DE89-21

大量データアクセストランザクションの並行制御方式

Concurrency Control of Bulk Access Transactions on Shared Nothing Parallel Database Machines

大森 匡 OHMORI, Tadashi
喜連川 優 KITSUREGAWA, Masaru
田中 英彦 TANAKA, Hidehiko
東京大学工学部電気工学科
Department of Electrical Engineering, The University of Tokyo

あらまし: 本論文では、大量データをアクセスする処理単位 (BAT と呼ぶ) の並行制御方式を提案する。従来の短時間処理とは異なり、BAT 処理ではデータ競合・リソース競合が非常に高い。そこでこれらの競合を低く抑えたスケジュールを行なう必要がある。そのため、本論文では「重みつき処理単位順序グラフ」(WTPG と呼ぶ) と、これを用いた二種類の先読みスケジューラを提案する。WTPG 上で最適化を用いることでデータ競合・リソース競合の小さい直列可能スケジュールを生成することができる。シミュレーションでは、これらのスケジューラは一括ロック規約・二相ロック規約よりも安定して高い性能を維持することが判った。

Abstract This paper proposes new concurrency control schemes for *Bulk Access Transactions* (BAT). The BAT is a transaction to access large bulk of data. Thus the scheduling of BATs is quite different from that of short term transactions. When scheduling BATs, the contentions of both data and resources are extremely high. Our approach is to schedule BATs so that these contentions are reduced.

We propose *Weighted Transaction Precedence Graph* (WTPG) and two cautious schedulers using the WTPG. The WTPG represents the costs required to run transactions under the serializability. Using the WTPG, the two schedulers reduce these contentions by global optimization and by local optimization respectively. Simulation results show that the schedulers achieve stable and much higher performance than the atomic lock and the two phase lock.

1 Introduction

In database services, a batch-job is usually a transaction to access large bulk of data. For instance, such a job reads files of 100 mega-byte size, computes their join-operation, and updates their major parts. We name this type of transaction a *Bulk Access Transaction* (BAT). This paper proposes new concurrency control schemes for BATs on a 'shared nothing' database machine.

The BAT is categorized as a Long Lived Transaction (LLT) in [Gra81], because it has a long lifetime by accessing bulk of data. Most of previous studies have discussed the concurrency control of short term transactions such as banking systems [Bhi88]. The LLT processing for CAD/CASE has been also discussed recently. But the concurrency control of BAT has not been well discussed so far.

The feature of the BAT processing is that the contentions of both data and resources are extremely high. These high contentions largely degrade the performance in concurrency control, as described in [Agr85, Tay85]. The data contention is defined as 'the contention over access to a data-granule' in [Tay85]. The resource contention refers to the congestion of resources such as disks.

The high data contention occurs in the BAT processing, because coarse granules of locking are used. For instance, a whole file is locked before starting to access all its data. Thus a BAT is often blocked by another, which is also blocked. This 'chain of blocking' degrades the performance because it reduces the number of practically active transactions. The resource contention is also very high in the BAT processing. It is because a BAT issues a bulk data processing such as scanning a whole file. Moreover such a bulk operation is too expensive to abort.

This situation is quite different from that in the processing of short term transactions, which has the very low data/resource contentions. Thus we must develop new schedulers for BATs.

We discuss the BAT processing on a 'shared nothing' database machine [Bhi88]. The machine is composed of a group of nodes interconnected by a network. Then the high data/ resource contentions cause the unbalance of load among the nodes.

Our approach is to estimate the degree of the data/ re-

source contentions in a schedule and to schedule BATs so that both contentions are reduced. For estimating this degree, we propose a *Weighted Transaction Precedence Graph* (WTPG). The WTPG represents the costs for running transactions under the serializability. We propose two *cautious schedulers* [Nis87] using a WTPG: Chain-WTPG scheduler and K-conflict WTPG scheduler. These schedulers estimate the degree of both contentions by a global strategy and by a local one respectively. In order to reduce the contentions, these schedulers grant a lock-request q only when they estimate that q causes the lowest contentions.

In both schedulers, a transaction must declare its sequence of read/write steps with their I/O demands at its start. Using this information, both schedulers have no abortion by deadlock, avoid chains of blocking, and keep a high number of practically active transactions. Simulation results show that these schedulers achieve stable and much higher performance than the atomic lock and the two phase lock.

The rest of the paper is organized as follows: Section 2 describes model and assumptions. In Section 3, the WTPG and the two cautious schedulers using it are proposed. Section 4 presents the global optimization algorithm in the Chain-WTPG scheduler. Section 5 summarizes the simulation results in [Ohm89]. Finally Section 6 concludes the paper.

2 Model and Assumptions

2.1 Target environment

Our target environment is a 'shared nothing' database machine whose file placement has high locality of reference. A shared nothing database machine is composed of 'data-nodes' interconnected by a network [Bhi88, DeW86, Cop88]. A 'data-node' is a computer with disks storing a part of database. We assume this type of machine because it supports the most huge scale of database processing.

As for a file placement on these data-nodes, we assume the range-partitioning, or the partial declustering, or clustering files of high affinity within a data-node. These placements achieve high 'locality of reference' within a data-node. It means that one data-node stores most of necessary data for a typical short term transaction.

