Scheduling Batch Transactions on Shared-Nothing Parallel Database Machines: Effects of Concurrency and Parallelism

Tadashi OHMORI Masaru KITSUREGAWA Hidehiko TANAKA

Dept. of Electrical Engineering, The University of Tokyo, JAPAN

Abstract This paper discusses concurrency-control scheduling of batch transactions on Shared-Nothing (or 'loosely-coupled') multiprocessor database machines. Usually, batch transactions access large bulks of data for data analysis or periodic database-update. This paper tests various schedulers for these batch transactions, and examines how well the schedulers perform when both inter-transaction parallelism and intra-transaction parallelism are limited. By using the best-performing schedulers for batch transaction processing, Shared-Nothing machines can freely tune their data-placements for short-term transaction processing. Simulation results show that the two new schedulers proposed in our previous study [13] are the best performers in various workloads.

1 Introduction

Today's On-Line Transaction Processing (OLTP) applications have heavy mixed-workload of short-term transactions and batch transactions. For these applications, Shared-Nothing (or 'loosely-coupled') multiprocessor database machines have been proposed as attractive platforms, such as Tandem VLX/CLX series [14], Gamma [7] and Teradata DBC1012 [6]. These machines are composed of loosely-coupled computer-nodes [3], and a database is partitioned on these nodes. Further, transactions are executed on the nodes storing the necessary data. Because of this architecture, Shared-Nothing machines need to tune their data-placements in order to provide load-balancing and high performance of short-term transactions [8] [6] [5].

On the other hand, today's OLTP applications must run high volume of batch transactions as well. Usually, batch transactions have file-scanning operations to read/write large bulks of data, and are used for statistic data analysis or periodic database-update. Shared-Nothing machines thus need to run these batches on the data-placements tuned for short-term transactions.

In this case, previous studies showed the following two design problems: The first problem is that, when tuning data-placements for short-term transaction processing, parallelism (i.e. intra-transaction parallelism) of a batch is limited as a side effect. For instance, range-partitioning strategy [7] and partial-declustering strategy [5] greatly improve performance of short-term transactions [5] [8] [6]. In these strategies, however, each file scanned by a batch is not distributed uniformly on all nodes [8] [5]. This fact limits parallelism within a file-scanning operation.

As the second problem, today's OLTP applications run a much greater number of batches which update large bulks of data. For example, periodic database-management often issues a batch transaction like this: "updating all records that match a given filtering-condition in a view relation". This batch has bulk-update operations as well as bulk-read operations. Concurrency (i.e. inter-transaction parallelism) between such batches is fairly limited because their lock-requests to files are frequently blocked. The traditional two-phase locking protocol does not work well in this case because of 'chains of blocking' [15], i.e., the disadvantage that transactions are blocked one after another.

Motivated by these problems, this paper discusses concurrency-control scheduling of batch transactions on Shared-Nothing database machines, where we concentrate on batch transactions having bulk-update operations. Our objective is to find the best schedulers for batch processing. Due to the above design problems, schedulers for batches need to perform well even when either concurrency or parallelism of batches is limited.

We test various schedulers, and examine their absolute performance and relative performance-gain either when increasing parallelism by data-placements or when increasing the degree of concurrency. By using the best-performing schedulers for batch transactions, Shared-Nothing machines can tune their data-placements for short-term transactions, and have no disadvantage to batch processing. Therefore such the dedicated schedulers for batch processing are new design alternatives.

Our previous study [13] already proposed two new schedulers for batch processing. This paper examines these new schedulers in addition to traditional locking protocols. This will show that our new schedulers perform best even when utilizing intra-transaction parallelism of batches.

A recent study [4] clarified performance effects of concurrency-control and parallelism on short-term transaction processing. However these effects on batch-transaction processing are still unclear. Some works [9] [5] proposed to tune data-placements for a given OLTP/batch workload, while our approach is orthogonal to theirs.

The next section describes models of batch processing. Section 3 introduces our new schedulers. After explaining a simulation model in Section 4, Section 5 discusses experimental results. Section 6 concludes this paper.

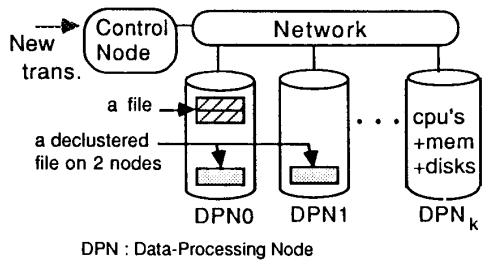


Figure 1: Shared-Nothing database machine model

2 Batch Processing Model

This section describes a model of a batch transaction. Fig.1 shows a model of a Shared-Nothing machine. This is composed of "data-processing nodes", which execute database processing in parallel. A detailed model is described in Section 4.1.

A batch is not a typical job like a debit-credit transaction. Thus batches utilize no indices in many cases. When executing any given selection-operation on a file, a batch locks the whole file (or the whole partition) as any given predicate-lock, and scans it. This observation gives the following model of a batch transaction:

1: A batch transaction is a sequential execution of steps, where each step reads or writes a file by file-scanning operations. A file is used as a locking-granule. A reading (or writing) step to a file must hold a shared(S) (or an exclusive(X)) lock on it. Also a batch holds all its acquired locks until its commitment. □

Here, a *file* represents either a partially-declustered relation in 'partial declustering' strategy [5] or subrange's partition in a 'range-partitioned' relation [7]. Fig.1 shows two files declustered on either one or two nodes. Intra-transaction parallelism of a batch is obtained by scanning a declustered file in parallel. Thus parallelism within a batch increases by declustering a file on more nodes [5] [8]. In this case, however, performance of short-term transactions is degraded owing to CPU overhead of message passing.

2: A cost model of a batch transaction is given only by its I/O demands: i.e. each reading/writing step has the cost of C. C is the number of objects to be accessed sequentially by this step. An object is a unit of data for a bulk-access operation, such as a cylinder of a disk. Only a part of a file can be accessed if indices are used. Also we assume that the updated data are flushed to disks soon after write-ahead logging. Thus the I/O demand from the commitment of a transaction to its completion is ignored. □

Example: Fig.2-(a) shows two batch transactions $T1$ and $T2$. Each batch has a sequence of three steps. Here, "*step*1 → *step*2" represents a sequential execution of steps. $\mathtt{r}_i(P:C)$ (or $\mathtt{w}_i(P:C)$) means a reading (or writing) step of T_i to a file P, and the step has the cost of C. A commitment step comes at the last of each transaction's sequence. This commitment step is omitted in the rest of this paper.

