# 用於平行資料庫系統之查詢優先式並行控制方法
# Query-First Concurrency Control Protocols for Parallel Database Systems *

唐學明 Shyueming Tang　　呂永和 Yungho Leu

台灣科技大學資管系
Department of Information Management
National Taiwan University of Science and Technology
{tang,yhl}@cs.ntust.edu.tw

## 摘要

過去許多平行資料庫方面的研究都著重於複雜查詢的最佳化，如果一個平行資料庫系統中同時進行複雜查詢和線上交易，則會發生資料競爭的問題。本研究的目的即在探究四種「查詢優先」的交易並行控制方法，並評估其效能。「查詢優先」的交易並行控制方法使唯讀的複雜查詢在取得資料項時具有較高的優先權，如此方可減少其延遲，以充分發揮平行資料庫系統的效益。研究結果顯示，在二階段鎖定中加入查詢優先的方法，不但可以提高複雜查詢的效率，亦可提高線上交易的效率，但必需是在系統在高負載狀態或具有足夠計算能量的前提下，方可達到到此一效果。

關鍵字：並行控制、平行資料庫系統、效益分析

## ABSTRACT

*Many researches on parallel database systems have focused on query processing, especially on the optimization of complex queries. The performance issue of concurrently executing a mix of both complex queries and update transactions has not been well addressed. In this paper, we investigate a class of concurrency control protocols, called query-first protocols, for parallel database systems. These protocols allow concurrent execution of complex queries and simple update transactions.*

*The performance of four query-first protocols which favor complex queries is studied. The simulation results reveal that, using locking-based protocols, the overall performance can be improved only when the system load is high or the system resources are sufficient.*

Keywords: Concurrency Control, Parallel Database Systems, Performance Analysis

*This work is supported by the Republic of China National Science Council under Contract NSC 86-2213-E-011-009

## 1 Introduction

A parallel database system can be defined as a database management system implemented on a tightly coupled multiprocessors [1]. During the past decade, parallel database systems have been widely used in the field due to their high performance, scalability and availability. For example, the Oracle database system is equipped with a component called Parallel Server, while the INFORMIX-OnLine Dynamic Server provides parallel query processing capability using a technique called PDQ (Parallel Data Query).

Although many issues on parallel database systems have been thoroughly studied, the problem of concurrency control has not been addressed adequately [2]. Complex queries tend to access huge amount of data and may easily conflict with update transactions; that is, the high data contention due to update transactions may hinder the execution of complex queries. As a result, the benefits of parallel query processing cannot be fully obtained in an environment where complex queries are executed together with update transactions.

There have been few performance studies on the concurrency control in parallel database systems. M. Carey and M. Livny used simulation to compare the performance of four distributed concurrency control protocols (2PL, wound-wait, OCC, and times tamp ordering) on a database machine that distributed over eight sites [3]. Another study proposed by B. Jenq et al. further addressed the issue of 2PL performance in parallel database systems with widely varying size (from 4 nodes to 256 nodes) [4]. These studies only considered simple transactions, and do not take complex queries into account. T. Ohmori et al. have proposed an approach to reduce the data contention between bulk access transactions [5]. Their study concentrated on the estimation of degree of contention and reducing the contention by global optimization.

