Unified Support for Stability and Bulk Data in a Persistent Store

M. G. Ashton and F. A. Henskens

Department of Computer Science & Software Engineering
University of Newcastle
N.S.W. 2308
email: mashton@ozemail.com.au, henskens@cs.newcastle.edu.au

KEYWORDS:

Persistence, concurrency-control, stability, bulk data

ABSTRACT

The results of research into the use of a unified set of mechanisms to support store stability and concurrency control is presented. It is shown that the use and extension of the data structures already maintained by the system to support stability provides support for a novel approach to provision of cooperative concurrency control in persistent systems.

1. INTRODUCTION

Conventional computer systems implement a dichotomy of storage mechanisms: a *file store* that provides a durable repository allowing data to exist after the creating program has ceased execution (such data is termed long-term data), and *virtual memory* which provides a repository for data during program execution (such data is termed short-term data). In typical conventional systems it is the responsibility of the programmer to manage the conversion of data between the transient form suitable for virtual memory and the permanent form suitable for the file store. This conversion uses up CPU cycles, and can result in data misinterpretation leading to reduced data protection.

Persistent systems, on the other hand, remove the distinction between long and short-term data by providing a single set of mechanisms for the management of data regardless of its lifetime. Work on persistence to date has largely concentrated on implementation of stores that adhere to the properties of orthogonal persistence defined in [1]. A much smaller body of work has investigated applications which exploit the benefits of persistence. Persistent stores have long been touted as alternatives to conventional database systems, but to our knowledge no-one, with the possible exception of IBM with the AS/400 [2], has actually implemented a database-style application interface to a persistent store. Such an interface would rely on the underlying persistent store providing a durable repository for data rather than having to provide this durability itself, as occurs with conventional database systems. Unwillingness to adopt persistent stores as a basis for bulk data management systems has in part been due to doubts about the performance of such stores.

Previously [3] the authors presented results showing that a properly designed persistent system could match a conventional system for basic bulk data storage. This paper presents the results of research into the use of a unified set of mechanisms to support (typically operating system provided) store stability and (typically application-level provided) concurrency control.

2. SUPPORT FOR STABILITY IN PERSISTENT STORES

A persistent store is said to be stable if it automatically recovers to a consistent state after a failure that has prevented orderly system shutdown. Stability in persistent stores is typically provided using operations called checkpoints that flush all modified data currently held in main memory to disk, and atomically create a snapshot of the store at that moment. Early stability schemes, for example [4], checkpointed the entire store at once, requiring processing on the store to cease during a checkpoint operation In a multi-user store involving multiple nodes this would result in unacceptable degradation of performance. Accordingly, systems have been developed which checkpoint parts of the store independently (for example [5]). The stable state of such a store is the collection of these stable parts. Checkpointing parts of the store independently however, creates the possibility of logical inconsistencies between data. Modified data from one object may influence the way a process modifies data in some other object. As a result the two objects have a dependency relationship that must be considered when checkpointing either of them. Such dependencies have been described using sets of clients in Casper [5], and more recently using directed graphs of entities [6].

These directed graphs, termed Directed Dependency Graphs (DDGs) are maintained by the operating system as follows:

- An $\rightarrow$ edge is used to specify the dependency relationship between two entities. $E_1 \rightarrow E_2$ means that E_1 depends on E_2. $\rightarrow$ is transitive, but not symmetric i.e. if $(E_1 \rightarrow E_2)$ and $(E_2 \rightarrow E_3)$, then it is implied that $(E_1 \rightarrow E_3)$, but $E_1 \rightarrow E_2$ does not imply $E_2 \rightarrow E_1$.
- The right hand side of a dependency relation may depend on the left hand side either through transitivity (for instance if $E_1 \rightarrow E_2$, we may also have $E_2 \rightarrow E_3$ and $E_3 \rightarrow E_1$ which implies that $E_2 \rightarrow E_1$), or when a process has modified an object, which results in two unidirectional edges with different directions between the two entities.
- In the case of a write operation which leads to a pair of dependencies, instead of indicating two

unidirectional edges ($E_1 \rightarrow E_2$ and $E_2 \leftarrow E_1$), we use the notation $E_1 \leftrightarrow E_2$.

