Optimistic Concurrency Control based on Cache Coherency in Distributed Database Systems

Tae-Young Choe,

Kumoh National Institute of Technology 1, YangHo Dong, Gumi, Korea

Summary
Optimistic concurrency control highly takes advantage of parallelism because there is no delay by lock. Unfortunately, I/O operations by transactions and cache operations are delayed in the optimistic concurrency control. In order to reduce such delay, we combine cache coherency control and optimistic concurrency control rather than they operate in separate. In this paper, we propose optimistic concurrency control, which includes cache control in parallel database system. Two basic concurrency control schemes, *direct validation scheme* and *reduced validation scheme* are devised according to the degree of cache *activity*. Based on these basic schemes, we designed four integrated concurrency control algorithms: GSO, GMS, GMM, and LMM. Experimental results show that GSO, GMS and GMM that use global cache show higher performance than LMM that uses local cache. GSO that is a direct validation scheme has larger communication overhead than reduced validation scheme. So, GMS shows the best performance among other algorithms as the portion of communication overhead is large.

Key words:
Optimistic Concurrency Control, Cache Coherency Control, Distributed Database System, Object Oriented Database.

1. Introduction

Concurrency control is a major part of transaction processing in conventional database systems. Let us consider a parallel or distributed database systems. In the database systems, a transaction should be executed atomically. In other words, transactions which occur in multiple processors should be coordinated so that the result remains the same as if they are executed sequentially. It is called concurrency control in database system. Concurrency control schemes are typically categorized as follows: pessimistic control and optimistic control [1], where multiple worker processors handle transactions and require I/O to disk servers. Each worker processor is composed of CPU, memories, clocks, etc. Disk servers are dedicated to store database files and data blocks.

In general pessimistic concurrency controls use lock mechanism in order to maintain serialization. Under lock-based pessimistic concurrency control, a transaction locks the object that will be accessed. After it finishes operations on the object, it unlocks the object. There are two types of lock: read-lock and write-lock. Read-locks are compatible with each other, but other types are not compatible. A transaction must obtain lock permission before it executes its read/write operation. At once it gets the permission, the operation is guaranteed as a valid operation as long as there is no error. The pessimistic concurrency control is vulnerable to deadlock. Pessimistic concurrency control and cache coherency control can be combined to improve system performance [2], where the cache control locates under concurrency control in general.

Optimistic concurrency control makes transactions start without checking any possible conflicts for their objects. The transactions are validated by checking whether they are finished without any collision. If there was any collision, the transactions causing the collision are aborted. When collisions occur frequently, optimistic concurrency control incurs quite large overheads. In the case of optimistic concurrency control, write operations are not guaranteed to be valid at the time of invocation. Thus cache subsystem must not use coherency control to the operations to preserve cache coherency. For example, let a write operation of a transaction occur and a cache control modify or invalidate copies in other clients. If the transaction is aborted or invalidated, effect of cache control must be rolled back, which is large overhead. After all, it is efficient that optimistic concurrency control manages cache or integrates with it.

Among many optimistic concurrency controls, some methods rearrange transaction orders in order to reduce the number of conflictions which invoke recover overheads. Such protocols base on dynamic adjustment of serialization order (DASO) [3][4][5]. Haritsa et al. proposed WAIT-50, an optimistic concurrency control that recognizes conflict stat and grants priority to more urgent transactions [6]. Some optimistic concurrency controls divide a transaction duration into two or three sub-cycles and validate more read-only transactions using enhanced forward validation schemes [7][8]. Also there are some optimistic concurrency controls that uses timestamp in order to reduce coarseness of conflict detections [9].

Unfortunately, these algorithms do not consider cache operations that enhance operation speed by maintaining copy of objects in each server. Although Yu and Vahdat proposed a consistency model, they assumed replicated servers as the base system [10]. As a method of reducing

Manuscript received November 5, 2008
Manuscript revised November 20, 2008

the number of conflictions, cache management can be utilized. We device protocols which reduce conflicts between transactions by carefully combining cache coherency control. We present two types of concurrency controls that utilize cache to control serializability in transaction: direct validation and reduced validation.