(a) Sequences of two transactions $T1$, $T2$:

$T1$: `r1(A:1)` → `r1(B:3)` → `w1(A:1).`

$T2$: `r2(C:1)` → `w2(A:1)` → `w2(C:1).`

(b) WTPG where $T1$ and $T2$ have just started.

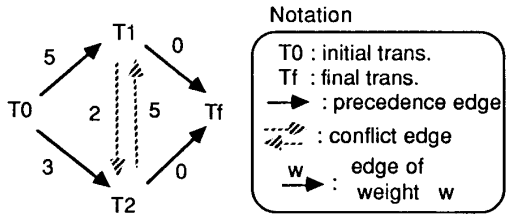


Figure 2: Examples of a WTPG

3 New Schedulers for Batches

This section outlines our new schedulers [13] designed for batch transaction processing. Apparently, batch-transaction scheduling has high contention of both data and resources [1]. These high contentions degrade performance greatly [1] [15]. Thus our schedulers generate a serializable schedule of batches so that these contentions are reduced as much as possible.

As a general scheme, our schedulers work as follows: First, the schedulers estimate the degree of the contentions in a scheduling state, and next the lock-request q is granted only if q causes the smallest degree of the contentions. For this estimation, our schedulers use a new tool called a *Weighted Transaction-Precedence Graph* (WTPG) [13]. A WTPG represents not only a serializable order between transactions but the I/O costs for executing them. The following subsections describe two new schedulers using a WTPG, each of which uses a different estimation strategy.

3.1 Weighted Transaction-Graph

This subsection introduces a WTPG [13]. Fig.2-(b) shows a WTPG where the two transactions of Fig.2-(a) have just started and have declared their sequences of Fig.2-(a). In Fig.2-(a), $T0$ (or Tf) is the initial (or the final) transaction [2] [2].

A WTPG is a variation of 'transaction-precedence graph' (or 'serialization graph' [2]), except that each edge has a *weight*. A weight between two transactions represents the I/O cost to be used between commitments (or completions) of these transactions. When a new transaction starts up, it

[1] Resource contention means congestion of resources, and data contention means locking conflict [15].

[2] $T0$ is a virtual transaction which precedes all the transactions in any serializable order. Tf is also a virtual one which is preceded by all the transactions.

must declare both its sequence of steps and I/O demands of each step. These 'access-declarations' with their I/O costs are used to build up a new WTPG.

A WTPG represents a serializable order between transactions as follows: In Fig.2-(b), a pair of shaded arrows, $\{T1 \rightarrow T2\}$ and $\{T2 \rightarrow T1\}$, is named a *conflict-edge* $(T1, T2)$. In general, (Ti, Tj) means that both Ti and Tj have declared conflicting accesses to the same file. (e.g. in this case, $T1$ and $T2$ have declared exclusive accesses to the file `A`). When it is determined that Ti precedes Tj in a serializable order, (Ti, Tj) is replaced by a *precedence-edge* $\{Ti \rightarrow Tj\}$.

Then each edge has the following weight:

1: Each edge from Ti to Tj has the weight of w. This weight has the following meaning: "Suppose that Tj is blocked by Ti and that Ti has just now committed. Then, from now, Tj must access w objects before its commitment.". e.g. In Fig.2-(b), $\{T1 \rightarrow T2\}$ has the weight of 2. That is, $T2$ is blocked by $T1$ at its second step, `w2(A:1)`. Thus $T2$ has the remaining cost of 2 objects (i.e. the cost of both `w2(A:1)` and `w2(C:1)`) before its commitment. □

Weights on edges of either $T0$ or Tf have the following special meanings:

2: A weight w on $\{T0 \rightarrow Ti\}$ means "At the current scheduling-state, Ti must access the remaining I/O cost of w objects before its commitment". e.g. In Fig.2-(b), $\{T0 \rightarrow T1\}$ has the weight of 5 because $T1$ has just started up. □

3: A weight on $\{Ti \rightarrow Tf\}$ represents the I/O cost to be paid between Ti's commitment and its completion. From the cost model in Section 2, this is set to 0 for all Ti. Tf and its edges are not depicted in the rest of the paper. □

When a schedule proceeds, only the weights of $T0$'s edges are adjusted [13]. The other kinds of weights are kept constant.

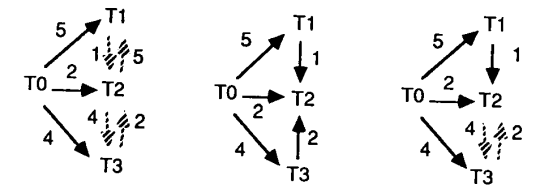
3.2 Scheduler using global optimization

Here we outline the *Globally-Optimized WTPG scheduler* [13] (GOW [3]). GOW uses a global optimization strategy to estimate the degree of the contentions, as follows: *Let W be a full serializable order of the output schedule. Then W should make the shortest critical path in the WTPG of the current state.*

(In a given WTPG, a serialzable order is called 'full' when it replaces all conflict-edges by precedence-edges. The critical path is the longest path from T0 to Tf)

Example: Suppose Fig.3-(a) shows a WTPG at the current scheduling-state. In this WTPG, a full serializable order, $W = \{T1 \rightarrow T2,\ T3 \rightarrow T2\}$, replaces all the conflict-edges by precedence-edges. W modifies Fig.3-(a) into Fig.3-(b), and makes the critical path $\{T0 \rightarrow T1 \rightarrow T2\}$ there. This critical path is shorter than that made by any other full serializable order in Fig.3-(a). Thus the serializable order of the output schedule should be W. □

[3] This was called the Chain-WTPG scheduler in [13].



(Length of path P = $\sum$ (weights of edges in P))

Figure 3: WTPG's in Globally-Optimized WTPG scheduler

INPUT: a lock-request q on a locking-granule d.
OUTPUT: granting, blocking, delaying, or aborting q.

