ALGORITMA ROCC

FAHREN BUKHARI

Departemen Matematika,
Fakultas Matematika dan Imu Pengetahuan Alam,
Institut Pertanian Bogor
Jln. Meranti, Kampus IPB Dramaga, Bogor 16680, Indonesia

SULASNO

Pusat Pengembangan Informatika Nuklir,
Badan Tenaga Nuklir Nasional (BATAN)
Jakarta, Indonesia

Abstrak: Algoritma concurrency control merupakan algoritma pengendalian akses konkurensi pada sistem sehingga objek yang diakses bersifat konsisten. Penelitian tentang concurrency control sudah dilakukan sejak 30 tahun lalu dan sudah banyak algoritma yang dihasilkan. Algoritma yang dihasilkan umumnya menggunakan asumsi bahwa Transaction Manager adalah satu-satunya modul yang digunakan pengguna untuk mengakses objek. Sekarang ini akses terhadap objek dilakukan orang tidak hanya melalui Transaction Manager, tetapi juga melalui aplikasi internet. Pola tingkah laku transaksi melalui aplikasi internet berbeda dengan aplikasi tradisional. Algoritma concurrency control yang ada seperti two phase locking kurang tepat dan berkinerja buruk pada aplikasi internet. Untuk itu dibutuhkan suatu algoritma concurrency control baru yang sesuai dengan aplikasi internet. Shi dan Perizzo memperkenal algoritma ROCC (Read-commit Order Concurrency Control). Banyak peneliti menilai algoritma ini sangat sesuai dengan aplikasi internet, tetapi algoritma ini melakukan restart yang tidak perlu. Penulis mencoba memperbaiki algoritma proses validasi sehingga restart dilakukan hanya pada akses atau transaksi yang tidak konsisten. Penelitian ini juga melakukan simulasi dalam upaya melihat perbedaan kinerja antara algoritma ROCC dan algoritma ROCC yang sudah diperbaiki.

Keyword: concurrency control, algoritma, ROCC, simulasi,.

1. PENDAHULUAN

Sistem berbasis objek menjadi banyak perhatian para peneliti di bidang komputasi, dan banyak produk yang sudah dihasilkan. Salah satu faktor utama yang membuat menarik para peneliti terhadap system berbasis objek adalah kesesuaian sistem tersebut terhadap aplikasi lanjut. Dengan berkembangnya

teknologi internet, jaringan komputer, dan perangkat keras pendukungnya, orang dapat bekerja bersama-sama dalam lingkungan *cooperative work environment* dan *computer assisted design*. Kedua hal lingkungan kerja ini melayani banyak orang atau pengguna yang bekerjasama untuk mencapai tujuan bersama, sehingga seringkali komponen-komponen atau objek-objek yang mereka akses saling tumpang tindih. Objek dalam hal ini dapat berupa data item, gambar, file, atau multi media objek (suara, photo, atau video).

Agar akses terhadap objek-objek tersebut dapat dilakukan *recovery* bila terjadi sesuatu, maka didefinisikan **transaksi**, yaitu kumpulan operasi-operasi terhadap objek (*write* dan *read*). Eksekusi transaksi secara *concurrence* (simultan) dibolehkan dalam sistem untuk meningkatkan kinerja. Semakin tinggi tingkat *concurrence* semakin baik kinerja sistem. Tetapi eksekusi transaksi yang tumpang tindih (*interleaving*) tidak dibolehkan, karena akan membuat objek dalam keadaan tidak konsisten. Untuk itu diperlukan *concurrency control* yang berfungsi untuk menata atau mengatur eksekusi transaksi yang simultan agar tidak saling tumpang tindih.

Banyak algoritma *concurrency control* yang sudah dipublikasikan. Algoritma-algoritma yang dipublikasikan umumnya dibuat khusus untuk sistem database. Pada sistem database tradisional, modul untuk mengakses data disebut *Transaction Manager* (suatu modul yang diinstalasi pada komputer user) adalah satu-satunya modul yang digunakan user untuk mengakses data. Selain itu algoritma yang ada mengasumsikan bahwa pola tingkah laku transaksi (pengguna) dapat diantisipasi. Salah satu algoritma tersebut dan banyak digunakan adalah *two phase locking*. Algoritma ini menggunakan mekanisme *locking* untuk mengatur eksekusi transaksi yang simultan. Algoritma ini dinilai kurang tepat atau berkinerja buruk bila digunakan pada sistem berbasis objek yang menggunakan aplikasi internet (aplikasi berbasis web) untuk mengakses objek, karena pola tingkah laku transaksi pada aplikasi internet sangat sulit diantisipasi. Suatu transaksi pada aplikasi berbasis web bisa saja tanpa diakhiri oleh *commit* atau *abort*, atau kalau diakhiri oleh *commit* atau *abort* mungkin dalam waktu yang cukup lama.

Shi dan Perizzo [10] memperkenalkan algoritma ROCC (*Read-commit Order Concurrency Control*) . Algoritma ini menggunakan struktur *RC-queue* untuk mengurut transaksi atau akses terhadap objek. Tetapi algoritma ini masih terdapat kelemahannya, yaitu algoritma ROCC melakukan *restart* atau *abort* pada suatu transaksi walaupun eksekusi transaksi tersebut tidak tumpang tindih dengan transaksi lain. Penelitian ini melakukan perbaikan terhadap algoritma ROCC sehingga *restart* atau *abort* hanya pada transaksi yang eksekusinya tumpang tindih dengan yang lain. Penelitian ini juga melakukan kajian simulasi untuk mengevaluasi kinerja algoritma yang diperbaiki dengan algoritma ROCC dan *two-phase locking*.

2. MODEL SISTEM BERBASIS OBJEK

Sistem berbasis objek dapat didefinisikan sebagai suatu kumpulan objek yang tersimpan secara terstruktur pada sistem komputer. Objek dicatat sebagai *x, y, z,* dst. Objek dapat berupa data, file, gambar, suara, atau potongan video.