This paper is organized as follows: In Section 2, we propose two cache schemes which control concurrency in database system. In Section 3, we present four concurrency control algorithms from above schemes. In Section 4, we show the result of simulation of the schemes. Conclusions appear in Section 5.

2. Classified Schemes

Combination of cache coherency control and concurrency control makes two types of schemes which are classified by the degree of cache activity. We call a concurrency control as *reduced validation scheme* if the control suppresses cache activity and reduces the number of disk access. If the control does not restrict operations of cache coherency control, and it checks validity of transactions just according to cache operation, then we call it as *direct validation scheme*.

2.1 Reduced Validation Scheme

In general, the number of I/O operations can be reduced by using cache in each processor. Such reduction can be applied to I/O operations in transactions. Multiple read operations for an object can be reduced to one real disk read by putting the object into cache at the time of the first read operation. Also multiple writes for an object can be reduced to one real disk write by writing the object into cache during transaction and by sending the last write request to disk server at the end of transaction as shown in Fig. 1. Also, cache control can enrich serialization. We know that a disk server serializes I/O-requests, because a disk server processes one I/O at a time. Thus if each transaction requests only one I/O operation, they are serialized by the disk server. From these properties, we manage transaction control and cache control as follows: First, we reduce disk I/O operations as much as possible with the aid of cache control. Second, serialization of transactions are automatically achieved, if each transaction contains only one I/O. Thus we can focus on the transactions that generate two or more I/O operations after reduction by cache. Fig. 1 shows the reduction of disk I/O operations.

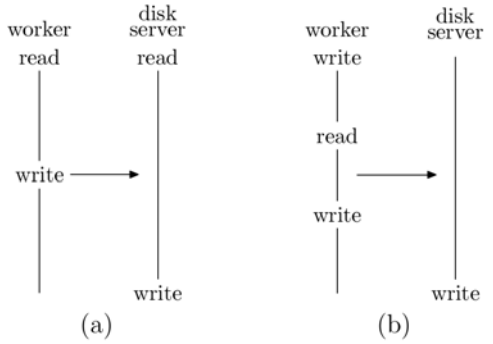

Fig. 1 Reduction of disk I/O by cache: Left side of each example shows I/O operations before reduction using cache. Right side of each example shows I/O operations after reduction using cache.

Let us explain I/O reduction using cache control on a case-by-case. Sequences of I/O operations for an object (or page) in transaction can be categorized into three cases. First, let I/O operations of a transaction start with a *read* and there is a following *write* operation as shown in Figure 1 (a). When there is a read request to an object which does not exist or invalid in local cache, the object is read from disk and is stored in the local cache. Following read or write operations to the object access the copy of the object in the local cache, which reduces the number of message communications to disk server. At the end of the transaction, a write request of the object is sent to the disk server. From the viewpoint of the disk server, there are one read and one write operations. This type of transaction is invalidated if other transaction modifies the object between the first read and write to disk server.

Second case is a case that a transaction requires only *read* operations. After an object is obtained by the first read request, a copy of the object is stored in the local cache. Following read requests access the copy in the local cache. Since only one disk read operation is required, this type of transaction need not be invalidated.

Final case is that the first I/O in a transaction is *write* operation as shown in Fig. 1 (b). After an object is written to cache, following read and write requests to the object access the copy of the object in the local cache. At the end of the transaction, one write to disk server is required. So this type of transaction also need not be invalidated.

Two cases of transactions except the first one require only one disk operation and they are not interfered by other transactions. So they are serialized naturally by disk servers without any further operations. After all, only the first case of transaction T_a needs to be checked for consistency. If there is any transaction T which writes the object to the disk server between first read and write at the end of transaction T_a, transaction T_a must be invalidated.

