ALGORITMA PENGENDALI KONKURENSI TERDISTRIBUSI (DROCC)

FAHREN BUKHARI

Departemen Matematika,
Fakultas Matematika dan Ilmu Pengetahuan Alam,
Institut Pertanian Bogor
Jl. Meranti, Kampus IPB Darmaga, Bogor, Indonesia

ABSTRAK. Penelitian ini memperkenalkan algoritma pengendalian konkurensi untuk sistem basis data terdistribusi atau dikenal dengan sebutan DROCC (*Distributed Read commit Order Concurrency Control*), karena algoritma DROCC merupakan pengembangan algoritma ROCC (*Read commit Order Concurrenct Control*) yang diperkenalkan oleh Shi dan Perizzo untuk sistem basis data terpusat. Sama halnya dengan ROCC, algoritma DROCC mengurut eksekusi transaksi tanpa menggunakan mekanisme *locking*, tetapi menggunakan struktur *Read Commit queue* (RC-*queue*) untuk mengurut akses terhadap basis data lokal dan menggunakan struktur *serial graph* untuk mengurut transaksi secara global. Proses validasi pada algoritma DROCC terdiri dari proses validasi lokal dan proses validasi global. Proses validasi lokal DROCC merupakan penyempurnaan proses validasi ROCC. Sedangkan proses validasi global memanfaatkan struktur *serial graph* yang dibangkitkan dari RC-*queue*. Pada penelitian ini mekanisme penghapusan transaksi yang sudah tervalidasi juga dirancang.Algoritma DROCC memiliki *feature*, (i) optimistik, setiap *request* langsung dieksekusi tanpa penundaan yang berarti, (ii) bebas *deadlock* baik lokal maupun global, (iii), masing-masing situs memiliki *full autonomy*.

Keyword: DROCC, ROCC, *distributed concurrency control*, *free locking*, *Serial Graph.*

1. PENDAHULUAN

Area penelitian pengendali konkurensi (*concurrency control*) telah lama menjadi perhatian para peneliti, seiring dengan dikenalnya teknologi basis data. Hal ini dicerminkan dengan banyaknya algoritma pengendali konkurensi yang diperkenalkan pada literatur. Salah satu algoritma pengendali konkurensi yang banyak dibicarakan para ilmuwan komputer adalah *Two-Phase Locking* (2PL). Algoritma ini menggunakan mekanisme *locking* untuk mengatur eksekusi transaksi. Kinerja algoritma pengendali konkurensi 2PL dinilai baik pada sistem basis data terpusat (*centralized database systems*), tapi tidak pada basis data terdistribusi (*distributed database systems*), apalagi pada sistem berbasis *web*. Shi dan Perizzo melakukan kajian tentang hal ini dalam [Shi04a], dan mereka juga

memperkenal algoritma pengendali konkurensi ROCC (*Read commit Order Concurrency Control*). Algoritma ROCC ini menarik perhatian banyak ilmuwan ilmu komputer, karena algoritma ini tidak menggunakan mekanisme *locking*, tetapi menggunakan struktur RC-*queue* (*Read Commit queue*). Algoritma ROCC ini dirancang untuk sistem basis data terpusat.

Perkembagan teknologi internet akhir-akhir ini memungkinkan akses terhadap basis data melalui aplikasi berbasis *web*. Shi dan Perizzo ([Shi04a]) menunjukan bahwa algoritma pengendali konkurensi yang menggunakan algoritma *locking* berkinerja buruk pada aplikasi berbasis *web*. Sehingga kebutuhan akan algoritma pengendali konkurensi yang berkinerja baik pada sistem berbasis *web* sangat diperlukan. Penelitian ini memperkenalkan suatu algoritma pengendali konkurensi terdistribusi DROCC (*Distributed Read commit Concurrency Control*). Algoritma ini tanpa menggunakan mekanisme *locking*, tetapi menggunakan struktur RC-*queue* (*Read Commit queue*) dan *serial graph*. Struktur RC-*queue* merepresentasikan urutan eksekusi elemen-elemen semua transaksi secara lokal, sedangkan *serial graph* merepresentasikan urutan eksekusi transaksi secara global. Struktur RC-*queue* diperkenalkan oleh Shi dan Perizzo ([Shi04a]), sedangkan *serial graph* didefinisikan cukup jelas dalam [Ber87].
Algoritma DROCC yang diperkenalkan pada penelitian ini memiliki *feature* sebagai berikut:

i. Bersifat optimistik, yaitu setiap *request* dari transaksi cenderung langsung dieksekusi. Bila sistem mendeteksi eksekusi suatu transaksi tumpang tindih dengan yang lain, maka sistem melakukan *abort* terhadap transaksi tersebut.

ii. Bebas dari masalah *deadlock* baik lokal maupun global.

iii. Setiap situs memiliki otonomi penuh dalam mengatur akses terhadap basis data lokal.

