SOLUÇÕES DE REPLICAÇÃO COM ISOLAMENTO SNAPSHOT

Tiago Delgado Dias
tdias@ptsoft.net

Apresentado como trabalho de síntese da
disciplina de Tópicos Avançados de
Sistemas Distribuídos no contexto do
Mestrado em Engenharia Informática

Universidade Nova de Lisboa –
Faculdade de Ciências e Tecnologia

Fevereiro de 2006

SUMÁRIO

O nível de isolamento snapshot (*snapshot isolation*) surge como uma opção pragmática, de resultados comprovados, para solucionar problemas de performance em aplicações transaccionais em que leituras relativamente complexas concorrem com actualizações de registos. Este é por exemplo o cenário em muitas aplicações web. De forma a permitir que soluções baseadas em isolamento snapshot escalem foi necessário criar soluções de replicação com esta semântica.

Apresentamos este nível de isolamento, o contexto da sua extensão para a replicação e por fim analisamos várias soluções de replicação que implementam o isolamento snapshot.

1. INTRODUÇÃO

O nível de isolamento snapshot permite efectuar leituras e escritas concorrentes (ganho na concorrência) com um nível de consistência relativamente elevado, próximo do nível de isolamento serializável.
Este nível de isolamento, passível de ter o mesmo efeito da execução em série sob condições particulares (a especificar na próxima secção), permite melhorar a performance dos níveis de isolamento clássicos em cenários de escritas e leituras concorrentes, particularmente em modos de utilização mista e em que não sejam frequentes os conflitos entre escritas.

Antes da materialização deste nível de isolamento em [1] existiu bastante trabalho efectuado sobre sistemas transaccionais multi-versões [12] [13] que em vários casos exibiam várias das propriedades do isolamento snapshot. Os sistemas multi-versão apresentam a particularidade de fazer persistir as versões por necessidades de auditoria ou outras inerentes à sua utilização.

Neste documento pretendemos apresentar a informação necessária sobre o isolamento snapshot, estendê-la para cenários de replicação que permitem dotar os sistemas de escalabilidade e finalmente analisar comparativamente várias soluções analíticas ou de implementação de replicação com esta semântica.

Na secção **2. Isolamento Snapshot** apresenta-se uma introdução e contextualização deste nível de isolamento comparando-se brevemente a outros níveis de isolamento. Apresenta-se uma forma de implementação em pseudo-código. Focam-se as duas principais anomalias verificadas neste nível: *write skew* e uma *read anomaly* [2]. Analisam-se formas de garantir que o resultado do isolamento snapshot equivale à execução em série e prova-se informalmente que a anomalia de [2] não se verificaria nas condições analisadas.

Na secção seguinte, **3. Replicação com Semântica de Isolamento Snapshot**, apresenta-se uma introdução e contextualização do isolamento snapshot na replicação.

A penúltima secção, **4. Soluções de Replicação com Semântica IS,** apresenta a análise de implementações e sistemas de replicação que suportam isolamento snapshot ou se relaccionam com este. Os sistemas são comparados entre si durante a análise sempre que se verifica oportunidade. No final é apresentada uma tabela comparativa e sumariamente consolidados os resultados das comparações.

A conclusão sumariza os principais resultados das análises efectuadas.

2. ISOLAMENTO SNAPSHOT

Dois dos objectivos dos sistemas transaccionais são garantir que o processamento das transacções é paralelizado (concorrência) e garantir que transacções realizam os seus objectivos da mesma forma que aconteceria se fossem executadas em sequência (consistência).

Os sistemas transaccionais utilizam diferentes níveis de isolamento de forma a suportar diferentes cenários de proporções entre concorrência e consistência. De forma a aumentar a concorrência é necessário relaxar os requisitos de consistência. Num dos extremos encontramos o nível isolamento serializável que garante a consistência totalmente uma vez que só permite ordenações equivalentes às ordenações em série das transacções, neste caso a concorrência é severamente limitada. A ausência de qualquer controlo, no extremo oposto, permite a máxima concorrência ao preço de não garantir consistência entre transacções.

As implementações de vários níveis de isolamento são usualmente efectuadas através de locks sobre registos (por exemplo linhas das tabelas). Existem dois tipos de locks (leitura e escrita), sendo que os locks de escrita só podem existir quando mais nenhum lock existe para o mesmo registo. Diferentes níveis de isolamento são implementados conforme o momento em que a transacção obtém os locks e os liberta, por exemplo os locks de escrita são sempre mantidos até ao fim das transacções mesmo nos níveis de isolamento mais baixos (os que oferecem menor nível de consistência) [1].

Passados vários anos em que as implementações comerciais apenas suportavam níveis de isolamento baseados em locking baseados nas definições do ANSI SQL (READ UNCOMMITED, READ COMMITED, REPEATABLE READ e SERIALIZABLE) a publicação de [1] veio consolidar o conhecimento existente sobre estes níveis de locking (colmatando algumas lacunas nas definições dos mesmos) e tornar visível um conjunto de novos níveis de isolamento dos quais destaca o nível SNAPSHOT ISOLATION, referido neste artigo como isolamento snapshot.

Desde o final dos anos 80 que têm sido referidos mecanismos de controlo de concorrência [12] [13] baseados em múltiplas versões em vez de locking. Verifica-se contudo que foi através da consolidação em [1] e da nomenclatura SNAPSHOT ISOLATION que este nível de isolamento se popularizou, tendo sido implementado em vários produtos comerciais ainda nos anos 90.

O isolamento snapshot gere múltiplas versões dos registos de forma a suportar operações de escrita e leitura simultâneas sobre o mesmo registo. As versões consolidam temporariamente um estado estável da base de dados (i.e. o estado após o *commit* de alterações), este estado é assim usado para servir as leituras enquanto outra transacção efectua a escrita do mesmo registo.