In this paper, we report on our efforts to develop

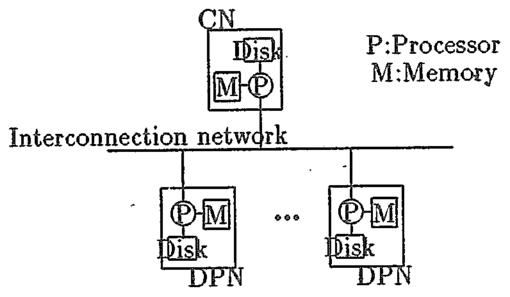

Figure 1: Conceptual Model of a Shared-Nothing Database System

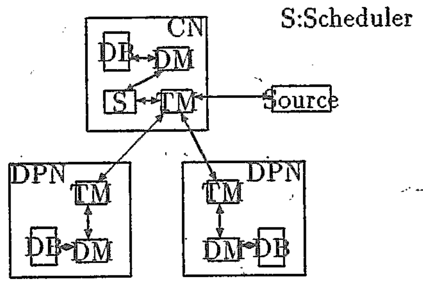

Figure 2: Components of an SN Parallel Database Management System

priority-based concurrency control protocols for parallel databases. These protocols are called *query-first protocols* because they give complex queries higher priority than that of update transactions. In this way, the execution of complex queries will not be blocked by update transactions.

The remainder of this paper is organized as follows. Section 2 gives the background of this research. In Section 3, we propose four query-first concurrency control protocols. In Section 4, we describe our simulation model used in our experiments. In Section 5, we study the effects of our protocols by analyzing the experiment results. Section 6 concludes this paper.

## 2 Background

In this section, we outline the architecture of parallel database systems, and briefly review the basic concurrency control protocols, including the 2PL protocol and the OCC protocol.

### 2.1 Target Environment

Our target environment for transaction processing is a *shared-nothing* (SN) parallel database system. This architecture consists of multiple *nodes* which connected by an *interconnection network* [6]. Each node contains a processor, a local cache memory, and a disk unit which is used to store the database. These nodes work in parallel to process queries posed by users. Figure 1 shows the conceptual model of an SN database system [4].

There are two types of nodes in an SN database system, one *control node (CN)* and multiple *data-processing nodes (DPN)*, as shown in Figure 2 [3] [5]. Application programs run typically on the CN, and the task of concurrency control is also centralized at the CN. In the CN, there are three distinct components: a *transaction manager (TM)*, which monitors the execution of transactions (including complex queries and update transactions) and coordinates the execution of a multi-node query; a *concurrency control manager* (or *scheduler*), which implements a particular concurrency control protocol and processes data requests issued by the TM; and a *data manager (DM)*, which is responsible for the actual execution of database operations. A *source* component is associated with the CN only; that is, only the CN can generate transactions on behalf of user's request. The components of a DPN is similar to those of CN, but lack of the concurrency control manager component because all data requests are scheduled in the CN. For simplicity, we assume that a fast interconnection network is used, and the actual wire time for data transmission is negligible [3].

### 2.2 Basic Concurrency Control Protocols

Two classes of concurrency control protocols related to our research are 2PL and OCC. A transaction is required to set a read lock on a data object before reading it and to set a write lock before writing on it. Our study assumes a *dynamic* locking strategy; that is, locks are set dynamically when the data objects are required.

In the OCC protocol, the execution of a transaction consists of three phases: the read phase, the validation phase and the write phase. There are two validation schemes — *backward validation* and *forward validation* [7]. We adopt the forward validation scheme in our study because it provides more flexibility for conflict resolution than the backward validation scheme.

## 3 Query-First Concurrency Control Protocols

In this section, we propose four query-first concurrency control protocols which can be divided into two families : 2PL-based protocols and OCC-based protocols.

### 3.1 Query-First Protocols Based on Two-Phase Locking

In the family of the 2PL-based query-first protocols, we propose two different protocols. They are *2PL with wounding lock-holders* (abbreviated as 2PL-Wound) protocol and *2PL with adjusting waiting order* (abbreviated as 2PL-Wait) protocol. In these protocols, each transaction follows the rules of 2PL protocol on lock-requesting and lock-releasing. However, in order to ensure that complex queries have priority in accessing data over update transactions, we augment the 2PL protocol with a priority-based conflict resolution scheme. In this scheme, priority is assigned to each transaction according to its class.

In our lock-requesting algorithms, we assume that, for each data object $x$, the scheduler maintains two queues, the *lock* queue (denoted as *lock_queue*[$x$]) and the *wait* queue (denoted as *wait_queue*[$x$]). The lock queue of a data object maintains the identifier of transactions that have been granted locks on this data object, whereas the wait queue of a data object maintains the identifier of transactions whose lock requests are blocked. There are two types of entries in both the lock queue and the wait queue — the *write* entry and the *read* entry. A write entry may only contain a transaction identifier (*tid*), while a read entry may contain multiple *tids*.