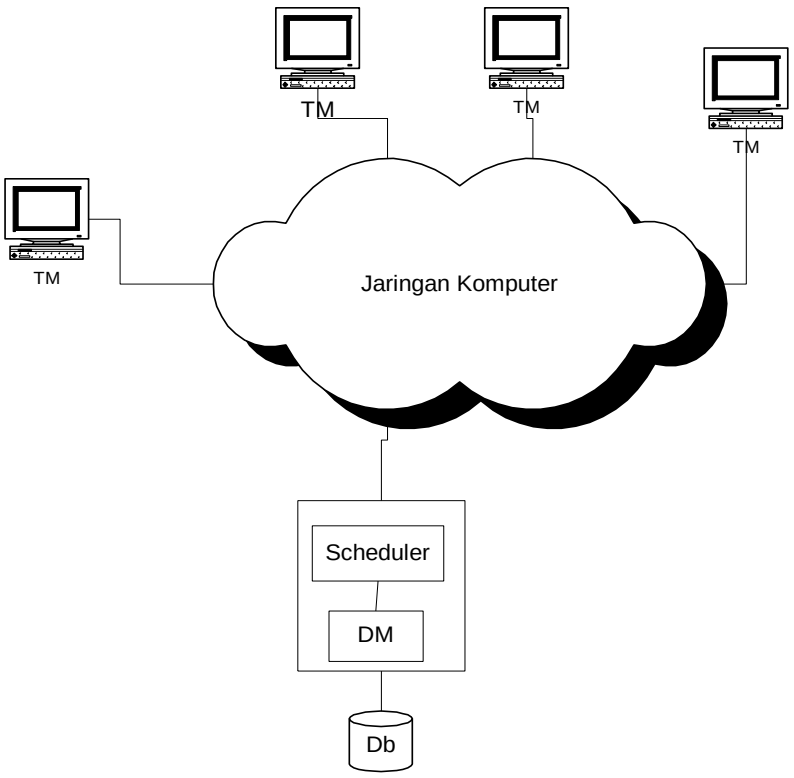
Selama ini kendala utama dalam implementasi sistem basis data terdistribusi ialah masalah pengendali konkurensi (*concurrency control*). Banyak algoritma pengendali konkurensi yang sudah diperkenalkan pada sistem basis data terpusat, tetapi pada saat dikembangkan untuk sistem basis data terdistribusi mengalami banyak masalah atau berkinerja buruk. Berdasarkan analisis penulis, algoritma DROCC ini akan mempermudah implementasi sistem basis data terdistribusi baik untuk aplikasi yang berbasis *web* maupun aplikasi tradisional. Sehingga salah satu kontribusi yang utama dalam penelitian ini ialah memperkenalkan suatu algoritma atau algoritma pengendali konkurensi untuk sistem basis data terdistribusi. Kontribusi lainnya ialah penyempurnaan prosedur validasi ROCC, memperkenalkan prosedur validasi global, dan prosedur pembangkitan *serial graph*.

2. DEFINISI ISTILAH

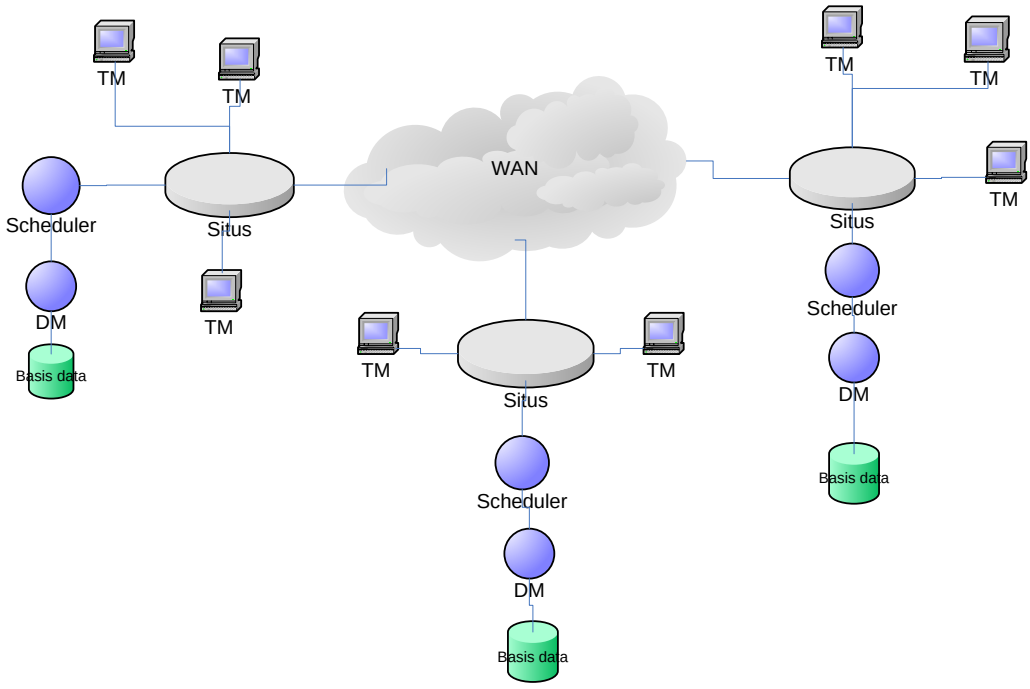
2.1. Arsitektur

Model sistem basis data yang digunakan pada penelitian ini mengacu pada model yang dibahas Bernstein dalam [Ber87]. Pada model yang diajukan Bernstein terdiri dari empat komponen, yaitu *Transaction Manager*, *Scheduler*, *Data Manager*, dan *Database* (basis data).

Transaction Manager (TM) adalah kumpulan program yang berfungsi sebagai *user interface*. TM menginterpretasikan transaksi dan mengirim *request* ke *Scheduler*. *Scheduler* berfungsi menjadwalkan operasi–operasi dari setiap transaksi, menjaga kekosistenan basis data, dan menjamin setiap transaksi memenuhi ACID (Atomik, *Consistency*, *Isolation*, dan *Durability*). Algoritma pengendali konkurensi dalam *scheduler*. *Data Manager* (DM) berfungsi mengeksekusi setiap operasi yang dikirim oleh *scheduler*, dan mengirim kembali hasilnya. *Scheduler* dan DM terletak pada situs yang sama. Terakhir *Database* (basis data) adalah kumpulan basis data relasional.