- When a process belonging to a DDG reads a modified object or modifies an object that belongs to another DDG, the two DDGs are merged using one of the above edges to create a single larger graph.
- When a process reads an unmodified object, nothing is added to any DDG.
- When a process P_1 reads a modified object O_1, the edge $P_1 \rightarrow O_1$ is added to the DDG(s) including P_1 and O_1.
- When a process P_1 modifies an object O_1, the edge $P_1 \leftrightarrow O_1$ is added to the DDG(s) including P_1 and O_1.
- As shown in figure 1, an $\rightarrow$ edge represents both $\overset{S}{\rightarrow}$ (i.e. in terms of checkpoint or *stabilise* dependency) and $\overset{R}{\leftarrow}$ (i.e. in terms of roll-back dependency). Thus $E_1 \rightarrow E_2$ implies that checkpoint of E_1 propagates to E_2 (but checkpoint of E_2 does not propagate to E_1) and that roll-back of E_2 propagates to E_1 (but roll-back of E_1 does not propagate to E_2). A consequence of this is that if $E_1 \leftrightarrow E_2$, checkpoint and roll-back of either entity propagates to the other.
- A DDG shrinks when a set of dependent entities is checkpointed or reverts to its last stable state (rolls back). Once a checkpoint or roll-back operation is initiated for an entity E, the operation propagates to each entity that is reachable from E in the DDG to which E belongs. Then, because each involved entity is now stable, all edges attached to them are removed.
- At any instant each entity belongs to one and only one dependency graph. To find the set of entities dependent on any entity, it is sufficient to find the location of the entity in its graph and then, subject to the kind of operation, traverse the directed graph starting from the entity. Thus the set of dependent entities may differ for entities in the same DDG.

Dependency Graph	Stabilising Graph	Roll-back Graph
$\rightarrow$	$\overset{S}{\rightarrow}$	$\overset{R}{\leftarrow}$
$\leftarrow$	$\overset{S}{\leftarrow}$	$\overset{R}{\rightarrow}$
$\leftrightarrow$	$\overset{S}{\leftarrow}$ and $\overset{S}{\rightarrow}$	$\overset{R}{\rightarrow}$ and $\overset{R}{\leftarrow}$

Figure 1: The relationship between edges in DDGs, Stabilise Graphs and Roll-back Graphs

With appropriate hardware support, it is possible to lazily construct DDGs by updating them to record dependency data at the completion of each process time slice [7]. This assumes that dependency is recorded at the virtual page rather than the individual object level, thus utilising and extending hardware that is already typically present to support virtual memory management. Conventional virtual memory management requires the presence of status data indicating whether the content of an in-memory page has been modified since the page was loaded (i.e. whether the page is *dirty*), allowing the system to determine whether the page must be flushed to disk before the page frame it occupies can be re-used. In order to efficiently support stabilise and roll-back operations, it is necessary to distinguish between in-memory pages that are unstable and unflushed (DIRTY) and unstable but flushed (MODIFIED). A page would have MODIFIED state if, for instance, as part of virtual memory management it had been loaded, modified, flushed and then reloaded.

Pages may remain in main memory for a period encompassing many process activations. The M_ACCESSED status data allows detection of process access to modified object data during the process' current time-slice. This status data is set for a page if the page is accessed while the MODIFIED status for the page is set. Dependencies between a process and the objects containing pages with the M_ACCESSED status set are represented by the addition of appropriate $\rightarrow$ edges to the dependency graph at the conclusion of the process' period of activation. All M_ACCESSED status data must be clear at the commencement of a process time-slice; this may be achieved in a single operation using appropriate hardware.

