Vei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactio on Computers*, C-37(12):1488–1505, December 1988.

VHBM90] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *Proceedings of the nine ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville*, pag 109–123, 1990.

N88] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proceedir of the International Symposium on Databases in Parallel and distributed Systems*, pages 177–18 1988.

or77] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Databa Systems*, 2(1):91–104, March 1977.

S90] E. Levy and A. Silberschatz. Log-driven backups: A recovery scheme for large memory databa systems. In *The Jerusalem Conference of Information Technology (JCIT 90)*, 1990. Also availal as technical report TR-89-24, Computer Sciences department, The University of Texas at Austir

HL^{+}90] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recove method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Tec nical Report RJ 6649 (63960), IBM Research, February 1990. A revised vesion. To appear in AC Transactions on Database Systems.

LC87] J. E. B. Moss, B. Leban, and P. K. Chrysanthis. Finer grained concurrency for the database cache. *Proceedings of the Third International Conference on Data Engineering, Los Angeles*, pages 96–1(February 1987.

oh87] C. Mohan. Directions in system architectures for high transaction rates. In *Proceedings of AC SIGMOD 1987 International Conference on Management of Data, San Francisco*, pages 6–7, 198 A panel session, whose participants were Mohan, C. (chairperson), Gawlick, D., Gray, J., Klein, Lassettre, E., and Neches, P.

oh90] C. Mohan. Commit-LSN: A novel and simple method for reducing locking and latching in tra action processing systems. In *Proceedings of the Sixteenth International Conference on Very Lar Databases, Brisbane*, 1990. A vesion available as IBM research report RJ 7344.

oh91] C. Mohan. A cost-effective method for providing improved data availability during DBMS resta recovery after a failure. Unpublished manuscript, 1991.

P91] C. Mohan and H. Pirahesh. ARIES-RRH: restricted repeating of history in the ARIES transacti recovery method. In *Proceedings of the Seventh International Conference on Data Engineerin Kobe, Japan*, April 1991.

GK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive dis (RAID). In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Da Chicago*, pages 109–116, 1988.

u86] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, (1):271–28 1986.

ap75] R. L. Rappaport. File structure design to facilitate on-line instantaneous updating. In *Proceedir of ACM-SIGMOD 1975 International Conference on Management of Data, San Jose*, pages 1– 1975.

eu84] A. Reuter. Performance analysis of recovery. *ACM Transactions on Database Systems*, 9(4):526–5 December 1984.

GM87a] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. Technical Repo CS-TR-126-87, Princeton University, Computer Science Department, 1987.

GM87b] K. Salem and H. Garcia-Molina. Crash recovery for memory-resident databases. Technical Repo CS-TR-119-87, Princeton University, Computer Science Department, 1987.

an87] Tandem Computers Corporation. *Remote Duplicate Facility (RDF), System Management Manu* March 1987.

SW88] C. Beeri, H.-J. Schek, and G. Weikum. Multi-level transaction management, theoretical art practical need? In *International Conference on Extending Database Technology, Lecture Notes Computer Science*, volume 303. Springer Verlag, 1988.

BDU75] K. M. Chandy, J. C. Brown, C. W. Dissly, and W. R. Uhrig. Analytic models for rollback a recovery strategies in database systems. *IEEE Transactions on Software Engineering*, SE-1(1):10 110, March 1975.

KKS89] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *Proceedings the Fifteenth International Conference on Very Large Databases, Amsterdam*, pages 327–336, 19

KO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, and M. R. Stonebraker. Implementati techniques for main memory database systems. In *Proceedings of ACM-SIGMOD 1984 Internatio Conference on Management of Data, Boston*, pages 1–8, 1984.

B84] K. Elhard and R. Bayer. A database cache for high performance and fast restart in database systen *ACM Transactions on Database Systems*, 9(4):503–525, December 1984.

ic86] M. Eich. Main memory database recovery. In *1986 Proceedings ACM-IEEE Fall Joint Compu Conference, Dallas*, pages 1226–1232, 1986.

ic87] M. Eich. A classification and comparison of main memory database recovery techniques. In *P ceedings of the Third International Conference on Data Engineering, Los Angeles*, pages 332–3 1987.

