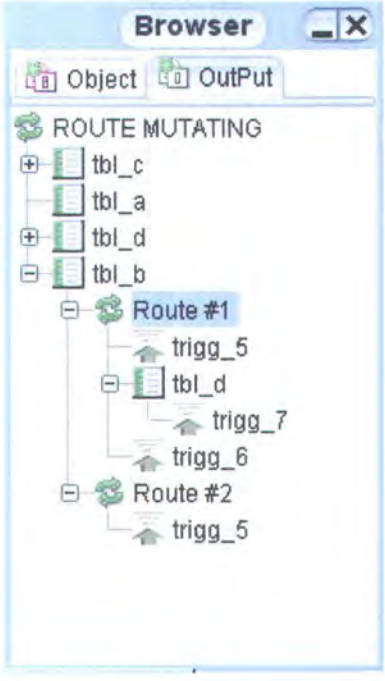
Berikut ini adalah pemberian tanda elips dari Diagram *Dependency Trigger* yang menunjukkan bahwa peramalan ke-*mutating*-an tabel *tbl_c* di atas adalah benar. Pemberian tanda ini didasarkan pada algoritma *CRV*. Adapun elips berwarna merah berarti *edge* gagal meneruskan *chain reaction*, sedangkan elips hijau berarti *edge* berhasil meneruskan *chain reaction*.



Gambar 5.20 Penandaan *Chain reaction* Rute *Mutating* untuk Tabel *tbl_c*

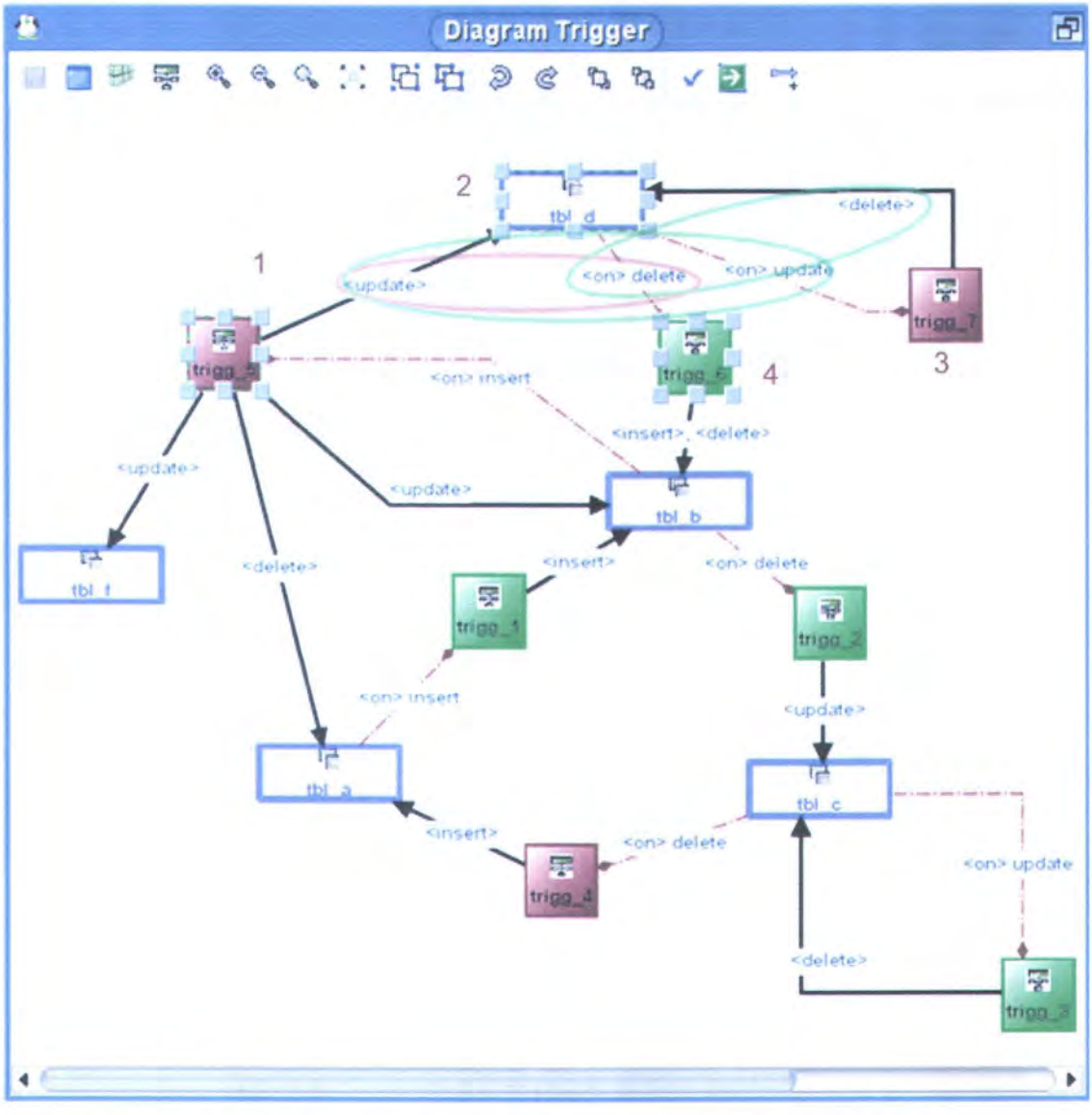
Berdasarkan penelusuran manual dari gambar di atas, terlihat bahwa *chain reaction* yang berasal dari tabel *tbl_c* berhasil kembali ke *tbl_c*. Dari rute tersebut, *trigger* pertama yang ditemui adalah *trigger trigg_4* yang merupakan *trigger row level*. Hal ini menyebabkan *tbl_c mutating*.