The inclusion of WRITTEN status data allows detection of object data modifications made by the current process. This data is distinct from the MODIFIED status described previously because it describes the modification behaviour of the current process rather than the status of the virtual page. The WRITTEN status is set together with the MODIFIED and DIRTY status, but is cleared as part of the dependency graph update at the conclusion of the process time-slice. In contrast the MODIFIED status is cleared at the next object checkpoint and the DIRTY status is cleared when the page is flushed to disk. Pages with the WRITTEN status set cause the inclusion of an appropriate $\leftrightarrow$ dependency graph edge. Operation of the described status data is shown in Figure 2.

3. CONCURRENCY CONTROL

Most descriptions of concurrency control concentrate on the transaction model used for database systems. This model represents one extreme of a spectrum of concurrency control mechanisms ranging from isolation to cooperation. The database transaction model enforces isolation and hides concurrency from the user. At the other extreme concurrency is achieved by cooperation between users. It is not clear which model of concurrency control is most suited to persistent systems. Some researchers [8] regard the cooperative model as the most appropriate, while others [9] prefer to offer a choice of models.

For over twenty years, the transaction has been acknowledged as the central abstraction in preventing concurrent applications from corrupting the contents of a database through errors such as lost update, dirty read or unrepeatable read [10]. The original concurrency control algorithm, strict two-phase locking with shared and exclusive locks, is still widely used in practice, since it is simple to implement and guarantees serializability. Many alternative algorithms have been proposed and, in commercial systems these include variants of key-range locking to avoid phantoms, and escrow reads to improve throughput on hotspot data, as discussed in [11]. New algorithms continue to appear. For example, a constrained shared lock has been proposed in [12]. Besides algorithms which offer alternative implementations for the traditional transaction semantics (ACID properties), there have been many new models proposed, for use in advanced application domains where cooperation is needed between concurrent activities. A detailed survey of these new ideas is found in [13].

Concurrency control techniques ensure that a set of concurrent transactions produce the same result as if they had executed serially, and may be broadly categorised as either optimistic or pessimistic. Pessimistic schemes typically use locks to prevent other concurrent transactions from accessing objects that are being used by the locking transaction. Optimistic schemes [14] proceed without locking but examine the transaction before it is committed, to determine its serialisability, leading to a decision to commit or abort. Both approaches have advantages and disadvantages: the pessimistic approach may cause transactions to deadlock, whilst the optimistic approach may require rollback of transactions after a considerable amount of work has been done.

Status Data Operation	DIRTY	MODIFIED	M_ACCESSED	WRITTEN
Unmodified page retrieved	Cleared	Cleared	Cleared	Cleared
Modified page retrieved	Cleared	Set	Cleared	Cleared
Process reads data from page	Unchanged	Unchanged	Copy *modified*	Unchanged
Process writes to page	Set	Set	Set	Set
End of process time-slice	Unchanged	Unchanged	Cleared	Cleared
Page flushed	Cleared	Unchanged	Unchanged	Unchanged
Object checkpoint	Cleared	Cleared	Unchanged	Unchanged

Figure 2:Effect of operations on object status data.

Optimistic concurrency control was proposed to avoid locking overheads, possibility of deadlock, congestion caused by locking in virtual memory systems and unnecessary acquiring and holding of locks perceived to be problems with pessimistic control schemes. It was proposed [14] that a transaction should be viewed in terms of three phases:

- A read phase, in which all objects read or written by a transaction are copied into a private work area associated with that transaction,
- A validation phase, in which it is established whether applying the modifications achieved during the read phase to the global store would result in loss of integrity of the data in that store, and
- Based on the result of the validation phase, a write phase in which modified objects in the private work area are copied to the global store and made globally visible.

In order to achieve validation every transaction T has an associated Readset(T) comprising objects read by the transaction and Writeset(T) comprising objects modified by the transaction. The transaction manager also records the temporal start of the read phase StartR(T), the end of the read phase FinishR(T) and the end of the write phase (FinishW(T) for every transaction.

A transaction Tj is validated if one of the following conditions is satisfied for every T_i such that $FinishW(T_i) < FinishW(T_j)$:

1. T_i completes its write phase before T_j starts its read phase, i.e. T_i is already serially ahead of T_j because it completed before T_j started.