Algorithm GOW(q)

Phase0: If q is the start step of a new transaction T, GOW() examines whether the WTPG of the current state keeps a 'chain-form' by starting T. If this examination fails, then q is aborted and GOW() returns. Otherwise, append T to the WTPG.

Phase1: If q conflicts with the current lock held on d, q is blocked and GOW() returns.

Phase2: Compute a full serializable order W, where W makes the shortest critical path in the WTPG of the current state.

Phase3: If a serializable order which is inconsistent with W is generated by granting q, then q is delayed and GOW() returns. Otherwise q is granted.

Phase4: Find a conflict-edge, (Ti, Tj), where it is newly determined that Ti precedes Tj. If such the edge exists, then replace it by $\{Ti \rightarrow Tj\}$, and GOW() returns.

Figure 4: Globally-Optimized WTPG scheduler

Apparently, a chain of blocking such as $\{T1 \rightarrow T2 \rightarrow T3\}$ makes a long critical path in Fig.3-(a). Also high contention of resources keeps large weights on $\{T0 \rightarrow Ti\}$. Thus the scheduling strategy of GOW reduces the contentions in the output schedule as far as GOW can predict.

Fig.4 shows the scheduling algorithm of GOW when a lock-request is submitted. In Fig.4, the *Phase2* finds the globally-optimized serializable order W, and then W is enforced to the output schedule at the *Phase 3*. Aborted or delayed lock-requests are submitted to GOW after some delay.

It is NP-hard to find the above serializable order, W, in any given WTPG. The *Phase0* of Fig.4 thus restricts a WTPG into a 'chain-form', where W is computed by $O((Number\ of\ Nodes)^2)$ in any given chain-form WTPG [13]. A "chain-form WTPG" means that each general transaction conflicts only with its adjacent nodes in the WTPG. e.g. WTPGs of Fig.3 are chain-forms.

Example: Fig.3-(a) shows the WTPG of the current

INPUT: a lock-request q.

Function $\mathcal{E}(q)$

Phase1: Let $W0$ be the WTPG of the current state. Then make the WTPG, $W1$, where q has been just granted in $W0$. Here, by granting q in $W0$, the serializable order between Ti and Tj may be determined. If so, replace such conflict-edge (Ti, Tj) by the precedence-edge $\{Ti \rightarrow Tj\}$. If q causes a deadlock, then return $\mathcal{E}(q) = \infty$.

Phase2: Ignore all the remaining conflict-edges in $W1$. Then $\mathcal{E}(q)$ is set to the length of the critical path from $T0$ to Tf in this WTPG.

Figure 5: Function $\mathcal{E}(q)$

scheduling-state. $W = \{T1 \rightarrow T2,\ T3 \rightarrow T2\}$ makes the shortest critical path in this WTPG. Then GOW works as follows: In Fig.3-(a), suppose that $T1$ requests a lock which conflicts with $T2$. If this lock is granted, then $(T1, T2)$ is replaced by $\{T1 \rightarrow T2\}$. This is consistent with W. Thus this lock is granted to $T1$ and then the current WTPG is modified into that of Fig.3-(c). On the other hand, if $T2$ requests a lock conflicting with $T1$ in Fig.3-(a), then this makes the inconsistent precedence-order, $\{\ T2 \rightarrow T1\ \}$, against W. Therefore the request of $T2$ is not granted but is delayed. □

3.3 Scheduler using local optimization

The 'chain-form' constraint of GOW fairly limits the number of running transactions when batches update a small part of database (i.e. 'hot' files such as master files). Because of this problem, we also proposed another scheduler using a less strict optimization strategy [13]. This subsection describes this *Locally-Optimized WTPG scheduler* (LOW) [4].

The scheduling strategy of LOW is as follows: *LOW grants a lock-request, q, only when q causes the smallest degree of the contentions in the current scheduling-state.* This strategy does not predict how much contentions may occur in the future scheduling-state. Based on this strategy, when a lock-request q is submitted, LOW computes the function $\mathcal{E}(q)$ shown in Fig.5. $\mathcal{E}(q)$ estimates how much degree of contentions have been caused when granting q in the current state.

Example: Fig.6 illustrates the procedure of $\mathcal{E}(q)$. For simplicity, the weight on $\{T0 \rightarrow Ti\}$ is set to 0 for all Ti. $T0$ with its edges is not displayed in this figure. Fig.6-(a) shows a WTPG of the current state. In Fig.6-(a), suppose that $T5$ issues a lock-request, q, which conflicts with $T6$. Then Fig.6-(b) shows the WTPG where $\mathcal{E}(q)$ is computed. Fig.6-(b) is made from Fig.6-(a) as follows: First, q gets granted in Fig.6-(a), and then $(T5, T6)$ is replaced by $\{T5 \rightarrow T6\}$. Then the path $\{T4 \rightarrow T5 \rightarrow T6 \rightarrow T7\}$ is generated. Thus $(T4, T7)$ in Fig.6-(a) is replaced by

[4] This was called the K-conflict WTPG scheduler in [13].

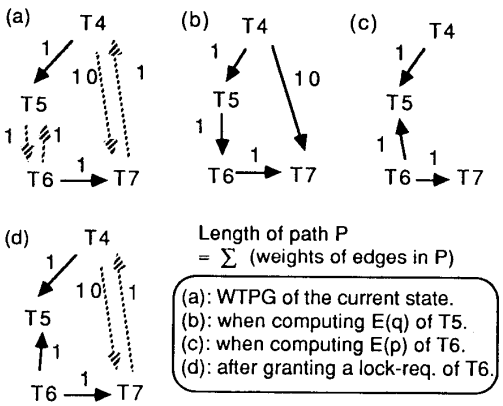


Figure 6: WTPG's in Locally-Optimized WTPG scheduler

INPUT: a lock-request q on a locking-granule d.
OUTPUT: granting q, or delaying q, or blocking q.

Algorithm LOW(q)

Phase1: If q conflicts with the current lock held on d, q is blocked and LOW() returns.

Phase2: Compute $\mathcal{E}(q)$. If q causes a deadlock, q is delayed and LOW() returns.

Phase3: If $\mathcal{E}(q) \leq \mathcal{E}(p)$ for all $p \in C(q)$, then q is granted. ($C(q)$ is a set of access-declarations which conflict with q on d.) Otherwise q is delayed and LOW() returns.