Gambar 1. Model Sistem Basis Data



Gambar 2. Model sistem basis data terdistribusi

Pada sistem basis data terdistribusi, basis data ditempatkan lebih dari satu situs. Pada sistus basis data tersimpan juga terdapat *scheduler* dan *Data Manager*. Masing-masing *scheduler* bertanggungjawab mengatur atau menjadwalkan akses terhadap basis data lokal. Sedangkan *Data Manager* (DM) bertugas mengeksekusi elemen atau operasi-operasinya yang dikirim dari *scheduler*.

2.2.Transaksi

Özsu dan P. Valduriez dalam [Özs99] mendefinisikan transaksi adalah kumpulan operasi baca atau tulis data item. Sehingga suatu transaksi T_1 yang membaca x dan menulis y dinotasikan sebagai $T_1 = \{r_1(x), w_1(y)\}$. Pada penelitian ini digunakan asumsi bahwa operasi tulis transaksi dilakukan diakhir eksekusi transaksi tersebut, yaitu pada saat transaksi melakukan proses *commit*. Untuk penyederhanaan masalah pada penelitian ini suatu transaksi didefinisikan sebagai kumpulan elemen-elemennya pada RC-*queue* (*Read Commit queue*). Sehingga transaksi T_1 pada saat setelah mengirim *commit request* dapat didefinisikan sebagai $T_1 = \{e_1^1(\{x, y\}), e_1^2(\{y\})\}$. Sedangkan pada saat transaksi berhasil melewati proses validasi lokal, maka transaksi T_1 didefinisikan sebagai $T_1 = \{e_1^3(\{x, y\}, \{y\})\}$. Bila transaksi T_1 berhasil melewati proses validasi global, maka transaksi tersebut didefinisikan sebagai $T_1 = \{e_1^4(\{x, y\}, \{y\})\}$. Walaupun definisi transaksi berubah-ubah berdasarkan kandungan elemen yang dimiliki transaksi tersebut pada RC-*queue*, tetapi definisi transaksi tidak berubah berdasarkan operasi dan data item yang diaksesnya.

Suatu transaksi mungkin saja memiliki lebih dari satu elemen *read*, tetapi hanya satu elemen *commit*. Bila transaksi selesai melakukan proses validasi dan berhasil, maka transaksi tersebut memiliki satu elemen yaitu elemen *validated*. Semua elemen transaksi tersebut dihapus, dan posisi elemen *commit* diganti dengan elemen *validated*. Sedangkan himpunan data item yang ditulis merupakan himpunan yang berasal dari elemen *commit*, sedangkan himpunan data item yang dibaca merupakan gabungan himpunan *rs* semua elemen *read*.

Transasi statik, adalah transaksi yang hanya mengirim *read request* satu kali, tidak perlu melewati proses validasi. Transaksi ini dalam RC-*queue* hanya memiliki satu elemen, yaitu elemen *validated* bagi transaksi yang membaca data item saja sehingga tidak memerlukan proses validasi lokal maupun global.

Setiap transaksi memiliki identifikasi yang unik. Identifikasi transaksi dapat terdiri dari asal situs dan nomor urut transaksi yang dibangkitkan oleh situs asal.

2.3. Elemen

Elemen didefinisikan sebagai himpunan data item yang diakses. Pada penelitian ini didefinisikan tiga buah elemen yang berbeda, yaitu elemen *read* (*read element*); dinotasikan sebagai $e_i^1(rs_i)$, dimana *i* merepresentasikan transaksi T_i, sedangkan rs_i adalah himpunan data item yang dibaca oleh transaksi bersangkutan. Elemen *read* dibangkitkan oleh *scheduler* atas respon pesan *read request* dari transaksi. Suatu transaksi dapat mengirim lebih dari satu pesan *read request* ke *scheduler*. Elemen *local commit* (*local commit element*); dinotasikan sebagai $e_i^2(ws_i)$; dimana ws_i adalah himpunan data item yang ditulis oleh transaksi bersangkutan. Elemen *local commit* dibangkitkan *scheduler* atas respon pesan *commit request* dari transaksi. Pesan *commit request* berisikan daftar data item yang akan diubah beserta nilainya.

Setelah elemen *local commit* dibangkitkan *scheduler*, proses validasi lokal dilakukan bagi transaksi tersbut. Bila transaksi berhasil melewati proses validasi lokal dan $ws_i \neq \phi$, maka elemen *local commit* berubah menjadi elemen *global commit*; dinotasikan sebagai $e_i^3(rs_i, ws_i)$, dimana rs_i adalah himpunan data item yang dibaca dan ws_i adalah himpunan data item yang akan ditulis oleh transaksi. Elemen *global commit* menunjukkan transaksi pemilik elemen tersebut sedang melakukan proses validasi global. Tidak semua transaksi memerlukan proses validasi global. Transaksi yang hanya mengakses basis data lokal (*read only transaction*) tidak memerlukan proses validasi global, tetapi hanya proses validasi lokal dan elemen *local commit* diubah menjadi elemen *validated* bila transaksi berhasil melewati proses validasi lokal.