+81] J. N. Gray et al. The recovery manager of the system R database manager. *ACM Computi Surveys*, 13(2):223–242, 1981.

MS90] H. Garcia-Molina and K. Salem. System M: A transaction processing testbed for memory reside data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, 1990.

ra78] J. N. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science, Operat Systems: An Advanced Course*, volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

ag86] R. B. Hagmann. A crash recovery scheme for memory-resident database system. *IEEE Transacti on Computers*, C-35(9):839–843, September 1986.

R83] T. Haerder and A. Reuter. Principles of transaction oriented database recovery — a taxonon *ACM Computing Surveys*, 15(4):289–317, December 1983.

GMHP88] R. P. King, H. Garcia-Molina, N. Halim, and C. Polyzois. Management of a remote backup co for disaster recovery. Technical Report CS-TR-198-88, Princeton University, Computer Scien Department, 1988.

am81] B. W. Lampson. Atomic transactions. In *Lecture Notes in Computer Science, Distributed Syste — Architecture and Implementation: An Advanced Course*, pages 246–265. Springer-Verlag, Berl 1981.

C87] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-reside database system. In *Proceedings of ACM-SIGMOD 1987 International Conference on Manageme of Data, San Francisco*, pages 104–117, 1987.

ev91] E. Levy. Incremental restart. In *Proceedings of the Seventh International Conference on D Engineering, Kobe, Japan*, April 1991.

n80] B. G. Lindsay. Single and multi-site recovery facilities. In *Distributed Databases*, chapter 10, pag 247–284. Cambridge University Press, Cambridge, U.K., 1980. Also available as IBM Resear Report RJ2571, San Juse, July, 1979.

can be adopted for such purposes, but this deserves separate attention. On the other hand, since t log-driven design is predicated on a partial-residence assumption, it can accommodate partially-reside databases efficiently by enforcing rules *Safe-Fetch*, and *Single-Propagation*.

e above comparison favors the log-driven approach. Among the rest, fuzzy algorithms seem to be close co titors. We note that fuzzy algorithms stand out (considering CPU overhead during normal operation) accordi the performance evaluation studies of Salem and Garcia-Molina [SGM87b].

We should note that other methods that are log-driven in spirit can be found in [Eic86] and [LN88]. It eresting to note that in [Eic86], log records of a transaction are marked after the transaction has committe that only log records of committed transactions would affect the BDB. It should also be mentioned that a lo iven approach is often used to manage remote backups for disaster recovery purposes (e.g., [KGMHP88, Tan87

0 Conclusions

e increasing size of contemporary databases, and the availability of stable memory and very large physic emories are bound to impact the requirements from, and the design of recovery components. In particular, f eckpointing and restart processing, the traditional approach becomes inappropriate for high rates of transactio d very large databases. An incremental approach, that exploits the new technological advances, is a natu ution. In this paper we described in a high-level manner such a solution.

The main thrust of this paper is the design of recovery techniques in a manner that would allow their i leaving with normal transaction processing. The techniques exploit stable memory and are geared to me e demands of systems that incorporate large main memories. We have proposed both restart algorithm (call *cremental restart*) and a checkpointing-like technique (called *log-driven backups*) that operate in an *incremen* anner, *in parallel* with transaction processing. The prominent original concepts motivating our design are lows:

- Associating restoration activities with individual data objects, and assigning priorities to these activit according to the demand for these objects. Consequently, recovery processing is interleaved with norm transaction processing. By contrast, the conventional restart procedure for example, treats the database a single monolithic data object, and enables resumed transaction processing only after its termination.

- A direct consequence of the previous point is the grouping of recovery-related information (e.g., log recor on data objects basis. This structuring is aimed to facilitate the efficient restoration of individual da objects.