Phase4: Find a conflict-edge (Ti, Tj) where it is newly determined that Ti precedes Tj. If such the edge exists, then replace it by $\{Ti \rightarrow Tj\}$, and then LOW() returns.

Figure 7: Locally-Optimized WTPG scheduler

$\{T4 \rightarrow T7\}$. Then, in Fig.6-(b), the critical path is $\{T4 \rightarrow T7\}$ of length 10. Hence $\mathcal{E}(q) = 10$. □

Fig.7 shows the algorithm of LOW when a lock-request q is submitted. $\mathcal{E}(q)$ is computed at its Phase2. At the Phase3, LOW grants q only when $\mathcal{E}(q) \leq \mathcal{E}(p)$ for all p, where p is any access-declaration conflicting with q. If $\mathcal{E}(q) > \mathcal{E}(p)$ for some p, then LOW judges that the lock requested by q should be granted primarily to the transaction declaring p.

Example: Suppose that $T5$ issues a lock-request q in Fig.6-(a), where q conflicts with $T6$'s access-declaration, p. LOW then computes $\mathcal{E}(q)$ and $\mathcal{E}(p)$. Fig.6-(c) shows the WTPG where $\mathcal{E}(p)$ is computed. Here the conflict-edge, $(T4, T7)$ in Fig.6-(a), has been ignored by *Phase2* of $\mathcal{E}(p)$. Thus $\mathcal{E}(p) = 1$. Because $\mathcal{E}(q) = 10 > \mathcal{E}(p) = 1$, LOW delays q. On the other hand, if $T6$ issues the lock-request p in Fig.6-(a), then p is granted and thereafter the WTPG is modified into that of Fig.6-(d). □

At the Phase3 of Fig.7, we limit the size of $C(q)$ to a constant, K $(= 0, 1, 2, \ldots)$. LOW of $K = i$ is computed by

Table 1: Simulation Parameters

Parameter	Meaning	Value
NumNodes	Number of Data-Processing Nodes	8
NumFiles	Number of Files	8 to 64
DD	Degree of Declustering	1, 2, 4, 8
mpl	multi programming level	infinite
λ	transaction arrival rate (TPS)	0 to 1.4
CPUspeed	CPU speed of Control Node	4 MIPS
netdelay	network delay time	0 ms
msgtime	message send/receive CPUtime	2 ms
sot_time	CPUtime of transaction startup	2 ms
cot_time	CPUtime of commitment	7 ms
ddtime	CPUtime of deadlock detection in C2PL	1 ms
kwtpgtime	CPUtime of computing $\mathcal{E}(q)$ in LOW	10 ms
chaintime	CPUtime of computing the optimized serializable order in GOW	30ms
toptime	CPUtime of chain-form test	5 ms
ObjTime	1 object processing time at a Data-Processing Node at $DD = 1$	1000 ms

$O((Number\ of\ Nodes)^2)$ [13]. LOW starts up a new transaction only when this limited size of $C(q)$ is not violated. Even at $K = 1$, LOW allows a non chain-form WTPG.

4 Simulation model

4.1 Machine model

Here we describe a simulation model of a Shared-Nothing machine of Fig.1. The model in Fig.1 has one control node and *NumNodes* nodes for database processing. Table 1 lists its parameters with their values.

1. Data Placement: *NumFiles* is the number of files, where each step of a batch scans a *file*, as described in Section 2. A file, *fileID*, is placed at its *home* node, where its *nodeID* = (*fileID* mod *NumNodes*). *DD* is called the degree of declustering: i.e. When a file is declustered on *DD* nodes, it is split into *DD* partitions and they are located at *DD* nodes (from its home node to the node of ((*home_nodeID* + *DD* − 1) mod *NumNodes*)).

2. Control Node (CN): CN has a lock table of file-level locking granules. CN grants or blocks or delays a lock-request to a file. The model of CN is set to a *CPUspeed* MIPS processor. A new transaction T arrives at CN in the exponential distribution of arrival rate λ. At CN, the number of active transactions is controlled to be less than multi-programming level, *mpl*. When T commits (or starts), CN spends *cot_time* (or *sot_time*) as a coordinator of two-phase commitment.

3. Data-Processing Node (DPN): The model of a DPN is given by *ObjTime*. *ObjTime* is the time used for processing 1 object at a DPN. This simple model is justified because a bulk-access operation is implemented by pipeline between processors and disks, and because its cost is in I/O-bound region when most of accessed data is filtered out. Control overheads such as initiating a cohort is ignored at DPN's because they are far smaller than *ObjTime*.

4. Execution model: A batch transaction T is executed as follows: First, when a step of T accesses a file declustered on *DD* nodes, CN sends T to its home node. Next T is split into *DD* cohorts and then they are sent to the appropriate nodes. Thereafter a DPN executes cohorts in a round-robin manner. When $DD = k$, the unit of the round-robin service is to scan the data of size $1/k$ object. When a cohort of T's step ends on a DPN, it returns to its home node. After all the cohorts of a step have returned to its home node, T returns to CN. Thereafter T requests its next step.

4.2 Parameter setting

Here we explain the parameter setting listed in Table1.

Concurrency-control parameters *ddtime*, *chaintime*, and *kwtpgtime* are set up from instruction counts of our simulator.

ObjTime of 1 second corresponds to a DPN model of a 4 MIPS processor per disk of 2.5 MB/second transfer rate. One object corresponds to sequential scan of 2.5 mega byte. From our experiences on Functional Disk System-R [10], this model can execute a join-operation in I/O-bound manner if 80% data of source relations are filetered out before the join. *ObjTime* is much larger in practical cases. Therefore this setting overestimates control overhead.

NumNodes is set to 8. The values of *NumFiles* in Table 1 are far smaller than those of previous studies about short-term transactions [4] [1] [3]. This is because a database has tens of relations, while it has millions of record-level locking granules.

In each experiment, a new transaction is given its sequence of steps by "*Pattern: step1* → ... → *stepN*". Steps are expressed as described in Section 2. Each new transaction is an instance of this pattern. In each pattern, the I/O cost of each step shows the cost in the case of $DD = 1$. When $DD = k$, k cohorts execute a step of cost C in parallel. Thus this step declares the cost of C/k when $DD = k$.