Bila suatu transaksi berhasil melewati proses validasi global, maka dibangkitkan elemen *validated* (*validated element*); dinotasikan sebagai $e_i^4(rs_i, ws_i)$; dimana rs_i adalah himpunan data item yang dibaca dan ws_i adalah himpunan data item yang tulis oleh transaksi bersangkutan. Transaksi statik, yaitu transaksi yang mengakses baca

saja dan mengirim hanya satu pesan *read request*, tidak memerlukan proses validasi lokal maupun global. Sehingga *local scheduler* pada saat menerima pesan *read request* dari transaksi statik, langsung membangkitkan elemen *validated*.

Dua elemen dikatakan konflik, bila kedua elemen tersebut berasal dari transaksi yang berbeda, mengakses data item yang sama, dan salah satu atau keduanya melakukan akses tulis. Sehingga secara formal dua elemen e_i^p dan e_j^q dikatakan konflik bila $i \neq j$, $p=1$, $q=2,3,4$, maka $rs_i \bigcap ws_j \neq \phi$, atau bila $p=2,3,4$, maka $rs_i \cap ws_j \neq \phi$, $ws_i \cap rs_j \neq \phi$, atau $ws_i \bigcap ws_j \neq \phi$.

Setiap elemen memiliki identifikasi unik. Identifikasi elemen dapat terdiri dari identifikasi transaksi dan nomor urut elemen dalam transaksi tersebut.

2.4. RC-queue

Bila transaksi T_i akan mengakses basis data, maka transaksi tersebut mengirim pesan atau *request*. Setiap *request* yang diterima, *scheduler* akan membangkitkan suatu elemen transaksi tersebut dan menyisipkan ke RC-*queue* (*Read Commit queue*) dan pada saat yang bersamaan elemen tersebut dikirim ke *Database Manager* untuk dieksekusi. RC-*queue* adalah suatu struktur *queue* yang memiliki *rear* dan *front*; *rear* untuk menyisipkan elemen yang baru dibangkitkan, sedangkan *front* untuk menghapus elemen *validated* (bila *front* menunjuk elemen *validated*). Secara umum RC-*queue* dapat digambarkan sebagai berikut:

$$rear \quad e,...,e,e_i^2,e,...,e,e_i^1,e,...,e,e_i^1,e,...,e \quad front.$$

Secara gamblang dapat dikatakan bahwa RC-*queue* mencerminkan urutan akses elemen-elemen transaksi. Sehingga sistem harus menjamin bahwa bila $e_i \prec e_j$ (e_i lebih awal disipkan ke RC-*queue* dibanding e_j), maka e_i dieksekusi oleh sistem lebih awal dibanding e_j terutama bila mereka konflik.

2.5. Serial Graph

Serial graph adalah suatu graph berarah dengan transaksi-transaksi pada *T* sebagai node pada graph, sedangkan suatu sisi $T_i \rightarrow T_j$ ada pada graph bila e_i^k konflik e_j^l dan $e_i^k \prec e_j^l$

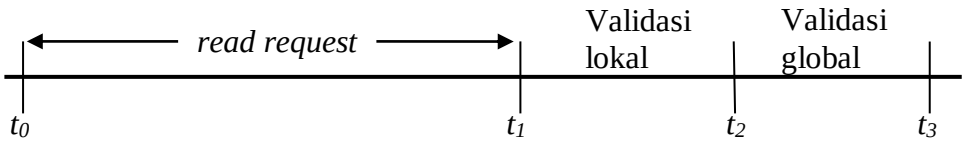
(e_i^k dieksekusi lebih awal dari e_j^l), untuk sembarang *k* dan *l*. Suatu sistem mengeksekusi transaksi-transaksinya dengan benar atau *serial* jika dan hanya jika *serial graph* dari eksekusi transaksi-transaksi tidak mengandung *cycle* (lihat [Ber87]).

3. Distributed *ROCC (DROCC)*

Algoritma *Distributed ROCC* (DROCC) adalah pengembangan algoritma ROCC yang dirancang untuk sistem basis data terdistribusi. Untuk menyederhanakan

permasalahan, algoritma DROCC dirancang khusus untuk sistem basis data terdistribusi yang bersifat *fully replicated*. Walaupun demikian algoritma DROCC dengan sedikit penyesuaian dapat juga diterapkan pada sistem basis data terdistribusi yang bersifat *partially replicated*.

Eksekusi suatu transaksi dimulai dengan *read request*, yaitu menyampaikan operasi baca ke sistem. *Read request* dapat dilakukan berkali-kali. Operasi tulis dilakukan pada akhir eksekusi transaksi, yaitu pada saat transaksi melakukan *commit*. Bila suatu transaksi meyampaikan pesan *commit request*, maka sistem akan melakukan proses validasi terhadap transaksi tersebut. Proses validasi dilakukan dalam dua tahapan, yaitu proses validasi lokal dan proses validasi global. Sehingga waktu eksekusi suatu transaksi secara



Gambar 3. Waktu eksekusi suatu transaksi

umum dapat dilihat pada Gambar 3. Waktu eksekusi suatu transaksiWaktu t_0 eksekusi transaksi dimulai dengan pengiriman pesan *read request*, suatu transaksi dapat mengirim pesan *read request* berkali-kali. Pesan *read request* berisi nomor identitas transaksi dan daftar data item yang akan dibaca. Pada waktu t_1 transaksi mengirim pesan *commit request*. Pesan ini berisikan nomor identitas transaksi dan daftar data item yang akan ditulis. Pesan ini memicu dilakukannya proses validasi lokal terhadap transaksi. Bila proses validasi lokal sukses (misalkan pada waktu t_2), maka proses validasi global dimulai. Proses validasi lokal hanya dilakukan pada situs lokal saja, sedangkan proses validasi global melibatkan situs-situs lainnya yang dikoordinasi oleh situs lokal, yaitu situs asal transaksi.