2. T_i completes its write phase (i.e. $FinishW(T_i)$ occurs) before T_j starts its write phase and $Writeset(T_i)$ does not intersect with $Readset(T_j)$ (i.e. $writeset(T_i) \cap readset(T_j) = \Phi$). In effect, logically T_j might as well have started after T_i completed.

3. T_i completes its read phase before T_j completes its read phase and $Writeset(T_i)$ does not intersect with the union of $Readset(T_j)$ and $Writeset(T_j)$ (i.e. $writeset(T_i) \cap (readset(T_j) \cup writeset(T_j)) = \Phi$). In effect T_j has no unrepeatable reads because of the behaviour of T_i.

It is apparent from the above description that data must be maintained for every committed transaction T_i to enable comparison with $Writeset(T_j)$ as required by the validation phases of temporally overlapping transactions. Additionally, a transaction executes almost to completion (in fact everything is completed except the commit) before inability to commit is detected. This occurs because transactions work in isolation of each other in their work area until the validation phase (and even then a transaction in read-phase works in isolation from other validating transactions).

As shown in Section 4, the DDGs, maintained to support stability, record dependency data that provides an alternate means of achieving optimistic concurrency control, and in some cases signal transactions to abort earlier than would occur with the requirement for a validation phase described above. Further, every transaction that completes to the end of the read phase (as defined above) is guaranteed success in the validation phase, removing the necessity for that phase.

4. SYSTEM SUPPORT FOR CONCURRENCY CONTROL

As described in Section 2, the system records a process' access behaviour during each time quantum, and uses this data to perform a DDG update as part of the overhead of the process switch that occurs at the cessation of the quantum. With the exception of clean read operations (the act of reading an object that has not be modified since it was last stabilised), all object accesses of interest to a transaction manager are recorded by the system as part of the stability mechanism. The — DDG edge provides support for recording of such clean read operations (note the edge is undirected because it does not represent a dependency-creating operation), making it possible to incorporate transaction control into the existing stability system as follows:

- At the commencement of a transaction, the initiating process must exist in a single-node DDG. If that is not the case, the process must initiate a stabilise operation, with isolation being the consequence. The process is then part of a DDG associated by the system with the fledgling transaction.

- As the process (and any parallel processes incorporated in the transaction) interacts with objects in the store, $\rightarrow$, $\leftarrow$, $\leftrightarrow$ and — edges are used to incorporate the entities into the transaction DDG. Construction of the graph is achieved lazily using access data collected as described above during each process time quantum.

- During each transaction DDG update, the system analyses any graph merge operations and determines whether the merge causes a violation of transaction isolation and whether any transaction must be aborted as a result.

- A transaction that completes, i.e. whose DDG could be constructed without a need for transaction roll-back, commits by stabilising its transaction DDG.

It should be noted here that this early roll-back optimistic concurrency control technique updates data in the global store on a real-time basis. There is no private work area associated with each transaction. Data modifications can be rolled back if required as a consequence of the stability technique (for example shadow paging) implemented for the store.

Updates to the DDG at the completion of a (transaction-implementing) process time quantum take the form of insertion of one of the edges $\rightarrow$, $\leftrightarrow$ or — between the graph node representing the process and a data object from the store. The system makes decisions about the effect of inclusion of such an edge in the DDG based on the following:

- Edges have a precedence order —, $\rightarrow$, $\leftrightarrow$ with respect to any process (transaction) – object pair. As a result insertion of an edge to the right in this order will replace an extant edge to the left. An extant edge to the right will not be replaced by an edge to the left, indeed an edge to the left will not be inserted if it occurs after an edge to the right.

- If there are no existing edges between any process node and the object, the appropriate edge is added and the object belongs to (and becomes a node in) the same DDG as the process.

- If all prior edge(s) between other process nodes and the node representing the object are to nodes in the same DDG as the process, the appropriate edge is inserted subject to the precedence rule.