Kumpulan objek tersebut seringkali saling berhubungan dan berkaitan satu dengan objek lainya, sehingga pada sistem tersebut terdapat suatu kendala integritas (*integrity* constraints) yang harus selalui dipenuhi oleh setiap objek. Sistem yang memenuhi kendala integritas disebut *consistent*. Sehingga akses terhadap sistem dapat dianggap sebagai proses transformasi sistem dari satu *consistent* ke *consistent* yang lain. Tetapi, sering kali ketika user mengakses objek untuk melakukan transformasi database ke *consistent* yang baru, kendala-kendala tersebut dilanggar secara temporer. Contohnya, bila user ingin melakukan transfer dana dari suatu rekening bank ke rekening bank yang lain, maka pelanggaran kendala konsistensi (bahwa jumlah saldo semua rekening sama dengan jumlah liabilitas bank) secara temporer tidak dapat dihindari, yaitu pada saat suatu rekening bank sudah didebit dan rekening bank yang lain belum di kredit. Atas dasar ini akses-akses terhadap sistem dikelompokkan menjadi suatu kesatuan konsistensi yang disebut **transaksi** (*transaction*). Dengan perkataan lain, transaksi memetakan sistem dari keadaan konsisten ke keadaan konsisten yang baru.

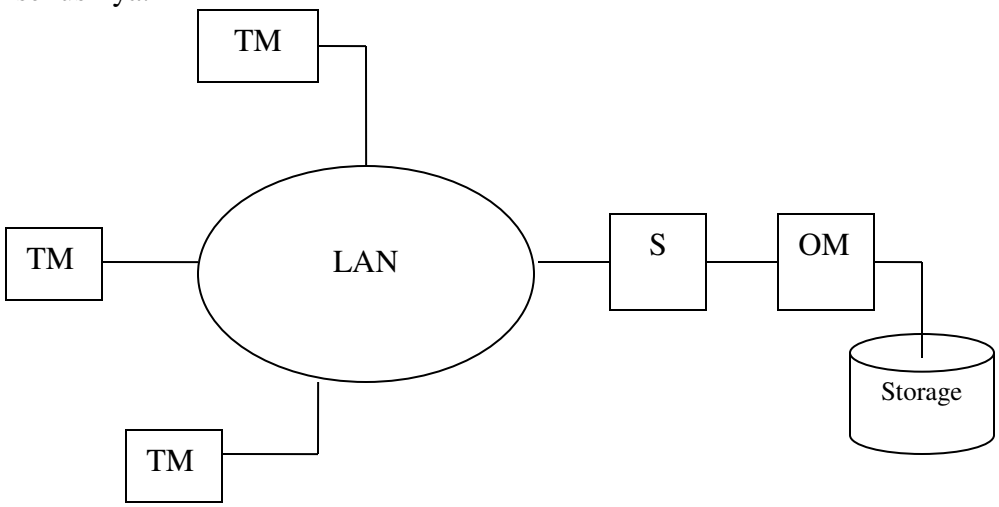
Untuk meningkatkan kinerja sistem, eksekusi transaksi-transaksi diperbolehkan simultan (*concurrence*). Tetapi eksekusi transaksi-transaksi yang tumpang tindih (*intervening*) tidak dibolehkan, karena eksekusi transaksi yang tumpang tindih ini akan membuat sistem tidak konsisten. Untuk itu sistem membutuhkan suatu mekanisme yang memantau dan mengendalikan eksekusi transaksi yang simultan sehingga tidak ada eksekusi transaksi yang tumpang tindih. Mekanisme tersebut dikenal sebagai *concurrency control*. Tujuan utama *concurrency control* adalah menjaga konsistensi sistem dan menjamin setiap transaksi dieksekusi tanpa melanggar propertisnya, yaitu *Automicity*, *Consistency, Isolation,* dan *Durability* (ACID).

Sistem terpusat terdiri dari lima komponen utama, yaitu transaksi, *Transaction Manager* (TM), *Scheduler*, *Objek Manager* (OM), dan *Storage*. TM berfungsi sebagai *front end object processing*. Eksekusi setiap transaksi dipandu oleh TM. Hal ini berarti transaksi dan operasi-operasinya diisukan oleh TM, kemudian TM mengirimnya ke *Scheduler*. Bentuk dan rancangan TM berorientasi pada aplikasi. *Scheduler* berada antara TM dan OM. *Scheduler* adalah modul yang mengatur urutan operasi pada objek yang diisukan oleh TM, dan *Scheduler* bertanggung jawab menjadwalkan operasi transaksi ke OM. OM adalah modul yang mengeksekusi operasi objek yang dikirim oleh *Scheduler*. OM bertanggung jawab bahwa operasi yang disampaikan kepadanya terekam pada *storage*. Sedangkan *storage* adalah kumpulan objek yang terintegrasi dan saling berhubungan.

3. ALGORITMA READ COMMIT ORDER CONCURRENCY CONTROL (ROCC)

Shi dan Perrizo ([10]) memperkenal suatu metode baru dalam *concurrency control*, yaitu metode yang menggunakan *Read commit queue* (*RC queue*) suatu struktur yang digunakan untuk mengurut transaksi yang konflik.

Concurrency control ini dirancang untuk sistem terpusat. Setiap *request* yang disampaikan oleh transaksi ke *scheduler* direpresentasikan oleh sebuah elemen pada *RC queue*. *Scheduler* menggunakan struktur *RC queue* untuk melakukan validasi transaksi yang akan melakukan *commit*. Transaksi yang gagal melalui proses validasi akan dibatalkan (*abort*) atau diulang (*Restart*) eksekusinya.



