Transactions and Concurrency Control

Arghanta Wijna Suryabrata, 33129-TE
Bismoko Seto Nugroho, 3358-TE
Hendra Ridwandhana, 33392-TE
Jurusan Teknik Elektro FT UGM,
Yogyakarta

1.1 PENGENALAN

Tujuan transaksi disini adalah untuk menjamin semua obyek yang ditangani oleh server tetap dalam keadaan konsisten pada saat obyek – obyek tersebut diakses oleh beberapa macam transaksi dan juga pada saat adanya crash pada server.

Obyek yang dapat dipulihkan setelah terjadinya crash disebut dengan recoverable objects. Umumnya, obyek yang dikelola oleh server dapat disimpan pada memori volatile maupun memori tetap. Ketika obyek disimpan pada memori volatile, server mungkin juga menyimpan beberapa informasi mengenai keadaannya pad memori tetap sehingga obyek dapat dipulihkan jika ada crash pada server. Sebuah transaksi ditetapkan oleh client sebagai suatu rangkaian operasi pada obyek yang akan dilakukan sebagai suatu unit yang terpisahkan oleh server yang mengelola obyek tersebut. Server harus menjamin bahwa ketika seluruh transaksi telah dilakukan, hasilnya disimpan pada storage yang permanen atau jika terdapat crash pada satu atau lebih obyek, efeknya harus dihilangkan seluruhnya. Transaksi seorang client juga dapat dianggap terpisahkan dari client lain dalam artian bahwa operasi dari satu transaksi tidak dapat digunakan untuk mengamati efek parsial operasi transaksi lainnya.

1.1.1 Sinkronisasi simpel (tanpa transaksi)

Salah satu isu yang sering ditemui berhubungan dengan masalah transaksi adalah saling mengganggunya informasi satu client dengan client yang lain. Hal tersebut dapat menimbulkan kesalahan nilai pada obyek.

Atomic operations pada server -> telah diketahui bahwa penggunaan multi threads sangat bermanfaat bagi server, selain itu penggunaan threads memperbolehkan operasi dari beberapa client untuk berjalan secara bersamaan dan mengakses obyek yang sama. Oleh sebab itu, method untuk obyek – obyek harus didesain untuk digunakan pada konteks multi thread.

Sebagai contoh, dibuat method pada kelas yang menjamin pengaksesan obyek unik untuk masing – masing kelas, yaitu :

```
Public synchronized void deposit(int amount) throws RemoteExceptions{
//adds amount to the balance of the amount

}
```

Jika satu thread memanggil sinkronisasi method pada sebuah obyek, maka obyek tersebut akan dikunci sehingga thread lain yang memanggil obyek tersebut akan diblok sampai kunci pada obyeknya dilepaskan. Sinkronisasi ini mengharuskan pengeksekusian thread terpisah dalam satuan waktu dan menjamin variabel dalam obyek diakses secara konsisten.

Operasi yang bebas dari gangguan operasi yang bersamaan dalam threads lain disebut atomic operations. Penggunaan method sinkronisasi dalam java adalah untuk mencapai atomic operations ini. Namun, dalam bahasa pemrograman lain untuk lingkungan multi thread server, operasi pada obyek masih membutuhkan

atomic operations untuk menjaga agar obyeknya konsisten. Hal ini dapat dilakukan dengan menggunakan berbagai mekanisme mutual exclusion semisal mutex.

Peningkatan kerjasama client dengan sinkronisasi operasi pada server -> client dapat menggunakan server sebagai media untuk berbagi resource yang dimilikinya. Hal ini dapat dilakukan dengan sebuah operasi untuk mengupdate obyek pada server dan client lain menggunakan operasi lain untuk mengakses obyek tersebut. Skema untuk sinkronisasi akses ke obyek tersebut telah menyediakan pelbagai hal yang diperlukan dalam banyak aplikasi – termasuk pencegahan gangguan antar thread. Akan tetapi, beberapa aplikasi membutuhkan suatu cara agar thread dapat berkomunikasi satu dengan yang lainnya.

Sebagai contoh, ada situasi dimana satu client tidak dapat meneyelesaikan operasi yang dilakukannya sampai operasi yang dilakukan client lain selesai. Hal ini misalnya situasi dimana satu client menjadi produser dan client lainnya menjadi konsumen. Client yang membutuhkan sesumber (konsumen) harus menunggu sampai client yang menjadi produser menyelesaikan operasinya.