Berikutnya adalah pembuktian ramalan ke-*mutating*-an tabel *tbl_b*. Untuk tabel *tbl_b* ini, terdapat dua buah rute yang menyebabkan tabel *tbl_b* menjadi *mutating*. Rute ini antara lain adalah:



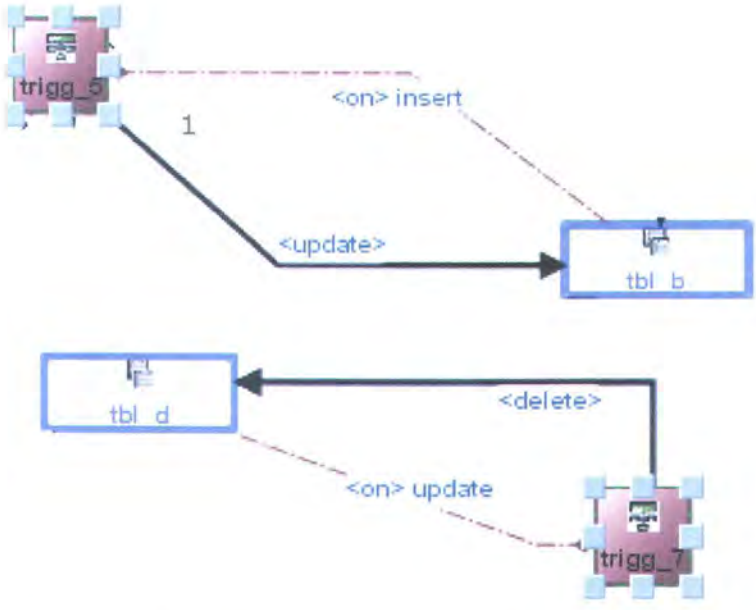
Gambar 5.21 Rute Penyebab *Mutating* untuk *tbl_b*

Dalam rute *Route #1,* terdapat satu kali ekspansi. Ekspansi ini terjadi pada tabel *tbl_d*. Pembuktian ke-*mutating*-an rute ini dengan menggunakan algoritma *CRV* secara manual bisa dilihat pada gambar berikut:



Gambar 5.22 Penandaan Rute *Chain reaction* untuk Tabel *tbl_b*

Untuk Rute *Route #2* milik tabel *tbl_b* dan *Route #1* milik tabel *tbl_d* terdapat kondisi *mutating* dengan tipe *direct access* (lihat bab 3.2.3.3). Pembuktian kasus *mutating* untuk *direct access* relatif mudah. Pembuktian ini bisa dilihat pada gambar berikut:



Gambar 5.23 *Direct Access* untuk *tbl_b* dan *tbl_d*

Semua ramalan ke-*mutating*-an yang terjadi pada tabel *tbl_c*, *tbl_d* dan *tbl_d* oleh aplikasi telah benar terbukti.

5.3.2 Pelaksanaan Skenario 2

Pada subbab ini akan dijelaskan mengenai pelaksanaan uji coba aplikasi untuk kasus database SIM akademik.