Figur 1 Model Sistem Berbasis Objek Secara Terpusat

Elemen-elemen pada *RC queue* terdiri diri dari empat jenis, yaitu elemen *Read*, *Commit*, *Validated*, dan *Restart*. Elemen *Read* merepresentasikan pesan yang dikirim oleh transaksi sebagai *request* untuk membaca objek. Elemen *Commit* merepresentasikan pesan *request* untuk *commit* dari transaksi. Elemen *Commit* dilengkapi dengan himpunan identitas objek dan perubahannya yang disimpan pada *storage*. Suatu transaksi diperbolehkan memiliki lebih dari satu elemen *Read*. Semua objek yang akan diubah oleh transaksi tersimpan pada elemen *Commit*. *Object Manager* mengeksekusi setiap operasi dengan urutan yang sesuai dengan urutan elemen pada *RC queue*. Dengan demikian urutan elemen di *RC queue* merepresentasikan urutan eksekusi yang sebenarnya, sehingga *RC queue* dapat digunakan untuk proses validasi transaksi. Proses validasi terhadap suatu transaksi akan dilakukan/dieksekusi bila transaksi tersebut mengirim *request* untuk *commit*. Bila validasi eksekusi transaksi gagal dan pengguna (*user*) masih ingin melanjutkan proses, maka sistem akan membuat elemen *Restart*. Elemen *Restart* ini berisikan semua identitas objek yang akan diakses dan diubah. Elemen *Validated* merepresentasikan transaksi yang berhasil melewati proses validasi atau transaksi yang tidak memerlukan proses validasi (*static* atau *restarted transaction*). *Static transaction* adalah transaksi yang mendeklarasikan semua objek yang akan diakses pada awal eksekusi transaksi, sehingga transaksi ini hanya memiliki satu elemen, yaitu elemen *Commit* pada *RC queue*. Bentuk umum elemen adalah sebagai berikut:

Tid	V	C	R	Reads	Writes	Next

Figur 2. Format Elemen RC queue

Tiap elemen berisikan atribut *Transaction id* (*Tid*), atribut jenis elemen, himpunan objek yang akan dibaca (*Reads*), himpunan objek yang akan ditulis (*Writes*), dan atribut untuk menyimpan *pointer* ke elemen berikutnya. Atribut *Tid* berisikan nomor identitas transaksi yang merupakan asal dari elemen. Atribut jenis elemen terdiri dari tiga bit, yaitu *V* bernilai 1 untuk elemen yang berhasil divalidasi, *C* bernilai 1 untuk elemen *Commit,* dan *R* bernilai 1 untuk elemen *Restart*.

Ketika *Scheduler* menerima pesan *request* dari transaksi T_i maka suatu elemen yang sesuai akan ditambahkan pada *RC queue*, kemudian operasi-operasi yang terkandung pada *request* atau elemen tersebut dikirim ke *Object Manager* untuk dieksekusi. *Scheduler* menjamin bahwa *Object Manager* mengeksekusi elemen dengan urutan yang sesuai dengan *RC queue* terutama untuk elemen yang konflik. Dua elemen dari transaksi yang berbeda dikatakan konflik bila terdapat paling sedikit satu operasi konflik dengan operasi yang terkandung pada elemen lain. Dua buah operasi dikatakan konflik bila paling sedikit satu operasi *write*.

Bila *scheduler* menerima pesan *commit* dari transaksi T_i, *scheduler* akan membuat elemen *commit* dan menambahkan di belakang *RC queue*. Selanjutnya *scheduler* akan melakukan proses validasi terhadap transaksi T_i. Proses validasi transaksi T_i dilakukan dengan menelusuri (mengamati) elemen-elemen transaksi lain pada *RC queue* yang berada diantara elemen *Read* pertama transaksi T_i dan elemen *commit* transaksi T_i. Bila ada elemen-elemen lain yang tumpang tindih eksekusinya (*interleaving*) dengan elemen transaksi T_i maka validasi transaksi T_i gagal, *scheduler* akan menghapus semua elemen transaksi T_i dan menempatkan elemen *Restart*. Elemen *Restart* berisikan semua objek yang akan diakses dan diubah. Atribut *V* (*validated*) dan atribut *R* (*Restart*) bernilai 1. Elemen ini tidak memerlukan proses validasi, sehingga setiap transaksi paling banyak hanya mengalami satu kali *restart*. Tetapi bila tidak ada elemen-elemen transaksi lain yang berada diantara elemen *Read* dan elemen *Commit* transaksi T_i yang tumpang tindih, maka proses validasi sukses, dan *scheduler* akan menghapus semua elemen-elemen transaksi T_i kecuali elemen *Commit.* Atribut *V* elemen *Commit* transaksi T_i diberi nilai 1, hal ini menunjukan elemen tersebut sudah berhasil divalidasi. Selanjutnya *Scheduler* mengirim operasi tulis (bila ada) transaksi T_i ke *Object Manager*. Agar *Object Manager* mengeksekusi operasi transaksi dengan urutan yang sesuai dengan urutan pada *RC queue*, maka *Scheduler* perlu menunda pengiriman operasi transaksi berikutnya yang konflik dengan transaksi T_i sampai *Object Manager* mengirim pesan *acknowledge* bahwa operasi tulis transaksi T_i sudah dieksekusi, dan *Scheduler* mengirim pesan ke transaksi bahwa *Commit* sudah dilakukan.

Elemen *Commit* transaksi T_i akan dihapus oleh *Scheduler* pada *RC queue* bila elemen ini berada pada posisi di depan (*front*) *queue*. Tetapi bila elemen *Commit* tidak (belum) berada diposisi depan pada *RC* queue, penghapusan elemen ini ditunda. Hal ini dilakukan karena elemen *Commit* transaksi T_i diperlukan keberadaannya pada *RC queue* untuk proses validasi bagi transaksi lain.