2.2 Direct Validation Scheme

Given a shared object, a write operation after a read operation has a possibility of breaking serializability. For example, in Fig. 2, transaction T_b is invalidated at the end of transaction because shared object x is already modified when T_b enters verification phase. However, T_b can be invalidated before it enters verification phase. If T_b knows that object x is already modified by other transaction just after T_a executes write operation for the object x, T_b can be invalidated instantly and can withdraw further operations. When a copy of the object is modified, stale copies in other workers can be invalidated or be modified by cache coherency control.

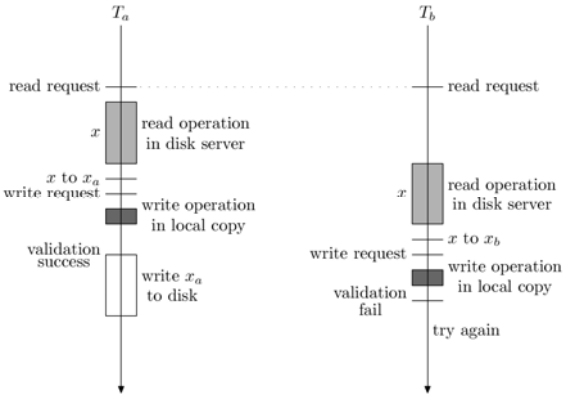

Fig. 2 Processing of two transactions in optimistic concurrency control

Instead of cache being managed by concurrency control, *direct validation scheme* makes cache coherency control fully operate by itself, and checks validity of transactions from result submitted by the cache control. The scheme invalidates transactions that access a copy of an object that is invalidated or modified by cache coherency control. If direct validation scheme is applied to Fig. 2, as soon as transaction T_a writes x_a to its local memory, copy of x in T_b is invalidated by the cache control. When transaction T_b requests write operation, concurrency control notices invalidated copy x and invalidates T_b as shown in Fig. 3.

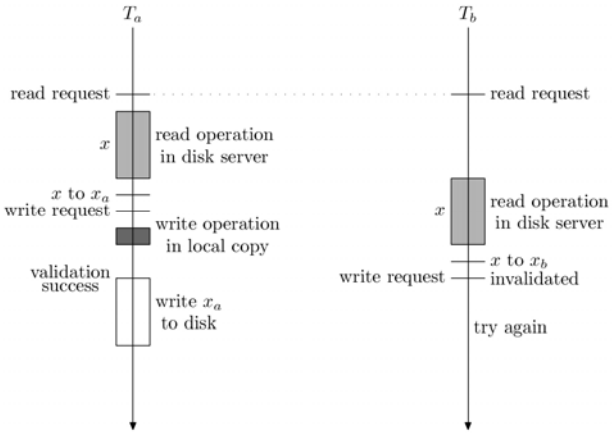

Fig. 3 Early invalidation using cache coherency control.

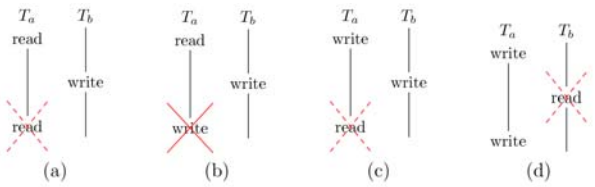

Fig. 4 Possible invalidation cases by cache in concurrency control.

Strict cache coherency control is not a unique solution for the concurrency control. Fig. 4 shows all possible invalidation cases when two transactions access a shared object. In Fig. 4 a transaction T_a executed I/O operations for an object and another transaction T_b accesses the object before T_a enters validation phase. A transaction shown in Fig. 4 (b) is an obvious invalidation case. Using cache coherency control, a copy of a shared object in local cache of transaction T_a is invalidated when transaction T_b writes to the object. When T_a tries to write to the copy of object, it notices invalidation of the copy and invalidates itself. Such invalidation can be applied to all cases in Fig. 4. However if we loosen invalidation of stale copies, transaction cases shown in Fig. 4 (a), (c), and (d) can be validated because they preserve serialization. An advantage of strict cache coherency control is that fast invalidation reduces wasted CPU time of transactions that would be invalidated. On the other hands, a disadvantage of strict cache coherency control is that some serializable transactions could be invalidated.