The 'range partitioning' divides a relation horizontally by ranges on an attribute [DeW86]. A fragment named 'partition' is located per node. Then a range-query hits partitions on a few data-nodes and issues bulk data processing there. Thus the unbalance of load occurs among the nodes. In the 'partial declustering', a file is spread over a part of all the data-nodes [Cop88]. A full scan of a file also causes the load-unbalance here. It is also true when placing files of high affinity within a data-node.

This load-unbalance degrades the performance of BATs. Thus BATs should be scheduled so that the load is balanced among the data-nodes.

These file-placements are superior in both the short term transaction processing and the mix-workload: Because the high locality of reference reduces the traffic of messages among the data-nodes. The partial declustering also reduces the message-load in the point-to-point communication [Cop88]. Since these reductions alleviate the overhead of processors, they improve the throughput of short term transactions, as described in [Cop88, Bhi88].

For simplicity, the rest of the paper assumes each relation is range-partitioned on all the data-nodes.

2.2 Transaction model

This subsection describes the model of a BAT on both the locking and the cost.

We model a transaction T as a sequential execution of read/write steps, such that each step reads or writes only one partition. Here we assume that a step has a range-selection whose range is that of one partition on the partitioned attribute. A read/write step is abbreviated to a step in the rest of the paper.

A read (or write) step to a partition must hold a shared(S) (or exclusive(X)) lock on it before executing the step. A X-lock conflicts with either a S-lock or a X-lock.

T declares both all the data to read and those to write at its start. We call these declared data 'lock-declarations'. A shared (or exclusive) lock-declaration on a data-granule d represents 'a transaction will read (or write) d in the future'. A lock-declaration on a data-granule is replaced by the real lock-request on it when T requests to hold the lock. All the locks are held until the commitment of T for the recovery. At the commitment of T, these locks are released.

We use a partition as a locking-granule. A lock on a partition d is a predicate-lock to the range of d on the partitioned attribute. In updating a partition, we do not use a record-level X-lock. Because a BAT is supposed to update a major part of the partition.

A size of a partition is given by the number of *objects* in it. An *object* is the unit of data in bulk data processing, e.g. a cylinder of a disk for a sequential scan.

Then the cost model of a BAT is defined as follows:

A step s is given the cost $costof(s)$ = the number of objects s accesses. It is the estimation of the I/O demand of the step. When s reads $a\%$ of data in a partition P, $costof(s)$ is $a|P|$. ($|P|$ is the size of P.) In updating $a\%$ of P, $costof(s)$ is $2a|P|$, because it must read data at first before writing them. In the rest of the paper, $r_i(P:C)$ (or $w_i(P:C)$) refers to a read (or write) step of cost C to a partition P by a transaction T_i.

Note that a step has a lock on the whole partition, while it can access a part of it by using its indices.

Updated data are written back to disks immediately, because it is expensive to keep all the bulk-updated data in memory until commitment. Hence we ignore the I/O

```
T1:    r1(A:1) → r1(B:3) → w1(A:1).
T2:    r2(C:1) → w2(A:1).
T3:    w3(C:1) → r3(D:3).
```

Figure 1: transaction model

demands from the commitment of a transaction to its completion.

We use the centralized concurrency control for BATs because of the partition locking-granules: The centralized control node (CN) manages a lock-table of the partition-granules. Then CN grants or blocks a lock-request to a partition. Once a lock is granted to a step of a transaction T, CN send T to a data-node. In the data-node, steps are executed in the round-robin service: a step switches to the next waiting step after processing 1 object.

Example2.1: Figure 1 illustrates three transactions $T1, T2, T3$. '*step1* → *step2*' means a sequential execution of the steps. In the figure, 100% of a partition is read and a half of it is updated. The commitment at the last of a read/write sequence is not displayed in the rest of the paper. □

3 Schedulers for BAT

3.1 Weighted transaction graph

In the BAT processing, schedulers should estimate the degree of the data/ resource contentions and should generate a schedule so that these contentions are reduced. For estimating this degree, we attach 'weights' to edges of a transaction precedence graph. The weights represent the costs for running transactions.

Definition 1 Weighted Transaction Precedence Graph (WTPG) is a graph $< N, C, E, w >$ such that:

1. N: the set of nodes representing transactions. ($T0$: the initial transaction. Tf: the final transaction. Ti and Tj are general ones.)

2. C: the set of *confliction-edges* (Ti, Tj). It is a pair of edges $Ti \rightarrow Tj$ and $Tj \rightarrow Ti$. (Ti, Tj) means that Ti conflicts with Tj in the serializable order. This edge is generated only when both Ti and Tj have issued the conflicting lock-declarations on a locking-granule. When it has been determined that Ti precedes Tj, (Ti, Tj) is replaced by a *precedence-edge* $Ti \rightarrow Tj$. We call this operation 'resolving (Ti, Tj) into $Ti \rightarrow Tj$'.

3. E: the set of *precedence-edges* $Ti \rightarrow Tj$. $Ti \rightarrow Tj$ means "Ti conflicts with Tj in the serializable order, and Ti precedes Tj in the serializable order". $Ti \rightarrow Tj$ is generated only by resolving (Ti, Tj). (for all Ti, there exist the edges $T0 \rightarrow Ti$ and $Ti \rightarrow Tf$).