Relativamente às escritas se duas transacções concorrentes (t1 ‖ t2, tal que i1[1] < i2 < c1[2] ou i2 < i1 < c2) escreverem o mesmo registo então uma delas terá que

[1] iX é o início da transacção X.

abortar[3]. Em comparação os mecanismos de locking nesta situação levariam a que a segunda transacção a tentar a escrita esperasse pelo fim da primeira, sem necessidade de abortar.

Optamos por instanciar um algoritmo que implementa o isolamento snapshot de forma a suportar a análise das secções seguintes, uma definição formal é facilmente acessível a partir de [1] [2] [5] [7] [10].

Seja t1 uma transacção com isolamento snapshot:

- Ao iniciar obtém o número de sequência do último *commit* na base de dados (*nc*), será o seu número de versão - *nv*
- Durante a transacção as operações processam-se da seguinte forma:
  - Leituras:
    É obtida a versão mais recente menor ou igual a *nv*.
  - Escritas:
    É gerada uma nova versão, indisponível a outras transacções até ao *commit* (conseguido por exemplo com v = ∞).
- Antes da operação de *commit* é necessário efectuar a validação de que as escritas da transacção não interferem com escritas de outras transacções concorrentes:
  - Se existe pelo menos um registo escrito pela transacção para o qual existe uma versão posterior a *nv* (diferente de ∞) a transacção terá que abortar.
  - Caso contrário pode-se efectuar o *commit*:
    i. Incrementar o número de sequência do último *commit* - *nc*
    ii. Alterar a versão das escritas efectuadas para *nc*, tornando-as visíveis.

Para além de potenciar a concorrência de leituras e escritas este nível de isolamento é bastante atractivo porque está relativamente equiparado aos níveis de isolamento mais fortes como o REPEATABLE READ. Através da abordagem de [1] que compara os vários níveis de isolamento distinguindo-os através das anomalias que verificam, vemos que o isolamento snapshot não verifica a anomalia *phantom*[4] ao contrário do REPEATABLE READ. Contudo permite a anomalia *write skew* não permitida pelo REPEATABLE READ.

A anomalia *write skew* consiste na violação de uma condição existente entre dois registos distintos, sendo que cada registo é alterado numa transacção distinta e em cada transacção isolada não se viola a condição.

Outra anomalia verificada no isolamento snapshot é apresentada em [2]. Exista uma condição entre duas variáveis X e Y na qual a escrita de Y seja alterada, e[5](Y

---

[2] cX é o commit da transacção X.

[3] Isto de forma a evitar uma anomalia conhecida como Lost Updates, ver [1].

[4] Uma vez que as leituras são todas efectuadas a partir das versões existentes no início da transacção.

[5] e(V) é a operação de escrita da variável V.

com condição(X, Y) falsa) ≠ e(Y com condição(X, Y) verdadeira), e tenhamos t1, t2 e t3 tal que t1 se inicie com condição(X, Y) verdadeira, tal como t2. t1 irá efectuar uma escrita sobre Y com resultado R' (diferente da operação da mesma operação de escrita com condição(X, Y) falsa: R). A transacção t2 efectua uma alteração a X de tal forma que condição(X, Y) passa a ser falsa. t3 efectua a leitura de ambas as variáveis X e Y. Tenhamos o seguinte condicionamento à ordem de execução c2 < i3, i3 < c1 e i1 < c2, então a leitura efectuada em t3 irá apresentar um estado de X, Y que não verifica a condição, uma leitura posterior a c1 irá mostrar que foi feita uma operação posterior a t3 sobre Y em que a condição entre X, Y era verdadeira. Desta forma a leitura de t3 não deveria ser possível. Esta anomalia não ocorreria sob isolamento serializável.

Por vezes seria desejável evitar estas anomalias no nível de isolamento snapshot, obtendo os resultados de uma execução em série. Em [3] é apresentado um conjunto de regras que permite garantir que a execução com isolamento snapshot generalizado[6] verifique as propriedades da execução em série.

É apresentada uma regra de validação adicional antes do processo de *commit* que pretende garantir que a transacção não efectuou leituras de registos que foram escritos posteriormente ao início da transacção, caso a regra seja violada a transacção deverá ser abortada.

Uma abordagem alternativa é também apresentada e consiste em aplicar uma regra mais restritiva mas que pode ser aplicada como validação de aplicações (em vez de durante a execução) de forma a verificar se cumprem os requisitos para garantir serialização com o nível de isolamento snapshot. Uma vez que esta abordagem aplica as regras a todos os pares de transacções possíveis numa aplicação verifica se duas transacções têm leituras e escritas em comum, o que não será o caso de um conjunto amplo de aplicações que falharão esta verificação.

Uma vez que as regras apresentadas em [3] se aplicam ao isolamento snapshot generalizado aplicam-se também ao isolamento snapshot convencional, visto que o isolamento snapshot convencional é um caso particular do isolamento snapshot generalizado em que a versão lida é sempre a mais recente ao início de cada transacção.

Para implementar a abordagem dinâmica proposta em [3] no algoritmo de implementação de IS apresentado anteriormente bastaria compilar durante a transacção, para além do *write-set*, o *read-set*. A transacção teria que abortar se desse *read-set* existisse algum registo com uma versão actual na BD mais recente do que a versão *nv* (correspondente à versão lida pela transacção).

Verificamos que esta abordagem garante a ausência tanto de *write skews* como da anomalia de leitura apresentada em [2], vejamos para o caso da anomalia de leitura:

- A transacção t2 continua a poder efectuar o *commit* uma vez que as leituras que efectua de X e Y não foram escritas após i2.