Consider a transaction that requests to access a data object, if the lock queue of the data object is empty or the entry in the lock queue is in a non-conflicting mode, the transaction is allowed to access the data object. If a lock request is allowed, we put the *tid* in an entry (read or write), and put the entry into the lock queue. If the entry in the lock queue is in a conflicting mode, the lock request is blocked. To prevent a deadlock, we use the *wait-depth-limited* (WDL) method to limit the wait-depth of a transaction [8]. A *wait-depth-limited* routine is invoked once a lock request is blocked. If the wait-depth of a transaction is greater than a constant $d$ (which is set to 3 in our algorithms), the transaction is aborted and restarted. Otherwise, its *tid* is put in an entry and appended to wait queue. Because a data object can be simultaneously read by multiple transactions, a read entry in a lock queue may contain more than one *tid*. Putting a *tid* into a lock queue is performed by first removing the existing read entry from the lock queue, then adding the *tid* to the entry, and finally putting the entry back into the lock queue.

#### 3.1.1 Two-Phase Locking with Wounding Lock-Holders (2PL-Wound)

In the 2PL-Wound protocol, if a complex query requests a lock on a data object held by an update transaction in a conflicting mode, the update transaction is immediately *wounded* (i.e., aborted). Conversely, if an update transaction requests a lock on a data object held by transactions (update transactions or complex queries) in a conflicting mode, the requester should *wait* for the lock holders to release the lock. The lock-requesting algorithm of the 2PL-Wound protocol is shown in Figure 3.

```
// let oid be the id of the requested data object
// let ltype be the lock type of the lock request
// let tid be the transaction id of the requester
if the lock_queue(oid) is not empty
    if lock type of the entry in lock_queue(oid) = WRITE
        let tno be the transaction id of the lock-holder
        if priority(tid) > priority(tno)
            abort transaction tno
            remove the entry from lock_queue(oid)
            add tid to the entry
            put the entry back into lock_queue(oid)
        else
            call WDL routine giving tid, oid
            if wait-depth(tid) > d
                abort transaction tid
            else
                put tid in an entry
                append the entry to wait_queue(oid)
            endif
        endif
    else
        if ltype = WRITE
            call WDL routine giving tid, oid
            if wait-depth(tid) > d
                abort transaction tid
            else
                put tid in a write entry
                append the entry to wait_queue(oid)
            endif
        else
            put tid in a read entry
            put the entry into lock_queue(oid)
        endif
    endif
else
    put tid in a new entry
    put the entry into lock_queue(oid)
endif
```

Figure 3. Lock-Requesting Algorithm of the 2PL-Wound Protocol

#### 3.1.2 Two-Phase Locking with Adjusting Waiting Order (2PL-Wait)

In the 2PL-Wait protocol, instead of immediately aborting an update transaction when it blocks a complex query, the scheduler permits the update transaction to keep its locks, but the read entry that belongs to the complex query is adjusted to the "first" position of the wait queue. Conversely, if an update transaction is blocked, its entry is append to the "last" position of the wait queue. The lock-requesting algorithm of the 2PL-Wait protocol is depicted as follows.

```
// let $oid$ be the id of the requested data object
// let $ltype$ be the lock type of the lock request
// let $tid$ be the transaction id of the requester
if the lock_queue($oid$) is not empty
    if lock type of the entry in lock_queue($oid$) = WRITE
        let $tno$ be the transaction id of the lock-holder
        if priority($tid$) > priority($tno$)
            call WDL routine giving $tid$, $oid$
            if wait-depth($tid$) > d
                abort transaction $tno$
                remove the entry from lock_queue($oid$)
                add $tid$ to the entry
                put the entry back into lock_queue($oid$)
            else
                put $tid$ in an entry
                store the entry as $first$ of wait_queue($oid$)
            endif
        else
            call WDL routine giving $tid$, $oid$
            if wait-depth($tid$) > d
                abort transaction $tid$
            else
                put $tid$ in an entry
                append the entry to wait_queue($oid$)
            endif
        endif
    else
        if $ltype$ = WRITE
            call WDL routine giving $tid$, $oid$
            if wait-depth($tid$) > d
                abort transaction $tid$
            else
                put $tid$ in a write entry
                append the entry to wait_queue($oid$)
            endif
        else
            put $tid$ in a read entry
            put the entry into lock_queue($oid$)
        endif
    endif
else
    put $tid$ in a new entry
    put the entry into lock_queue($oid$)
endif
```