4. w gives a weight to $Ti \rightarrow Tj$ in $E \cup C$ as follows:

- $w(T0 \rightarrow Ti)$ = the number of objects Ti must access during the period from the present time to the commitment of Ti.
- $w(Ti \rightarrow Tf)$ = the number of object Ti must access during the period from the commitment of Ti to its completion.
- $w(Ti \rightarrow Tj)$ = the number of objects Tj must access during the period from the commitment of Ti to the commitment of Tj.

□

In this definition, only the weights $w(T0 \rightarrow Ti)$ depend on the present state of the schedule. Since the transaction-model in Section2.2 accesses objects sequentially, a weight on an edge in a WTPG expresses the shortest period (i.e. length of time) between two events. In this case, a unit of period is equal to the time required to process one object on a data-node.

Figure 2-(a) shows a WTPG. In the rest of the paper, a WTPG is depicted as follows: A precedence-edge is depicted by a solid arrow. A confliction-edge is represented by a pair of shaded arrows between nodes. A weight of an edge is depicted beside the edge. Moreover, $Ti \rightarrow Tj$ is also notated by $Tj \leftarrow Ti$. e.g. in Figure 2-(a), the confliction-edge $(T2, T3)$ is a pair of edges $T2 \rightarrow T3$ of the weight 4 and $T2 \leftarrow T3$ of the weight 2.

Example3.1: Figure 2-(a) shows the WTPG where all the transactions in Figure 1 have just started up. In Figure 2-(a), $w(T1 \rightarrow T2) = 1$: because, if $T1$ precedes $T2$, $T2$ can start `w2(A:1)` only after $T1$ releases a X-lock on `A`. Then, even if $T2$ is not blocked and it runs exclusively, $T2$ must spend the time required to process 1 object before its commitment.

By the cost model in Section2, $w(Ti \rightarrow Tf) = 0$ for all Ti. Tf and its edges are not depicted in the rest of the paper. □

From the transaction model in Section2.2, the weights in a WTPG are computed as follows:

At the start of a transaction Ti, it declares its sequence of steps $\{s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_N\}$ and the estimation of their I/O demands $costof(s_i)$. Let due(s_i) be the number of objects Ti must access before Ti commits after the start of s_i. It is defined by the formulae:

due$(s_N) = costof(s_N)$.

due$(s_i) = costof(s_i) +$ due(s_{i+1}). $\quad (i < N)$

Then, at the start of Ti, $w(T0 \rightarrow Ti)$ is set as follows:

$w(T0 \rightarrow Ti) =$ due(s_0).

When a lock-declaration of a step s_i of Ti conflicts with that of a step s_j of Tj, the weights on (Ti, Tj) are set as follows:

$w(Tj \rightarrow Ti) =$ due(s_i).

$w(Ti \rightarrow Tj) =$ due(s_j).

$w(Ti \rightarrow Tj)$ and $w(Tj \rightarrow Ti)$ take the largest values among the values computed for them respectively.

For all s_j, due(s_j) is kept with the lock-declaration of s_j in the lock table. Thus the weights on the edges of Ti are computed when Ti declares its information at its start.

As a schedule proceeds, $w(T0 \rightarrow Ti)$ is adjusted as follows: If a step of Ti ends accessing 1 object on a

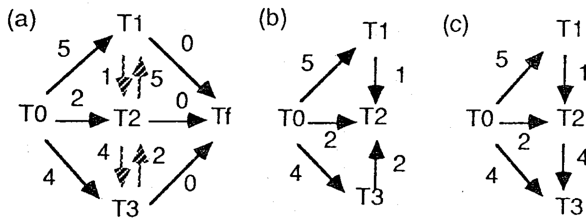

Figure 2: examples of WTPG

data-node, Ti issues a message to the control node and decrements $w(T0 \rightarrow Ti)$. The weights on (Ti, Tj) are not updated once their values have been determined.

In the rest of the paper, a '*full* SR-order' refers to a serializable order which consists of all the nodes in a WTPG. A 'WTPG resolved by a full SR-order S' is the WTPG where all its confliction-edges has been resolved into the precedence-edges in S. A 'critical path' refers to the longest path from $T0$ to Tf.

3.2 Chain WTPG scheduler

Using a WTPG, we estimate the degree of the data/resource contentions by a global strategy as follows:

Suppose that a full SR-order S has resolved all the confliction-edges in a WTPG. Then the length of its critical path from $T0$ to Tf represents the earliest possible completion time of a total schedule under S. The shorter the path is, the less contentions of both data and resource occur. Because chains of blocking make the critical path long, while high congestion on a data-node keeps $w(T0 \rightarrow Ti)$ large.

Example 3.2: In Figure 2-(a), a full SR-order $W = \{T1 \rightarrow T2, T3 \rightarrow T2\}$ resolve all the confliction-edges in the WTPG so that it has the shortest critical path. Figure 2-(b) is the WTPG resolved by W. Its critical path is $T0 \rightarrow T1 \rightarrow T2$ of length 6.

In another full SR-order $\{T1 \rightarrow T2 \rightarrow T3\}$, its critical path has length of 10 as in Figure 2-(c). Clearly a chain of blocking $\{T1 \rightarrow T2 \rightarrow T3\}$ degrades the performance in this case. □

Then our strategy is:

> The serializable order of the generated schedule is the full SR-order W, where the WTPG resolved by W has the shortest critical path.

This strategy uses a global optimization: In order to reduce the data/ resource contentions in a schedule, this strategy estimates the degree of the contentions including what may occur in the future.