5.3.2.1 Pembuatan *Trigger hist_trigg*

Trigger yang akan diuji pada skenario 2 ini berjumlah satu buah. *Trigger* ini adalah *trigger* yang berfungsi mencatat *history* peserta kuliah yang mengundurkan diri. *Trigger* ini dieksekusi ketika salah satu *record* di tabel *pesertakul* dihapus. Record yang di-*delete* ini akan ditulis ke tabel *peserta_kul_deleted*. *Trigger* ini bernama *hist_Trig*.

```
CREATE OR REPLACE TRIGGER "TESTBED3"."hist_trigg"

  after delete on pesertakul
  for each row

declare
    recordDeleted pesertakul%rowtype;
    kodesem varchar(1);

begin
    select semester.kodesem
    into kodesem
    from matakul, semester where :old.kodekul = matakul.kodekul
and matakul.kodesem = semester.kodesem;

    insert into
peserta_kul_deleted(kodekul,nrp,kelas,nilai,nhuruf,kodesem)

values(:old.kodekul,:old.nrp,:old.kelas,:old.nilai,:old.nhuruf,kod
esem);

end hist_trigg;
```

Alur dari *trigger* ini adalah sebagaimana berikut: mula-mula *trigger hist_trigg* akan dieksekusi ketika terjadi *event delete* di tabel *pesertakul*. *Record-record* dari tabel *pesertakul* yang ter-*delete* tersebut akan dicari tahu pada semester berapa *record* tersebut berasal. *Query* untuk menentukan semester tersebut terdapat pada *trigger body*. Hasil dari *query* tersebut disimpan dalam sebuah variabel bernama *kodesem*. Kode tersebut adalah seperti ini:

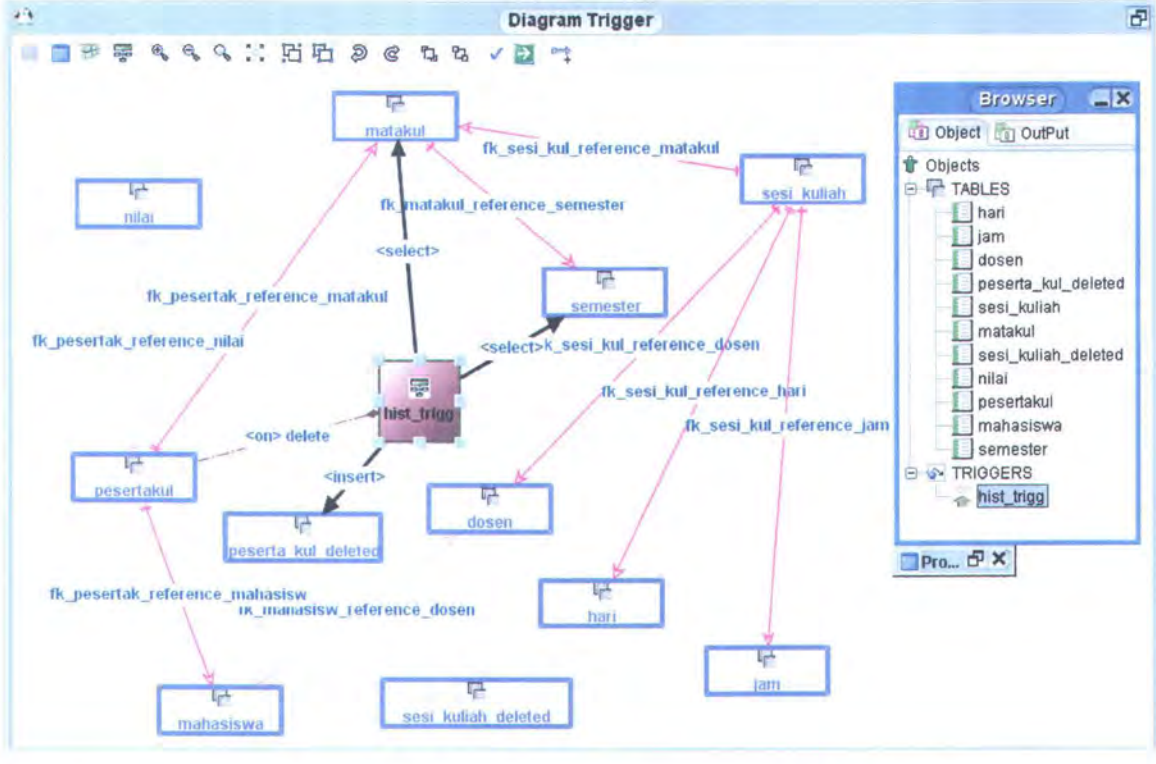
```
select semester.kodesem
    into kodesem
    from matakul, semester where :old.kodekul =
matakul.kodekul and matakul.kodesem = semester.kodesem;
```

Langkah proses dari *trigger hist_trigg* berikutnya adalah meng-*insert*-kan data yang telah terhapus tadi ke dalam tabel *peserta_kul_deleted*. Kolom-kolom dalam tabel *peserta_kul_deleted* adalah sama persis dengan tabel *pesertakul*, namun ditambah satu buah kolom yang menyatakan semester. Kolom ini nantinya akan diisi oleh variabel *kodesem*.