2. Implementation Issues and Some Proposed Algorithms

We investigated how to combine cache coherency control and transaction concurrency control. When we implement

an integrated algorithm in a system environment as shown in Fig. 5, the following issues must be determined:

- When is a modified copy transfers from a local cache to disk server?
- When is a stale copy in a local cache invalidated?
- When will invalidated transactions be stopped?
- The number of write-copies: a write-copy is a copy of an object that is modified by a transaction and is not written to a disk yet.
- The number of read-copies: a read-copy is a copy of an object that is generated when a transaction read the object from a disk.
- Global cache or local cache

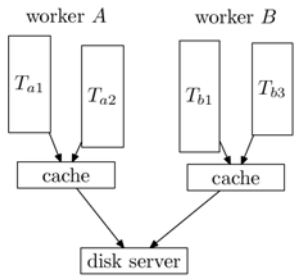

Fig. 5 Multi-transactions in a distributed disk server model.

From various combinations of the alternatives, four algorithms GSO, GMS, GMM, and LMM are devised. Representative properties that distinguish the algorithms are shown in Table 1. To simplify implementation of concurrency control and *abort* operation, all algorithms delay invalidations of old copies as late as possible. That is, a transaction does not know whether a copy of an object in other transactions is modified. Invalidation of an object copy is noticed when another transaction sends a verification message with write requests to a disk server at the end of the transaction. GSO, GMS, and GMM algorithms adopt global cache mechanism proposed in [11].

Global Single copy Only (GSO) algorithm is a strict design of direct validation scheme. At most one copy for an object exists in entire caches. Each object copy in worker has a timestamp which is a read request completion time of the object [9] and an object modification time. Also transaction has a timestamp which is a newest timestamp of an object which is possessed by the transaction. If a transaction requires a read for an object to a disk server, the server checks whether it is the unique owner of the copy of the object. In the case, the server sends the copy to the transaction with timestamp of the current time. Otherwise, the server forwards the request to a worker which has the copy. The worker checks the request by comparing a timestamp in the request and a timestamp of the object copy. If the transaction first requested the copy, it has no timestamp. Also the request has no timestamp. If the request has no timestamp or two timestamps are the same, the worker transfers the copy to the transaction which invokes the read request and invalidates its local copy. Otherwise, the transaction is invalidated. When a transaction tries to modify the copy, a cache manager looks for the copy. If there is no copy in the cache, the cache manager requires it to the server with a timestamp of the transaction. The protocol is the same as the read request for the copy. After the transaction receives the copy, it modifies the copy and increases its timestamp. Write operation does no effect on validation in the algorithm.

Global Multiple read-copies Single write-copy (GMS) algorithm is less strict than GSO in terms of cache coherency. Multiple read-copies for each object are allowed. However, as soon as a copy is written to a disk server, other copies are invalidated. An object in a disk server has two states: opened or closed. If an object is opened, any request is possible. Otherwise, any transaction that issues a request to the object is invalidated. If a transaction issues an object read request to a disk server, the server returns a copy of the object as long as the object is opened. Before a transaction issues an object write request to a disk sever, it checks whether the object copy is valid. In the case, the object is written to the server. The server invalidates other copies and closes the object. Otherwise, the transaction is invalidated. If another transaction just writes the object after the invalidation, the transaction is invalidated because the object is closed. The closed object is reopened when the transaction that wrote the object finishes.