It is NP-hard to compute the above W in any WTPG (see appendix). We use a restricted topology of a WTPG and compute W by $O(N^2)$ in it. (N is the number of nodes in the WTPG).

Definition 2 A *chain-form WTPG* is a WTPG such that all its nodes except both $T0$ and Tf are labeled $1, \ldots, N$ as follows: for all k in $2, 3, \ldots, N-1$, $n[k]$ conflicts only with either $n[k-1]$ or $n[k+1]$ in the serializable order. ($n[k]$ is the node of label k.) □

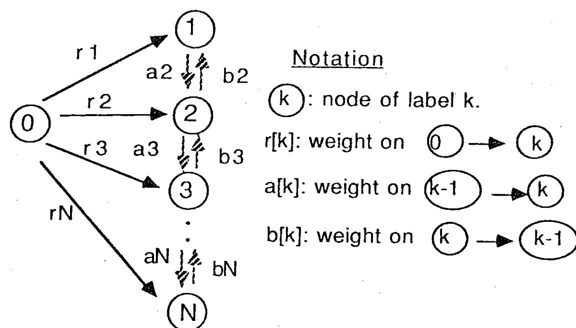

Figure 3: A chain-form WTPG $G(1, N)$

Figure 3 shows a chain-form WTPG. ($T0$ is labeled 0. Tf and its edges are not depicted.) The WTPG in Figure 2-(a) is also a chain-form WTPG.

Using this strategy, the scheduler CC1 behaves as follows when a lock-request q is issued on a locking-granule d:

CC1(q: lock-request on d: data)

Step0: If q is the start step of a transaction T, CC1 adds T into the WTPG and checks its topology. If it is not a chain-form, T is aborted.

Step1: If q conflicts with the lock held on d, q is blocked.

Step2: Compute a full SR-order W, such that the WTPG resolved by W has the shortest critical path.

Step3: If a serializable order which is inconsistent with W is generated by granting q, q is delayed. Otherwise q is granted.

□

Either the delayed lock-requests or the aborted ones are resubmitted to CC1 after a specified delay. The *Step0* of CC1 keeps the WTPG in a chain-form by the depth first traverse. By the *Step3*, the generated schedule is kept consistent with W. We name CC1 'Chain-WTPG scheduler' or 'Chain'.

Example 3.3: In Figure 2-(a), $W = \{T1 \rightarrow T2, T3 \rightarrow T2\}$ makes the shortest critical path in this WTPG. Suppose that the step `r2(C:1)` of $T2$ in Figure 1 is requested. Then $(T2, T3)$ is resolved into $\{T2 \rightarrow T3\}$ if the step is granted. It is inconsistent with W. Thus CC1 delays `r2(C:1)`. □

3.3 K-conflict WTPG scheduler

When a small part of database is often accessed by conflicting locks, the 'chain-form' constraint in Chain limits the number of active transactions. Consequently the performance is degraded in this case.

We call this case 'the data contention on a hot-set'. Then a scheduler should accept any topology of a WTPG against this contention. Thus we must estimate the degree of the data/ resource contentions without a global optimization.

The following function $\mathcal{E}(q)$ estimates the degree of the contentions in the case a lock-request q is granted.

$\mathcal{E}(q$: lock-request of a transaction $T)$
Step1: Make the WTPG where q has been granted. Identify $before(T)$ and $after(T)$ in this WTPG. They are a set of transactions which precede T in the serializable order and a set of ones which T precedes respectively. If q causes a deadlock, return $\mathcal{E}(q) = \infty$.

Step2: For all the confliction-edges (Ti, Tj) such that $Ti \in before(T)$ and $Tj \in after(T)$, resolve it into $Ti \rightarrow Tj$.

Step3: Delete all the confliction-edges in the above WTPG. Then $\mathcal{E}(q)$ is the length of the critical path from $T0$ to Tf in it.
□

Example 3.4: Figure 4 illustrates the above procedure. For simplicity, $w(T0 \rightarrow Ti)$ is set to 0 for all Ti. $T0$ and its edges are not displayed in the figure. In the WTPG in Figure 4-(a), suppose that $T5$ now issues a lock-request q which conflicts with $T6$. Then, if q is granted, $T5 \rightarrow T6$ is generated. At this time, we set $before(T5) = \{T4\}$ and $after(T5) = \{T6\}$. Thus $(T4, T6)$ is resolved into $T4 \rightarrow T6$, as shown in Figure 4-(b). Its critical path in Figure 4-(b) is $T4 \rightarrow T6$ of length 10. So $\mathcal{E}(q) = 10$. □

$\mathcal{E}(q)$ is computed by $O(max(n, e))$. (n: number of nodes, e: number of edges in a WTPG), because *Step2* in $\mathcal{E}(q)$ is computed by the depth-first traverse and because its *Step3* is by the topological sort.

Let $C(q)$ be the set of lock-declarations which conflict with the lock-request q. $C(q)$ is found in the lock-table. Then our strategy is:

> q is granted only when q has the smallest value of $\mathcal{E}(q)$ in $C(q)$.

This strategy reduces the data/ resource contentions in a schedule by a local optimization: It estimates only the degree of the contentions which occur at the present state of the schedule.

Using this strategy, CC2 behaves as follows when q is issued to a locking granule d:

CC2(q: lock-request on d: data)
Step1: If q conflicts with the lock held on d, q is blocked.

Step2: Compute $\mathcal{E}(q)$. If q causes a deadlock, q is delayed.