Figure 4. Lock-Requesting Algorithm of the 2PL-Wait Protocol

## 3.2 Query-First Protocols Based on Optimistic Concurrency Controlling

In the family of OCC-based query-first protocols, we proposed two different protocols. They are *OCC with sacrificing validating transaction* (abbreviated as OCC-Sac) protocol and *OCC with waiting for complex queries* (abbreviated as OCC-Wait) protocol. In these protocols, database consistency is maintained by checking the write set of a validating transaction against the read set of concurrently executing (active) transactions. Once a conflict is detected, the active transaction is aborted and restarted later. In order to ensure that complex queries may not be interrupted by update transactions, we augment the OCC protocol with a priority-based conflict resolution scheme. In this scheme, the priorities of complex queries are set higher than those of update transactions. It should be noted that a complex query needs not to be validated because its write set is empty.

### 3.2.1 Optimistic Protocol with Sacrificing Validating Transaction (OCC-Sac)

In the OCC-Sac protocol, if a conflict is detected in the validation phase, the low-priority transaction should be restarted; that is, the low-priority transaction (update transaction) is *sacrificed* in an effort to help the high-priority transaction (complex query) complete its works. On the other hand, if the conflicting transactions have identical priority, the active transaction should be restarted. The validating algorithm of the OCC-Sac protocol is shown in Figure 5.

```
// let $T_A$ be the validating transaction, and
// let $T_B$ be a transaction in the active transaction set
move $T_A$ from active transaction set
for all $T_B$ in active transaction set
    if the read set of $T_B$ is conflicted with the write set of $T_A$
        if priority($T_B$) > priority($T_A$)
            restart $T_A$
            return
        else
            restart $T_B$
        endif
    endif
endloop
```

Figure 5. Validating Algorithm of the OCC-Sac Protocol

The OCC-Sac protocol satisfies the goal of favoring complex queries. However, it suffers from the *cyclic restart* problem by repeatedly restarting the update transactions. We use a restart waiting policy to solve the problem. That is, an update transaction is forced to delayed a fixed period of time before it is restarted, in order not to conflict with the same complex query again.

### 3.2.2 Optimistic Protocol with Waiting for Complex Queries (OCC-Wait)

In the OCC-Wait protocol, when a validating transaction (i.e., an update transaction) conflicts with a complex query, the validating transaction is forced to wait until the corresponding complex query completes, or until it is aborted by other validating transactions. It is noted that once a validating transaction is forced to wait, it must exit the validation phase and put into the active transaction set. The validating algorithm of the OCC-Wait protocol is shown in Figure 6.

```
// let $T_A$ be the validating transaction, and
// let $T_B$ be a transaction in the active transaction set
move $T_A$ from active transaction set
for all $T_B$ in active transaction set
    if the read set of $T_B$ is conflicted with the write set of $T_A$
        if priority($T_B$) > priority($T_A$)
            $T_A$ is put back to active transaction set
            return
        else
            restart $T_B$
        endif
    endif
end loop
```

Figure 6. Validating Algorithm of the OCC-Wait Protocol

## 4 The Simulation Model

In order to study and measure the performance of the proposed protocols described in the previous section, a simulation is performed using SIMSCRIPT II.5 .