- If one or more edges exist between other process nodes and the node representing the object, and these process nodes do not belong to the same DDG as the process, the system either inserts the appropriate edge or causes transaction abort (roll-back) operation(s) as described below. Transaction roll-back operations are achieved by appropriate DDG roll-backs.

In the following discussion of the effect of DDG edge insertion we assume the existence of an object O_n and transactions T_a, the transaction creating the new edge, and T_b, some other transaction. Decisions on the validity of T_a's edge-producing access are made by considering the edge to be inserted E_a with respect to each individual existing edge between O_n and each other concurrent transaction T_b, as follows (this discussion assumes that the system has already determined that there is no existing edge of higher or equal priority to E_a between O_n and T_a):

(1) If there is no edge between T_a and O_n the new edge is inserted.

(2) If there is an existing — edge between T_b and O_n and the access by T_a was a read, a new — edge is inserted between the node representing O_n and the node representing T_a.

(3) If there is an existing — edge between T_b and O_n and the access by T_a was a write, a policy decision is made. Either a new $\leftrightarrow$ edge is inserted between the node representing O_n and the node representing T_a, and T_b is forced to abort (roll-back), as a consequence removing the — edge, or T_a is forced to roll-back.

(4) If there is an existing $\rightarrow$, edge between T_b and O_n and the access by T_a was a read, a new $\rightarrow$ edge is inserted between the node representing O_n and the node representing T_a. Because the system has already eliminated the possibility of an existing $\leftrightarrow$ edge (i.e. of higher priority) between O_n and T_a, there must also be an existing $\leftrightarrow$ edge between O_n and some other transaction T_b.. Subject to T_b committing before T_a, T_a is allowed to continue.

(5) If there is an existing $\rightarrow$, edge between T_b and O_n and the access by T_a was a write, because the system has already eliminated the possibility of an existing $\leftrightarrow$ edge (i.e. of equal priority) between O_n and T_a, there is an existing $\leftrightarrow$ edge between O_n and some other transaction T_x in which case all transactions with edges to O_n must be aborted (rolled back).

(6) If there is an existing $\leftrightarrow$, edge between T_b and O_n and the access by T_a was a read, a new $\rightarrow$ edge is inserted between the node representing O_n and the node representing T_a. Subject to T_b committing before T_a, T_a is allowed to continue.

(7) If there is an existing $\leftrightarrow$, edge between T_b and O_n and the access by T_a was a write, both transactions must be aborted (rolled back).

A consequence of the described system support is that it is possible to distinguish whether a read operation involving a modified object occurred prior to ('clean') or after ('dirty') the object had been modified. As shown above in points (4) and (6), T_a is permitted to continue after a 'dirty' read subject to the future behaviour of the writing process T_b. Point (3) shows that, on the update, any previously (i.e. 'clean') reading transaction must abort, avoiding the occurrence of unrepeatable read. This is in contrast to conventional optimistic schemes that evaluate ability to commit based on read and write sets with no temporal properties (a 'dirty' read is not possible in conventional schemes). As a result the reading transaction always aborts after the writing transaction commits. A consequence of this ability to read data modified by an as-yet-uncommitted transaction (subject to (4)) allows interleaving of transactions under certain circumstances with a subsequent improvement in concurrency.

The presented concurrency control support allows a scheme that performs as well as conventional concurrency control for read – read situations, and outperforms conventional schemes for write – read situations as discussed in the previous paragraph. For conventional optimistic concurrency control systems, read – write and write – write situations result in ability to commit for the first transaction to achieve that stage, and abort for the second transaction. In this newly-presented scheme, read – write is managed as described in (3), with one of the involved transactions being forced to roll-back and the other continuing. The decision on which option prevails is based on issues such as respective DDG sizes and transaction longevity with effect similar to that of conventional schemes. Write – write is managed as described in (7), which appears to be more draconian than the conventional approach. It should be noted, however, that the conventional approach will certainly abort one of the transactions after all involved have run to completion. The new scheme aborts both as soon as the conflict is detected, potentially reducing the incidence of unproductive work. Subject to granularity of concurrency it may be possible to allow the second writer to continue while aborting the initial writer. This issue, together with others such as measuring the comparative merits of the traditional and new approaches is the subject of further work.