Step3: If q has the smallest value of $\mathcal{E}(q)$ in $C(q)$, q is granted. Otherwise q is delayed.
□

Example 3.5: Suppose the same case in Example 3.4. Let q' be a lock-request of $T6$, such that q' conflicts with q of $T5$. Figure 4-(c) is the WTPG where q' has been granted. $(T4, T6)$ has been deleted by *Step3* of $\mathcal{E}(q')$ in the figure. Since $\mathcal{E}(q) = 10 > \mathcal{E}(q') = 1$, CC2 delayes q when q is submitted. □

For alleviating the complexity in the *Step3* of CC2, we limit the size of $C(q)$ to $K = 0, 1, 2, \ldots$ by the constraint:

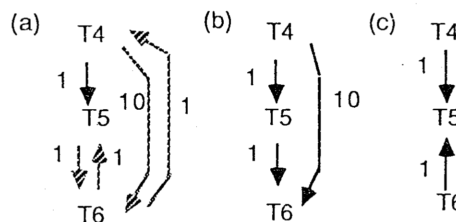

Figure 4: examples of K-conflict WTPG

> Each lock-declaration may conflict with K lock-declarations at most.

We name this constraint 'K-conflict'. At the start of a new transaction T, CC2 tests this constraint. If it fails, T is aborted.

With K fixed, *Step3* of CC2 is computed by $O(K \times max(n, e))$. We name CC2 with this constraint 'K-conflict WTPG scheduler' or 'K-WTPG'. Even with $K = 1$, K-WTPG accepts a WTPG which is not a chain-form.

3.4 Reducing control overhead

For reducing the control overhead, both Chain and K-WTPG use their previous results of computation if neither
1) a specified period has elapsed after the last computation, nor
2) a transaction commits/ aborts/ starts after the last computation.

If so, Chain uses the full SR-order W which was computed at the last time in the *Step2* of CC1. K-WTPG estimates $\mathcal{E}(c)$ at its most recently computed value if it exists. In K-WTPG, $\mathcal{E}(c)$ is kept with the lock-declaration of c in the lock-table. $\mathcal{E}(c)$ is replaced by a new value when it is recomputed.

4 Algorithm for Chain-WTPG

This section describes the algorithm to compute the full SR-order W, such that W makes the shortest critical path in any chain-form WTPG.

Figure 3 shows a chain-form WTPG $G(1, N)$, as defined in Section 3.2. Although $n[k]$ (the node of label k) need not always conflict with its adjacent nodes in the definition2, we only deal with $G(1, N)$ here. If $(n[k], n[k+1])$ has been resolved, we set the weights either `b[k+1]` or `a[k+1]` to ∞ in Figure 3 accordingly.

The following notations are used:

1. with k fixed, $n[k]$ is abbreviated to nk.

2. $G(i, j)$: the subgraph of $G(1, N)$ which is composed of $n0$ and $\{n[i], n[i+1], .., n[j]\}$.

3. '$(n[i], n[i+1])$ is set upwards' means that the edge is resolved into $n[i] \leftarrow n[i+1]$. '$(n[i], n[i+1])$ is set downwards' means that it is resolved into $n[i] \rightarrow n[i+1]$.

4. $min(A, B)$ (or $max(A, B)$) is the minimum (or the maximum) between A and B.

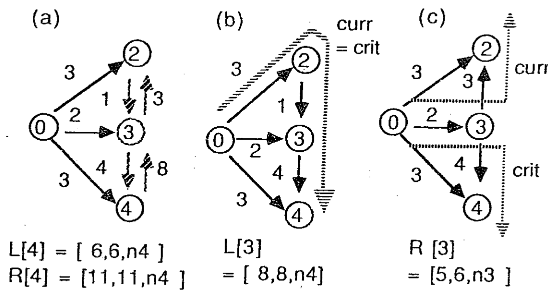

Figure 5: examples of $L[k]$ and $R[k]$

The following structural parameters $L[k]$ and $R[k]$ represent the information about the critical paths in $G(k-1, N)$. By computing these parameters, we can find the shortest critical path in $G(1, N)$. Each member of these parameters is referred to by *parameter.member*.

Definition 3 $L[k]$ and $R[k]$ are the triplets $[curr, crit, rev]$ defined below:

Suppose that $(n[k-1], n[k])$ has been set downwards in $G(k-1, N)$. Let $S1(k-1, N)$ be a full SR-order which makes the shortest critical path in this case. Then $L[k]$ is defined on the edge $n[k-1] \rightarrow n[k]$ as follows:

- $L[k].crit$: the length of the shortest critical path in $G(k-1, N)$ which has been resolved by $S1(k-1, N)$.
- $L[k].rev$: the smallest label among the labels i, such that $(n[i], n[i+1])$ is set upwards in $S1(k-1, N)$. $L[k].rev = N$ if such the label does not exist.
- $L[k].curr$: the length of the path $n0 \rightarrow n[k-1] \rightarrow n[k] \rightarrow .. \rightarrow n[L[k].rev]$.