Berbeda dengan algoritma ROCC, pada algoritma DROCC terdapat 4 elemen, yaitu elemen *read, local commit, global commit*, dan *validated*. Elemen *read* dibangkitkan untuk merespon pesan *read request* dari suatu transaksi. Sama halnya dengan algoritma ROCC, transaksi pada algoritma DROCC dapat memiliki lebih dari satu elemen *read*. Bila algoritma ROCC langsung menyisipkan elemen *read* ke RC-*queue* dan langsung mengeksekusi setelah membangkitkannya, maka pada algoritma DROCC sedikit berbeda. Bila suatu situs menerima pesan *read request*, maka elemen *read* dibangkitkan dan sebelum disisipkan ke struktur RC-*queue*, sistem (*local scheduler*) akan melihat apakah ada transaksi lain yang sedang melakukan proses validasi global. Bila elemen *read* konflik dengan elemen *global commit* yang ada pada RC-*queue*, maka elemen *read* tersebut ditunda eksekusinya dan ditempatkan pada **BlockedElement**. Pada saat suatu transaksi selesai proses validasi global, maka elemen-elemen dalam **BlockedElement** akan diperiksa apakah ada elemen yang bisa dieksekusi. Sedangkan bila tidak ada elemen *global commit* yang konflik dengan elemen *read* tersebut, maka elemen tersebut langsung dieksekusi dan disisipkan pada struktur RC-*queue*.

Bila *local scheduler* menerima pesan *commit request* dari suatu transaksi, maka elemen *local commit* dibangkitkan dan sebelum elemen tersebut disisipkan pada RC-*queue*, sistem akan memeriksa apakah elemen *local commit* konflik dengan elemen *global commit* yang ada di RC-*queue*. Bila ada elemen *global commit* yang konflik dengan elemen *local commit*, maka elemen tersebut ditempatkan pada **BlockedElement**. Bila elemen *local commit* tidak konflik dengan elemen-elemen *global commit* di RC-*queue*, maka elemen tersebut disisipkan pada RC-*queue* dan proses validasi lokal dilakukan terhadap transaksi tersebut.

Setiap transaksi yang gagal melewati proses validasi lokal atau proses validasi global, maka semua elemen transaksi bersangkutan pada RC-*queue* dihapus dan transaksi di-*restart*. Tetapi bila transaksi berhasil melewati proses validasi lokal, maka untuk transaksi yang tidak memiliki operasi tulis elemen *local commit* diganti dengan elemen *validated*. Sedangkan untuk transaksi yang memiliki operasi tulis, elemen *local commit* diganti dengan elemen *global commit* dan operasi tulis transaksi bersangkutan dikirim ke DM lokal untuk dieksekusi. Kemudian *local scheduler* melakukan proses validasi untuk transaksi bersangkutan.

4. PROSES VALIDASI

Untuk melakukan proses validasi global, *local scheduler* mengirim pesan *global commit request* ke situs lainnya. Pesan *global commit request* berisi data item yang akan ditulis beserta nilainya dan identifikasi transaksi. *Scheduler* yang menerima pesan *global commit request* akan membangkitkan elemen *global commit* dan menyisipkannya pada RC-*queue* setempat, kemudian *scheduler* setempat membangkitkan *serial graph* setempat dan mengirimnya ke situs (*scheduler*) asal transaksi.

4.1. Proses Validasi Lokal

Prosedur validasi pada algoritma ROCC yang diusulkan oleh Shi dan Perizzo dalam [Shi04a] perlu dilakukan perbaikan atau penyempurnaan, karena prosedur tersebut melakukan *restart* transaksi yang tidak perlu. Suatu transaksi di-*restart* bila eksekusinya tumpang tindih dengan transaksi lain. Penulis bersama Sulasno pada [Buk06] yang melakukan kajian pada algoritma ROCC, menjumpai bahwa ada transaksi yang tidak tumpang tindih eksekusinya tapi di-*restart*, karena prosedur validasi yang dirancang Shi dan Perizzo melakukan deteksi kurang akurat terhadap eksekusi transaksi. Walaupun pada [Buk06] penulis sudah melakukan perbaikan, tetapi prosedur validasi masih perlu disempurnakan lagi. Sehingga pada penelitian ini penulis melakukan penyempurnaan terhadap algoritma ROCC, khususnya prosedur validasi yang penulis namakan sebagai prosedur validasi lokal untuk membedakan dengan prosedur validasi global.

Proses validasi lokal dilakukan algoritma DROCC dengan menelusuri (*traversal*) struktur RC-*queue* lokal. Suatu transaksi akan gagal melewati proses validasi lokal bila eksekusinya tumpang tindih dengan eksekusi transaksi lain. Prosedur validasi lokal akan mendeteksi eksekusi suatu transaksi yang tumpang tindih berdasarkan

urutan eksekusi elemen-elemennya diantara eksekusi elemen-elemen transaksi lain pada RC-*queue* lokal.