Shi dan Perizzo juga mengembangkan algoritma yang mereka sebut *"intervening" validation algorithm*. Algoritma ini digunakan untuk melakukan

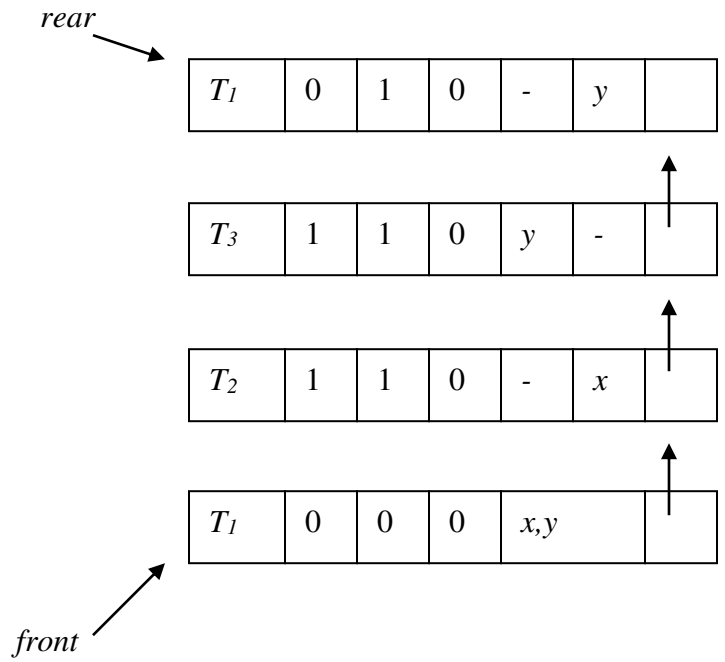
proses validasi suatu transaksi yang akan *commit*. Bila transaksi berhasil melewati proses validasi, maka proses *commit* akan dieksekusi. Tetapi bila transaksi gagal melewati proses validasi, maka transaksi akan dilakukan proses *restart* terhadap transaksi tersebut. Proses *restart* suatu transaksi adalah *Scheduler* akan memerintahkan *Object Manager* untuk membaca ulang semua objek yang diakses oleh transaksi tersebut, dan *Scheduler* mengirimnya ke *Transaction Manager* serta menawarkan ke user apakah proses akan dilanjutkan atau *abort*, dan *Scheduler* menghapus semua elemen milik transaksi tersebut dan menambahkan elemen *Restart* pada *RC queue*. Operasi perubahan objek oleh transaksi belum ada yang dieksekusi hanya operasi pembacaan objek yang sudah dieksekusi, sehinggak eksekusi *restart* tidak perlu melalui proses yang *roll back* yang rumit.

```
Conflict=0;" First"=NULL;" Second"=NULL; Success=1;
Search RC queue for transaction's first Read in queue;
If (Such is not found) Return Validated = success;
"First" = the first reached Read element;
While (1)
{ Compare "First" with all elements of other
     transactions behind it until it reaches an
     element of the same transaction;
  If (There is no element conflict)
  { Add "First" to reached element of same trans;
     Remove the "First" element from the RC queue;
     If (the Reached element is the commit element)
         Return Validated = success;
     "First" = the reached element;
  }
  Else
  {Insert "First" in RC queue before conflicting elt;
    Remove the original "First" from the RC queue;
    Conflict=1;
    "Second" = Commit element;
    While (1)
    {Compare"2nd" with all elts of other trans before
          it until it reaches an elt of same trans;
       If (There is no element conflict)
       {Add "2nd" elt to reached elt of same trans;
         Remove "Second" element from RC queue;
         If (reached element is "First")
            Return Validated = success;
         "Second" = the reached element;
       }
       Else
       { If (Conflict==1)
         { Remove all elements of trans from RC queue;
           return validated = failure;
} } }  } }
```

Figur 3. Algoritma validasi "Intervening"

Algoritma validasi "intervening" menggunakan konsep konflik elemen. Dua buah elemen dari transaksi yang berbeda dikatakan konflik bila paling sedikit terdapat satu buah operasi konflik dengan operasi elemen lainnya.

Proses validasi dilakukan algoritma *"intervening"* dengan menelusuri (*traversal*) struktur *RC queue*. Suatu transaksi akan gagal melewati proses validasi bila ada elemen-elemennya yang konflik dengan dua elemen dari transaksi lain (tidak harus berasal dari transaksi yang sama). Misalkan terdapat tiga buah transaksi yaitu $T_1 = \{r_1(x), w_1(y)\}$ dan transaksi statik $T_2=\{w_2(x)\}$ dan $T_3=\{r_3(y)\}$. Transaksi T_1 pertama kali datang dan mengirim *Read Request* untuk membaca objek *x* dan *y*. Kemudian datang transaksi statik T_2 melakukan perubahan sehingga transaksi T_2 hanya mengirim pesan *commit* dan nilai objek *x* yang akan diubah. Operasi transaksi T_2 langsung dieksekusi oleh *Scheduler*. Selanjutnya datang transaksi T_3 yang hanya membaca atau mengakses objek *y*. Terakhir transaksi T_1 mengirim *commit request*, sehingga struktur *RC queue* akan berisi seperti pada figur 4.



Figur 4. Struktur *RC queue*

Proses validasi transaksi T_1 dinyatakan gagal, karena dari *front* elemen transaksi T_1 konflik dengan elemen transaksi T_2; operasi read objek *x* ($r_1(x)$) konflik dengan operasi tulis objek *x* ($w_2(x)$). Kemudian dari *rear*, elemen transaksi T_1 konflik dengan elemen T_3 ; operasi tulis objek *y* ($w_1(y)$) konflik operasi baca *y* ($r_3(y)$). Sehingga transaksi T_1 mengalami *restart*. Padahal eksekusi transaksi-transaksi tidak ada yang tumpang tindih. Eksekusi transaksi-transaksi tersebut setara dengan eksekusi *serial*, yaitu $T_3 \rightarrow T_1 \rightarrow T_2$. Banyak contoh kasus yang

diberikan untuk menunjukan bahwa algoritma ROCC melakukan *restart* yang tidak perlu. Bila masalah ini bisa diatasi tentu akan diperoleh algoritma yang menghasilkan *restart* yang minimum dan *throughput* yang lebih baik.