- Carrying out recovery processing and transaction execution in parallel implies decoupling the respecti resources to reduce contention as much as possible. In the log-driven backups technique both data a processing resources for checkpointing are separate from the resources required for forward transacti processing.

eferences

HG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databa Systems*. Addison-Wesley, Reading, MA, 1987.

it86] D. Bitton. The effect of large main memory on database systems. In *Proceedings of ACM-SIGM 1986 International Conference on Management of Data, Washington*, pages 337–339, 1986. A pa session, whose participants were Bitton, D. (chairperson), Garcia-Molina, H., Gawlick, D., a Lomet, D.

R87] B. R. Badrinath and K. Ramamritham. Semantic-based concurrency control: Beyond commu tivity. In *Proceedings of the Third International Conference on Data Engineering, Los Angel* 1987.

Delaying restart activities was first described in [Rap75]. There, restart does not perform any recovery activi
stead, reading a data item triggers a validity check that finds the committed version of the data item that shou
read. The incremental restart procedure we propose resembles this early work in that data items are recover
ly once they are read.

A more conventional approach to speeding up restart is proposed in [MP91] in the context of the ARI
nsaction processing method. The idea there is to shorten the redo pass of conventional restart by performi
lective redo. Instead of repeating the history by redoing all the actions specified in the log, only those actic
ecified in winner log records are redone. It is also mentioned there that undo of loser transactions can
erleaved with the processing of new transactions if locks (similar to RS-locks) protect the uncommitted da
ms updated by the loser transactions. During the analysis pass of restart, the identity of these data items
scovered, whereas in our scheme such data items are already marked as stale.

The concept of deferred restart (which is similar to incremental restart) is discussed in [MHL$^+$90] also in t
ntext of ARIES. It is mentioned that in IBM's DB2 redo/undo for objects that are off-line can be deferre
ie system remembers the LSN ranges for that objects and makes sure that they are recovered once they a
ought on-line and before they are made accessible to other transactions. DB2 employs physical, page-le
gging. Problems related to logical undoing and deferred restart are also discussed in [MHL$^+$90]. Our wo
fers from the ARIES work in exploiting stable memory and as it presents a simple algorithmic description
fundamentals of incremental restart in the context of both physical and operation logging.

Another noteworthy approach to fast restart is the Database Cache [EB84]. There, dirty pages of acti
nsactions are never flushed to the backup database. At restart, the committed state is constructed immediat
loading the recently committed pages from a log device (called safe there). The main disadvantages
is approach are that locking is supported only at the granularity of pages, full-page physical logging is us
contrast to our entry logging, update-intensive transactions need to be treated specially, and that comn
ocessing includes a synchronous I/O. The DB cache idea is refined to accommodate finer granularity locki
[MLC87], however this extension does not deal with operation logging and concurrency among semantica
mpatible operations.

Work on improving restart processing is reported in [Moh91]. The approach there is to adapt the pass
traditional restart and admit new transactions during these passes. Also, associating freshness status wi
committed pages is discussed there and in [Moh90].

A thorough survey of different MMDBS checkpointing policies, their impact on overall recovery issues, a
eir performance can be found in [SGM87b].

Next, we compare our log-driven backups scheme with several variations of MMDBS checkpointing (e.
KO$^+$84, Pu86, Hag86]).

- Checkpointing interferes in one way or another with transaction processing, since both activities compe for the PDB and the main CPU. Taking a consistent checkpoint requires bringing transaction activity a quiescent state, since a transaction-consistent checkpoint reflects a state of the database as produced completed transactions. In the extreme case, transactions have to be aborted to guarantee the consisten of the checkpoint [Pu86]. Even in fuzzy algorithms, which do not produce consistent checkpoints [Hag8 memory contention is inevitable since both normal transactions and the checkpointer must access the ve same memory. By contrast, in the log-driven backups scheme, transaction processing and propagation to t BDB do not use the same memory and may use different processors. This separation is the key advanta of the scheme.

- It has been observed in [SGM87b] that consistent checkpoints must be supported by two copies of t database on secondary storage, since there is no guarantee that the entire checkpoint will be atomic. Mc precisely, there is always one consistent checkpoint of the entire database on secondary storage that w created by the penultimate checkpoint run, while the current run creates a new checkpoint. This proble does not arise in the log-driven backups technique since the propagation to the BDB is continuous and n periodic.