Pada bahasa java, ada method yang dapat digunakan untuk menyelesaikan problem mengenai komunikasi antar threads tersebut. Method tersebut adalah method ‘wait’ dan ‘notify’. Threads memanggil method wait pada suatu obyek sehingga method lain pada obyek tersebut dapat digunakan oleh thread lain. Method notify dipanggil untuk memperingatkan thread lain yang mengakses obyek tersebut bahwa telah terjadi perubahan data pada obyek tersebut. Kedua method tersebut akan mempengaruhi status lock yang dijalankan pada obyek tersebut sehingga memungkinkan thread lain mengaksesnya.

Sinkronisasi thread ini mempunya kelebihan lain, yaitu menghandle request dengan cepat sehingga client cepat puas. Hal tersebut sukar dilakukan tanpa adanya sinkronisasi thread, jika misalnya satu client ingin melakukan suatu operasi yang belum dapat dipenuhi oleh server, client tersebut akan mencoba lagi sampai operasinya terpenuhi. Hal tersebut berpotensi menimbulkan monopoli oleh client tersebut karena client lain mungkin akan melakukan suatu operasi ketika client pertama mencoba menghubungi server lagi.

1.1.2 Model yang gagal untuk transaksi

Lampson [1981a] membuat suatu model untuk menggambarkan distribusi transaksi yang gagal untuk disk, server dan komunikasinya. Model ini dapat mengklaim bahwa algoritma yang digunakannya berhasil untuk mendeteksi adanya kesalahan, namun tidak dapat memprediksi mengenai kelakuan obyek saat terjadi kesalahan. Modelnya dapat dijelaskan sebagai berikut :

- Proses untuk menulis ke storage dapat gagal (dapat berupa penulisan nilai yang salah atau bahkan tidak menuliskan apapun).
- Server dapat mengalami crash secara tiba – tiba. Ketika server yang mengalami crash digantikan proses baru, memori volatile yang digunakannya diset ke keadaan sebelum terjadi crash. Setelah itu, dia memanggil prosedur recovery menggunakan informasi pada storage dan proses lain untuk mengatur nilainya.
- Adanya kemungkinan delay sebelum pesan sampai. Pesan yang dikirimkan pada suatu komunikasi dapat hilang, terduplikasi ataupun corrupt.

Model yang gagal untuk storage permanen, prosesor serta komunikasinya dapat digunakan untuk merancang suatu sistem yang stabil dimana komponennya dapat bertahan terhadap kesalahan yang terjadi. Sorage yang stabil dapat menyediakan atomic operation untuk operasi write jika ada kesalahan operasi tulis ataupun kegagalan crash pada prosesnya. Prosesor yang stabil menggunakan storage di atas untuk menyediakan proses recovery setelah terjadinya crash.

1.2 TRANSAKSI

Dalam berbagai situasi, client membutuhkan suatu urutan pemisahan rekues ke server agar :

- Client terbebas dari gangguan operasi client lain yang sedang berjalan bersamaan.
- Operasi client harus diselesaikan sepenuhnya atau operasi tersebut tidak boleh mempunyai efek ketika server mengalami crash.

Transaksi berhubungan dengan sistem manajemen database. Transaksi adalah eksekusi program yang mengakses database. Transaksi dikenalkan ke sistem terdistribusi dengan bentuk transaksi file server semisal XDFS [Mitchell and Dion 1982] yang pada konteksnya, transaksi diartikan sebagai eksekusi urutan rekues client untuk operasi file. Transaksi pada obyek terdistribusi disediakan beberapa sistem, semisal Argus [Liskov 1988] dana Arjuna [Shrivastava et al. 1991]. Dari sudut pandang client, transaksi dapat diartikan sebagai urutan operasi yang berbentuk satu langkah tunggal, yang mengubah data dari server dari satu bentuk ke bentuk lainnya.

Pada semua konteksnya, transaksi berlaku untuk obyek yang dapat diperbaiki dan dimaksudkan untuk menjadi atomic operations. Hal ini sering disebut dengan atomic transactions. Ada dua aspek mengenainya :

- All or nothing : transaksi dapat berhasil sepenuhnya atau tidak sama sekali. Ada dua aspek lebih lanjut mengenai aspek ini :
  1. Failure automicity : efeknya kecil meskipun server mengalami crash.
  2. Durability : setelah transaksi berhasil, datanya akan disimpan dalam storage. Data yang disimpan dalam storage akan tetap ada meskipun server mengalami crash.
- Isolation : tiap – tiap transaksi harus dilakukan tanpa adanya gangguan dari transaksi lain.