4. PERBAIKAN ALGORITMA VALIDASI

Perbaikan algoritma ROCC dilakukan dengan mengubah prosedur validasi yang dilakukan oleh algoritma validasi intervening yang akan diuraikan dalam penjelasan di bawah ini. "First" adalah elemen operasi read dari transaksi yang melakukan commit. "Combine" adalah kumpulan elemen yang operasinya konflik dengan elemen commit ("Second") maupun operasinya konflik dengan elemen "Combine" sebelumnya. Sebagai inisialisasi awal "Combine" = 0. Langkah melakukan validasi setelah suatu transaksi mengirimkan commit request, pada ROCCM selengkapnya adalah sebagai berikut :

1. Bandingkan "First" dengan elemen dari transaksi lain yang terdapat diantara "First" sampai elemen commit. Bila pada saat penelusuran menemukan elemen read dari transaksi yang sama (first-down reached element) maka gabungkan elemen "First" ke dalam elemen transaksi yang sama berikutnya. Kemudian bandingkan "First" hasil gabungan tersebut, dengan elemen dari transaksi lain yang terdapat diantara "First" sampai elemen commit. Proses penelusuran dilakukan terus untuk menemukan elemen yang konflik atau elemen read berikutnya. Bila transaksi berikutnya yang ditemukan, adalah elemen commit dari transaksi yang sama, dan tidak terdapat konflik maka validasi dinyatakan sukses.

2. Jika "First" konflik dengan elemen dari transaksi lain, pindahkan elemen "First" ke posisi transaksi sebelum elemen dari transaksi lain yang konflik. Hapus elemen "First" yang asli dari RC-queue.

3. Bandingkan "Second" atau "Combine" dengan elemen dari transaksi lain yang terdapat diantara "Second" sampai ditemukan elemen dari transaksi yang sama ("First up reached element"). Setiap elemen yang konflik, lakukan insert ke "Combine".

4. Bandingkan "Combine" dengan "First up reached element" Jika terdapat konflik maka validasi dinyatakan gagal. Jika tidak terdapat konflik lakukan pengecekan apakah "First up reached element" adalah "First", jika merupakan "First maka validasi dinyatakan sukses, tetapi jika bukan elemen "First" lanjutkan langkah 5.

5. Gabungkan "Second" dengan "First up reached element" hapus "Second" asli dari RC-queue. Lanjutkan langkah 3.

Langkah-langkah prosedur validasi bagi algoritma ROCC yang diperbaiki (disebbut algoritma ROCCM) lebih rincinya ditulis pada *pseudo-code* yang diperlihatkan pada figur 5.

```
Conflict=0; Combine = Null; Success=1;
First = The transation's first read element in rc queue;
Second = The transaction's commit element;
NextElement = get next element in rc queue before First;
IF (First==Null) Return Validated = success;
WHILE (1)
   IF (NextElement is the next read element of the transaction)
      Remove First read in the rc queue;
      First = Merge First and NextElement;
      Replace NextElement with First in the rc queue;
   ELSE IF (NextElement == Second)
      Return Validated = success;
   ELSE IF (First conflict with NextElement)
      NextElement = get previous element of the transaction's
commit
                    element in the rc queue;
      WHILE (1)
         IF (NextElement is First)
            IF (Combine conflict with First)
               Return Validated = Failure;
            ELSE
               Remove Second in the rc queue
               Second = Merge Second with First;
               Replace First with Second;
               Return Validate = Success;
            END IF
         ELSE IF (NextElement is read-up element)
            IF (Combine conflict with NextElement)
               Return Validated = Failure;
            ELSE
               Remove Second in the rc queue;
               Second = Merge NextElement and Second;
               Replace NextElement with Second;
               NextElement = get previous element in the rc
queue;
            END IF
         ELSE IF (NextElement conflict with Combine or Second)
            Insert NextElement into Combine;
            NextElement = get previous element;
         END IF
      END WHILE
   ELSE
      NextElement = get next element in the rc queue;
   END IF
END WHILE
```

Figur 5. Algoritma validasi "intervening" ROCCM.

Kinerja algoritma ROCC yang sudah diperbaiki (ROCCM) dibandingkan dengan algoritma ROCC dan *Two-phase Locking* melalui simulasi.

5. MODEL SIMULASI ROCC

5.1. Asumsi-asumsi simulasi. Simulasi dilakukan dengan melihat *throughput*; yaitu banyaknya transaksi yang dapat diselesaikan per satuan waktu, *restart ratio*; yaitu ratio banyaknya transaksi yang mengalami restart per satuan waktu, dan *response time*; yaitu waktu diantara ketika sebuah terminal mengirim sebuah transaksi baru dan ketika hasil transaksi baru tersebut dikembalikan ke terminal. Kajian simulasi dilakukan pada beberapa level multiprograming.. Level multiprogramming adalah level yang menunjukan maksimum banyaknya transaksi yang dilayani server, bila suatu transaksi melakukan request awal, dan ternyata jumlah transaksi yang dilayani server sudah maksimum, maka transaksi tersebut diblok sampai jumlah transaksi yang dilayani berkurang.

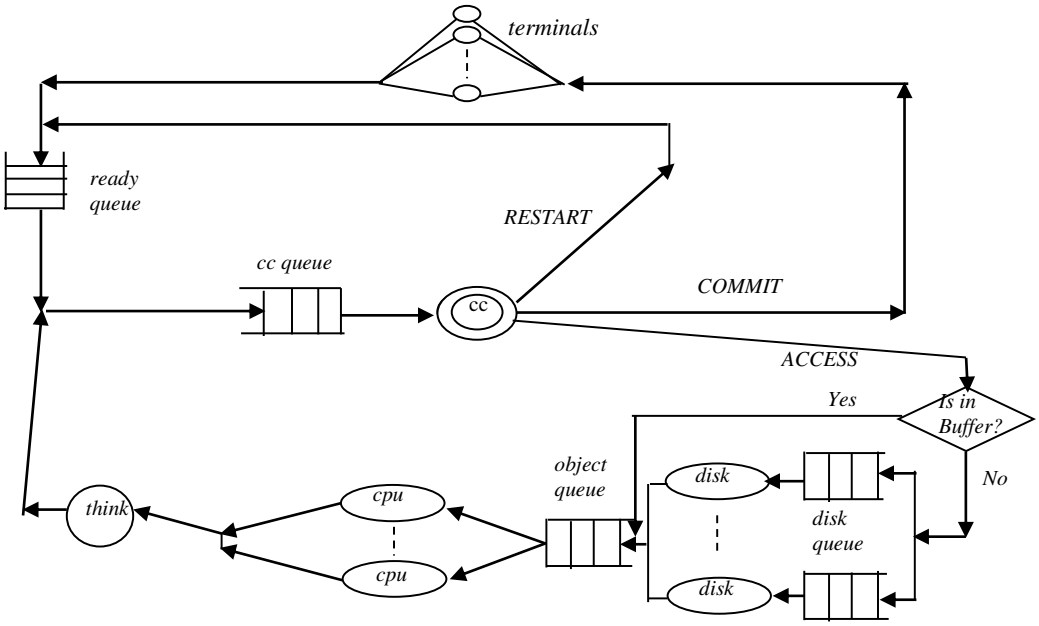
Simulator yang digunakan untuk mengukur kinerja algoritma *concurrency control* (CC) tersebut, menggunakan model antrian tertutup pada suatu sistem basis data terpusat yang diambil dari penelitian sebelumnya (Shi & Perrizo 2003). Simulator tersebut diperlihatkan pada Figur 6 dan Figur 7. Pada simulator terdapat sejumlah terminal, untuk membangkitkan transaksi.