Misalkan terdapat tiga buah transaksi yaitu $T_1 = \{r_1(x), w_1(y)\}$ (transaksi lokal) dan transaksi $T_2 = \{w_2(x)\}$ (transaksi yang berasal dari situs lain) dan $T_3 = \{r_3(y)\}$ (transaksi lokal). Transaksi T_1 pertama kali datang dan mengirim pesan *read request* untuk membaca objek *x* dan *y*. *Scheduler* lokal menyisipkan elemen $e_1^1(\{x, y\})$ pada RC-*queue*. Kemudian datang pesan *global commit* transaksi T_2 yang berasal dari situs lain. Transaksi T_2 melakukan perubahan nilai data item *x*, sehingga *local scheduler* menyisipkan elemen *global commit* $e_2^3(\{x\})$. Elemen e_2^3 langsung dieksekusi oleh *local scheduler* tanpa perlu melakukan proses validasi. Selanjutnya datang transaksi T_3 yang hanya membaca atau mengakses objek *y*, *scheduler* menyisipkan elemen $e_3^4(\{y\},\{\})$ pada RC-*queue*. Terakhir transaksi T_1 mengirim *commit request*, sehingga struktur *RC queue* akan berisi sebagai berikut,

$$rear \quad e_1^2(\{y\}), e_3^4(\{y\},\{\}), e_2^3(\{\},\{x\}), e_1^1(\{x, y\}) \quad front$$

Berdasar prosedur validasi yang diusulkan Shi dan Perizzo, transaksi T_1 akan gagal dan di-*restart*. Elemen e_1^1 konflik dengan elemen e_2^3, karena operasi baca pada e_1^1 konflik dengan operasi tulis e_2^3, dan elemen e_1^2 konflik dengan elemen e_3^4. Sehingga transaksi T_1 akan mengalami *restart*. *Restart* yang dilakukan oleh algoritma ROCC seharusnya tidak perlu dilakukan, karena pada kenyataannya eksekusi transaksi T_1 tidak tumpang tindih dengan transaksi lain. Transaksi T_1 membaca data item *x* sebelum diubah oleh transaksi T_2, karena elemen $e_1^1(\{x, y\})$ dieksekusi sebelum elemen $e_2^3(\{\},\{x\})$ (lihat RC-*queue*). Transaksi T_3 membaca *y* sebelum diubah transaksi T_1, karena elemen $e_3^4(\{y\},\{\})$ dieksekusi sebelum elemen $e_1^2(\{\},\{y\})$. Sehingga eksekusi transaksi-transaksi tersebut setara dengan eksekusi *serial*, yaitu $T_3 \rightarrow T_1 \rightarrow T_2$, dan transaksi T_1 seharusnya sukses melewati proses validasi.

Perbaikan prosedur validasi algoritma ROCC perlu dilakukan perbaikan, sehingga algoritma tidak me-*restart* transaksi yang tidak perlu. Dengan menggunakan Algoritma 1. Prosedur validasi lokal algoritma DROCC yang merupakan penyempurnaan prosedur validasi algoritma ROCC, tidak ada transaksi pada kasus tersebut yang di-*restart*. Transaksi T_1 akan sukses melewati proses validasi, dan RC-*queue* akan berubah menjadi,

$$rear \quad e_2^3(\{\},\{x\}), e_1^3(\{x\},\{y\}), e_3^4(\{y\},\{\}) \quad front$$

Prosedur validasi lokal akan dipanggil atau digunakan bila suatu transaksi mengirim pesan *commit request*. Pada saat pesan *commit request* diterima, elemen *local commit* dibangkitkan dan disisipkan pada RC-*queue*. Bila suatu transaksi gagal melewati proses validasi lokal; artinya eksekusi transaksi tersebut tumpang tindih dengan transaksi lain, maka transaksi akan di-*restart*, semua element transaksi tersebut akan

dihapus pada RC-*queue*, dan pesan *restart* akan disampaikan ke transaksi bersangkutan. Bila transaksi sukses melewati proses validasi lokal; artinya eksekusi transaksi tersebut tidak tumpang tindih dengan eksekusi transaksi lain, maka elemen *local commit* berubah menjadi elemen *global commit* dan proses validasi global dilakukan terhadap transaksi tersebut.

```
ConflictElement = Null; Success=1; Failure=0;
First = get first element of the transaction from the front of rc queue;
Second = get The transaction's commit element;
NextElement = get next element in rc queue after First;
IF (First==Null) Return Validated = Success;
WHILE (1)
    IF (NextElement == another read element) /*another 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;
        NextElement = get next element;
    ELSIF (NextElement == Second)
        Replace Second with (Merge First and Second);
        Remove First;
        Return Validated = success;
    ELSIF (First conflict with NextElement)
        Remove First from rc queue;
        Insert First before NextElement in rc queue;
        NextElement = get previous element of Second in rc queue;
        WHILE (1)
            IF (NextElement is First)
                IF (ConflictElement conflict with First)
                    Remove all elements of the transaction in rc queue;
                    Return Validated = Failure;
                ELSE
                    Remove Second in rc queue;
                    Second = Merge Second with First;
                    Replace First with Second in rc queue;
                    Move ConflictElement to the previous of Second in rc queue;
                    Return Validate = Success;
                END IF
            ELSIF (NextElement == e_i^1 ) /* another read element */
                IF (ConflictElement conflict with NextElement)
                    Remove all elements of the transactions in rc queue;
                    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;
                    Move ConflictElement to the previous of Second in rc queue;
                END IF
            ELSEIF (NextElement == ConfilctElement) /*another element of
ConflictElement*/
                Insert NextElement to ConflictElement;
                NextElement = get previous element in rc queue;

            ELSIF (Second or ConflictElement conflict with NextElement)
                Insert NextElement to ConflictElement;
                NextElement = get previous element in rc queue;
            ELSE
                NextElement = get previous element in rc queue;
            END IF
        END WHILE
    ELSE
        NextElement = get next element in the rc queue;
    END IF
END WHILE
```