- It is not clear how checkpointing algorithms can be adjusted to support our assumption of a partially reside database. The correctness of these algorithms may be jeopardized by arbitrary fetching and flushing database pages. It seems that fuzzy checkpointing, which is the simplest type of checkpointing algorith

HL^{+}90, Gra78]. In our case, since updates are not propagated before the commit of an operation, the WA le means that the high-level undo record should be written to the high-level log prior to the commit point e corresponding operation.

By structuring the high-level recovery on top of our incremental restart method, we intend to give the over covery scheme incremental flavor. The major challenge in making this multi-level recovery scheme incremen the fact that we can no longer treat single pages as the individual unit for recovery, since operations affe veral pages. Had we used single pages, we would have violated the high-level action atomicity requireme entioned above. For this reason we devise the notion of a *recovery unit (RU)*. An RU is a set of pages, such th is not possible for any high-level operation to affect more than one RU. For instance, if an INSERT operati used for updating both index and data files, then the index and the corresponding data file constitute an R is the responsibility of the base recovery to bring an RU to an operation-consistent state before any high-le do can be applied to it.

When a post-crash transaction requests to access an RU, the incremental restart algorithm is applied to all t ges of that RU. Once this phase is completed, the RU is in an operation-consistent state. Then, the high-le covery brings the RU to its committed state by applying the high-level undo operations for loser transactio the reverse order of the appearance of the corresponding log records. To facilitate fast restoration of individu Js, high-level log records should be grouped on an RU basis on the high-level log (see [Lev91] for techniques f ouping log records). A high-level undo operation is treated as a regular operation, keeping both base and hig el logging in effect. Care should be taken to undo only operations whose effect actually appears in the back tabase (the high-level action idempotence requirement of [WHBM90]). Therefore, the base recovery passes e high-level recovery an indication which of the operations of loser transactions were winner operations, a nce were redone, in the base recovery phase.

By partitioning the database to RUs the incremental effect is obtained. RUs can be of coarse granulari ereby diminishing the benefits of incremental restart. For example, an entire relation and the correspondi dex structure must be recovered before a post-crash transaction may read any of the tuples. This observati lls for as small RUs as possible.

Example 2. Consider again the three transactions of Example 1. This time, however, T_{11} and T_{12} a gh-level operations (subtransactions of T_1), and T_{21} is the sole operation of T_2. The same sequence of events ed. The stale/fresh marking of a, b and c, and the winner/loser status of the operations remain as in Examp n this execution. Pages a, b and c constitute an RU, and the high-level log for that RU is as follows (present the logged undo information for operation T_{ij} as $undo(T_{ij})$):

$$undo(T_{11}),\ undo(T_{21}),\ c_2,\ undo(T_{12})$$

terms of transactions, T_1 is a loser, whereas T_2 is a winner. Base recovery for the three pages takes pla actly as in Example 1 (i.e., only page a is recovered). In the high-level recovery phase, only T_{11} is undone, sin $_2$ was a loser in the base-level.

The presented scheme is not efficient mainly since it performs excessive log I/O while committing the hig el operations. A more efficient version of the scheme would probably employ the improvements outlined e second approach in [WHBM90]. The goal of presenting the above scheme was only to demonstrate ho cremental restart can be used as the base for a more complex and higher-level recovery, using the modul lti-level model of [WHBM90].

Related Work

e work reported in this paper is a continuation of our earlier work in this area [Lev91, LS90]. A gene le/fresh marking algorithm that is not based on no-steal buffer management is presented in [Lev91].

A proposal for incremental restart is presented in [LC87] in the context of a main-memory database (MMD able memory is used extensively to implement this approach. There are several aspects that distinguish o rk from the work on [LC87]. Some aspects there are peculiar to entirely-resident MMDBs. Namely, there is nsideration of paging activity. Integrating full-fledged operation logging is not discussed in [LC87] at all. Al le/fresh partition and the improvements it entails are lacking from the work in [LC87].