We use the following three performance metrics: mean response time, *RT* (the time from arrival of a transaction to its completion), throughput (Number of completed Transactions Per Second, TPS), and response-time speedup at a fixed arrival rate. The response-time speedup of any given scheduler, S, is given by (*RT* of S at $DD = k$)/(*RT* of S at $DD = 1$). This represents relative performance gain of each scheduler when it utilizes intra-transaction parallelism. At each measured point of a fixed arrival rate, the simulation ran during 2,000,000 clocks (1 clock = 1 millisecond) with *mpl* = infinity.

We examine the following six schedulers: GOW, LOW of $K = 2$ (LOW), Atomic Static Locking (or 'conservative two-phase locking' in [2]) (ASL) [15], Cautious Two-Phase Locking (C2PL) [12], Optimistic Locking (OPT) [11], and NO Data Contention (NODC). NODC grants any lock at any time so that it shows upper bound of performance. C2PL is a variation of strict two-phase locking [12], and

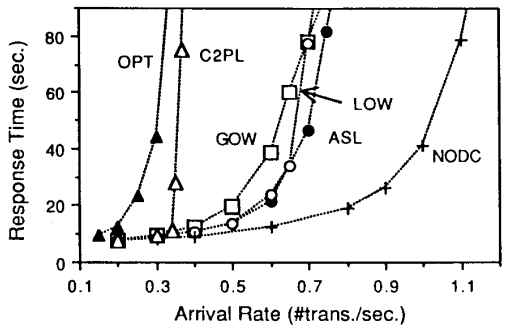


Figure 8: Exp.1: Arrival Rate vs. Resp. Time. ($DD = 1$, $NumFiles = 16$)

has a WTPG (without weights) for predicting deadlock. C2PL grants a lock-request q if and only if q is not blocked and does not cause a deadlock. ASL is the two-phase locking where a transaction has to get all the necessary locks at its start. OPT certifies serializability when a transaction commits, and then a transaction is aborted if this certification fails. The schedulers except OPT do not have either deadlock or rollback.

5 Experiments and Results

This section shows and discusses experimental results. Batch transactions have no fixed form of operations, so their workloads are unfixed. Thus we examine the following two cases distinctly:
(1): the case when batch transactions are frequently blocked.
(2): the case when batch transactions update a hot set.

If a scheduler keeps high performance in the above two cases, it can keep stable and high performance in any given workload of batch transactions.

5.1 Performance effect on blocking

First, we examine the case where transactions are frequently blocked. This examination shows how well the schedulers avoid chains of blocking when parallelism increases. The following experiment is used throughout this subsection:

Experiment1:
Pattern1: `r(F1:1)`→ `r(F2:5)` → `w(F1:0.2)` → `w(F2:1)`. $NumFiles = 16$ (in default), $DD = 1$, 2, 4, or 8. The two files, F1 and F2, are randomly chosen among $NumFiles$ files. X-locks are requested at the first two steps. These steps cause chains of blocking. □

5.1.1 Characteristic of batch processing

Table 2: Exp.1: Number of Files (#files) vs. Throughput (TPS) at Resp.Time = 70 sec., DD=1.

#files	NODC	ASL	GOW	LOW	C2PL	OPT
8	1.02	0.45	0.44	0.44	0.25	0.16
16	1.04	0.72	0.67	0.65	0.35	0.24
32	1.04	0.9	0.86	0.83	0.5	0.3
64	1.04	0.96	0.95	0.94	0.62	0.38

First of all, we describe characteristics of batch-transaction processing in comparison with short-term transaction processing.

Fig.8 shows response time, RT, as a function of arrival rate in Experiment 1, where $DD = 1$ and $NumFiles = 16$. In this figure, let λ_S be the arrival rate where any given scheduler, S, has RT=70 seconds. Then, without locking conflict, resources are saturated at $\lambda_{NODC} = 1.04$ TPS. [5]. The ratio of λ_S/λ_{NODC} shows the ratio of useful resource-utilization [6] in the case of using the scheduler S. Then this figure shows the following characteristic:

#1: When batch transactions have bulk-update operations, high data-contention degrades performance at much smaller arrival rate than resource congestion does. e.g. In Fig.8, for all scheduler S, λ_S/λ_{NODC} is less than 70 %. □

This characteristic shows that inter-transaction parallelism of batches is limited. Thus parallel execution within a batch is necessary to have high ratio of useful resource-utilization.

In contrast, short-term transaction processing has extremely low data-contention, which causes thrashing near or over the load of resource saturation. In this case, thus, locking protocols make small difference of performance by reducing overhead such as message passing or rollback.

5.1.2 Effect of concurrency control

Next we examine how the schedulers perform by using only concurrency-control of batches. Table.2 shows throughputs in Experiment1 where each scheduler has the response time of 70 seconds at $DD = 1$ and $NumFiles = 8$, 16, 32, or 64. Here, when $NumFiles$ varies, the ratio with which transactions are blocked varies drastically. This drastic change is due to unfixed workloads of batch processing, thus schedulers need to benefit from such the limited concurrency.

As for absolute performance in Table.2, ASL, GOW and LOW have almost the same performance. All of them have 1.6 to 2.0 times higher performance than C2PL and OPT. Therefore we can see that GOW and LOW avoid chains of blocking as perfectly as ASL does. In contrast C2PL has poor performance owing to chains of blocking. OPT also

[5] We observed that at λ_{NODC}, resource utilization ratio of NODC is about 95%.

[6] Resources are used usefully if a transaction using them is not aborted.