[6] O nível de isolamento snapshot generalizado é uma classificação mais geral que engloba o nível de isolamento comum que temos estado a abordar. No nível generalizado as leituras podem utilizar qualquer versão anterior ao seu início.

- A transacção t1 ao tentar efectuar *commit* terá que ser abortada porque o valor que leu de X foi alterado durante a sua execução por t2 que efectua *commit* antes de t1. t3 passa assim a ler uma situação estável da base de dados uma vez que quando t1 fôr executar novamente já a condição (envolvendo X e Y) não se verificará.

Em [8] é proposto um outro algoritmo com o mesmo fim (e algumas semelhanças), o de garantir a serialização das transacções utilizando apenas o isolamento snapshot.

3. REPLICAÇÃO COM SEMÂNTICA DE ISOLAMENTO SNAPSHOT

De forma a permitir que as soluções baseadas em isolamento snapshot escalem é necessário suportar o isolamento snapshot ao nível dos mecanismos de replicação.

Antes de mais devemos debruçar-nos um pouco sobre as soluções de replicação utilizadas para além do isolamento snapshot.

Existem dois extremos de abordagens distintos:

- Soluções *eager* em que a replicação é efectuada no contexto da transacção local
- Soluções *lazy* em que a replicação é efectuada de forma desacopolada da transacção local, i.e. após o *commit* local

As soluções *eager* premeiam a consistência em prol da concorrência, as soluções *lazy* têm a abordagem inversa. Na prática verifica-se que as soluções *eager* não escalam para cenários com um volume não negligenciável de escritas. [6] apresenta um conjunto de gráficos que relaccionam a factor de escala em função do peso das escritas, verificando-se que quando este peso não é bastante baixo o factor de escala varia como função logarítmica do número de nós. Ora para as soluções *eager* típicas (*read-one write-all*) o processo de escrita envolve todos os nós tornando-se extremamente pesado, daí a natural relacção entre os maus resultados analíticos e práticos.

Dentro das abordagens *lazy* é possível utilizar uma cópia primária para efectuar as escritas, existem várias soluções bem sucedidas com esta abordagem, desde que o percentagem de actualizações não seja especialmente elevada, [5] é um tal caso. Contudo são frequentes as soluções hibrídas (entre *eager* e *lazy*) em que existe um processo de coordenação entre os nós antes de efectivar o *commit* (através de protocolos de comunicação em grupos em [4] [6] [7]) que determina o seu sucesso futuro.

As soluções de replicação com semântica de isolamento snapshot oferecem usualmente a visão ao cliente de um sistema único no modo de isolamento snapshot (*1-copy-snapshot isolation*). Para este efeito tomam a forma de um

componente central ou distribuído, frequentemente de *middleware* ou sob a forma nativa de um SGBD ou protótipo do mesmo.

Este tipo de dimensões nas soluções são analisadas de forma comparativa na secção seguinte sendo o resultado sistematizado numa tabela no final da secção.

Verifica-se que é através de uma implementação replicada do algoritmo centralizado de IS, que são implementadas grande parte das versões de replicação com isolamento snapshot. Nas soluções de *middleware* parte do algoritmo é garantido pelo isolamento snapshot das bases de dados de cada uma das réplicas, contudo, por exemplo [7], implementa toda a componente de verificação de interferências. No extremo temos soluções como em [5], em que todo o algoritmo de isolamento snapshot (gestão de versões, verificações de interferências) é executado pela base de dados da cópia primária, e o *middleware* está apenas a efectuar um papel moderador, de canalização e propagação dos *write-sets*.

4. SOLUÇÕES DE REPLICAÇÃO COM SEMÂNTICA IS

Prefix-consistent Snapshot Isolation [3]

Em [3] é apresentado um sistema de replicação que garante um nível de isolamento pertencente ao mesmo grupo do isolamento snapshot, em que as leituras obtém uma versão anterior ou igual à existente no início da transação: o isolamento snapshot generalizado.

Seja t1 uma transacção local a determinado nó, se c2 < i1, onde t2 é uma transacção de um nó remoto, o nó local poderá não ter disponível o valor X escrito por t2 quando necessita de efectuar a sua leitura em t1. Desta forma, endereçando este problema de uma forma pragmática, o PCSI utiliza a versão mais recente localmente disponível no instante que a transacção local inicia. Posto de outra forma após o *commit* de uma transacção em determinado nó é possível que outros iniciem transacções de leitura e respondam às suas leituras com valores mais antigos (pré-*commit*).

Um dos objectivos desta implementação é evitar bloqueios, necessários nos sistemas que garantem isolamento snapshot, a ocorrem na espera das leituras pelo último valor *commited*.

Na implementação proposta as transacções correm localmente em cada nó, utilizando somente recursos locais, até ao momento do *commit*. É proposta uma abordagem em que a validação das transacções em fase de *commit* é feita num único nó e outra em que é distribuída, focamo-nos sobre a última.

O algoritmo é semelhante a uma versão distribuída do algoritmo que apresentamos na secção 2., com as seguintes alterações:

- o *nv* de uma transacção é igual ao valor local de *ns* (que é sempre menor ou igual a *max(nv)* de entre todos os nós).
- a operação de *commit* é distribuída, propagando-se o *write-set*[7] através de um *broadcast* (que garante que todos os nós o irão receber pela mesma ordem em relacção a outros *commits*).

Cada nó irá validar a transacção (uma vez que já processou todas as transacções a efectuar *commit* antes devido à ordenação do canal de comunicações). A decisão será igual em todos os nós. Após validação, se bem sucedida, é efectivado o *commit*, sendo o valor local de *nv* incrementado e associado às versões das escritas da transacção.

Para o processo de recuperação de um nó falhado é proposto que sejam mantidas nos outros nós as alterações (*write-sets*) desde o momento da falha, sendo que irão fornecer estas alterações ao nó falhado quando em recuperação.