Suppose that $(n[k-1], n[k])$ has been set upwards in $G(k-1, N)$. Let $S2(k-1, N)$ be the full SR-order to make the shortest critical path in this case. Then $R[k]$ is defined on the edge $n[k-1] \leftarrow n[k]$ as follows:

- $R[k].crit$: the length of the shortest critical path in $G(k-1, N)$ which has been resolved by $S2(k-1, N)$.
- $R[k].rev$: the smallest label among the labels i such that $(n[i], n[i+1])$ is set downwards in $S2(k-1, N)$. $R[k].rev = N$ if such the label does not exist.
- $R[k].curr$: the length of the critical path from $n0$ to $n[k-1]$ in $G(k-1, R[k].rev)$ which has been resolved by $S2(k-1, N)$.

□

Example 4.1: Figure 5-(a) illustrates $G(2, 4)$, $L[4]$, and $R[4]$. Figure 5-(b) shows the case computing $L[3]$:

Suppose that $(n2, n3)$ has been set downwards. Then, between $n3 \rightarrow n4$ and $n4 \rightarrow n3$, the former makes the shorter critical path of length 8. Hence $S1(2, 4) = \{n2 \rightarrow n3 \rightarrow n4\}$. In $S1(2, 4)$, there is no $n[i]$ such that $(n[i], n[i+1])$ is set upwards. Moreover, the critical path in Figure 5-(b) is $n0 \rightarrow n2 \rightarrow n3 \rightarrow n4$. Thus we set $L[3] = [8, 8, n4]$.

Figure 5-(c) displays $R[3]$: Clearly $S2(2, 4)$ is $\{n3 \rightarrow n2, n3 \rightarrow n4\}$, not $\{n4 \rightarrow n3 \rightarrow n2\}$. Thus $R[3] = [5, 6, n3]$. □

By the definitions of $L[k]$ and $R[k]$, theorem 1 holds:

Theorem 1 $L[i]$ and $R[i]$ for all $i = k+1$ to N are given in $G(k, N)$. Then $S1(k, N)$, $S2(k, N)$ in definition 3 and the full SR-order $S(k, N)$ which makes the shortest critical path P in $G(k, N)$ are computed by the formulae:

$$S1(k,N) = \{n[k] \rightarrow n[k+1] \rightarrow \ldots \rightarrow n[L[k+1].rev]\} \cup S2(L[k+1].rev, N).$$

$$S2(k,N) = \{n[k] \leftarrow n[k+1] \leftarrow \ldots \leftarrow n[R[k+1].rev]\} \cup S1(R[k+1].rev, N).$$

such that $S1(i,i) = S2(i,i) = \phi$ for all i.

The length of $P = min(L[k+1].crit, R[k+1].crit)$.

if $L[k+1].crit \leq R[k+1].crit$ then
  $S(k,N) = S1(k,N)$.
else
  $S(k,N) = S2(k,N)$.

□

Example 4.2: In Example 4.1, $L[3].crit = 8 > R[3].crit = 6$. Hence $S(2,4) = S2(2,4) = \{n2 \leftarrow n3\} \cup S1(3,4) = \{n2 \leftarrow n3 \rightarrow n4\}$. □

Theorem 2 Suppose that $G(k-1, N)$ and all the parameters $L[i]$ and $R[i]$ from $i = k+1$ to N are given. Then $L[k]$ and $R[k]$ are computed by $O(N-k)$. □

Outline of the proof:

We show the algorithm `Lcomp()` for computing $L[k]$, in the C language-like notation. The variables `a[k]`, `b[k]` and `r[k]` are the weights, as defined in Figure 3.

```
Lcomp() {
        /* procedure for L1[k] */
temp = L[k+1].curr - r[k] + r[k-1] + a[k];
if ( temp <= L[k+1].crit )
  L1[k] = [ temp, L[k+1].crit, L[k+1].rev ];
else {
  L1[k].crit
  = min( max(V(h), R[h+1].crit)); /* EXPR1 */
    for all h = k+1 to L[k+1].rev

  L1[k].rev  = h0;
  L1[k].curr = C(h0);

/*    h= h0 takes the minimum in EXPR1.
 *    C(h) and V(h) are defined as follows:
 *  V(k-1) = C(k-1)   = r[k-1];
 *  V(h) = max(r[h], V(h-1)+a[h]);(k =< h)
 *  C(h) = C(h-1) + a[h]; ( k =< h )
 */
}  /* end of L1[k] */
   /* procedure for L2[k]:*/
```

```
L2[k].curr = r[k-1] + a[k];
L2[k].crit = max(L2[k].curr, R[k+1].crit);
L2[k].rev  = k;    /* end of L2[k] */
if  ( L1[k].crit <= L2[k].crit )
    L[k] = L1[k];
else
    L[k] = L2[k];
}
```

In Lcomp(), $L[k]$ is either $L1[k]$ or $L2[k]$ according to their values of *crit*. $L1[k]$ represents $L[k]$ in the case $(n[k], n[k+1])$ is set downwards. $L2[k]$ does when it is set upwards. $L2[k]$ is computed only in the case $G(k-1, N)$ is resolved by $\{n[k-1] \rightarrow n[k]\} \cup S2(k, N)$.

$L1[k]$ is computed at first in the case $G(k-1, N)$ is resolved by $\{n[k-1] \rightarrow n[k]\} \cup S1(k, N)$.

In this case, a new path $P0 = \{\ n0 \rightarrow n[k-1] \rightarrow n[k] \rightarrow n[k+1] \rightarrow ... \rightarrow n(L[k+1].rev)\ \}$ is generated. The variable temp in Lcomp() is the length of $P0$.