Kode yang menyatakan perintah tersebut adalah seperti ini:

```
insert into
peserta_kul_deleted(kodekul,nrp,kelas,nilai,nhuruf,kodesem)

values(:old.kodekul,:old.nrp,:old.kelas,:old.nilai,:old.nhur
uf,kodesem);
```

5.3.2.3 Pembuatan Diagram *Dependency Trigger*



Gambar 5.24 Diagram *Dependency Trigger* untuk Skenario 2

5.3.2.4 Pengujian Fungsionalitas *hist_trigg*

Sebelum proses peramalan *mutating table* dijalankan, mula-mula dibuktikan dulu apakah *trigger hist_trigg* sudah bisa bekerja dengan baik sesuai harapan atau tidak. Pengujian fungsionalitas *trigger hist_trigg* adalah dengan jalan menghapus salah satu *record* dalam tabel *pesertakul.*

Isi awal dari tabel *pesertakul* adalah bisa dilihat pada *screen shot* berikut ini:

	NHURUF	KODEKUL	NRP	KELAS	NILAI
1	A	SP	11020002	A1	83
2	E	SP	11020003	A1	24
3	AB	SP	11020004	A1	76
4	D	SP	11020005	A1	55
5	E	SP	11020006	A1	13
6	E	SP	11020007	A1	38
7	C	SP	11020008	A1	57
8	E	SP	11020009	A1	33
9	B	SP	11020010	A1	67
10	E	SP	11020011	A1	40
11	A	SP	11020012	A1	82
12	E	SP	11020013	A1	20
13	E	SP	11020014	A1	28
14	A	SP	11020015	A1	88
15	A	SP	11020016	A1	88
16	BC	SP	11020017	A1	64
17	BC	SP	11020018	A1	61
18	D	SP	11020019	A1	51
19	E	SP	11020020	A1	39

Gambar 5.25 Isi Awal dari Tabel *pesertakul*

Berikutnya adalah mengekseskusi perintah *DML* untuk menghapus salah satu *record* di tabel *pesertakul*. *Record* yang dihapus ini adalah *record* dengan NRP = '11020002'. *DML* tersebut adalah seperti ini:

```
delete from pesertakul where nrp = '11020002'
```

Begitu *DML* di atas dieksekusi, dengan segera *record* yang terhapus akan pindah ke tabel *peserta_kul_deleted.*

	KODEKUL	NRP	KELAS	NILAI	NHURUF	KODESEM
1	SP	11020002	A1	83	A	1
2	DW	11020002	A1	36	E	1
3	TD	11020002	A1	54	D	1
4	VB	11020002	A1	30	E	1
5	PD	11020002	A1	28	E	1
6	SS	11020002	A1	45	D	2
7	DL	11020002	A1	39	E	2
8	PW	11020002	A1	35	E	2
9	AS	11020002	A1	56	C	2
10	VL	11020002	A1	28	E	2