Algoritma 1. Prosedur validasi lokal algoritma DROCC

Peubah **ConflictElement** pada Algoritma 1. Prosedur validasi lokal algoritma DROCCMerupakan *pointer* yang menunjuk ke suatu *link list*; yaitu suatu untaian dengan komponen elemen-elemen yang konflik dengan **Second** atau **ConflictElement**, urutannya disesuaikan dengan urutan posisi elemen-elemen tersebut di RC-*queue*. Suatu elemen pada RC-*queue* dikatakan konflik dengan **ConflictElement,** bila elemen tersebut konflik (paling sedikit satu elemen) dengan elemen yang ada pada **ConflictElement**.

Prosedur validasi lokal hanya mendeteksi eksekusi suatu transaksi pada suatu situs. Pada sistem basis data terdistribusi akan terdapat lebih dari satu situs (lebih dari satu sistem basis data) yang terlibat, sehingga diperlukan mekanisme untuk mendeteksi eksekusi transaksi secara global.

4.2. Proses Validasi Global

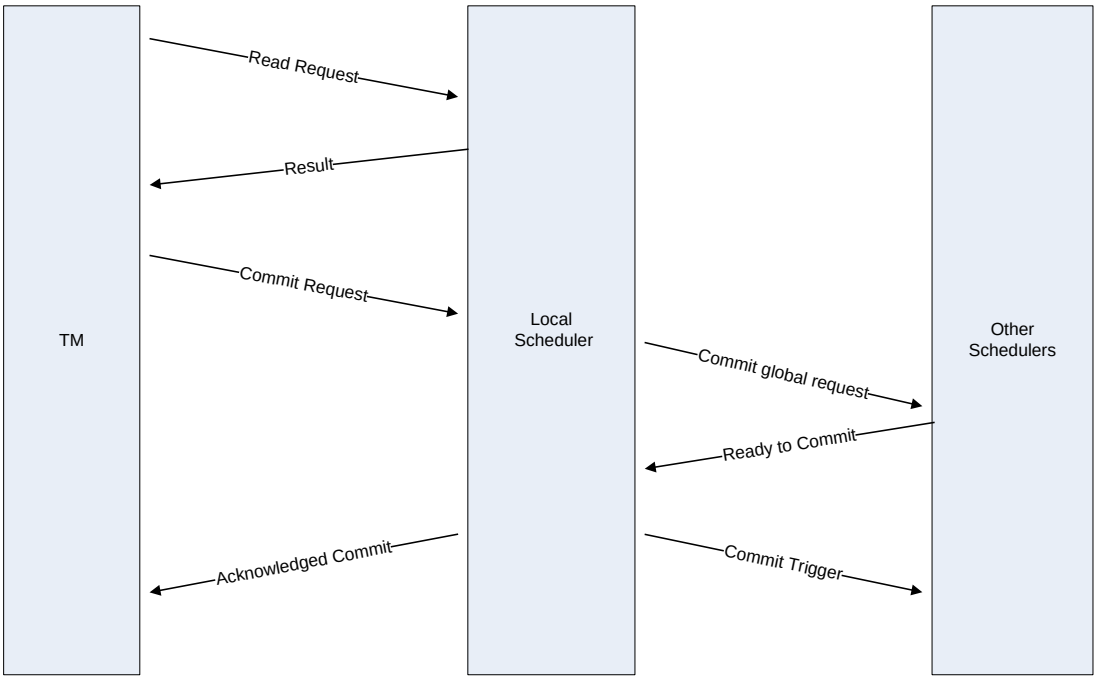
Setelah suatu transaksi T_i berhasil melakukan proses validasi lokal, elemen *local commit* diubah menjadi elemen *global commit*; mengindikasikan transaksi T_i sedang melakukan proses validasi global. Kemudian *local scheduler* membangkitkan *local serial graph* dengan menggunakan Algoritma 2. Pembangkitan *Serial Graph*. Kemudian *local scheduler* mengirim pesan *global commit request* ke semua situs lainnya. Pesan ini berisi nomor identifikasi transaksi dan daftar data item yang akan diubah oleh transaksi tersebut. Pada saat suatu *scheduler* menerima pesan *global commit request*, elemen *global commit* dibangkitkan dan disisipkan pada RC-*queue*. Kemudian *scheduler* tersebut mengirim *serial graph* setempat ke situs (*scheduler*) asal transaksi tersebut. Semua *serial graph* yang diterima oleh *local scheduler* dari situs lain digabungkan dengan *local serial graph*, sehingga terbentuk *global serial graph*.

```
FOR transaksi T_i ∈ {T_1, T_2, ..., T_n}
      TransElement = Get the first elemen e_i dalam RC-queue;
      FOR elemen e_j setelah TransElement dalam RC-queue menuju rear
            IF (e_j berasal dari transaksi T_i)
                  TransElement = Merge TransElement with e_j;
            ELSIF (TransElement konflik dengan e_j)
                  Insert T_i → T_j pada Graph;
            ENDIF
      ENDFOR
ENDFOR
```