If $P0$ is longer than the shortest critical path of $G(k, N)$, $P0$ is the critical path when using $\{n[k-1] \rightarrow n[k]\} \cup S1(k, N)$. But $P0$ may get shortened by setting upwards $(n[h], n[h+1])$ in $P0$. i.e. using $S(h) = \{n[k-1] \rightarrow n[k] \rightarrow \ldots \rightarrow n[h]\} \cup S2(h, N)$ where $k+1 \leq h \leq L(k+1).rev$. The expression EXPR1 in Lcomp() computes $L1[k]$ in this case.

In EXPR1, $C(h)$ is the length of the path $P(h) = n0 \rightarrow n[k-1] \rightarrow n[k] \rightarrow \ldots \rightarrow n[h]$. $V(h)$ is the length of the critical path in $G(k-1, h)$ which has been resolved by $P(h)$.

In $S(h)$, $G(k-1, h)$ is not connected with $G(h, N)$. By its definition, $S2(h, N)$ makes the shortest critical path in $G(h, N)$ if $(n[h], n[h+1])$ is set upwards. Thus $S(h)$ makes the critical path of length $max(V(h), R[h+1].crit)$ in $G(k-1, N)$. So EXPR1 correctly finds $h = h0$ such that $S(h0)$ makes the shortest critical path in $G(k-1, N)$.

$V(h)$ and $C(h)$ are computed by $O(1)$ in each loop of EXPR1. Thus $L1[k]$ is computed by $O(N-k)$.

The other cases for $L1[k]$ are trivial and they are omitted here. The algorithm for $R[k]$ is described in the appendix.

□

Computing $L[k]$ and $R[k]$ from $k = N$ to 2, both $L[2]$ and $R[2]$ are computed. Then Corollary 1 holds by theorem 1 and theorem 2.

Corollary 1 In a chain-form WTPG $G(1, N)$ in Figure 3, the full SR-order which makes the shortest critical path in it is computed by $O(N^2)$. □

Example 4.3: Figure 6-a shows the WTPG where a node $n1$ and its edges $n0 \rightarrow n1$ of weight 5 and $n1 \rightarrow n2$ of weight 10 are appended to the WTPG in Figure 5-b. Figure 6-a shows the behavior of Lcomp() to compute $L1[2]$. Appending $n1 \rightarrow n2$ to $G(2, 4)$ under $S1(2, 4) = \{n2 \rightarrow n3 \rightarrow n4\}$, then the new path P is $n0 \rightarrow n1 \rightarrow n2 \rightarrow n3 \rightarrow n4$ of length 20. P is longer than $L[3].crit = 8$. Thus P is now the critical path in $G(1, 4)$. But P gets shortened by setting upwards $(n3, n4)$, as shown

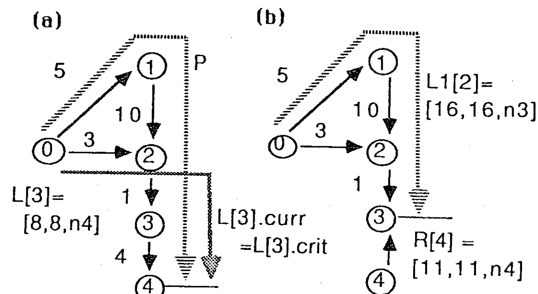

Figure 6: example of the algorithm for $L1[k]$

in Figure 6-b. In Figure 6-b, the critical path has the length of $max(|n0 \rightarrow n1 \rightarrow n2 \rightarrow n3|, R[4].crit) = 16$. EXPR1 in Lcomp() computes $L1[2]$ in this case and sets $L1[2] = [16, 16, n3]$. □

5 Evaluation

This section summarizes the simulation results in [Ohm89] because of the space-limitation.

The protocols we test are: Cautious Two Phase Lock (C2PL), Atomic Static Lock (ASL) in [Tay85], Chain, and K-WTPG of $K = 2$ (K2). C2PL is a variant of the two phase lock in the cautious schedulers [Nis87].

The simulation-results have shown the following:

1: In the high data contention by chains of blocking both Chain and K2 avoid chains of blocking as perfectly as ASL. These three protocols achieve 100% higher performance than C2PL in the experiments.

2: In the high data contention on a hot-set, K-WTPG achieves 50% higher performance than both C2PL and Chain, which in turn outperform ASL by 100%. These differences are caused by the topological constraints on a WTPG. K2 is better than C2PL because of the chains of blocking in C2PL.

3: Even with the low resource contention, the extremely high data contention limits the inter-transaction parallelism of BATs. Thus a file-placement and the intra-transaction parallelism should be highly utilized.

4: K-WTPG is more sensitive to the erroneous I/O demands than Chain. Even when the error ratio is the normal distribution of the standard deviation 0.5 and the average 0, both keep 70% higher throughput than C2PL in the blocking contention.

6 Conclusion

This paper proposed new concurrency control schemes for the Bulk Access Transactions (BAT). The BAT is a transaction which accesses large bulk of data. Our target environment is a shared nothing database machine whose file placement has the high locality of reference.

We proposed Weighted Transaction Precedence Graph (WTPG) and two cautious schedulers: Chain-WTPG scheduler (Chain) and K-conflict WTPG scheduler (K-WTPG). Both schedule BATs so that the data/

resource contentions are reduced. These schedulers reduce the contentions by a global optimization and by a local optimization respectively.