Global Multiple read-copies Multiple write-copies (GMM) algorithm allows multiple read or write copies. GMM is a implementation which is nearer to reduced validation scheme than to direct validation scheme. GMM is a slight modification of a timestamp implementation proposed at [12]. If a transaction issues the first read request of an object, a disk server returns a copy of the object with a new timestamp. When a transaction issues a write request, if there is a copy, the copy is changed with a new value and the transaction set a flag that there is a write operation. If a write request to an object is the first I/O operation of the transaction, a copy of the object is written to the local buffer without timestamp. When the transaction with any write request enters verification phase, it sends the copy of the object with timestamp. If the timestamp of the copy is smaller than the timestamp of the object in the server, the transaction is aborted. Otherwise, the copy is written to the object in the server and the timestamp of the object is changed to the current server

time. If the copy sent by transaction has no timestamp, the transaction is validated because the transaction just makes a copy of an object by writing instead of reading from the server, which preserves serializability. Transactions shown in Fig. 4 (a), (c), and (d) are validated if each transactions are located in other processors and GMM is applied.

Local site Multiple read-copies Multiple write-copies (LMM) algorithm allows at most one read-copy and multiple write-copies for an object in each local cache of a transaction. GMM invalidates transaction cases shown in Fig. 4 (a), (c), and (d) if two transactions T_a and T_b run in the same processor. If there is a write operation to an object copy in a processor, following operations are all invalidated because they share the local cache. In the case of LMM, each transaction has its local buffer. Thus above cases are validated in LMM. Table 1 arranges properties of each algorithm.

Table 1: Classification of integrated concurrency control algorithms. 1/proc means that there is at most one copy in a processor and 1/trans means that a copy can exist in a transaction

	scheme	Time of copy management	No. copies		times tamp
			write	read	
GSO	direct	as soon as it modified	0	1	use
GMS	reduced	verification phase	1/p	1	
GMM	reduced	verification phase	1/p	1/p	use
LMM	reduced	verification phase	1/t	1/t	use

4. Experimental Results

4.1 Experiment Environments

In order to measure the performance of the proposed algorithms, we simulated a simple object-oriented database environment as shown in Fig. 5 with 500 object types. The simulation is programmed in language C++ and uses C++SIM [13] as a basic simulation package. There are a schema and multiple instances for each object, which are evenly distributed in each disk. A class schema is striped over five disks. Table 2 shows the database configuration. We used a small-sized database to reduce simulation time. Instances of object type 499 occupy almost disk space and 50% of access rate. Thus instances of object type 499 fill large amount of cache. We use 5 disks. Total database size is 179 Mbytes.

Table 2: Specification of object-oriented database used in experiment.

Object id	0~399	400~498	499
Block no. of class	0~399	400~498	499
Block no. of instance	1000~1399	1400~1498	1499
Schema size	1 KBs	2 KBs	3 KBs
Instance size	2 KBs	3 KBs	4 KBs
The number of instances in each disk	25	50	250
Access rate of sequential transactions	10%	40%	50%
Access rate of parallel transactions	45%	35%	20%

Table 3 shows constant values that are used in the experiments. Internal network means the communication between processes in a processor. External network means the communication between processes in different processors. User think time is the time interval between read operation completion time and write operation start time on an instance in modify transaction. Event inter-arrival time is the time interval between the end of transaction and start of new transaction in a worker processor. Workload of the system is controlled by the number of clients instead of event interarrival time. We use LRU (Least Recently Used) cache scheme as base cache strategy.

Table 3: Constants used in the experiment. Disk specification is based on Seagate BARRACUDA 2.26 Gbyte Ultra-SCSI disk (8 bit SCA).

internal network setup time	1 μsec
internal network speed	20 Mbytes/sec
external network setup time	100 μsec
external network speed	2 Mbytes/sec
average disk seek time(r/w)	8.8/9.8 msec
average disk rotation latency	4.17 msec
disk transfer time	50 nsec/byte
average user think time	0.5 sec
event interarrival time	0.1 sec
cache strategy	LRU

4. Simulation Results

The performance of GMM algorithm is almost the same as that of GMS algorithm. Difference between GMS and GMM is a decision policy. Assume the condition that a transaction T_a reads an object, other transaction T_b write the object, and T_a reads the object again as shown in Fig. 4 (a). T_a is invalidated in GMS, but it is not invalidated in GMM. Because our experimental environment does not generate such repeated read, GMM and GMS have same performance.