Untuk mendukung persyaratan mengenai failure automicity dan durability, obyeknya harus dapat diperbaiki. Sehingga setiap ada perubahan mengenai obyek harus disimpan pada storage.

Server yang mendukung proses transaksi harus mensinkronkan operasinya untuk menjamin setiap persyaratan mengenai aspek isolation terpenuhi. Satu cara yang dapat dilakukan untuk melakukannya adalah dengan menjalankan transaksi secara serial. Namun sayangnya, cara ini secara umum ditolak oleh server yang sesumbernya dibagi untuk beberapa user.

Tujuan dari server yang mendukung transaksi adalah untuk memaksimalkan penggunaan secara bersamaan. Oleh sebab itu, transaksi diperbolehkan untuk dieksekusi secara bersamaan jika efeknya serupa dengan eksekusi secara serial.

1.2.1 Kontrol Konkurensi

- **Lost update problem** -> adalah suatu masalah yang timbul akibat dihiraukannya informasi pada saat ada update dari operasi lain yang hampir bersamaan waktunya.
- **Inconsistent retrievals** -> adalah permasalahan mengenai informasi update suatu operasi yang belum didapatkan oleh operasi lain saat operasi tersebut berjalan.
- **Serial equivalence** ->adalah kriteria untuk mengkoreksi eksekusi secara bersamaan yang bertujuan untuk menghindari adanya lost update problems dan inconsistent retrievals.
- **Conflicting operations** -> adalah adanya konflik antara dua atau lebih operasi yang efeknya saling bertentangan.

Serial equivalence sering digunakan sebagai suatu kriteria protokol concurrency control. Protokolnya berusaha untuk menjadikan transaksi sebagai suatu serial saat mengakses obyek. Pada dasarnya, kontrol konkurensi dapat dicapai dengan memaksa klien menunggu klien lain menyelesaikan operasinya atau dengan merestart transaksi klien setelah terdeteksi adanya konflik, atau kombinasi dari keduanya.

1.2.2 Kemampuan Recovery dari Pembatalan Operasi

Server harus mencatat setiap efek dari transaksi yang dilakukan dan tidak mencatat efek pembatalan transaksi. Meski demikian, server harus mencegah pembatalan transaksi mengganggu transaksi lain yang sedang berlangsung bersamaan.

Ada beberapa masalah yang umum berkaitan dengan pembatalan transaksi. Masalah ini semisal ‘dirty read’ dan ‘premature writes’ yang mana keduanya dapat terjadi pada saat pengeksekusian transaksi.

- **Dirty read** -> adalah masalah yang disebabkan oleh interaksi antara operasi read dan operasi write yang lebih dahulu berlangsung, namun dibatalkan pada obyek yang sama.

Ada beberapa hal yang perlu diperhatikan bersangkutan dengan adanya dirty read yang terjadi. Hal tersebut misalnya :

  - **Recoverability of transactions** -> masalah yang ditimbulkan oleh adanya proses dirty read tidak dapat disembuhkan secara langsung. Masalah yang timbul harus dideteksi sebelumnya dan dicegah agar tidak terjadi. Pencegahannya dapat dilakukan dengan menambah waktu delay operasi sesudah pembatalan, dan atau ikut membatalkan operasi yang berlangsung pada saat pembatalan operasi.
  - **Cascading aborts** -> adalah masalah yang timbul pada transaksi sesudah pembatalan. Transaksi – transaksi lain sesudah pembatalan yang telah menerima efek dari pembatalan suatu transaksi harus dibatalkan juga. Untuk menghindari adanya cascading abort, suatu transaksi hanya diperbolehkan untuk membaca obyek yang sudah mengalami proses write yang berhasil. Untuk menjamin hal ini, setiap operasi read harus ditunda sampai operasi write pada obyek yang bersangkutan selesai atau dibatalkan.

- **Premature writes** -> adalah masalah yang timbul akibat adanya pembatalan operasi write pada suatu obyek yang mempengaruhi operasi write pada transaksi lain dalam obyek tersebut.