Incremental Recovery for High-Level Recovery Management

common way of enhancing concurrency is the use of semantically-rich operations instead of the more primiti **ad** and **write** operations. Having semantically-rich operations allows refining the notion of conflicting vers mmutative operations [BR87, Wei88]. It is possible to examine whether two operations commute (i.e., t conflict); such operations have the nice property that they can be executed concurrently. Semantics-bas ncurrency control is often cited as a very attractive method for handling high contention to data (i.e., 'h ots') [MHL$^+$90, BR87, Wei88]. The problem, however, is that the simple state-based (i.e., physical) recove ethods no longer work correctly in conjunction with these operations. Only *operation logging*, referred to al logical-transition logging [HR83], can support this type of enhanced concurrency. For instance, consider t *crement* and *decrement* operations which commute with each other and among themselves. A data item can cremented concurrently by two uncommitted transactions. If one of the transactions aborts, its effect can done by decrementing the item appropriately. However, reverting to the before image may erase the effects e second transaction also, resulting in an inconsistent state.

One of the problems of using operation logging is that the logged high-level operations may be implement a set of lower-level operations, and hence their atomicity is not guaranteed. Therefore, when logged operatio e undone or redone after a crash, they should not be applied to a backup database that reflects partial effec operations. Therefore, a key assumption in any operation logging scheme is that operations must appear ough they were executed atomically. This requirement is a prerequisite to any correct application of operati g records to the BDB at restart time, and is referred to as *high-level action atomicity* in [WHBM90]. As ustration, we mention System R [G$^+$81] which employs operation logging. There, at all times, the BDB is in *eration-consistent state* — a state that reflects the effects of only completed operations, and no partial effec operations. This property is obtained by updating the BDB atomically, and only at checkpoint time, usi shadowing technique [Lor77]. At restart, the operation log is applied to the consistent shadow version of t tabase.

The problem of implementing operation logging is best viewed as a multi-level recovery problem. A ve egant and simple model of (standard, non-incremental) multi-level recovery is introduced in [WHBM90]. at follows, we make use of that model to construct an incremental multi-level recovery scheme.

A transaction consists of several high-level operations. A high-level operation is defined over fine-granulari ms (e.g, tuples, records), and is implemented by several base-level primitives that collectively may affect mo an a single page. The base level primitives are **read** and **write** that affect single pages—primitives that a nsistent with our page-level model.

In other words, transactions are nested in two levels. Serializability of transactions is enforced by a multi-lev ncurrency control that uses strict two-phase locking at each level [BSW88].

Recovery is also structured in two levels. Our page-based incremental method constitutes the base recove ensures persistence and atomicity of higher-level operations and not of complete transactions. That is, t gh-level operations are regarded as transactions as far as the base recovery module is concerned. Persistence committed transaction is obtained as a by-product of the persistence of its operations (i.e., if all operations o nsaction have committed, then the transaction itself has committed). Observe that both the log-driven backu d marking algorithms refer to operations rather than transactions in the current context. Any occurrence o nsaction there should be substituted with an operation.

We still require that dirty pages are not flushed unless the operation that updated them is committed (i. -steal policy with respect to operations is enforced). This is not a major restriction since operations update nall number of pages. Imposing this restriction also helps avoiding the extra overhead due to the hierarchic yering. Consequently, the log of the base recovery, called the base log, is a redo log and there is no need rform base-level undo at restart.

The high-level recovery is based on operation logging and it guarantees atomicity of complete transactio e high-level log is separate from the base log and it holds only high-level undo information. The high-lev ndo log does not participate in the log-driven backups flow, and may, in fact, be implemented as a traditional l disk. The overall plan is to use the base recovery to redo committed transactions and committed operatio ereby bringing the BDB to an operation-consistent state, and then apply high-level undo in order to undo t erations of loser transactions.

Since the high-level log deals with Undo log records, it should obey the Write–Ahead–Log (WAL) ru

requested held	R	W
R		X
W	X	X
RS	X	

Figure 2: Lock compatibility matrix.

owed even greater flexibility. Indeed, stale pages cannot be read by post-crash transactions; however, writi ta items in a stale page is possible.