Table 4 shows the performance of GSO scheme, GMS scheme, and LMM scheme. Table 4 (a) and (b) show that workload and write ratio little effect on the hit ratio. GMS is superior to LMM in all cases, which mean that global cache is better than local cache in parallel database system. From Table 4 (a), hit ratio of LMM with 2048 Kbytes cache is similar to that of GMS with 512 Kbytes cache. Our cache control uses simple LRU scheme, thus we can see that global cache is scalable without cooperative operations between caches. GSO and GMS are similar in hit ratio and the percentage of conflicts as shown in Table 4 (a), (b), and (d). But GMS is a little better than GSO in throughput. The reason is that since

GSO uses long data messages more frequently than GMS, response time of GMS is shorter than GSO, and the simulation model uses closed queuing system. The percentage of conflict in Table 4(d) reveals effect of ``When is the stale copy in local cache invalidated?" Since LMM does not invalidate stale copy in other local copy, the percentage of conflict is higher than that of GSO or GMS.

5. Conclusions

In optimistic concurrency control, validity of write operation is known not in time when it invoked but in time when transaction of it is validated. Thus write operation must be delayed, and cache operation for the write operation also be delayed. This property shows that cache control is heavily related with concurrency control. It is efficient to integrate cache control to concurrency control rather than cache exists as isolated layer.

According that how many caches retain its activity, concurrency controls are classified as *direct validation scheme* and *reduced validation scheme*. Direct validation scheme preserves activity of cache, and it invalidates transaction which tries to accesses invalidated object again. Reduced validation scheme uses 'reduction of I/O frequency' of cache, and reduces the number of access for an object to one or two. In the case that the access number is one, the transaction is valid. In the case that the access number is two, the object must not be modified between the accesses. It simplifies validity check of transaction.

Implementation details generate various sort of concurrency control algorithm: GSO, GMS, GMM, and LMM. GSO is an algorithm of direct validation scheme. GMS, GMM, and LMM are algorithms reduced validation scheme. GSO, GMS, and GMM use global cache scheme.

Acknowledgments

This work was supported by Research Fund, Kumoh National Institute of Technology.

References

[1] M. A. Bassiouni, “Single-site and distributed optimistic protocols for concurrency control”, *IEEE Transactions on Software Engineering*, vol 14 no. 8, pp. 1071-1080, August 1988.

[2] Michael J. Franklin, Michael J. Carey, and Miron Livny, “Transactional client-server cache consistency: Alternatives and performance”, *ACM Transactions on Database Systems*, vol. 22, no. 3, pp. 315-363, September 1997.

[3] Jan Lindstrom and Kimmo Raatikainen, “Dynamic adjustment of serialization order using timestamp intervals in real-time databases”, in *6th International Conference on Real-Time Computing Systems and Applications*. 1999, pp. 13-20, IEEE Computer Society Press.

[4] Juhnyoung Lee and Sang H. Son. “Performance of concurrency control algorithms for real-time database systems”. in *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, 1996, pp. 429-460, Prentice Hall.

[5] Yongyan Wan, Qiang Wang, Hongan Wang, and Guozhong Dai, “Dynamic adjustment of execution order in real-time databases”, in Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS’04), 2004, p. 87a.

[6] Jayant R. Haritsa, Michael J. Carey, and Miron Livny, “Dynamic real-time optimistic concurrency control”. in *IEEE Real-Time Systems Symposium*, December 1990, pp. 94-103.

[7] Li Guohui, Yang Bing, and Chen Jixiong, “Efficient optimistic concurrency control for mobile real-time transactions in a wireless data broadcast environment”, in *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA05)*, August 2005, pp. 443-446.

[8] Levent Grgen, Claudia Roncancio, Cyril Labb, and Vincent Olive, “Transactional issues in sensor data management”, in *Proceedings of the 3rd workshop on Data management for sensor networks: in conjunction with VLDB 2006 (DMSN06)*, September 2006, pp. 27-32.