Selanjutnya terdapat batasan transaksi yang aktif pada suatu saat di dalam sistem, *mpl*. Apabila transaksi yang digenerasi oleh sejumlah terminal melebihi transaksi yang aktif, yang diijinkan oleh sistem (melebihi *mpl*), maka transaksi tersebut diletakkan dalam *ready queue* untuk menunggu transaksi dalam sistem selesai atau ada yang di*abort*. Sebaliknya apabila transaksi yang aktif dalam sistem tidak melebihi *mpl,* maka transaksi yang digenerasi oleh terminal atau transaksi yang antri di *ready queue*, masuk ke *cc queue* (*concurrency control queue*) dan membuat permintaan operasi akses basis data melalui CC. Jika lolos dari seleksi yang dilakukan oleh CC**,** selanjutnya transaksi tersebut mengakses data. Jika data tersebut ada di *buffer* maka eksekusi dilanjutkan ke CPU. Tetapi jika data tersebut tidak terdapat di *buffer* maka eksekusi akan dilewatkan ke disk untuk mengakses data yang seterusnya dilanjutkan ke CPU. Untuk menuju ke *disk* dan *cpu* harus melalui antrian di disk dan CPU yaitu *disk_queue* dan *cpu_ queue.*

Pada simulasi juga diasumsikan bahwa sebuah transaksi melakukan operasi *read* terlebih dahulu sebelum melaksanakan operasi *write*. Diasumsikan juga jaringan yang digunakan adalah jaringan *Local Area Network* (LAN) dalam keadaan handal pada saat transmisi data dari terminal ke server. Jalur *think* memberikan nilai *random delay* pada waktu mengakses item data. Pada **Strict 2PL**, jika hasil dari CC memutuskan bahwa suatu transaksi harus di *block* maka transaksi tersebut dimasukkan dalam *block queue* sampai permintaan akses data dapat diproses. Jika CC menetapkan untuk melakukan *restart* pada suatu transaksi, maka transaksi tersebut akan di*restart* dan selanjutnya dimasukan ke dalam *ready queue*. Jika suatu transaksi telah komplit (selesai) maka CC akan memberikan *commit* pada transaksi tersebut dan mengirimkan *commit succes message* ke terminal.

Pada simulator terdapat dua model *logical queuing* yaitu yang pertama model antrian untuk **ROCCM** dan **ROCC** yang diambil dari penelitian sebelumnya (Shi & Perrizo 2003), serta yang kedua model antrian untuk **Strict 2PL**.

Figur 6 di bawah ini memperlihatkan model antrian yang pertama pada simulator.

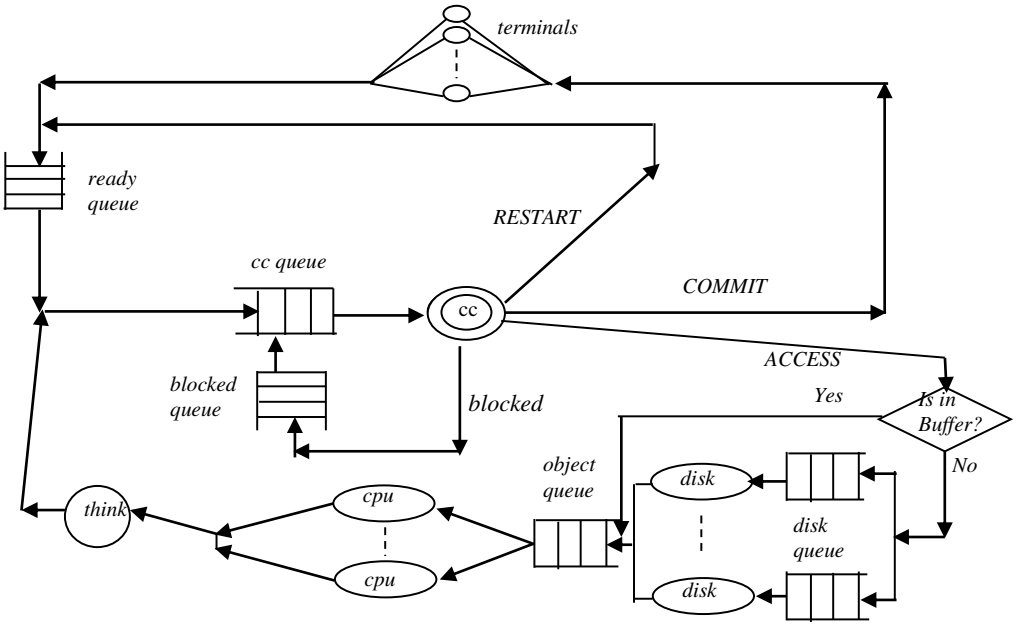


Figur 6. Model *logical queuing* untuk **ROCCM** dan **ROCC** pada simulator (Shi & Perrizo 2003).