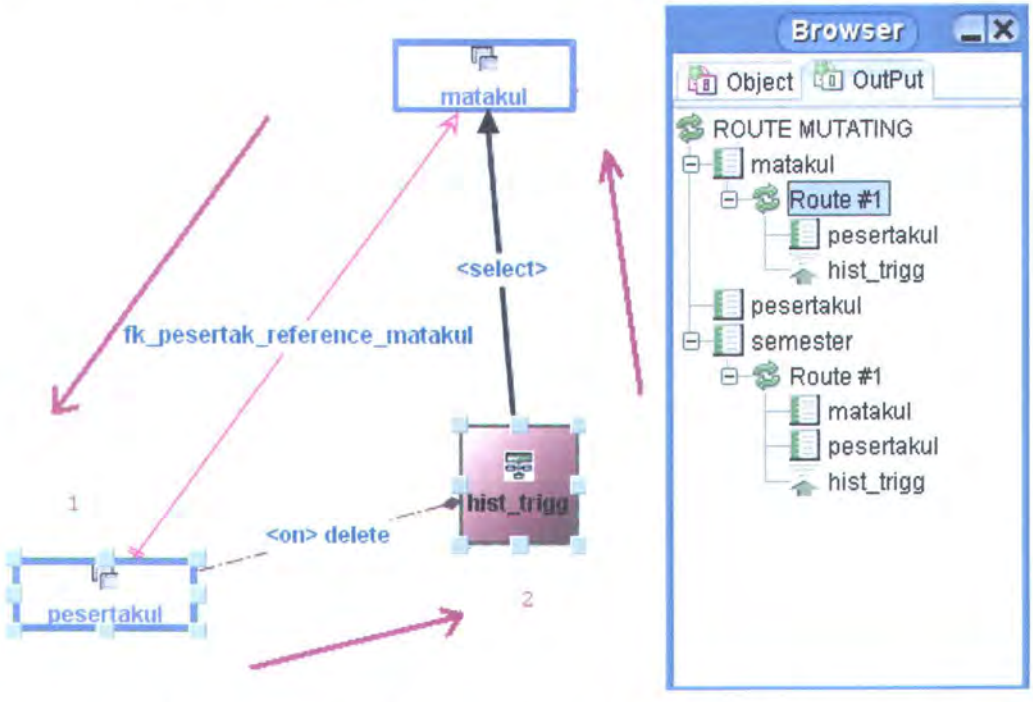
Gambar 5.26 Isi dari Tabel *peserta_kul_deleted*

Dari pengujian ini, bisa diketahui bahwa *trigger hist_trigg* telah berhasil bekerja dengan baik tanpa memunculkan pesan kesalahan.

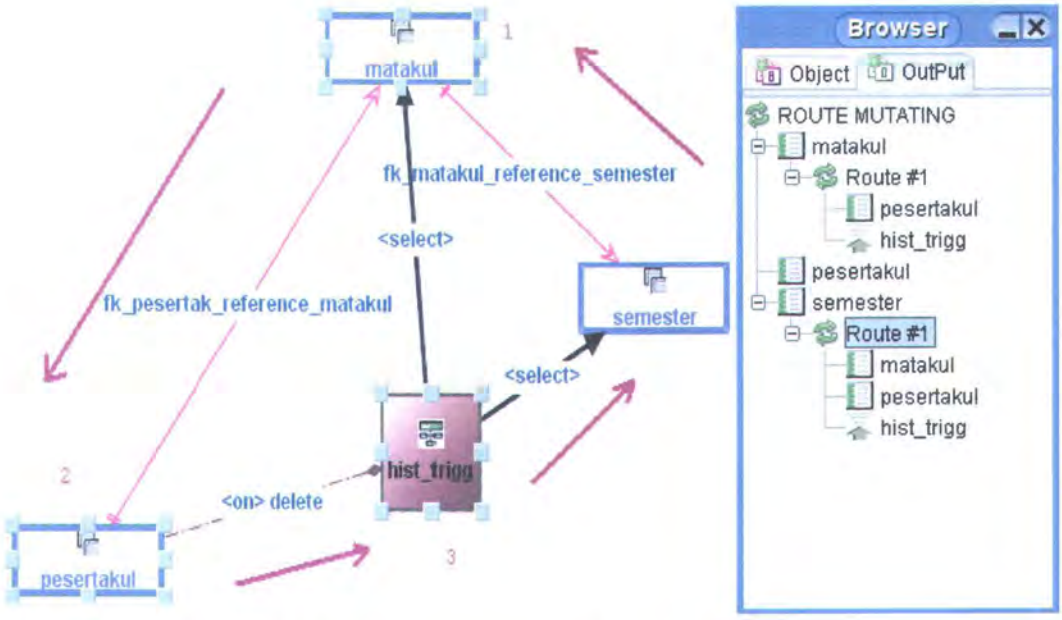
5.3.2.5 Peramalan *Mutating Table*

Kemudian langkah berikutnya adalah meramalkan tabel apa saja dari Diagram *Dependency Trigger* yang mungkin terjadi kasus *mutating table*.