Esta solução é particularmente adequada se os nós recuperarem após as falhas e estas sejam curtas. No caso de os nós permanecerem falhados durante períodos longos ou nunca recuperarem seria necessário adoptar um processo de limpeza, após o qual a recuperação de um nó implicaria uma cópia total.

A performance do algoritmo é calculada analiticamente e por simulação. Dos resultados apresentados há a destacar que o PCSI é mais útil em casos em que o rácio latência da rede / latência de processamento dos nós é maior, i.e. o PCSI obtém melhores resultados comparado com o isolamento snapshot convencional quando a latência da rede é alta e o CPU é rápido. Face ao cenário actual em que a performance da rede cresce a um ritmo mais acelerado do que a performance dos CPUs este resultado pode ser menos interessante do que os autores indicam. Contudo segundo os resultados numéricos a performance do PCSI só é muito inferior (para um rácio de latência de rede/CPU relativamente baixo) à do isolamento snapshot convencional quando a versão utilizada pelas leituras é bastante desactualizada (o que deverá ser minorado em função da rapidez da rede).

O modelo também assume uma uniformidade nos acessos a dados contudo nos sistemas reais é frequente existir um subconjunto de dados que é acedido com maior intensidade (*hotspot*).

[7] *Write-set* é um conjunto de pares (registo, valor escrito). *Write-set* de uma transacção é um tal conjunto ao qual pertencem todos e apenas os registos escritos no decorrer da transacção.

Bettina et al. [4]

Este é um protocolo em grande parte suportado pelas propriedades oferecidas por mecanismos de comunicação entre grupos. É por exemplo o mecanismo de comunicação que define a ordem dos *commits* ao garantir a ordem total de entrega das mensagens de *commit*.

Neste sistema as leituras são efectuadas localmente em cada nó, sendo as operações de escrita compiladas num *write-set* enviado para todos os nós na operação de *commit*.

Esta abordagem é comum a [3] [6] [7]. Varia, relativamente a [3], na medida em que as leituras são efectuadas da versão local mais recente (que pode não ser a global), enquanto que em [4], é sempre a última versão. Em [6] a ordem de serialização é imposta no início da transacção através da sua entrega com ordem total.

Para além de apresentar um algoritmo para suportar isolamento serializável é apresentado um algoritmo para isolamento snapshot. As versões necessárias para as leituras sobre isolamento snapshot (correspondentes ao instante de início da transacção) são reconstruídas a partir do *log* local. As escritas vão compilando o *write-set* da transacção que é transmitido de forma fiável e com garantia de ordem total no instante de *commit* (estes passos não são apresentados na Figura 2 [4] contudo são necessários e referidos no texto).

Ao receber o *write-set* cada nó irá analisar cada uma das escritas nele contidas. Caso exista um lock posterior ao início da transacção ou a versão escrita fôr mais recente do que o início da transacção esta será abortada. Face às propriedades de transmissão utilizadas para os *write-sets*, no instante de processamento do *write-set* o estado de cada nó será igual ao dos restanres quando receberam ou venham a receber o *write-set*, resultando na mesma decisão de *abort* ou *commit* em todos os nós.

São analisados três cenários de tratamento de falhas de um nó durante a transmissão de uma mensagem com o *write-set* (pré-*commit*). Os três cenários abordam níveis de consistência diferentes, dependendo da exigência de atomicidade na transmissão dos *write-sets* (e de uma mensagem de *commit* necessária para o isolamento serializável analisado nesta secção). À medida que se relaxam as exigências de atomicidade, removendo carga do protocolo, torna-se mais complexo o processo para atingir um concenso e posteriomente de recuperação do nó falhado.

Não é apresentado o algoritmo de sincronização de nós falhados em recuperação, contudo poderia-se aplicar uma abordagem equivalente à de [3] analisada anteriormente.

[4] propõe ainda um modelo híbrido de isolamento em que é utilizado *two-phase-locking* estrito (uma implementação de isolamento serializável) para as escritas e isolamento snapshot para as leituras. Este modelo garante a serialização podendo facilmente verificar-se que anomalias como a referida em [2] não se poderiam verificar neste modelo (dependo da ordem de execução: t2 não poderia escrever X antes de c1 ou t1 não poderia ler X antes de c2).

São apresentados resultados de um modelo de simulação dos quais destacamos:

- A latência da rede afecta negativamente os tempos de resposta, conforme esperado, especialmente quando as transacções são longas e sobre isolamento serializável (degradação exponêncial). Para os casos de isolamento snapshot e *cursor stabilitiy* a degradação aparenta ser linear.
- O isolamento snapshot tem taxas de *abort* relativamente elevadas para situações intensivas de escrita, contudo a performance não é visivelmente afectada.

Ganymed [5]

O sistema descrito em [5] implementa o algoritmo RSI-PC onde as transacções de escrita são efectuadas numa cópia primária e as transacções só de leitura nos nós restantes.

É explorada e oferecida a opção de correr as transacções de escrita num nível de isolamento em que o isolamento snapshot, no que diz respeito às versões de leitura, é aplicado a cada instrução SQL como se esta compusesse uma transacção isolada, obtendo uma versão para leitura potencialmente diferente a cada instrução.

Conforme previsto em [3] as leituras locais são bloqueadas sempre que a versão da réplica não está sincronizada com a versão global e até que esta sincronização se verifique. Em [5] é contudo proposta uma solução à base de um valor mínimo de "frescura" solicitado pelos clientes, permitindo ler versões locais mais antigas do que a versão global. Esta solução degenera o isolamento snapshot em PCSI, sendo a mesma que é apresentada em [3].