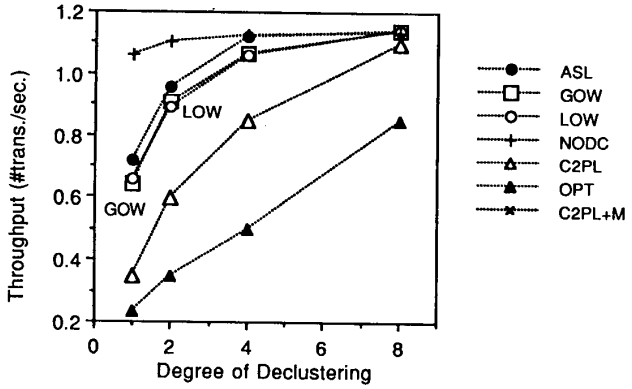


Figure9: Exp.1: Declustering vs. Thruput.
(at Resp.Time=70sec., NumFiles=16)

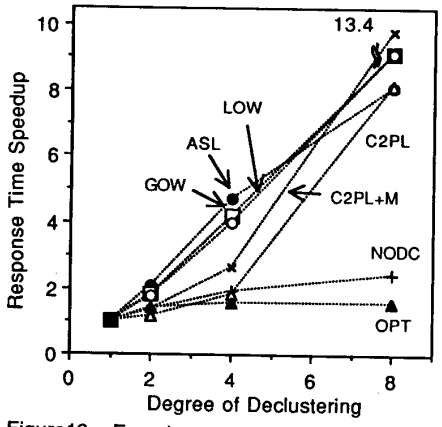


Figure10: Experiment 1:
Declustering vs. Resp.Time SpeedUp.
(NumFiles=16, Arrival Rate = 1.2tps)

Table 3: Exp.1: Declustering vs. Resp. Time (seconds). ($NumFiles = 16$, $\lambda = 1.2$ TPS)

DD	NODC	ASL	GOW	LOW	C2PL+M	OPT
1	141	387	429	430	669	783
2	103	183	233	245	479	555
4	74	83	102	107	250	494
8	58	48	47	47	50	490

has poor performance because high data-contention causes high ratio of rollback.

In terms of relative performance gain in Table.2, ASL, LOW, and GOW have high gain of throughput (of about 0.45 TPS) from $NumFiles$=8 to 32, while C2PL and OPT have poor gain (of less than 0.25 TPS).

ASL, LOW and GOW avoid chains of blocking and have no rollback of transactions. Because of this property, these three schedulers have much better absolute/relative performance through concurrency control than C2PL and OPT.

5.1.3 Effect of parallel execution

Next we examine performance effect of parallel execution in the case of blocking. Experiment1 is used here, where $NumFiles$=16 and parallelism of a batch increases from $DD = 1$ to 8.

Fig.9 shows throughputs where each scheduler has the response time of $RT = 70$ seconds. Table.3 shows response time of each scheduler at $\lambda_1 = 1.2$ TPS, where $DD = 1$ to 8. Also Fig.10 shows response-time speedup at λ_1 as a function of DD.

At the load of λ_1, NODC has the speedup of 2 at DD=8 in Fig.10. This shows that resources are fairly saturated at λ_1 if there is no data contention. Further we can see that parallel execution does not have great speedup at λ_1 if there is no locking conflict. Therefore response-time speedup at λ_1 shows how well each scheduler utilizes intra-transaction parallelism by its unique concurrency-control facility.

From Fig.9 and Table.3, the following observation of #1 hold about absolute performance:

#1: ASL, GOW and LOW keep much higher performance than C2PL and OPT when parallelism increases to a limited degree. e.g. At DD= 2 in Fig.9, ASL, LOW and GOW have high ratio of useful resource-utilization (of about 85 %), and their throughputs are 1.5 times higher than that of C2PL. Also at $DD = 2$ to 4 in Table.3, ASL, LOW and GOW have 2 to 2.5 times shorter response time than C2PL+M [7] and OPT do. □

This observation of #1 shows that parallel execution of batch transactions has greater performance-effect than their concurrency-control.

Next we discuss relative performance gain through parallelism. We can see the following observation in Fig.10:

#2: OPT has the smallest speedup of 1.5 at $DD = 4$, which is smaller than NODC's speedup of 2. Also C2PL(+M) has better speedup than OPT and NODC. □

This observation of #2 also holds in the case of short-term transaction processing. A recent study [4] explained this as follows: OPT saturates resources greatly by restarting transactions. Therefore OPT has much heavier load than NODC does. As a result, effective parallel execution of cohorts is very limited in OPT. In contrast, C2PL has no restart and blocks transactions. Thus C2PL executes cohorts effectively in parallel because C2PL makes its load much lighter than NODC does.

On the other hand, the following observations, #3 and #4, are unique to batch transaction processing. Both of them do not hold in the case of short-term transaction processing:

[7] C2PL+M is the best C2PL to control multi-programming level in order to avoid chains of blocking. C2PL+M has better response time than C2PL, but they have the same peak-throughput.

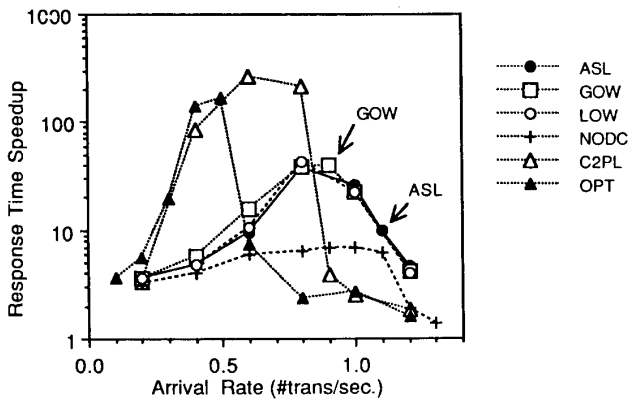


Figure 11: Arrival Rate vs. Time Speedup
(Exp.1. NumFiles=16, DD=4)

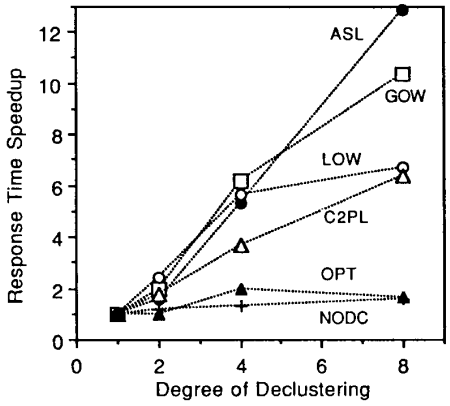


Figure12: Experiment 2:
Declustering vs. Resp.Time SpeedUp.
(at Arrival Rate = 1.2 tps)

#3: When the load is heavy, C2PL cannot obtain great speedup of response time from the limited parallelism. This is due to high data-contention of batch processing. e.g. At DD=4 in Fig.10, C2PL has the low speedup of 2.5 (in the case of C2PL+M), while ASL, LOW and GOW have the high speedup of 4 to 5. □

This observation of #3 is explained as follows: Parallelism makes the waiting time of blocked transactions much shorter, but high data-contention still raises chains of blocking at the heavy load of λ_1. From this fact, C2PL still has the poor response time although the blocking lightens the effective load at λ_1. Thus C2PL has much smaller speedup than ASL does.