  Hal lain yang perlu diperhatikan mengenai masalah recoverability ini adalah :

  - **Strict executions of transactions** ->secara umum, transaksi perlu menunda operasi read dan operasi write untuk mencegah terjadinya dirty read dan premature write. Eksekusi transaksi dikatakan ‘strict’ jika ada penundaan operasi read dan write pada suatu obyek sampai semua transaksi yang sebelumnya menuliskan operasi write pada obyek tersebut telah selesai atau dibatalkan.
  - **Tentative versions** -> server yang digunakan untuk memperbaiki obyek harus didesain agar server dapat menghapus setiap update obyek yang transaksinya dibatalkan. Untuk melakukan hal ini, semua update selama proses transaksi harus disimpan pada memori volatile pada versi eksperimen. Setiap transaksi harus disediakan dengan masing – masing set eksperimen pada obyeknya.

Versi eksperimen tersebut kemudian ditransfer ke obyek ketika transaksi terjadi, yang juga sekaligus menuliskannya pada storage. Ketika transaksi dibatalkan, versi eksperimennya juga dihapus.

1.3 TRANSAKSI BERSARANG

Transaksi bersarang adalah sebuah mekanisme sebuah transaksi dibentuk oleh transaksi lainnya.transaksi terluar pada sebuah set transaksi bersarang disebut transaksi level atas. Selain dari pada transaksi level atas maka akan disebut dengan transaksi Sub.

Transaksi sub pada level yang sama dapat berjalan secara konkuren tetapi akses nya ke objek objek umum menggunakan mekanisme terserialisasi. Seperti pada skema penguncian(locking). Ketika kita akan membedakan bentuk original dari sebuah transaksi dari yang bersarang kita dapat menggunakan skema transaksi datar. Ada beberapa kelebihan dari transaksi bersarang diantaranya :

° Transaksi – transaksi sub yang berada pada level yang sama dapat bekerja secara konkuren. Ketika transaksi sub dijalankan pada server yang berbeda mereka dapat bekerja secara parallel.

° Transaksi sub dapat melakukan operasi atau mebatalkan operasi secara independen.

Aturan aturan komitmen dari transaksi bersarang :

° Sebuah transaksi dapat berjalan atau batal hana setelah transaksi anakannya selesai.

° Ketika sebuah transaksi sub selesai, dia akan membuat keputusan independen diantara menjalankan provisionally atau batal.

° Ketika transaksi top levelnya batal maka transaksi anakannya akan otomatis dibatalkan.

° Ketika transaksi anakannya batal maka transaksi top levelnya akan mengambil keputusan akan membatalkan atau tidak.

Contoh dari transaksi bersarang

- Transfer $ 100 dari A ke B
- a.simpan(100)
- b.ambil(100)

dari contoh diatas dapat dilihat bahwa transaksi top level Transfer dapat dibagi menjadi 2 buah transaksi sub yaitu simpan dan ambil. Ketika 2 buah transaksi sub itu commit / berjalan maka transaksi top levelnya hamper pasti berjalan. Tetapi jika transaksi top levelnya dibatalkan dalam hal ini transaksi transfer maka transaksi sub simpan dan ambil juga akan ikut dibatalkan.

1.4 KUNCI/LOCK

Transaksi harus dijadwalkan sehingga efek dari data bersama dapat terserialisasi. Sebuah mekanisme simple untuk menserialisasikan adalah dengan menggunakan penguncian secara eksklusif. Pada saat sebuah objek dilakukan penguncian maka operasi lain yang membutuhkan objek untuk operasinya akan menunda operasinya hingga objek tersebut bebas dari penguncian.

Contoh operasi lock

Transaction T:		Transaction U:	
balance = b.getBalance()		balance = b.getBalance()	
b.setBalance(bal*1.1)		b.setBalance(bal*1.1)	
a.ambil(baL110)		c.ambil(bal,110)	
Operation	Locks	Operation	Locks
bukaTransaksi			
bal = b.getBalance()	kunci B		
b.setBalance(bal*11)		bukaTransaksi	
a.ambil(ba/I0)	kunci A	bal = b.getBalance()	Tunggu sampai
tutupTransaksi	bebaskan A, B	…	T mengunci B

	Kunci B
b.setBalance(bal*1.1)	
c.withdraw(bal/10)	Kunci C
tutupTransaksi	Bebaskan B,C

Pada contoh diatas terlihat bahwa terdapat 2 buat transaksi T dan U sedang transaksi T melibatkan objek A dan B dan transaksi U melibatkan objek B dan C . dari uraian contoh diatas terlihat bahwa transaksi T bekerja lebih dulu. Sehingga dia akan melakukan penguncian terhadap objek B dan Objek A pada saat penguncian ini terjadi maka transaksi U akan menunggu sampai tansaksi T benar benar membebaskan objek B. Barulah saat itu transaksi U dapat menggunakan objek B untuk untuk menjalankan transaksinya.