Esta solução necessita que os clientes submetam as transacções só de leitura explicitamente identificadas. Caso contrário irão ser executados na cópia primária compromentendo a escalabilidade do sistema.

Relativamente à tolerância a falhas é brevemente proposta uma solução de número NC componentes, onde NC << NN, sendo NN o número de nós, isto visto que a carga esperada no middleware é muito inferior à esperada nos nós de BD.

Dos resultados experimentais verifica-se que a capacidade de processamento escala de forma linear, já o tempo de resposta por transacção escala de forma sub-linear. Isto deverá dever-se ao facto de existirem poucos conflitos entre escritas, permitindo às transacções recebidas irem sendo escalonadas à medida que podem ser processadas. Ao processar mais transacções a capacidade de processamento disponível por transacção é menor e daí que o tempo de processamento por transacção aumente de forma não linear. Eventualmente pode-se prescindir da capacidade de processamento através de um limite máximo de transacções simultâneas por nó com o objectivo de garantir um tempo máximo de processamento inferior, por transacção.

Um facto interessante é que os resultados com um e dois nós são equivalentes. Como é indicado uma boa parte da carga dos testes TPC-W utilizado é na execução de leituras pesadas. Executando as transacções de leitura com isolamento snapshot é equivalente que sejam executadas na cópia primária ou numa réplica uma vez que não entram em conflito com outras transacções, a capacidade de processamento de transacções vai estar apenas limitada pela capacidade de processamento dos nós.

Jiménez-Peris et. al [6]

[6] introduz o conceito de classes de conflitos. As classes de conflitos básicas identificam grupos de registos disjuntos, agrupando-se em classes de conflitos compostas. O objectivo das classes de conflitos é definir um mecanismo de particionamento da carga (para as transacções de escrita), sendo que cada nó controla um conjunto de classes de conflitos compostas.

As transacções têm que ser submetidas pelo cliente com a identificação explícita da classe de conflitos composta a que acedem, a transparência da solução é bastante afectada com este requisito.

Vejamos um exemplo do particionamento em classes de conflitos. Sejam C1, C2 e C3 classes de conflitos básicas que particionam uma tabela de passagens em portagem pelas classes dos veículos que passaram. A tabela é totalmente abragida pela classe composta {C1, C2, C3}. Sejam N1 e N2 os dois nós que suportam a replicação da tabela de passagens, vamos assumir o particionamento da seguinte forma: N1 gere {C1}, {C2} e {C1, C2}, N2 gere {C3}, {C1, C3}, {C2, C3} e {C1, C2, C3}[8]. Seja t1 uma transacção que escreve duas passagens, uma de uma viatura da classe 2 e outra da classe 1 a classe de conflito de t1 seria {C1, C2}, logo t1 iria executar em N1.

O algoritmo implementado inicia-se com uma entrega (*to-delivery*) em ordem total da transacção a todos os nós (total order broadcast with optimistic delivery), precedida de uma entrega sem ordenação (*opt-delivery*). Logo após a *opt-delivery* todos os nós particionam a transacção e inserem-na em filas específicas para cada classe de conflitos básica[9]. O nó que gere a classe de conflitos composta da transacção irá iniciar a execução da transacção assim que todos os seus elementos seja primeiros nas filas, o que pode ocorrer antes ainda da *to-delivery*.

Um broadcast não ordenado do *write-set* (composto durante a execução) pode ser efectuado quando duas condições forem cumpridas: a transacção tenha terminado e a *to-delivery* tenha acontecido. Cada um dos nós restantes irá aplicar o write-set assim que receber o broadcast e todos os elementos da transacção sejam os primeiros das filas respectivas.

A utilização da *opt-delivery* permite iniciar a transacção assim que seja recebida, em vez de ter que se esperar pela recepção em ordenação total, assumindo que a ordenação garantida pelo meio físico é muitas vezes total no caso de redes locais.

[8] Note-se que a abragência dos nós é sempre total para as classes de conflitos compostas possíveis.

[9] Uma transacção só de leitura irá ser executada localmente no nó inicialmente contactado pelo cliente, segundo um escalonamento particular apresentado mais adiante.

Caso a *opt-delivery* não corresponda à ordem total a situação é avalida no momento de recepção da *to-delivery*.

As transacções só de leitura são enviadas apenas para o nó local (onde o cliente está ligado), são efectuadas localmente, sendo colocadas nas filas das classes de conflictos básicas respectivas, após as transacções *to-delivered* e antes das restantes, em vez de na ordem de recepção.

Ao contrário do que ocorre no nível de isolamento snapshot, as operações de leitura em transacções de escrita não podem ser concorrentes com outras transacções.

Se o algoritmo particionasse as operações dentro das transacções e aplicasse o escalonamento apresentado para as transacções de leitura às operações de leitura não se obteria uma execução equivalente ao nível de isolamento snapshot, nem sequer isolamento snapshot generalizado [3], uma vez que as leituras iriam poder ler versões distintas conforme a fila em que residissem, em vez de uma só versão, a disponível no início da transacção.

Foi já verificado que a ordenação total do broadcast das transacções define a sua ordem de serialização global.

Analisemos uma situação particular: seja t1 uma transacção *to-delivered* em todos os nós, tendo já iniciado a execução no nó que gere a sua classe (Ct1). Após este estado é entregue (*to-delivery*) uma nova transacções (t2), da qual $Ct2 \cap Ct1 \neq \emptyset$. Desta forma existirão filas em que t1 precede t2, logo t2 não iniciará a sua execução antes de t1. No caso de t1 ser uma transacção longa, e existindo N transacções curtas nas condições de t2 todas terão que esperar por t1, diminuindo a concorrência. Este comportamento advém da utilização da ordem total de entrega em vez da ordem total da efectivação do *commit* (como acontece em [3], [4] e [7]).