[9] Quazi Ehsanul Kabir Mamun and Hidenori Nakazato, “Timestamp based optimistic concurrency control”, in *TENCON2005*, November 2005, pp. 1-5.

[10] Haifeng Yu and Amin Vahdat, “Design and evaluation of a conit-based continuous consistency model for replicated services”, *ACM Transactions on Computer Systems*, vol. 20, no. 3, pp. 239-282, August 2002.

[11] Avraham Leff, Joel L. Wolf, and Philip S. Yu, “Efficient LRU-based buffering in a LAN remote caching architecture”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 2, pp. 191-216, February 1996.

[12] Michael J. Carey, “Improving the performance of an optimistic concurrency control algorithm through timestamps and versions”, *IEEE Transactions on Software Engineering*, vol. SE-13, no. 6, pp. 746-751, June 1987.

[13] “C++sim home page”, http://cxxsim.ncl.ac.uk/.

Tae-Young Choe received the B.S. degrees in Mathematical Education from Korea University in 1991, M.S. and Ph.D degrees in Computer Science and Engineering from POSTECH in 1996 and 2002, respectively. He is an Assistant Prefessor at Kumoh National Institute of Technology – Korea since 2002. He has been a visiting researcher at GEORGIA TECH – USA during 2007. His current research interests include parallel and distributed algorithms, high performance storage systems, and computer security.

Table 4: Performance of GSO/GMS/LMM scheme with 5 disks in sequential transactions environment (a) Cache hit ratio (%) according to cache size and write ratio, (b) Cache hit ratio (%) according to the number of workers per disk and write ratio when cache size is 2048KB, (c) Throughput (Mbytes/sec) according to the number of workers per disk and write ratio, (d) The percentage of conflicts per transaction (%) according to write ratio and the number of clients when cache size is 2048KB.

Cache size	GSO				GMS				LMM			
	0%	33%	66%	100%	0%	33%	66%	100%	0%	33%	66%	100%
512MB	27.8	27.7	27.8	28.3	28.5	27.7	27.7	28.1	19.8	19.3	18.9	18.6
2048MB	51.9	51.4	52.0	51.8	52.1	51.6	51.9	51.8	25.5	23.8	22.8	22.3
8196MB	77.9	77.6	77.2	76.2	78.0	77.6	77.2	76.2	44.9	38.6	35.6	34.1

(a)

Workers per disk	GSO				GMS				LMM			
	0%	33%	66%	100%	0%	33%	66%	100%	0%	33%	66%	100%
1	51.4	51.7	50.5	51.5	51.6	51.6	51.6	51.4	25.6	23.9	22.9	22.3
10	51.4	51.5	51.5	51.5	51.7	51.6	51.5	51.5	25.6	23.8	22.8	22.1
20	51.9	51.4	52.0	51.8	52.1	51.6	51.9	51.8	25.5	23.8	22.8	22.3

(b)

Workers per disk	GSO				GMS				LMM			
	0%	33%	66%	100%	0%	33%	66%	100%	0%	33%	66%	100%
1	0.55	0.24	0.11	0.11	0.64	0.25	0.15	0.11	0.60	0.24	0.15	0.11
10	3.35	2.34	1.30	1.01	4.44	2.19	1.42	1.05	3.48	2.02	1.37	1.01
20	4.39	2.63	2.03	1.67	5.23	3.46	2.41	1.91	3.82	2.78	2.13	1.77

(c)

workers per disk	GSO				GMS				LMM			
	0%	33%	66%	100%	0%	33%	66%	100%	0%	33%	66%	100%
1	0.0	0.01	0.01	0.08	0.0	0.01	0.03	0.06	0.0	0.13	0.39	0.90
10	0.0	0.22	0.40	0.78	0.0	0.19	0.46	0.89	0.0	0.29	0.85	1.38
20	0.0	0.26	0.77	1.34	0.0	0.31	0.92	1.49	0.0	0.41	1.21	2.19

(d)