Hasil peramalan dari aplikasi menunjukkan bahwa ada dua buah tabel yang kemungkinan terjadi *mutating*, yaitu tabel *matakul* dan tabel *semester*. Ke-*mutatin*-gan kedua tabel tersebut bisa dimengerti dengan mudah dengan melihat struktur potongan Diagram *Dependency Trigger* berikut ini:



Gambar 5.27 Penjelasan Rute Penyebab Ke-*mutating*-an Tabel *matakul*

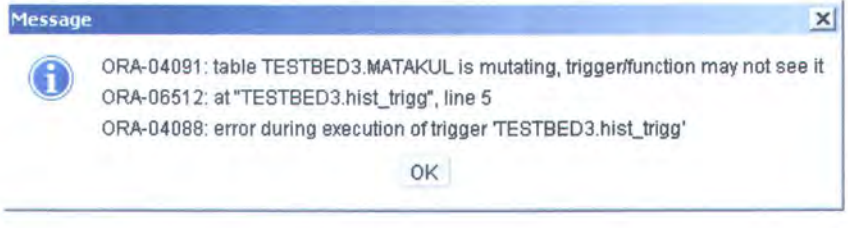


Gambar 5.28 Penjelasan Rute Penyebab Ke-*mutating*-an Tabel *semester*

5.3.2.6 Pembuktian Hasil Ramalan

Dari salah satu potongan struktur Diagram *Dependency Trigger* di atas, bisa diketahui bahwa kasus *mutating table* pada tabel *matakul* terjadi apabila terjadi *event delete* pada tabel tersebut. Berikut ini adalah pembuktiannya:

```
delete from matkul where kode = 'AS'
```

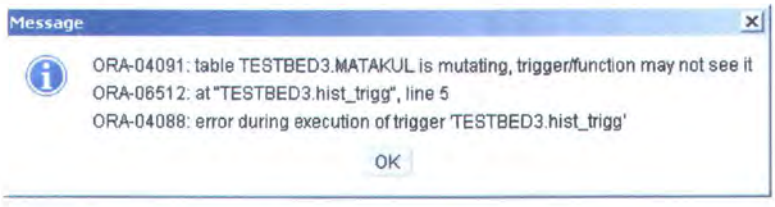


Gambar 5.29 Pesan Kesalahan dari *Oracle* bahwa Tabel *matakul Mutating*

Ramalan ke-*mutating*-an tabel *matkul* telah terbukti.

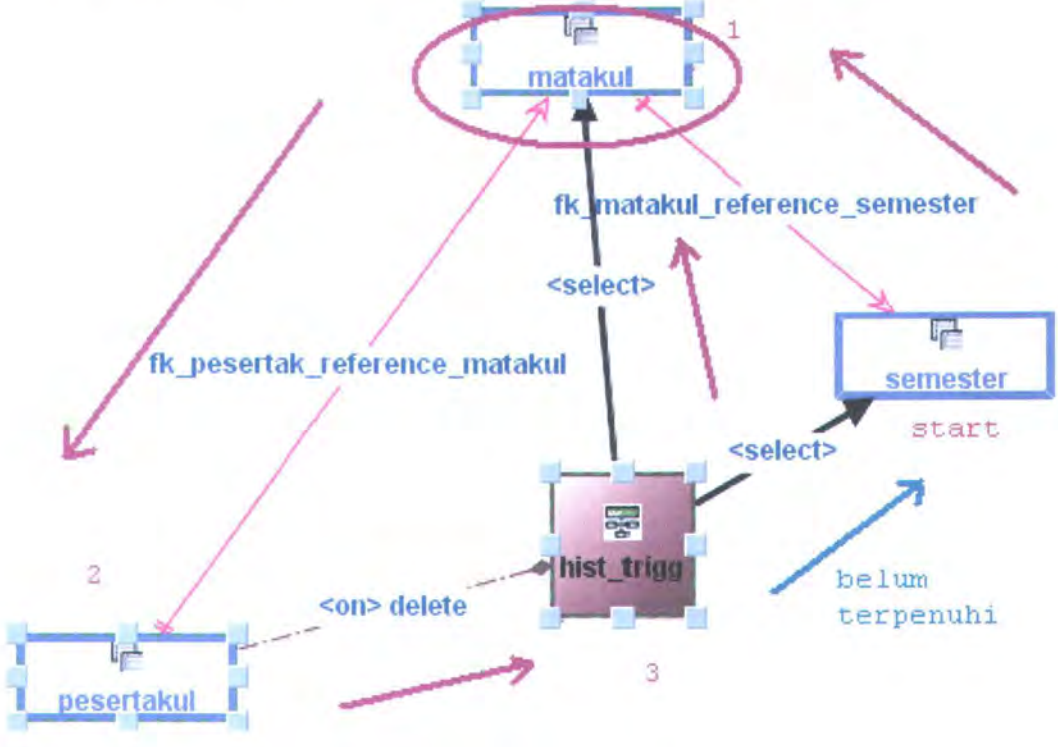
Berikutnya adalah membuktikan ke-*mutating*-an tabel yang kedua, yaitu tabel *semester*. Sama seperti tabel *matkul*, ke-*mutating*-an tabel ini terjadi apabila dalam tabel *semester* terjadi *event delete*. Berikut ini adalah pembuktiannya:

```
delete from semester where kodesem = '1'
```