As filas funcionam como mecanismo de *locking* estrito (*locks* obtidos no início da transacção e libertos apenas no fim), implementado com o nível de granularidade definido pelas classes de conflitos básicas. No modelo apresentado as transacções com conflitos são executadas em série, sendo ainda mais restritivo do que o modelo híbrido definido em [4].

Uma outra limitação deste algoritmo pode ser a necessidade de submeter as transacções de forma integral, uma vez que após a recepção necessitam de ser particionadas e inseridas em cada uma das filas das classes de conflitos. Esta limitação é identificada em [5]. Contudo, no caso de a aplicação poder identificar as classes básicas acedidas antes de *runtime* pode-se implementar o algoritmo proposto em que o broadcast inicial envia uma assinatura da transacção, com a classe de conflitos composta. No momento em que a transacção inicia a sua execução no nó que gere a classe indicada iria receber o *stream* aberto para envio da transacção no qual poderia submeter as respostas a queries.

Dos resultados experimentais destacamos o facto de a escala da solução com níveis elevados de escritas ser sub-linear.

SI-Rep [7]

[7] apresenta duas versões do mesmo algoritmo. Uma centralizada (SRCA) depois estendida para uma versão totalmente distribuída (SRCA-Rep). Debruçamo-nos sobre a primeira por uma questão de simplicidade:

- No início de cada transacção obtém-se a estampilha mais recente para um nó onde a transacção será local.
- As operações de leitura e escrita são implementadas da forma usual (neste caso suportadas pelo isolamento snapshot na base de dados).
- O *commit* implica a obtenção do *write-set*, a validação relativamente às transacções concorrentes já *committed* (verificar se os *write-sets* se interceptam, caso afirmativo é necessário abortar a transacção) e, finalmente a adição à fila de transacções para *commit*, mantida por cada nó.
- Ao processar o primeiro elemento da fila de *commit*, em cada nó, é incrementada a estampilha (mantida por nó) e são aplicadas as alterações na BD, de forma atómica.

Uma vez que a estampilha de início da transacção é local em cada nó, e quando alterada existem sempre as versões respectivas na base de dados, não se verifica a necessidade de esperar pela versão de início da transacção para as leituras, ao contrário do que acontece em [5]. É contudo necessário esperar por uma operação de *commit* em curso (se existir) uma vez que é um processo atómico, espera esta que está de acordo com a presença de bloqueios, nas soluções de isolamento snapshot, proposta em [3].

O algoritmo apresentado em [7] não apresenta remoções da lista dos *write-sets* das transacções efectuadas. Poderia aplicar-se a solução apontada para a recolha de transacções para um nó falhado em [3], sendo contudo mais viável a manutenção de uma lista de *write-sets* estritamente operacional. Esta lista poderia ter apenas o resultado das transacções *committed* posteriores a pelo menos uma transacção em execução, garantido-se que a validação continuava possível.

É de destacar que na solução proposta os clientes não necessitam de explicitar transacções só de leitura uma vez que estas transacções são tratadas da mesma forma que as transacções de escrita.

Na versão distribuída a ordem de *commit* é definida pela ordem total da entrega do *write-set*. Existe uma validação local antes da transmissão do *write-set*, que visa detectar antecipadamente violações, contudo a validação definitiva é executado por todos os nós antes de aplicar o *write-set*, como o estado em que esta validação ocorre é o mesmo em todos os nós, também o resultado (*commit* ou *abort*) será o mesmo.

Relativamente à tolerância a falhas, em particular durante um *commit* numa réplica do *middleware*, se o nó falha após a transmissão do *write-set* (portanto garantida) e antes de o receber ou efectuar o *commit* deste na sua BD, a transacção

será dada como *committed* nas restantes réplicas e para o cliente (que irá ser redirecionado para outra réplica), mas não ficará *committed* na BD gerida pelo middleware falhado. Desta forma é necessário que o processo de recuperação de nós falhados garanta a reconciliação adequada.

Quando um nó falha a sua eventual recuperação em [7] implica a paragem do sistema e a cópia integral de uma das réplicas. È apontada uma solução similar à adoptada em [3] e já abordada neste documento. Esta mesma solução poderia serviria também na reconciliação conforme necessário face ao exposto no parágrafo anterior.

Relativamente aos resultados experimentais verifica-se que a proporção carga processada / tempo de resposta escala em função do número de réplicas, naturalmente de forma mais suave para cenários só de leitura.

É apresentado um resultado comparativo com o protocolo de [6] que tem uma prestação nesta mesma escala ainda inferior à de uma solução centralizada. Este comportamento deverá dever-se às várias limitações do protocolo utilizado em [6], em particular a serialização das escritas. Prevê-se que o protocolo de [5] tivesse uma prestação próxima da de [7] desde que para operações de escrita pouco intensivas.

Phatak et al. [8]

[8] apresenta um sistema em que existe um base de dados central e N bases de dados móveis capazes de operar em modo desconectado. Quando em operação normal (conectada) as transacções são executadas no servidor central. Antes de iniciar um período de operação desconectada cada nó tem que obter uma cópia dos registos que pretende oferecer ao cliente.

No modo de operação desconectada pode ser utilizado qualquer nível de isolamento, sendo apenas necessário garantir a recolha dos *read-sets* e *write-sets* de cada transacção, cruciais para a fase de reconciliação.

Os clientes podem ainda definir funções de reconciliação e custo de reconciliação para cada transacção de forma bastante similar ao utilizado em [15]. É de notar que este sistema pretende resolver um problema comum ao BAYOU [15] de uma forma mais generalista e para transacções sobre bases de dados.

Cada reconciliação irá testar o cenário de reconciliação contra todas as versões activas na base de dados central. Como se a transacção ocorre-se no passado representado pelas versões.