Chain predicts the globally optimized serializable order W and grants a lock only if it is consistent with W. This prediction is NP-hard in general. We restrict a topology of a WTPG into a 'chain-form' and compute W in polynomial time.

K-WTPG estimates $\mathcal{E}(q)$: the degree of the contentions a lock-request q causes at the present. K-WTPG grants the lock-request q only if $\mathcal{E}(q)$ is the smallest.

Chain uses a more strict scheduling strategy than K-WTPG. But K-WTPG permits any topology of a WTPG, while Chain does not. In both schedulers, a transaction must declare its sequence of read/write steps and their I/O demands at its start.

In the simulation results, both schedulers achieve stable and much higher performance than ASL and C2PL. It is also true when transactions declare erroneous I/O demands.

In the mix-workload of different job-classes, a database-computer should support different concurrency control protocols according to the classes of jobs. We believe Chain and K-WTPG are good candidates for the class of Bulk Access Transactions when a file placement has the high locality of reference.

Appendix

Theorem 3 It is NP-hard to compute the full SR-order which makes the shortest critical path in any WTPG.

Outline of the proof: The static general job-shop scheduling problem *JS* is represented by a *disjunctive graph* [Gra79]. *JS* is the problem to resolve all the *disjunctive edges* in the graph so that the critical path is the shortest. The disjunctive graph is just the WTPG where all the outgoing edges of a node have the cost of the node. Since *JS* is NP-hard, the theorem holds. □

Lemma 1 In theorem 2, $R[k]$ is computed by $O(N-k)$.

Outline of the proof: we show the algorithm Rcomp() for computing $R[k]$ from both $G(k,N)$ and its parameters $R[i]$ and $L[i]$ $(k+1 \leq i \leq N)$. The variables a[k], b[k] and r[k] are defined in figure 3.

```
Rcomp() {
        /* procedure for R1[k]   */
temp = R[k+1].curr + b[k];
if ( max( r[k-1], temp ) =< R[k+1].crit )
    R1[k] = [ temp, R[k+1].crit, R[k+1].rev ];
else if ( max(r[k-1], temp ) == r[k-1] )
    R1[k] = [ r[k-1], r[k-1], R[k+1].rev ];
else {
    R1[k].crit
     = min( max( V(h), L[h+1].crit) ); /* EXPR2 */
       for all h = k+1 to R[k+1].rev

    R1[k].rev  = h0;
    R1[k].curr = V(h0);
/*    h= h0 takes the minimum in EXPR2.
 *    V(h), C(h) are defined below.
 */
}       /* end of R1[k] */

        /* procedure R2[k]   */
    R2[k].curr = max( r[k]+b[k], r[k-1] );
    R2[k].crit = max( R2[k].curr, L[k+1].crit);
    R2[k].rev  = k;        /* end of R2[k]    */
if  ( R1[k].crit <= R2[k].crit )
    R[k] = R1[k];
else
    R[k] = R2[k];
}
```

In Rcomp(), $R1[k]$ represents $R[k]$ when setting upwards $(n[k], n[k+1])$. $R2[k]$ does when setting it downwards.

The expression EXPR2 assumes $S(h) = \{n[k-1] \leftarrow n[k] \leftarrow .. \leftarrow n[h]\} \cup S1(h,N)$. It finds $h = h0$, where $S(h0)$ makes the shortest critical path in $G(k-1,N)$.

The variable V(h) is the length of the critical path in $G(k-1,h)$ under $S(h)$. C(h) is the length of the path $n0 \rightarrow n[h] \rightarrow n[h-1] \rightarrow \ldots \rightarrow n[k-1]$. They are computed by the formulae:

V(k-1) = C(k-1) = r[k-1];
C(h) = C(h-1) - r[h-1] + r[h] + b[h]; $(k \leq h)$
V(h) = max(C(h), V(h-1)); $(k \leq h)$

All the proofs are similar to those in theorem 2 and they are omitted here.
□

References

[Agr85] Agrawal,R. et al. Models for Studying Concurrency Control Performance: Alternatives and Implications. In *Proc. ACM-SIGMOD '85*, pages 108–121, 1985.

[Bhi88] Bhide,A. An Analysis of Three Transaction Processing Architectures. In *Proc. 14th Int'l Conf. Very Large Data Bases*, pages 339–350, 1988.

[Cop88] Copeland,G. et al. Data Placement in Bubba. In *Proc. ACM-SIGMOD'88*, pages 99–108, 1988.

[DeW86] DeWitt,D.J. et al. Gamma - High Performance Dataflow Database Machine. In *Proc. 12th Int'l Conf. Very Large Data Bases*, 1986.

[Gra79] Graham,R. et al. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of Discrete Mathematics 5*, North Holland, pages 287–326, 1979.

[Gra81] Gray,J. The Transaction Concept: Virtues and Limitations. In *Proc. 7th Int'l Conf. Very Large Data Bases*, pages 144–154, 1981.

[Nis87] Nishio,S. et al. Performance Evaluation on Several Cautious Schedulers for Database Concurrency Control. In *Proc. 5th Int'l Workshop Database Machines*, pages 212–225, 1987.

[Ohm89] Ohmori,T. et al. to appear in Proc. 39th Annual Convention, IPS Japan, 1989.

[Tay85] Tay,Y.C. Locking Performance in Centralized Databases. *ACM Trans. Database Syst.*, 10(4):415–462, 1985.