Serial equivalence membutuhkan akses transaksi ke particular objek yang diserialisasikan untuk dapat diakses oleh transaksi yang lain. Semua pasangan yang terlibat konflik operasi dari 2 buah transaksi harus dieksekusi pada urutan yang sama. Untuk memastikan ini, sebuah transaksi tidak diijinkan melakukan penguncian terhadap objek baru setelah dia membebaskan pengunciannya. Fase satu dari setiap transaksi adalah sebuah “fase tumbuh”, ketika penguncian baru diperoleh. Pada fase kedua, penguncian dibebaskan (fase menyusut). Keadaan ini disebut “Two-phase locking”.

Sebuah transaksi dapat batal, sehingga dibutuhkan eksekusi yang ketat untuk mencegah “dirty reads” dan “Premature Write”. Pada saat ini sebuah transaksi yang akan membaca atau menulis sebuah objek harus ditunda hingga transaksi lain yang sedang beraktivitas dengan objek objek tersebut menyelesaikan / membatalkan aktivitasnya. Situasi ini disebut “strict two-phases locking”. Sehingga dapat disimpulkan aturan dari konflik operasi tersebut adalah :

- Jika transaksi T sedang melakukan kegiatan baca pada sebuah objek maka transaksi U tidak boleh melakukan aktivitas tulis pada objek tersebut.
- Jika transaksi T sedang melakukan aktivitas tulis maka transaksi U tidak diperbolehkan melakukan aktivitas baca maupun tulis.

Penggunaan mekanisme strict two-phase locking

1. Ketika sebuah operasi mengakses sebuah objek untuk sebuah transaksi :
    a. Jika objek belum terkunci , maka transaksi akan mengunci objek tersebut dan transaksi dilakukan.
    b. Jika objek sedang dipakai /dikunci oleh transaksi lain maka. Operasi lain dari sebuah transaksi yang berkenaan dari objek tersebut harus menunggu hingga objek tersebut dibebaskan.
    c. Jika sebuah objek memiliki sifat “non-conflicting lock”. Penguncian akan di share dan operasi dijalankan.
    d. Jika sebuah objek telah terkunci pada transaksi yang sama. Penguncian akan dipromosikan dan operasi dijalankan . (promosi dalam hal ini sama dengan konteks aturan b)
2. Ketika sebuah transaksi dibatalkan maka server akan membebaskan semua objek yang digunakan.

1.4.1 Deadlock

Resiko dari penggunaan lock adalah deadlock. Hal ini bisa terjadi manakala keadaan saling tunggu akan pembebasan penguncian terjadi pada tabel 1.

Sehingga dapat disimpulkan bahwa deadlock adalah keadaan dimana setiap member dari group transaksi menunggu member lainnya untuk membebaskan penguncian seperti pada situasi di atas yang kerap disebut keadaan wait-for graph.

Pencegahan -> sebuah solusi pencegahan yang dapat digunakan adalah dengan mengunci semua objek yang digunakan ketika sebuah transaksi mulai berjalan. Sehingga transaksi tersebut tidak akan pernah menuju keadaan deadlock dengan transaksi lainnya. Tetapi masih terdapat kendala yakni meprediksi objek objek mana yang kemungkinan akan dipakai dalam sebuah transaksi tersebut. Dengan mengimplementasikan cara pencegahan ini dapat mengurangi potensi deadlock tetapi memiliki efek samping terjadinya premature locking dan menurunnya konkurensi.

Deteksi deadlock -> deadlock dapat dideteksi dengan menemukan/ mengidentifikasi cycles pada wait-for graph. Setelah itu dipilihlah sebuah transaksi pembatalan untuk merusak cyclenya.