É ainda de notar que todas as transacções efectuadas em modo desconectado por cada cliente irão ser reconciliadas numa mesma transacção, o que aumenta a duração média das transacções num factor N (onde N é o número médio de transacções efectuadas por cada sessão desconectada), e irá ter um impacto negativo ao nível da disponibilidade. Desta forma [8] propõe a paralelização das várias transacções de cada sessão desconectada desde que se utilize uma exclusão mútua para acesso à versão mais recente da *snapshot*.

SQL 2005 [9]

Em [9] apresenta-se a implementação de isolamento snapshot no Microsoft SQL Server. O artigo centra-se na especificação dos cenários de utilização e modelos de utilização respectivos. Destacamos os seguintes cenários dos apontados como mais vantajosos para utilizar isolamento snapshot:

- Sistemas OLTP com tipos de carga mista (cenário típico em aplicações web)
- Geração de relatórios adhoc em tempo real
    a. Pode ser efectuado sobre uma réplica da base de dados (transaccional[10]. de forma a conseguir resultados próximos do tempo real) de forma a minorar o impacto dos relatórios sobre a OLTP.

Relativamente à implementação não são revelados factos suficientes para uma análise mais adequada (por exemplo se a replicação pode utilizar este nível de isolamento) É indicado contudo que as versões, com excepção da actual, são mantidas numa base de dados de sistema, a *tempdb*. Esta base de dados é mantida em memória, daí que seja volátil. Contudo, em caso de falha, uma vez que as versões diferentes da actual só servem as transacções já existentes, que são abortadas na situação de falha, este facto não se apresenta como um problema.

Um outro campo de aplicação do isolamento snapshot é nos sistemas de federação. Um sistema de federação suporta transacções que abarcam sistemas de bases de dados heterógeneos, com modelos de dados distintos que necessitam de ser afectados numa mesma transacção. Desta forma um sistema de federação não poderá efectuar o mesmo aproveitamento dos *write-sets* que os sistemas até agora focados.

Os sistemas de federação são por necessidade de negócio sistemas de replicação do tipo *eager* na medida em que o resultado de uma transacção depende sempre de todos os sistemas (BDs) envolvidos. Por exemplo só é possível garantir que uma compra ficou paga e o cliente será servido quando o banco tiver retirado o valor da compra ao cliente e, na mesma transacção, a entidade vendedora tenha garantido a disponibilidade do serviço e que será oferecido ao cliente.

Em [10] é apresentado um sistema de federação que utiliza o suporte de isolamento snapshot por cada uma das bases de dados envolvidas para oferecer isolamento snapshot ao nível global da transacção federada.

O interesse do isolamento snapshot não é exclusivo dos sistemas relaccionais. Em [11] é apresentado um exemplo de como se pode aplicar o nível de isolamento snapshot a uma lista unidireccionalmente ligada, destacando o efeito da anomalia

[10] De notar que o SQL Server implementa uma mecanismo básico de replicação baseado na cópia de snapshots temporais (por exemplo de hora a hora) e que tem o nome de Snapshot Replication, em nada relaccionado com o isolamento snapshot.

write skew na manipulação da lista, é ainda mostrando que esta anomalia não seria verificada numa lista duplamente ligada.

Devido à necessidade de manter múltiplas versões do mesmo objecto de forma a ter um histórico sempre disponível em [14] é apresentado um sistema que gere a consistência de múltiplas versões de uma base de objectos. O problema endereçado é inerentemente complexo uma vez que é necessário que todas as versões estejam consistentes. Seria interessante uma análise de como o isolamento snapshot poderia tirar proveito deste tipo de sistema uma vez que as versões geradas são persistentes, ao contrário do que acontece usualmente, em que só interessa o estado final e as versões existem para alimentar leituras iniciadas num contexto que passou a ser histórico após escritas mais recentes.

Na Tabela 1 apresenta-se uma comparação sumária dos principais sistemas analisados. Quase todos implementam a semântica de isolamento snapshot (em particular *1-copy-snapshot isolation*), excepto [3] que explora um modelo menos restrito e de implementação mais simples. Todas as abordagens implementam as escritas em todos os nós, excepto [5].

Esta análise foi distribuída entre sistemas de *middleware* e sistemas nativos, sendo que os sistemas de *middleware* são todos suportados por bases de dados que suportam o nível de isolamento snapshot.

Apesar de prôr uma solução distribuída [5] é a única solução implementada de forma centralizada, uma vez que o algoritmo implementado é leve a escalabilidade da solução pode não ficar comprometida ao nível do *middleware* centralizado até um factor de carga particularmente elevado.

Todas as soluções propagam as alterações através de *write-sets*, qualquer solução que não o faça estará a incorrer em carga de processamento adicional nas réplicas (com um factor polinomial: $n - 1$).

A maioria das soluções é baseada na comunicação entre grupos, excepto [5] (devido ao modelo simplificado) e [8] (por ser uma solução centralizada).

Relativamente à transaparência avaliamos as soluções de m*iddleware* sendo que apenas a [7] é totalmente transparente, no caso de [5] é necessário explicitar as transacções só de leitura, em [6] classificar a transacção relativamente aos registos que escreve.

Finalmente grande parte das soluções implementa a detecção de conflitos entre transacções de escrita, no caso de [5] não necessita uma vez que as transacções com escritas são executadas todas na mesma BD, em [6] devido ao particionamento e ao algoritmo de filas. A solução [8] implementa ainda a resolução de conflitos.