Pada figur 6 diperlihatkan sejumlah terminal untuk membangkitkan transaksi. Ketika suatu transaksi baru dibangkitkan, sistem akan melewatkan transaksi pada *ready queue* untuk diteruskan ke *cc queue*. Transaksi yang lolos validasi akan dilanjutkan untuk mengakses data di *buffer* atau *disk.* Setelah mendapatkan data operasi transaksi diteruskan ke *cpu*. Setelah dieksekusi oleh *cpu* terdapat pemberian nilai *int_think* dan *ext-think* diantara transaksi untuk selanjutnya masuk ke *cc queue*. Transaksi yang gagal validasi akan dilakukan *restart* dan kembali masuk *ready queue*. Eksekusi transaksi yang telah komplit akan dilaporkan ke terminal.

Figur 7 memperlihatkan model antrian untuk **Strict 2PL** pada simulator. Model tersebut merupakan modifikasi dari model *logical queuing* penelitian sebelumnya dan ditambahkan dengan jalur transaksi yang mengalami *blocked* serta *blocked queue* untuk menampung transaksi yang mengalami *blocked*. Eksekusi antrian hampir sama dengan model antrian pada **ROCC** dan **ROCCM**. Perbedaannya pada model antrian **Strict 2PL** adalah terdapat *block queue* untuk menampung transaksi yang mengalami *blocked* untuk kemudian diteruskan ke *cc queue.*

5.2 Input Simulasi. Parameter input yang digunakan pada pelaksanaan simulasi diperlihatkan pada Tabel 4. Nilai parameter input yang terdapat pada table 2 adalah nilai *default* yang terdapat pada simulator (Shi & Perrizo 2003). Pada pelaksanaan simulasi dilakukan perubahan beberapa nilai dari nilai *default* untuk beberapa parameter, guna mendapatkan hasil simulasi yang maksimal.



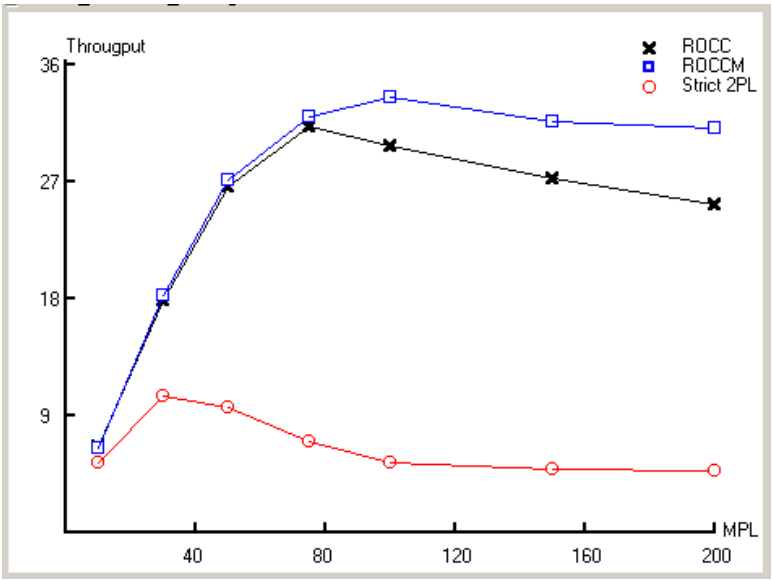
Figur 7. Model *logical queuing* untuk **Strict 2PL** pada simulator, modifikasi dari model Shi dan Perrizo (2004).

Tabel 2. Nilai parameter input *default* pada pelaksanaan simulasi

No	Parameter	Nilai	Keterangan
1	*db_size*	1000 *pages*	Ukuran database
2	*max_trans*	12 *pages*	Ukuran maksimum transaksi
3	*min_trans*	4 *pages*	Ukuran minimum transaksi
4	*write_prob*	0,25	Peluang banyaknya data item yang di*update*
5	*int_think*	1 *ms*	Rata-rata lamanya internal *think*
6	*xt_think*	1 ms	Rata-rata eksternal *think*
7	*max_req*	3	Maksimum *request* suatu transaksi
8	*mean_time*	3	Rata-rata antar kedatangan transaksi
9	*commit_num*	800	Akhir simulasi, setelah 800 transaksi melakukan *commit*
10	*hit_ratio*	0,5	Peluang suatu objek ada di buffer
11	*obj_io*	35 *ms*	Lamanya mengakses suatu objek pada hard disk
12	*obj_cpu*	15 *ms*	Lamanya menggunakan cpu untuk satu objek

13	*num_cpu*	4	Banyaknya cpu
14	*num_disk*	8	Banyaknya hard disk
15	*Mpl*	5,10,25,50, 75,100,200	Tingkat multiprogramming

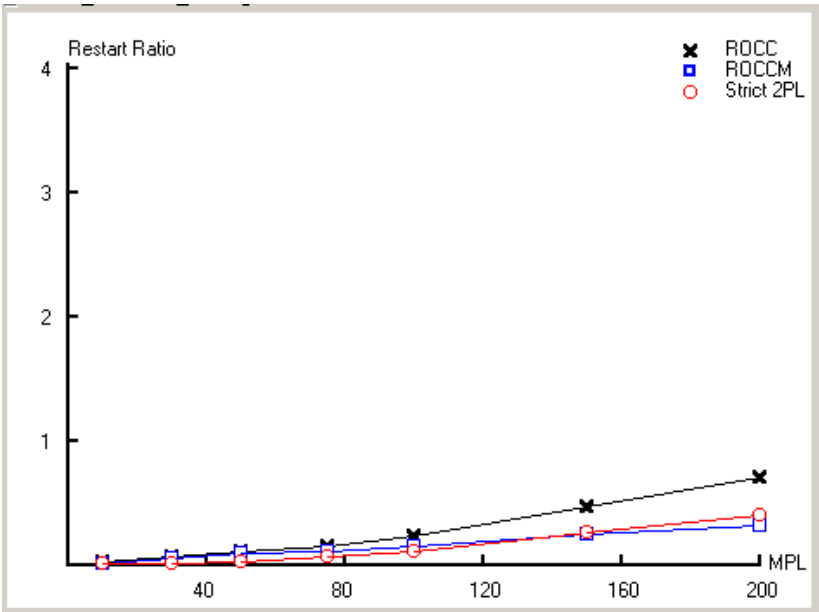
Hasil simulasi menunjukkan bahwa secara umum algoritma ROCCM lebih baik dari algoritma ROCC atau *strict 2pl*. dapat dilihat pada figur 8. Pada tingkat mpl 150 keatas perbedaan algoritma ROCCM dan ROCC berbeda nyata, sedangkan untuk tingkat mpl 150 kebawah walaupun terlihat *throughput* ROCCM lebih baik dari yang lain, tetapi tidak berbeda nyata. Bentuk kecenderungan grafik ROCCM maupun ROCC akan menurun disekitar tingkat mpl 200 atau lebih, karena banyaknya transaksi yang mengalami restart.