Gambar 5.30 Pesan kesalahan dari *Oracle* Bahwa Tabel *matakul mutating*

Ternyata pesan yang muncul adalah pesan ke-*mutating*-an tabel *matkul*, bukan pada tabel *semester*. Hal ini terjadi karena ke-*mutating*-an tabel *semester* terjadi sesudah ke-*mutating*-an tabel *matkul*.



Gambar 5.31 Ke-*mutating*-an Tabel *semester* Sesudah Tabel *matkul*

Dengan membaca *source code* dari *trigger body* bisa diketahui bahwa tabel *matakul* diakses terlebih dahulu sebelum tabel *semester* diakses. Karena eksekusi *trigger* terhenti pada tabel *matkul*, maka ke-*mutating*-an tabel *semester* belum

terjadi. Akan tetapi, hal ini bukan berarti ramalan aplikasi salah. Namun sesuai dengan batasan masalah pada bab 1, disebutkan bahwa hasil peramalan aplikasi ini adalah hanya berupa *warning*. Artinya, *output* peramalan valid apabila semua *event* dan *action* yang terdapat dalam rute terpenuhi. Dalam hal ini rute untuk *action select* belum terpenuhi, sebab *action* tersebut yang ada di dalam *trigger_body* belum sempat tereksekusi, sehingga *mutating table* belum terjadi.

BAB VI

KESIMPULAN DAN SARAN

BAB 6

KESIMPULAN DAN SARAN

6.1 Kesimpulan

Kesimpulan yang bisa diambil dari Tugas Akhir ini adalah:

- Penggunaan Diagram *Dependency Trigger*, disamping penggunaan Diagram *PDM* standar, lebih memudahkan administrator database dan pihak *developer PL/SQL* untuk menganalisis dan mengatur *trigger-trigger* serta relasi-relasi dalam suatu struktur database. Terutama untuk lebih memahami alur logika dari *trigger-trigger* yang terdapat dalam suatu database.
- Diagram *Dependency Trigger* bisa digunakan untuk meramalkan suatu kasus *mutating table* secara lebih mudah daripada harus dengan membaca *source code*.

6.2 Saran

Saran yang dapat digunakan untuk pengembangan lebih lanjut dari aplikasi ini adalah:

- Pembuatan Diagram *Dependency Trigger* tidak hanya diterapkan pada *Oracle* saja.
- Pembacaan *action* pada *trigger* hendaknya tidak hanya dilakukan pada *trigger body* saja, akan tetapi diperluas hingga ke *stored procedure* yang diacu oleh *trigger* tersebut dalam *trigger body*-nya.
- Aplikasi bisa memberikan suatu solusi pemecahan untuk kasus *mutating table* yang ditemukan.

DAFTAR PUSTAKA

DAFTAR PUSTAKA

[ALD03] Alder, Gaudenz, *Design and Implementation of the JGraph Swing Component*, Switzerland, 2003
http://www.jgraph.sourceforge.net/

[ALD05] Alder, Gaudenz, Benson, David, *JGraph User Manual*, 2005
http://www.jgraph.com/

[CAL05] Caldwell, Chris, *Graph Theory Glossary*, University of Tennessee at Martin, 2005
http://www.utm.edu/departments/math/graph/glossary.html

[CHR06] Wikipedia.org, *Chain Reaction,* wikipedia.org, 2006
http://en.wikipedia.org/wiki/Chain_reaction

[KAM03] Kamil, Amir, *Graph Algorithms*, UC Berkeley, California, 2003
www.cs.berkeley.edu/~kamil/sp03/041403.pdf

[RAC05] Rachmawati, Ruli Dyah, *Case Tool Pembangkit Script Trigger Untuk DBMS Oracle*, Teknik Informatika-ITS, Surabaya, 2005

[ROD05] Rodrigue, Jean-Paul, *Graph Theory: Definition and Properties*, Hofstra University, 2005
http://people.hofstra.edu/geotrans/eng/ch2en/meth2en/ch2m1en.html