The physical resources of a parallel database system are processors, memory, disks and interconnection network. Instead of modeling these physical resources, we take CN and DPN as the components of physical queuing model. Figure 7 shows the physical queuing model of a parallel database system. Resource overhead is associated with each concurrency control request, database access, and subsequent operation. For example, a lock request requires the CN service, while the subsequent data access and operation require the DPN service. The CN has an FIFO queue to keep the concurrency control requests. Each DPN also owns a FIFO queue to keep the data access requests and operation requests.

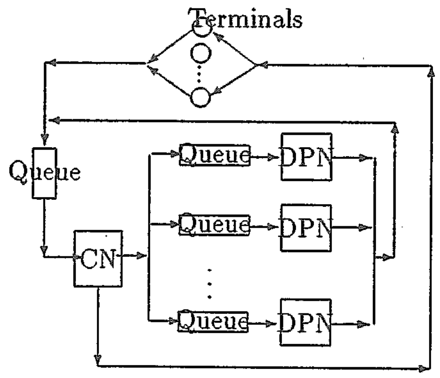

Figure 3: Physical Queuing Model of a Parallel Database System

## 5 Experiments and Results

In this section, we present and analyze the results of various simulation experiments for query-first protocols. Two families of protocols (2PL-based and OCC-based) are compared separately. The performance metric used in this paper is *transaction turnaround time*, which is the time between the origination of a transaction at a terminal to the completion of the transaction.

### 5.1 Effect of Multiprogramming Level

We first evaluate the effect of multiprogramming level on the performance of the four query-first protocols. The experiment is based on a system with 64 DPNs.

Figure 8 and 9 show the transaction turnaround time for the two families of protocols. In Figure 8, both the 2PL-Wound protocol and the 2PL-Wait protocol have lower turnaround time than that of the 2PL protocol when the multiprogramming level is greater than 40 (terminals). This result implies that, in an environment of high data contention, the overall performance can be improved using the query-first protocols. Due to conflict resolution and parallel data processing, a complex query can be completed within a short period of time. As a result, the blocking time of update transactions is also reduced. In Figure 9, both the OCC-Sac protocol and the OCC-Wait protocol have higher turnaround time than that of the OCC protocol. This result seems adverse to our expectation. However, the OCC-based query-first protocols can be used to improve the throughput of complex queries which are hard to complete when ordinary OCC protocol is used.

### 5.2 Effect of the Number of Data-Processing Nodes

In this section, we evaluate the effect of the number of DPN on the performance of the query-first protocols. In this experiment, we consider systems with DPN_CNT being 8 to 64 in steps of 8, while NUM_TERMINAL is fixed at 50.

Figure 10 and 11 show the transaction turnaround time for the two families of protocols. In Figure 10, both the 2PL-Wound protocol and the 2PL-Wait protocol have lower turnaround time when the number of DPN is greater than 48. This result highlights the fact that abundant system resources have positive impact on the overall performance of the query-first protocols. In Figure 11, both the OCC-Sac protocol and the OCC-Wait protocol have higher turnaround time than that