One way to view this improvement is to consider a new type of locks, called *restart locks*, that lock all sta ges, and no other pages, after a crash. An imaginary restart transaction acquires these locks as soon as t stem is rebooted and before post-crash transactions are processed. In Figure 2, we present the lock compatibil atrix for the three lock modes **read (R), write (W)**, and **restart (RS)**. Since restart locks are not requeste t rather are held by convention by the restart transaction, the compatibility matrix lacks the request colur the new lock type. An entry with "X" in this table, means that the corresponding locks are incompatible.

Observe that restart locking does not interfere with the normal concurrency control. This can be shown serving that the imaginary restart transaction is a two-phased transaction that is serialized before any po ash transaction that attempts to access a stale page. Also, restart locking cannot introduce deadlocks, sin e restart transaction is granted the **RS** locks on all the stale-marked pages unconditionally at reboot time.

An **RS** lock held on a stale page x is released when the page is brought up-to-date. This happens only wh is explicitly brought up-to-date by the incremental restart procedure, by applying log records to the back age.

A **write** of a stale page results in an update log record containing only the after image of the update, sin e page has not been recovered yet. Such a log record will actually affect the relevant page once the page covered and brought up-to-date (unless the transaction that generated the record aborts).

In summary, the above protocol allows post-crash transactions to be processed concurrently with the inc ental restart processing. Some transactions are scheduled without being delayed by the recovery activity at d some are delayed only as a result of recovering data items they need.

2 Further Improvements

this subsection, we briefly mention several points that can further improve an implementation of the incremen start algorithm.

- RS-locking can be used to combine incremental and standard restart for different sets of pages, there avoiding the need to maintain stale/fresh marking for too many pages. The set of pages that are recover using standard restart should be RS-locked until they are made consistent. Only predicted 'hot spot' da can be supported by incremental restart (and the stale/fresh marking). This improvement allows a ve attractive and flexible use of incremental restart even in very large databases.

- Background process(es) can recover the remaining portions of the database, while priority process(es) recov pages demanded by executing transactions. Once a page is recovered and made consistent, the RS lock c be released. This technique provides even greater concurrency between restart and transaction processin

- It is not necessary to log restart activities in order to guarantee its idempotence. It is advised, though, flush previously stale pages that are made up-to-date, thereby marking them fresh. Doing this will sa recovery efforts in case of repeated failures.

- Assuming a very large number of pages for which stale/fresh marking is managed using a sophisticat data structure, updating the marking data structure can become a bottleneck. A queue in stable memo that records recent updates to the marking can prevent this undesirable phenomenon. Applying the queu updates to the actual marking data structure can take place whenever the CPU is not heavily loaded.

Correctness Aspects

e prove two claims that underlie the correctness of our integrated architecture. The correctness of the marki gorithm is stated concisely by the hypothesis of Lemma 1 below:

Lemma 1. *At all times, in particular following a crash, if a page x is stale then $x.stale$ holds. Formal* $x :$ $(backup[x] \neq committed[x]) \Rightarrow x.stale)$ *is invariant.*

Proof. Consider the state space formed by the variables we have introduced. We model the executi transactions and fetching and flushing of pages, as transitions over that state space. We prove the claim owing that the invariant holds initially and that it is preserved by each of these transitions.

Assuming that initially all pages are fresh, the invariant holds vacuously when the algorithm starts. Flus g a page is modeled as an assignment to $backup[x]$, and committing a page is modeled as an assignment mmitted$[x]$. There are four state transitions that may affect the validity of the invariant: an execution e assignment statement specified in one of the rules *Dirty–Stale, Flush–Fresh*, the commitment of an updati nsaction, and the flushing of a page. We prove that the invariant holds by showing that each of these st nsitions preserves the invariant:

- **Rule Dirty–Stale**: Under no circumstances setting $x.stale$ to true can violate the invariant.
- **Commit of T**: Consider an arbitrary page x updated by the just committed transaction T (i.e., x $T.writeset$). Since a strict concurrency protocol is employed at a page level, we are assured that no oth transaction has updated x subsequently to T's update and before T's commitment. If x is dirty, th T's commitment renders it stale. However, since the assignment in *Dirty–Stale* is executed prior to t commitment of T, $x.stale$ holds, and the invariant still holds.
- **Flushing** x: According to our assumptions regarding buffer management policy, flushing a page x alwa renders it fresh (since only committed pages are flushed). Therefore, the invariant holds vacuously.
- **Rule Flush–Fresh**: Since this rule's execution follows immediately the flushing of x, x is fresh after t flush, and hence falsifying $x.stale$ preserves the invariant.

us, the invariant holds.

It should be realized that if $x.stale$ holds it does not necessarily mean that x is indeed stale, however t nverse implication does hold, as stated in Lemma 1. Hence, notice that $x.stale$ and "x is stale" are n erchangeable.

Lemma 2. *For all pages x, if x is not in the PDB, then x is fresh. Formally:* $(\forall x : \; x \notin PDB$ $ackup[x] = committed[x]))$.

Proof. A backup page can be updated by either the buffer manager or the propagator. If a page is n the PDB the propagator does not update it because of the *Safe–Fetch* rule. Regarding the buffer manag shing a page is allowed only if the page is committed. Therefore, all pages that are not in the PDB are fres

Improvements

this section we present several possible enhancements and refinements to the techniques we have present rlier.

1 Improving Restart Processing

ing the fresh/stale marking post-crash transactions can access fresh pages as soon as the system is up. tempt to access a stale page triggers the recovery of that individual page. The transaction that requested t cess is delayed until the page is recovered. Interestingly, aided by the marking, post-crash transactions can

1. Each page x is assigned a boolean variable $x.stale$ that is used for the stale/fresh marking. This set variables is the only data structure that is maintained in stable memory. All other data structures are ke in volatile memory and are lost in a crash. We stress that the boolean variables are introduced only present the algorithm, and we do not intend to implement them directly.

2. Each transaction T is associated with a set, $T.writeset$, that accumulates the IDs of the pages it modifi

The algorithm is given by the following two rules, each of which includes assignment that is coupled with t mporal event that triggers it:

- **Dirty & Stale**. Prior to the commit point of T: **if** $(x \in T.writeset \wedge x.dirty)$ **then** $x.stale := true$
- **Flush & Fresh**. After flushing a dirty page x: $x.stale := false$

ı assignment and its triggering event need not be executed as an atomic action. All that is required is th events that affect the variables we have introduced occur between the triggering event and the correspondi signment. The key idea in the algorithm is to always set $x.stale$ to true just prior to the event that actua uses x to become stale. As a consequence, a situation where $x.stale$ holds but x is still fresh is possib kewise, falsifying $x.stale$ is always done just following the event that causes x to become fresh.

We illustrate the marking scheme with the following example.

Example 1. Consider the following three transactions[1] that read and write (R/W) the pages a, b and c.

$$\begin{aligned} T_{11} &= R/W(a), R/W(b) \\ T_{12} &= R/W(a), R/W(b) \\ T_{21} &= R/W(c), R/W(a) \end{aligned}$$

ıe following sequence lists **write** operations of T_{ij} on page x ($w_{ij}(x)$), commit points of of T_{ij} (c_{ij}), and pa shes ($flush(x)$) in their order of occurrence in a certain execution that is interrupted by a crash:

$$w_{11}(a),\ w_{11}(b),\ c_{11},\ flush(b),\ w_{21}(c),\ w_{21}(a),\ c_{21},\ flush(c),\ w_{12}(a),\ w_{12}(b),\ CRASH$$

ter the crash, $a.stale$ holds (by *Dirty–Stale* prior to c_{21}), $b.stale$ does not hold (by *Flush–Fresh* after $flush(b$ d $c.stale$ also does not hold (by *Flush-Fresh* after $flush(c)$). Note that T_{11} and T_{21} are committed whereas T s to be aborted. We say that T_{11} and T_{21} are winner transactions, whereas T_{12} is a loser transaction. Usi e marking, only the updates of the winner transactions to page a need to be redone, since only a is mark ıle.