Dimensão de comparação	PCSI [3]	Kemme et. al [4]	Ganymed [5]	Jiménez-Peris et. al [6]	SI-Rep [7]	Phatak et al. [8]
Nível de isolamento ofercido	GSI/PCSI	IS entre outros	IS	IS	IS	IS
Locais de actualização	*Write-all* *Read-all*	*Write-all* *Read-all*	*Write-one* *Read-all*	*Write-all (particionado)* *Read-all*	*Write-all* *Read-all*	*Write-all* *Read-all*[11]
Tipo de solução	Nativa	Nativa	*Middleware*	*Middleware*	*Middleware*	Indefinida
Depende de BDs que suportem IS[12]	-	-	Sim	Sim	Sim	Sim
Solução centralizada / distribuída	Distribuída	Distribuída	Centralizada, c/ nó de *backup*[13]	Distribuída	Distribuída	Centralizada
Utiliza comunicação entre grupos	Sim	Sim	Não	Sim	Sim	Não
Propaga alterações através de *write-sets*	Sim	Sim	Sim	Sim	Sim	Sim
Transparência	-	-	Necessário indicar trans. só de leitura	Necessário indicar classe de conflito	Total	-
Detecção de conflitos	Implementada pela solução	Implementada pela solução	Suportada pela BD	Suportada pela BD	Implementada pela solução	E resolução implementada pela solução

Tabela 1. Comparação dos principais sistemas analisados.

[11] Desde que os dados lidos/escritos tenham sido replicados para utilização *offline*.

[12] Só aplicável ao middleware

[13] É muito brevemente proposto um modelo distribuído, contudo não foi considerado uma vez que não se apresenta o protocolo de sincronização entre componentes.

5. CONCLUSÃO

Analisámos o nível de isolamento snapshot, contextualizando-o relativamente a outros níveis baseados em locking. Apontamos anomalias que o distinguem do isolamento serializável e mostramos como atingir o mesmo nível. Quase todas as soluções comerciais de bases de dados implementam este nível de isolamento, focou-se brevemente o caso do recente Microsoft SQL Server 2005.

Verificou-se onde a extensão para a replicação situa o isolamento snapshot, entre soluções *eager* e *lazy*.

Apresentaram-se as soluções de [3] [4] [5] [6] [7] [8], entre as quais existem bastantes semelhanças (a propagação de alterações através de *write-sets*, a utilização de protocolos de comunicação fiável em grupos, etc.). Focou-se uma solução que relaxa o nível de isolamento snapshot na dimensão da idade das versões históricas lidas [3]. Abordaram-se várias soluções que acentam sobre uma plataforma de comunicação fiável em grupos [4] [6] [7]. Em [5] abordou-se a única solução de cópia primária ainda assim com resultados de performance bastante impressionantes. Em [6] analisamos um conjunto de limitações que o afastam relativamente às restantes soluções analisadas. Em [7] encontrámos uma das soluções mais versáteis e escaláveis onde existe ainda algum espaço de desenvolvimento/maturação, nomeadamente na componente de recuperação de nós falhados. Em [8] encontramos uma abordagem para a replicação em bases de dados com operação desconectada, especialmente interessante por implementar mecanismos de reconciliação baseados no isolamento snapshot.

Desta análise retirámos várias linhas de arquitectura de uma solução como seja a necessidade de propagar as alterações em vez de valores, a possibilidade de aproveitar a funcionalidade oferecida por bases de dados que implementem já o nível de isolamento snapshot ou a utilidade de protocolos de comunicação em grupo para a obtenção da ordem total dos *commits*.

Por fim abordámos outros dois temas, com respectivos sistemas, relaccionados com o isolamento snapshot, nomeadamente um sistema de federeção e sistemas transaccionais sobre estruturas de dados orientadas aos objectos.

REFERÊNCIAS

[1] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. *Proceedings of ACM SIGMOD Conference*, págs. 1-10, 1995.

[2] A. Fekete, E. O'Neil, P. O'Neil. A Read-Only Transaction Anomaly Under Snapshot Isolation. 1997.

[3] S. Elnikety, F. Pedone, W. Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. SRDS, Miami, FL, Outubro 2005.

[4] B. Kemme, G. Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. *Proceedings of ICDCS'98*, Amesterdão, Maio 1998.

[5] C. Plattner, G. Alonso. Ganymed: Scalable replication for transactional web applications. Middleware, 2004.

[6] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, G. Alonso. Improving the scalability of fault-tolerant database clusters. ICDCS, 2002.

[7] Y. Lin, B. Kemme, M. Patiño-Martínez, R- Jiménez-Peris. Middleware based data replication providing snapshot isolation. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 14 a 16 de Junho, 2005, Baltimore, Maryland

[8] S. H. Phatak and B. R. Badrinath. Multiversion Reconciliation for Mobile Databases. *Proceedings of the 15th International Conference on Data Engineering*, 23 a 26 de Março 1999, Sydney, Austrialia, págs. 582–589. IEEE Computer Society, Março 1999.

[9] K. L. Tripp. SQL Server 2005 Beta 2 Snapshot Isolation. MSDN, Fevereiro 2005.

[10] R. Schenkel, G. Weikum, N. Weissenberg, and X. Wu. Federated transaction management with snapshot isolation. FMLDO, 1999.

[11] M. Hollins. Transaction Isolation Levels and Object Oriented Data Structures. [http://citeseer.ist.psu.edu/693130.html]

[12] D. Agrawal and V. Krishnamurthy. Using multiversion data for non-interfering execution of writeonly transactions. *Proceeding of the ACM SIGMOD conference*, págs. 98-107, 1991.

[13] D. Agrawal, S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. *Proceedings of ACM SIGMOD Conference*, págs. 408-417, 1989.

[14] W. Cellary, G. Jomier. Consistency of versions in object-oriented databases. *Proceedings of 16th International Conference on Very Large Data Bases, Morgan Kaufmann*, págs. 432-441, 1990.

[15] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dezembro 1995.