Selain dari pada cara diatas maka digunakan pula cara/ mekanisme timeouts yang memberikan batas waktu operasi kepada setiap transaksi. Sehingga jika deadlock terjadi maka timeouts ini akan menghitung mundur dan jika waktu habis maka transaksi yang terlibat deadlock akan dibatalkan.

1.4.2 Peningkatan Konkurensi pada Skema Penguncian

Two-version locking, pada dasarnya metode ini menekankan pada operasi read yang harus menunggu jika transaksi lain sedang bekerja dengan objek yang sama. Skema ini mengijinkan konkurensi yang lebih ketimbang read-write lock. Transaksi tidak dapat menjalankan operasi penulisannya secara langsung jika transaksi lain yang belum selesai masih melakukan operasi read pada objek yang sama.

Hierarchic locks, pada penguncian hirarkis di setiap level setting parents lock mempunyai efek yang sama seperti setting semua child lock yang sebanding. Pada skema ini setiap node pada hirarki dapat dikunci dan memberikan pemilik dari kunci sebuah akses eksplisit ke node dan akses implicit ke anak anaknya. Contohnya misalkan terjadi operasi read/write di di cabang maka secaara implisit terjadi read/write locks di setiap akun.

Penguncian hiararki mempunyai kelebihan yaitu mampu mereduksi jumlah penguncian ketika mixed-granularity locking dibutuhkan. Selain itu juga mengijinkan setiap transaksi untuk mengunci bagian yang ukurannya dipilih bergantung dari kebutuhanannya.

1.5 KONTROL OPTIMISTIK KONKUREN

Kung dan Robinson telah mengidentifikasi sejumlah kekurangan penguncian dari pewarisan dan pengajukan pendekatan optimistik untuk serialisasi transaksi yang dapat mencegah kerugian tersebut. Kekurangan penguncian tersebut antara lain :

° Pemeliharaan kunci mewakili sebuah overhead yang tidak tampak pada sistem yang tidak mendukung konkuren akses ke data bersama. Akantetapi penguncian tetap dibutuhkan untuk berjaga-jaga.

° Penggunaan kunci dapat berakhir pada sebuah buntu.

° Untuk mencegah pembatalan, kunci tidak dapat dilepas sampai pada akhir transaksi.

Pendekatan alternatif yang diajukan olek Kung dan Robinson dikatakan ‘optimistik’ karena berdasarkan pada obserfasi yang menyatakan bahwa kemungkinan dua klien mengakses data yang sama adalah kecil.

Saat terjadi sebuah konflik , beberapa transaksi secara umum akan dibatalkan dan harus diulang oleh klien. Setiap transaksi memiliki fase berikut :

- *Fase bekerja* : pada fase ini, setiap transsaksi memiliki versi tentatif dari setiap objek yang diperbarui. Kegunaanya adalah memperbolehkan transaksi terhenti disaat fase kerja maupun bila gagal saat validasi, tanpa mempengaruhi objeknya. Opreasi *read* akan langsung dilakukan. Operasi ini akan mengakses versi tentatif dari transaksi tersebut bila sudah ada. Operasi *write* akan merekam nilai terbaru dari objek yang ada sebagai nilai tentatif. Bila ada beberapa transaksi yang bersamaan, beberapa nilai tentatif berbeda untuk objek yang sama dapat terjadi.
- *Fase validasi* : saat permintaan *closeTransaction* diterima, transasksi tersebut divalidasi untuk mengetahui apakah operasi pada objek bermasalah dengan operasi pada transaksi atas objek yang sama. Bila validasi berhasil terbukti sukses, maka transaksi dapat diteruskan. Akan tetapi bila validasi gagal, maka harus ada pembenaran konflik harus dilakukan.
- *Fase pembaruan* : bila transaksi sudah dilakukan dan terbukti valid, maka semua perubahan pada versi tentatif akan permanen.

Validasi transaksi -> validasi menggunakan aturan konflik *read-write* untuk memastika penjadwalan atas sebuah transaksi tertentu telah setara secara serial. Untuk membantu proses valdasi, setiap transaksi diberi sebuah nomer transaksi saat memasuki fase validasi. Nomer tersebut akan permanen saat transaksi sudah divalidasi. Nomer tersebut berupa sebuah interger yang disusun dengan urutan naik, sehingga sebuah transaksi selalu selesai fase kerjanya setelah semua transaksi yang memiliki nomer yang lebih kecil. Misalnya bila sebuah transaksi diberi nomer T_i selalu mendahului transaksi dengan nomer T_v bila $i < j$.