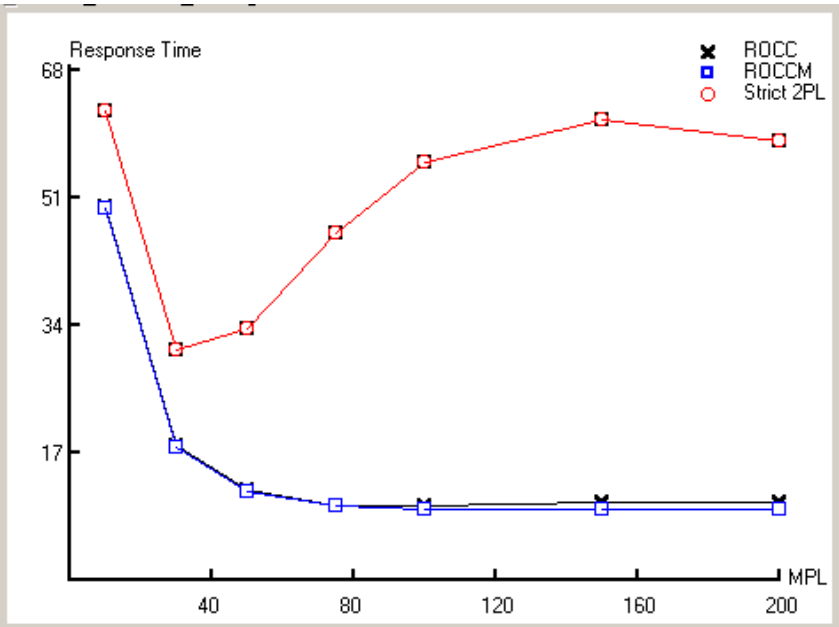
Figur 8. *Throughput* masing-masing algoritma pada berbagai tingkat mpl.

Figur 9 memperlihatkan *restart-ratio* (banyaknya transaksi yang di*restart* persatuan waktu) tiap-tiap algoritma. Terlihat dengan jelas bahwa hampir pada tiap tingkat mpl, *restart-ratio* algoritma ROCCM lebih rendah dibanding *restart-ratio* algoritma ROCC. Artinya banyak transaksi yang di*restart* pada algoritma ROCC lebih banyak daripada algoritma ROCCM.

Algoritma ROCCM dan ROCC memberikan *response time* (waktu antara transaksi dibangkitkan dan transaksi selesai) yang lebih baik dibanding algoritma *strict 2pl* (figur 10). Hal ini berkenaan dengan bahwa algoritma *strict 2pl* cenderung melakukan penundaan eksekusi transaksi, bila *lock* objek yang diinginkan transaksi tersebut tidak tersedia. Sedangkan algoritma ROCCM atau ROCC cenderung mengeksekusi setiap transaksi dan validasi dilakukan pada saat transaksi tersebut ingin melakukan *commit*. Bila eksekusi transaksi tersebut tumpang tindih, maka transaksi tersebut di*restart*.



Figur 9. *Restart-ratio* masing-masing algoritma pada berbagai tingkat mpl.



Figur 10. *Response time* masing-masing algoritma pada beberapa tingkat mpl.

6. PENUTUP

Setelah dilakukan perbaikan terhadap prosedur validasi algoritma ROCC yang diperkenalkan oleh Shi dan Perrizo terbukti algoritma ROCC memberikan hasil yang lebih baik. Hasil kajian simulasi juga menunjukan bahwa algoritma ROCC lebih baik kinerjanya dibanding algoritma *strict 2pl*.

Pada saat ini algoritma ROCC hanya dirancang untuk sistem database terpusat. Sehingga rancangan algoritma ROCC untuk sistem database terdistribusi (*distributed database system*) adalah suatu yang perlu dilakukan.

DAFTAR PUSTAKA

[1]. Agrawal, R., M.J. Carey, and M. Livny, "Concurrency Control Performance Modelling: Alternatives and Implications", ACM Transactions on Database Systems, vol. 12, no. 4, December 1987, p.609-654.

[2]. Bernstein, P.A., V. Hadzilacos, dan N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987

[3]. Bukhari, Fahren, *Two Fully Distributed Concurrency Control Algorithms,* Department of Computer Science, Univeristy of Western Ontario, 1990.

[4]. Bukhari, Fahren dan S.L. Osborn, "Two Fully Distributed Concurrency Control Algorithms", IEEE Transactions of Knowledge and Data Engineering, 5(5): 872 – 882, 1993.

[5]. Franaszek, P.A., J.T. Robinson, dan A. Thomasian, "Concurrency Control for High Contention Environments", *ACM Transactions on Database Systems*, vol. 17, No. 2, June 1992, p.304-345.

[6]. Peter Graham dan Ken Barker, "Effective Optimistic Concurrency Control in Multiversion Object Bases", Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, 1994

[7]. Kung, H.T. dan John T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM transactions on Database Systems*, vol. 6, No. 2, June 1981, pages 213-226.

[8]. M.T. Özsu dan P. Valduriez, *Principles of distributed database systems,* Prentice Hall, 1999.

[9]. Papadimitriou, C., *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[10]. Shi, Victor T. S. & William P. "A New Method for Concurrency Control in Centralized Database Systems." http://www.cs.ndsu.nodak.edu/~perrizo/classes/766/rocc.doc. [14 Juni 2003]

[11]. Basu, Samidip. "Application of Snapshot Isolation and ROCC in ADO/ASP.NET".