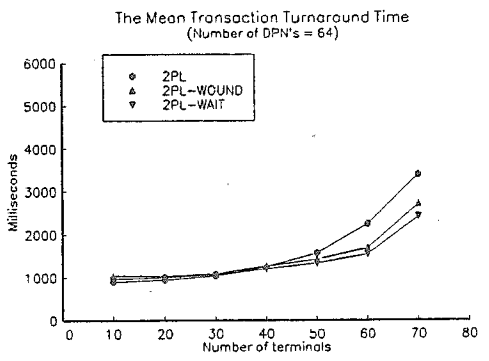

Figure 8. Comparison of 2PL-based protocols

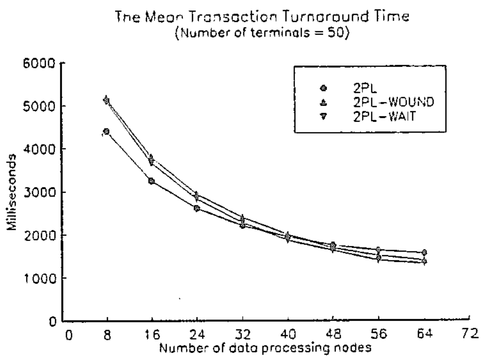

Figure 10. Comparison of 2PL-based protocols

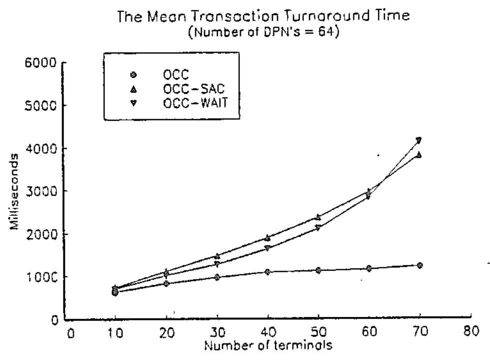

Figure 9. Comparison of OCC-based protocols

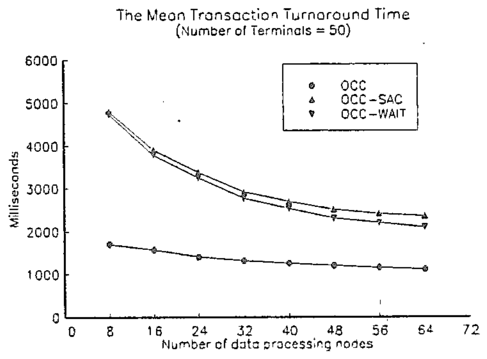

Figure 11. Comparison of OCC-based protocols

of the ordinary OCC protocol.

## 6 Conclusion

In this paper, we propose four query-first concurrency control protocols for parallel database systems, and investigate their performance issues. Using a simulation model, these protocols were evaluated under varying levels of multiprogramming (the number of terminals) and amount of underlying physical resources (the number of DPN). The experiment results show that, the locking-based query-first protocols exhibit substantial performance improvements in an environment with high multiprogramming level or sufficient physical resources. The optimistic-based query-first protocols do not exhibit overall performance improvement, yet they can be used to improve the throughput of complex queries. In general, the query-first protocols which favor complex queries have negative impact on the performance of update transactions. Unless the performance gains obtained from the speedup of complex queries exceed the performance loss of update transactions, the overall performance can not be improved.

## References

[1] Patrick Valduriez, *Parallel Database Systems: Open Problems and New Issues* , Distributed and Parallel Databases, Vol. 1, No. 2, Apr. 1993, pp.137-166.

[2] D. DeWitt and J. Gray, *Parallel Database Systems: The Future of High Performance Database Systems* , Communications of the ACM, Vol.35, No.6, June 1992, pp.85-98.

[3] M. Carey and M. Livny, *Parallelism and Concurrency Control Performance in Distributed Database Machines* , Proc. ACM SIGMOD, 1989, pp.122-133.

[4] B. Jenq et al., *Locking Performance in a Shared Nothing Parallel Database Machine* , IEEE Trans. Knowledge and Data Eng., Vol.1, No.4, Dec. 1989, pp.530-543.

[5] T. Ohmori et al., *Scheduling Batch Transactions on Shared Nothing Parallel Database Machines: Effects of Concurrency and parallelism* , Proc. of 7th Conf. on Data Eng., 1991, pp.210-219.

[6] A. Bhide, *An Analysis of Three Transaction Processing Architectures* , Proc. VLDB Conf., 1988, pp.339-350.

[7] T. Harder, *Observations on Optimistic Concurrency Control Schemes* , Inform. Systems, Vol.9, No.2, 1984, pp.111-120.

[8] P. A. Franaszek et al., *Concurrency Control for High Contention Environments* , ACM Trans. on Database Systems, Vol.17, No.2, June 1992, pp.304-345.