Algoritma 2. Pembangkitan *Serial Graph*

Jika *global serial graph* mengandung *cycle* dan transaksi T_i terlibat pada *cycle* tersebut, maka transaksi T_i akan di-*abort* dan pesan *abort* untuk transaksi T_i akan dikirim kesemua situs. Sedangkan bila *global serial graph* tidak mengandung *cycle* atau transaksi T_i tidak terlibat dalam *cycle*, maka pesan *commit trigger* akan dikirim

kesemua situs (lihat Gambar 4. Proses validasi global). Elemen *global commit* pada tiap situs akan dieksekusi, kemudian elemen *global commit* diubah menjadi elemen *validated*. Selanjutnya masing-masing *scheduler* melihat apakah ada elemen pada **BlockedElement** setempat **yang** dapat dieksekusi.



Gambar 4. Proses validasi global

Untuk membangkitkan *serial graph*, setiap situs menyimpan himpunan *T*. Himpunan $T = \{T_1, T_2, ..., T_n\}$ adalah himpunan transaksi yang elemennya terdapat pada RC-*queue*. Himpunan *T* berisi transaksi yang mengirim *request* ke situs tersebut. Kapan suatu transaksi dihapus pada himpunan *T*?. Mekanisme penghapusan transaksi lokal; yaitu transaksi yang mengakses database lokal saja, berbeda dengan transaksi global; yaitu transaksi yang mengakses lebih dari satu situs. Bila elemen transaksi lokal berupa elemen *validated* dan berada diposisi *front* dalam RC-*queue* atau elemen-elemen yang ada diantara elemen transaksi tersebut dan elemen pada posisi *front* pada RC-*queue* merupakan elemen *validated* semua, maka transaksi lokal tersebut akan dihapus dari himpunan *T* begitu juga dengan elemennya dalam RC-*queue*.

Penghapusan transaksi global sedikit lebih rumit dibanding transaksi lokal. Bila *local scheduler* mendapatkan elemen *validated* suatu transaksi global (asal transaksi dari situs lokal), berada pada posisi *front* pada RC-*queue* atau elemen-elemen yang ada diantara elemen transaksi tersebut dan elemen pada posisi *front* pada RC-*queue* merupakan elemen *validated* semua, maka *local scheduler* akan mengirim pesan *remove request;* pesan ini berisi nomor identifikasi transaksi yang akan dihapus, ke semua situs. Situs yang menerima pesan *remove request* akan mengirim pesan *ready to remove* ke situs asal bila elemen transaksi tersebut adalah elemen *validated* yang berada di *front* dalam RC-*queue*. Bila situs asal sudah menerima pesan *ready to*

remove bagi suatu transaksi dari semua situs, maka pesan *remove trigger* akan dikirim kesemua situs. Situs-situs yang menerima pesan *remove trigger* akan menghapus transaksi yang berkaitan dan elemennya pada RC-*queue*.

5. PENUTUP

Algoritma pengendali konkurensi terdistribusi yang diperkenalkan dalam penelitian ini memiliki *feature* sebagai berikut (i) bersifat optimistik, setiap *request* langsung dieksekusi tanpa ada penundaan, kecuali bila ada elemen *global commit* pada RC-*queue* setempat dan elemen yang dibangkitkan berkaitan dengan *request* tersebut, konflik dengan elemen *global commit*, (ii) bebas dari masalah *deadlock* baik secara lokal maupun global, (iii) setiap situs memiliki otonomi penuh dalam mengatur akses terhadap basis data lokal. Dengan *feature* ini algoritma yang diperkenalkan pada penelitian ini akan menjadi menarik perhatian para peneliti algoritma pengendali konkurensi.

Algoritma ini akan berkinerja baik pada sistem yang memiliki peluang suatu transaksi akan konflik dengan transaksi lain lebih kecil dari setengah (atau 50%). Karena algoritma yang diperkenalkan pada tulisan ini tanpa menggunakan mekanisme *locking*, maka algoritma ini sangat sesuai untuk aplikasi berbasis *web*.

Penelitian ini merupakan penelitian awal dalam perancangan algoritma pengendali konkurensi terdistribusi. Sehingga untuk mendapatkan algoritma pengendali konkurensi terdistribusi (*distributed concurrency control algorithm*) yang siap diterapkan pada suatu aplikasi sistem informasi, diperlukan kajian lanjutan. Kajian lanjutan yang diperlukan adalah sebagai berikut:

1. Kajian simulasi algoritma pengendali konkurensi. Kajian ini dimaksudkan untuk membandingkan kinerja algoritma yang dirancang dengan algoritma-algoritma lainnya pada sistem terdistribusi.

2. Penyusunan bukti secara matematis bahwa algoritma yang dirancang setara adalah benar atau urutan eksekusi transaksi-transaksi bersifat *serial*.

DAFTAR PUSTAKA

[Ber87] Bernstein, P.A., V. Hadzilacos, dan N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987.

[Buk06] Bukhari, Fahren dan Sulasno, *Algoritma ROCC*, Jurnal Matematika dan Aplikasinya, 2006

[Özs99] M.T. Özsu dan P. Valduriez, *Principles of distributed database systems,* Prentice Hall, 1999

[Shi04a] Shi, Victor T. S. & Perrizo, W., *Read-commit Order for Concurrency Control in Centralized High Performance Database Systems*. Information Journal of International Information Institute 7(1):95-106, 2004

[Shi04b] 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 2004].

.