#4: Even when the load is heavy, ASL, LOW and GOW can obtain rich gain of performance from the limited parallelism. e.g. In Fig.10, these three schedulers have the linear speedup of response time at DD = 1 to 8. Particularly they have high speedup even at the low degree of parallelism of $DD \leq 4$. Their speedup is the best among the six schedulers. □

This observation of #4 is explained as follows: Because of high data-contention, there are only a small number of transactions which can start without locking conflict. ASL starts only such the transactions without rollback. Consequently its load is made much lighter at λ_1 than that of NODC. Also ASL has no chains of blocking. Thus it executes much more transactions concurrently than C2PL does. Therefore ASL executes more cohorts effectively in parallel at lighter load than C2PL does. It follows that ASL has the linear speedup even at the heavy load of λ_1. GOW and LOW avoid chains of blocking as successfully as ASL. Thus they also have linear speedup of response time through the limited degree of parallelism.

We also examined response-time speedup of the schedulers at various loads. There we observed that, when DD = 1 to 4, the above observations from #2 to #4 hold at heavy loads. Here, the throughputs of C2PL in Fig.9 show the boundary between heavy loads and light loads. At each DD, any load over the throughput of C2PL is the heavy-load region. e.g. In Fig.11, we show this speedup when $DD = 4$ as a function of λ. At DD = 4, C2PL has the throughput of 0.85 TPS as shown in Fig.9. Thus, in Fig.11, the observations about relative performance gain hold at the heavy loads of $\lambda \geq 0.85$ TPS. [8].

The above observations, #3 and #4, show that blocking also limits performance benefit of parallelism. ASL, GOW and LOW reduce both blocking and resource saturation. Thus they obtain high absolute/relative performance from the limited parallelism.

5.2 Effect of parallelism on hot set

Here we examine performance effect of parallel execution in the case when batches update a hot set. The following experiment is used here.

Experiment2:
Pattern2: `r(B:5)` → `w(F1:1)` → `w(F2:1)`. The file `B` is chosen randomly among 8 'read-only' files. `F1` and `F2` are chosen randomly among the other 8 'hot' files. Every file is declustered on 1 to 8 nodes. Each node is set to a home node for both one 'read-only' file and one 'hot' file. The read/write steps in this pattern request S/X-locks respectively. □

Table.4 shows throughputs where DD = 1, 2, or 4 and each scheduler has the response time of RT = 70 seconds. This table also shows response time at λ = 1.2 TPS. Absolute performance at DD = 1 in this table shows effect of concurrency control on the schedulers in Experiment2. As

[8] At very light loads, every scheduler has almost the same response-time (of about 4 seconds when $DD = 4$). Thus C2PL and OPT have greater speedup at such light loads than the others, as shown in Fig.11. This shows that C2PL and OPT need richer parallelism in order to compete with the others.

Table 4: Exp.2: Throughput (TPS) and Response Time (seconds at λ = 1.2 tps) at DD=1, 2, 4.

	NODC	ASL	GOW	LOW	C2PL	OPT
Thruput						
$DD = 1$	1.1	0.4	0.57	0.77	0.7	0.38
2	1.11	0.7	0.88	1.01	0.92	0.55
4	1.13	1.03	1.1	1.12	1.09	0.85
Resp.Time						
$DD = 1$	112	611	500	321	432	751
2	97	380	252	133	242	746
4	87	116	80	57	118	457

discussed in our previous study [13], LOW is the best in Table.4, followed by C2PL, followed by GOW, and ASL is the worst except OPT. This gap of absolute performance is straightly due to how many transactions can start in each scheduler. Although C2PL starts the greatest number of transactions, its performance is degraded by chains of blocking on a hot set.

As for relative performance gain through parallelism, Fig.12 shows the speedup of response time at $\lambda = 1.2$ TPS as a function of DD. (The arrival rate of 1.2 TPS makes a very heavy load because NODC has the low speedup of 1.57 at $DD = 8$.) The following observation is obtained from this figure:

#1: ASL has worse response-time than C2PL. However ASL has better speedup of response time than C2PL and OPT. This is because chains of blocking on a hot set limit performance gain of C2PL. On the other hand, LOW and GOW have the best speedup as ASL does. In particular, LOW has the best throughput and the best speedup by parallel execution. □

From this observation, we can see that if no blocking occurs, relative performance gain is independent of data contention on a hot set.

In summary of Section 5.1 and 5.2, GOW and LOW utilize the limited concurrency and the limited parallelism most effectively. Both of them have the best absolute/relative performance. Particularly when updating 'hot' files, LOW is better than GOW. ASL is the best among the traditional locking protocols. However, when a workload has 'hot' files to be updated, ASL has poor and unstable performance if parallelism is limited.

5.3 Effect of parallelism on sensitivity

Although GOW and LOW perform well in the above experiments, they depend on how correctly transactions declare their I/O demands. This subsection examines this 'sensitivity' when increasing parallelism. We use the following experiment:

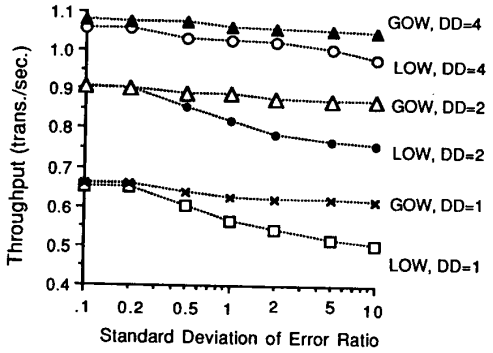


Figure 13: Exp.3: Error Ratio vs. Throughput. (DD = 1 to 4, $NumFiles$=16, Resp.Time = 70 sec.)