[TCR01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms,* Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Section 22.3: Depth-first search, pp.540–549.
http://en.wikipedia.org/wiki/Depth_first_search

وَسِيقَ ٱلَّذِينَ ٱتَّقَوْا۟ رَبَّهُمْ إِلَى ٱلْجَنَّةِ زُمَرًا ۖ حَتَّىٰٓ إِذَا جَآءُوهَا وَفُتِحَتْ أَبْوَٰبُهَا وَقَالَ لَهُمْ خَزَنَتُهَا

سَلَٰمٌ عَلَيْكُمْ طِبْتُمْ فَٱدْخُلُوهَا خَٰلِدِينَ ﴿٧٣﴾

Dan orang-orang yang bertakwa kepada Tuhan dibawa ke dalam *jannah* berombong-rombongan (pula). Sehingga apabila mereka sampai ke *jannah* itu sedang pintu-pintunya telah terbuka dan berkatalah kepada mereka penjaga-penjaganya: "*Salaamun 'alaikum..* (Kesejahteraan dilimpahkan atasmu). Berbahagialah kamu.. maka masukilah *jannah* ini, sedang kamu kekal di dalamnya."

(*Al Qur'an surah Az Zumar 73*)

وَأُزْلِفَتِ ٱلْجَنَّةُ لِلْمُتَّقِينَ غَيْرَ بَعِيدٍ ﴿٣١﴾ هَٰذَا مَا تُوعَدُونَ لِكُلِّ أَوَّابٍ حَفِيظٍ ﴿٣٢﴾ مَّنْ خَشِىَ

ٱلرَّحْمَٰنَ بِٱلْغَيْبِ وَجَآءَ بِقَلْبٍ مُّنِيبٍ ﴿٣٣﴾ ٱدْخُلُوهَا بِسَلَٰمٍ ۖ ذَٰلِكَ يَوْمُ ٱلْخُلُودِ ﴿٣٤﴾ لَهُم مَّا يَشَآءُونَ

فِيهَا وَلَدَيْنَا مَزِيدٌ ﴿٣٥﴾

Dan didekatkanlah *jannah* itu kepada orang-orang yang bertakwa pada tempat yang tiada jauh. "Inilah yang dijanjikan kepadamu, kepada setiap hamba yang selalu kembali (kepada Allah) lagi memelihara (semua peraturan-peraturan-Nya)". "(yaitu) orang yang takut kepada Tuhan yang Maha Pemurah sedang dia tidak kelihatan (olehnya) dan dia datang dengan hati yang bertaubat". "Masukilah *jannah* itu dengan aman, Itulah hari kekekalan". Mereka di dalamnya memperoleh apa yang mereka kehendaki dan pada sisi kami ada tambahannya.

(*Al Qur'an surah Qaaf 31-35*)

مَّثَلُ ٱلْجَنَّةِ ٱلَّتِى وُعِدَ ٱلْمُتَّقُونَ ۖ فِيهَآ أَنْهَٰرٌ مِّن مَّآءٍ غَيْرِ ءَاسِنٍ وَأَنْهَٰرٌ مِّن لَّبَنٍ لَّمْ يَتَغَيَّرْ طَعْمُهُۥ

وَأَنْهَٰرٌ مِّنْ خَمْرٍ لَّذَّةٍ لِّلشَّٰرِبِينَ وَأَنْهَٰرٌ مِّنْ عَسَلٍ مُّصَفًّى ۖ وَلَهُمْ فِيهَا مِن كُلِّ ٱلثَّمَرَٰتِ وَمَغْفِرَةٌ مِّن رَّبِّهِمْ

كَمَنْ هُوَ خَٰلِدٌ فِى ٱلنَّارِ وَسُقُوا۟ مَآءً حَمِيمًا فَقَطَّعَ أَمْعَآءَهُمْ ﴿١٥﴾

(apakah) perumpamaan (penghuni) *jannah* yang dijanjikan kepada orang-orang yang bertakwa yang di dalamnya ada sungai-sungai dari air yang tiada berubah rasa dan baunya, sungai-sungai dari air susu yang tidak berubah rasanya, sungai-sungai dari *khamar* yang lezat rasanya bagi peminumnya dan sungai-sungai dari madu yang disaring; dan mereka memperoleh di dalamnya segala macam buah-buahan dan ampunan dari *Rabb* mereka, sama dengan orang yang kekal dalam *Jahannam* dan diberi minuman dengan air yang mendidih sehingga memotong ususnya?

(*Al Qur'an surah Muhammad 15*)