Karena fasi validasi dan pembaruan terhitung singkat bila dibandingkan denga fase kerja, maka sebuah menyederhanaan dapat dibuat dengan cara membuat sebuah aturan dimana hanya boleh ada satu transaksi yang boleh berada dalam fase validasi dan pembaruan di saat yang bersama. Bila tidak ada dua transaksi dapat overlap pada fase update, aturan ke 3 diberlakukan. Untuk menghindar terjadinya overlap, seluruh fase validasi dan pembaruan dapat diimplementasikan sebagai sebuah sektor kritis agar hanya satu klien yang dapat mengaksesnya pada saat yang sama. Agar dapat meningkatkan kemampuan untuk konkuren, sebagian dari validasi dan pembaruan dapat diimplementasikan diluar sector kritis, tetapi sangat penting untuk melakukan nomer transaksi secara berurutan.

Validasi atas sebuah transaksi harus dipastikan bahwa aturan 1 dan 2 dipatuhi dengan cara mencoba mencari overlap yang terjadi antara kedua objek T_i dan T_u. ada dua bentuk validasi, yaitu *backward* dan *forward*. Validasi *backward* memeriksa transasksi yang sedang dilakukan dengan melanjutkan transaksi overlap yang lain, sedangkan validasi *forward* memeriksa transaksi yang sedang divalidasi dengan transaksi lainnya yang masih aktif.

Validasi backward -> karena semua operas *read* yang terjadi pada transaksi yang overlap dilakukan sebelum validasi $T_v$dilakukan, maka transaksi tersebut tidak dapat diganggu oleh operasi *write* pada transaksi yang sama. Validasi transaksi T_v memeriksa apakah read set pada transaksi tersebut overlap dengan write set pada transaksi overlap T_i sebelumnya. bila terjadi overlap, maka validasi gagal.

Pada validasi *backward*, read set pada transaksi divalidasi dengan membandingkan pada write set transaksi yang sudah dilakukan sebelumnya. sehingga satu-satunya cara untuk menyelesaikan konflik yang ada adalah dengan menghentikan transaksi yang sedang di validasi.

kontrol optimistik yang konkuren dengan menggunakan validasi *backward* memerlukan write set pada versi yang terdahulu dari objek yang terkoresponden dengan transaksi yang sedang dilakukan sampai tidak ada transaksi overlap yang tidak valid

Validasi forward -> dalam validasi *forward* pada transaksi T_v, write set T_v dibandingkan dengan read set pada semua transaksi overlap yang aktif. Validasi *forward* memperbolehkan sebuah fakta dimana read set pada sebuah transaksi aktif dapat berubah pada saat validasi. Saat transaksi dibandingkan dengan transaksi valid yang masih aktif, kita memiliki pilihan untuk membatalkan validasi transaksi atau mengambil cara alternatif untuk menyelesaikan konflik. Härder memberikan beberapa solusi alternatif :

- Menunda validasi hingga waktu transaksi yang bermasalah telah selesai.
- Membatalkan semua transaksi aktif yang bermasalah dan menyatakan transaksi tersebut sudah tervalidasi.
- Membatalkan proses validasi transaksi tersebut.

Perbandingan backward dan forward -> dapat dilihat bahwa validasi forward memperbolehkan kelonggaran dalam menangani konflik, sedangan pada validasi backward hanya memiliki satu solusi, membatalkan proses validasi. Akan tetapi validasi backward memiliki keunggulan dalam menyusun write set lama sampai mereka tidak dibutuhkan lagi, sedangkan pada validasi forward harus memperbolehkan transaksi baru dimulai saat proses validasi terjadi.

Starvation -> transaksi yang dibatalkan akan dimulai kembali oleh program klien, akan tetapi transaksi tersebut tetap memiliki kemungkinan untuk gagal saat proses validasi. Ini dapat terjadi bila transaksi tersebut mengalami konlik dengan transaksi lainnya karena penggunaan objek setiap kali transaksi tersebut diulang. Pencegahan transaksi untuk tidak dapat dilaksanakan disebut starvation yang jarang kali terjadi.