Experiment3: *Pattern1* of Experiment 1, with $NumFiles$=16 and DD = 1, 2, 4, or 8. I/O demand of each step is estimated at C by the formula: $C = C_0 \times (1 + x)$, where C_0 is the exact I/O demand of the step, and x is an error ratio. x is given by the normal distribution where its mean = 0 and its standard deviation, σ. ($C = 0$ when $x \leq -1$). □

At greater σ, GOW and LOW estimate the degree of data/resource-contentions incorrectly, and then their performance is degraded by chains of blocking.

Fig.13 displays throughput at RT=70 seconds as a function of σ, where parallelism increases from DD = 1 to 4. Also Table.5 shows degradation ratio of throughput as a function of DD. This degradation ratio of a scheduler is set to (its throughput at $\sigma = 10$)/(its throughput at $\sigma = 0$) in Fig.13. Further C2PL cannot avoid chains of blocking at all. Thus the throughputs of C2PL shown in Fig.9 are the lower-bound performance of both GOW and LOW.

From Fig.13, the following observations hold in the case when no parallelism is utilized:

#1: GOW and LOW avoid chains of blocking even when very incorrect I/O demands are declared. e.g., At $\sigma = 1$ and DD= 1 to 2 in Fig.13, GOW and LOW keep 1.45 to 1.7 times higher throughput than C2PL does. □

#2: LOW is more sensitive to incorrect I/O demands than GOW, as shown in Table.5. GOW is insensitive because of its chain-form constraint [13]. □

When increasing parallelism from DD = 1 to 4 in Fig.13, the following observations hold:

#3: When parallelism increases, GOW and LOW get more insensitive to the error ratio. e.g. At DD = 1 to 4 in Table.5, LOW and GOW have less degradation of throughput when increasing DD. This is because less chains of blocking occur by increasing parallelism. This reason can be seen because C2PL has higher performance at higher DD in Fig.9. □

#4: Even when increasing parallelism at high error-ratio, GOW and LOW still outperform C2PL greatly. e.g. At $DD = 4$ with $\sigma = 1$ in Fig.13, GOW and LOW have 1.23

Table 5: Experiment 3: Sensitivity Test
Degradation Ratio = (TPS at $\sigma = 10$)/(TPS at $\sigma = 0$)

	DD=1	DD=2	DD=4
GOW	94%	96%	97.5%
LOW	77%	84%	93%

times higher throughputs (of about 1.05 TPS) than C2PL does (0.85 TPS). Thus GOW or LOW should be used instead of C2PL when parallelism increases. □

In Table.5-(a), throughput of LOW is improved greatly by increasing DD. Therefore, considering performance in the case of updating a hot set, LOW is better than GOW if parallelism is utilized.

6 Conclusion

This paper has discussed batch-transaction processing on Shared-Nothing database machines, and has examined how well its performance is improved by using both concurrency control and parallel execution. The best-performing scheduler for batches is a new alternative so that these machines freely tune their data-placements for short-term transactions. We have examined cautious two-phase locking [12] (C2PL), atomic static locking (ASL) [15], optimistic locking (OPT) [11], and two new schedulers proposed in our previous study[13]: the Globally-Optimized WTPG scheduler (GOW) and the Locally-Optimized WTPG scheduler (LOW).

Simulation results have clarified the following requirements for utilizing the limited concurrency:
(1) avoiding chains of blocking,
(2) running many transactions when updating a hot set,
(3) making no rollback of transactions.

In order to obtain high gain of performance through parallel execution, the requirements of both (1) and (3) are necessary, but the requirement of (2) are not. Only LOW satisfies these three requirements. GOW has both of (1) and (3), and partially satisfies the requirement of (2). In contrast, C2PL, ASL, and OPT lack the requirements of (1), (2), and (3), respectively.

Comparing absolute performance between the schedulers, when parallelism is fairly limited (i.e. declustering a file on one or two nodes among 8 nodes), LOW and GOW keep 1.5 to 2.0 times higher throughput than the others. Particularly when updating a hot set like master files, LOW is much better than GOW. Among the traditional locking protocols, ASL is better than C2PL and OPT. However ASL has poor and unstable performance when parallelism is limited and a workload has hot files to be updated.

As for relative performance gain, we examined the case of increasing parallelism through partitioning data. There ASL, LOW and GOW have the best speedup of response time even at heavy loads. In contrast, C2PL and OPT have poor speedup at such heavy loads because of high data-contention and because of high resource-contention, respectively. Particularly, high data-contention limits relative performance gain of C2PL. This is contrast to the relatively-high gain of C2PL in the case of short-term transaction processing.

In summary, GOW and LOW utilize both concurrency-control and parallel execution of batches most effectively. Therefore, by using these dedicated schedulers for batch transactions, a Shared-Nothing machine can freely tune its data-placements for short-term transactions and has no great disadvantage to batch processing. As a further work, it would be interesting to improve these new schedulers for resource-level load-balancing on Shared-Nothing database machines.

References

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

[2] Bernstein,P.A. et al. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

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

[4] Carey,M.J. et al. Parallelism and Concurrency Control Performance in Distributed Database Machines. In *Proc. ACM-SIGMOD '89*, pages 122–133, 1989.

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

[6] DeWitt,D.J. et al. *A Single User Performance Evaluation of the Teradata Database Machine.* Technical Report DB-081-87, MCC, 1987.

[7] DeWitt,D.J. et al. Gamma - High Performance Dataflow Database Machine. In *Proc. 12th Int'l Conf. Very Large Data Bases*, pages 228–237, 1986.

[8] Ghandeharizadeh,S. et al. A Multiuser Performance Analysis of Alternative Declustering Strategies. In *IEEE Proc. 6th Int'l Conf. Data Engineering*, pages 466–475, 1990.

[9] Ghandeharizadeh,S. et al. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proc. 16th Int'l Conf. Very Large Data Bases*, pages 481–492, 1990.

[10] Kitsuregawa,M et al. Query Execution for Large Relations on Functional Disk System. In *IEEE Proc. 5th Int'l Conf. Data Engineering*, 1989.

[11] Kung,H. et al. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[12] 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.

[13] Ohmori,T. et al. Concurrency Control of Bulk Access Transactions on Shared Nothing Parallel Database Machines. In *IEEE Proc. 6th Int'l Conf. Data Engineering*, pages 476–485, 1990.

[14] Tandem Performance Group. A Benchmark of NonStop SQL on the Debit Credit Transaction. In *Proc. ACM-SIGMOD '88*, pages 337–341, 1988.

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