5. CONCLUSION

Stability schemes for persistent stores ensure that data in the store remains consistent even after the store has been shut down in an uncontrolled way after a hardware or system software failure (in contrast to orderly system shutdown that ensures the most recent, possibly in-core, version of data is written to stable disk storage prior to removal of power from core memory). This is achieved by regularly writing snapshots of all or part of the store to disk during system operation in operations called checkpoints. If parts of the store are checkpointed independently, it is necessary to consider the dependencies that are created between entities in the store during processing when scheduling checkpoint operations. The use of Directed Dependency Graphs (DDGs) has been shown to improve the efficiency of stability mechanisms by reducing the cascade effect for checkpoint (stabilise) and roll-back operations.

In this paper it was shown that the use and extension of the DDG data structures already maintained by the system provides support for a novel approach to provision of cooperative concurrency control in persistent systems. The use of a single mechanism to support both stability and concurrent data access removes the duplication present in other systems, thus enhancing system performance. The technique implements a form of optimistic concurrency control that matches or improves on the efficiency of conventional implementations in most situations.

REFERENCES

[1] Atkinson, M. and Morrison, R. "Persistent System Architectures", *Proceedings of the Third International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 73-97, 1989.

[2] Soltis, F. "Inside the AS/400", Duke Press, Loveland, Colorada, 1995.

[3] Henskens, F. and Ashton, M. “Persistent Databases That Perform?”, Proceedings, IASTED International Conference on Software Engineering SE ‘97, IASTED/ACTA Press, ISBN 0-88986-244-3, San Francisco, U.S.A., pp. 168-172, November 1997.

[4] Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. “Stability in a Persistent Store Based on a Large Virtual Memory”, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

[5] Vaughan, F., Basso, T. L., Dearle, A., Marlin, C. and Barter, C. “Casper: a Cached Architecture Supporting Persistence”, *Computing Systems*, 5(3):337-359, 1992.

[6]

[7] Jalili, R. and Henskens, F. A. "Entity Dependency in Stable Distributed Persistent Stores", *Proceedings of the 28th Hawaii International Conference on System Sciences*, vol 2, Hawaii, U.S.A., IEEE, pp. 665-674, 1995.

[8] Henskens, F. A., Koch, D. M., Jalili, R. and Rosenberg, J. "Hardware Support for Stability in a Persistent Architecture", *Workshops in Computing: Proceedings of the Sixth International Workshop on Persistent Operating Systems*, Tarascon, France, ed M. Atkinson, D. Maier and V. Banzaken, Springer-Verlag and British Computer Society, ISBN 3-540-19912-8, pp. 387-399, 1994.

[9] Lindstrom, A. G., "User-level Memory Management and Kernel Persistence in the Grasshopper Operating System", *Ph.D Thesis*, Basser Department of Computer Science, University of Sydney, 1996.

[10] Munro, D. S., "On the Integration of Concurrency, Distribution and Persistence", *Ph.D Thesis*, Department of Mathematical and Computational Sciences, University of St. Andrews, 1993.

[11] K.Eswaran, J. Gray, R. Lorie, I. Traiger, “The Notion of Consistency and Predicate Locks in Database Systems” in *Comm. ACM 19(11): 624-633, November 1976.*

[12] J. Gray and A. Reuter, *Transaction Processing*, Morgan Kaufmann 1993.

[13] D. Agrawal and A. El Abbadi, “Locks with Constrained Sharing” in *Proceeedings ACM PODS: 85-93, April 1991.*

[14] A. Elmagarmid (ed), *Database Transaction Models for Advanced Applications*, Morgan Kauffman 1992.

[15] Kung, H. and Robinson, J. "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, 6(2), pp. 213-226, 1981.