1.6 TIMESTAMP ORDERING

Pada kontrol konkuren yang berdasarkan pada timestamp ordering, setiap opersai pada semuah transaksi divaldidasi saat dilaksanakan. Bila operasi tidak dapat divalidasi, transkaksi tersebut langsung dibatalkan dan kemudian diulang oleh program klien. Setiap transkasi ditandai dengan nilai timestamp yang unik pada saat dimulai. Timestamp tersebut mendefinisikan posisinya pada urutan waktu transaksi. Permintaan sebuah transaksi dapat dipesan berdasarkan timestampnya.

Aturan pada timestamp dibuat untuk memastikan agar setia transaksi mengakses sebuah nilai objek yang konsisten. Aturan tersebut juga harus memastikan bahwa versi tentatif pada setiap objek harus diselesaikan dengan urutan yang ditentukan oleh timestamps dari transaksi yang membuatnya.

Seperti biasa, operasi write dorekam dalam versi tentatif pada objek dan tidak terlihat sampai transaksi memanggil *closeTransaction* dan transaksi selesai. Write timestamp dari objek akan lebih awal dari versi tentatifnya, dan read timestamps dapat diwakili oleh jumlah anggota maksimalnya. Pada timestamp ordering, setiap permintaan leh transaksi untuk read atau write pada sebuah objek akan diperiksa apakah sesuai dengan konflik pada operasi. Saat sebuah koordinator menerima permintaan untuk sebuah transaksi, koordinator tersebut akan selalu dapat melakukannya karena setiap operasi dari sebuah transaksi diperiksa secara konsisten oleh transaksi yang sudah dilakukan sebelunya.

Sebuah timestamp method mencegah terjadinya deadlock (buntu), tetapi sangat rawan terhadap terjadinya pengulangan. Sebuah modifikasi yang dikenal sebagai aturan “ignore obsolete write” telah dimplementasikan.

Multiversion timestamp ordering -> pada multiversion timestamp ordering, sebuah daftar objek maupun nilai tentatif objek tersebut disimpan pada masing-masing objek. Daftar ini mewakili sejarah dari nilai objek tersebut. Keuntungan menggunakan beberapa versi adalah operasi read yang datang terlambat tidak perlu ditolak.

1.7 PERBANDINGAN METODE KONTROL KONKUREN

Kita telah mendeskripsikan tiga metode terpisah untuk mengontrol akses konkuren ke data bersama : strict two-phase locking, optimistic methods dan timestamp ordering. Semua metode memiliki beberapa kelebihan dan tempat yang diperlukan, dan ketiganya memiliki batas-batas tertentu untuk operasi konkuren. Beberapa penelitian sistem terdistribusi telah mencari kegunaan dari kunci semantic, timestamp ordering dan pendekatan baru untuk transaksi jarak jauh.

Bekerja pada dua area aplikasi telah menunjukan mekanisme kontrol konkuren tidak selalu memadai. Salah satu dari area yang mengkhawatirkan adalah aplikasi multiuser dimana setiap user berharap dapat melihat pandangan umum objek yang sedang diperbarui oleh user lain. Aplikasi seperti ini membutuhkan data mereka seperti atom saat dihadapkan pada pembaruan yang konkuren dan tabrakan server, dan teknik traksaksi tampak memberikan sebuah pendekatan pada rancangan mereka. Akan tetapi, aplikasi ini harus memiliki dua kebutuhan baru berhubungan dengan kontrol konkuren :

- Pengguna membutuhkan notofikasi langsung atas perubahan yang dilakukan oleh pengguna lain
- Pengguna harus bias mengakses objek sebelum pengguna lain menyelesaikan transaksi mereka, yang menjadi dasar alasan pengembangan tipe baru penguncian yang memicu sebuah aksi saat objek diakses.

1.8 Tabel

TABEL 1. DEADLOCK

Transaksi T		Transaksi U	
Operasi	Lock	Operasi	Lock
a.simpan(100);	Kunci A		
		b.simpan(200)	kunci B
b.ambil(100)			
…	tunggu U membebaskan B	a.ambil(200);	tunggu T membebaskan A
		…	
…		…	
…		…	

a. tabel deadlock

REFERENCES

[1] Coulouris George, Dollimore Jean, and Kindberg Tim, “Distributed Systems Concepts and Design,” Pearson Education Ltd, 2001.

[2] R. Hidayat, “Paper Template in One-Column Format”, http://www.te.ugm.ac.id/~risanur