3 The Integrated Architecture

summarize the integrated architecture we list the five components we have introduced and their correspondi ıctionality. We refer the reader to Figure 1 for a schematic description of this architecture.

- **Buffer manager**: Enforces no-steal policy.
- **Accumulator**: Operates entirely within the stable memory. Accumulates log records as they are produc by transactions and forwards log records of committed transactions. In order to amortize page I/O, t accumulator groups log records that belongs to the same page together, so that the propagator will app them all in a single I/O.
- **Propagator**: Applies page-updates to BDB based on Redo log records.
- **Logger**: Writes Redo log records to the log on disk
- **Marker**: Reacts to page flushes by the buffer manager and BDB updates by the propagator and maintai the fresh/stale marking in stable memory.

[1] We use double subscripts for transactions since the same example is used again in the context of subtransactions in Section 8

this case, we are assured that all the updates have been applied to the BDB already (by the propagator) a
ere is no need to flush the page.

Implementing *Single–Propagation* can be very effective in large memory systems, where we assume that pagi
tivity is quite rare. By the time a page needs to be flushed to the BDB, it is quite possible that all t
evant updates have been propagated to the BDB by the propagator. We emphasize that incorporating *Sing*
opagation is only for performance reasons, and has nothing to do with correctness. By enforcing *Safe-Fetch* a
ngle-Propagation, the combination of propagator updates and page flushes as means for update propagation
ade optimal.

The log-driven backups technique ensures that the gap between the committed and backup images of t
tabase is not too wide. The technique is well-suited to MMDBs where most of the time all the accesses a
isfied by the PDB.

2 Stale/Fresh Marking

e goal of the marking technique is to enable very fast restart after a crash. The key observation is th
nsaction processing can be resumed immediately as the system is up, provided that access to stale pages
nied until these pages are recovered and brought up-to-date. An attempt by a transaction T to access a pa
riggers the following algorithm:

```
if $x$ is stale then begin
        fetch the backup image of $x$;
        Retrieve all the relevant log records for $x$ from the log;
        Apply these log records to $x$'s image in order to make $x$ up-to-date;
    end
Let $T$ access $x$
```

To support this approach to restart, a stale/fresh marking that indicates which pages are (potentially) st
eds to be implemented. The updates needed to bring a stale page up-to-date are always Redo updates becau
our assumptions. The log records with the missing updates can be found either in the log tail or on the log di
cording to the trade-off presented earlier regarding the timing of discarding a log record from stable memo
[Lev91] we elaborate on how to support efficient retrieval of the needed log records from a disk.

The stale/fresh marking of data pages is the crux of the algorithm. The marking enables resuming transa
n processing immediately after a crash, while preserving the consistency of the database. Typically, the l
ores enough information to deduce the stale/fresh status of pages. However, this information is not availa
mediately. The marking also controls the recovery of data pages one by one according to the transactio
mands. In order for the algorithm to be practical, it is critical to both maintain the stale/fresh marking
ain memory, as well as have it survive a crash. Therefore, we underline the decision to *maintain the stale/fr*
rking in stable memory. We do not elaborate on how to manage the marking efficiently. However, in light
scale of current databases, an appropriate data structure holding page IDs that supports efficiently inser
letes, and searches is deemed crucial. Observe that the functions of the analysis pass [MHL$^+$90] in standa
start procedures are captured by the stale/fresh marking, and are ready for use by restart without the need
alyze the log first.

The partition of the set of the backup pages into a set of stale pages and a set of fresh ones varies dynamica
transaction processing progresses. There are two events that trigger transitions in that partition:

- the commit event of an updating transaction, and
- the updates to BDB pages by either the buffer manager or the propagator.

hen a transaction commits, its dirty pages become stale since they were not written to the BDB (see ru
rty–Stale below). When flushing occurs, the transitions depend on whether the page is committed or n
nce we enforce the no-steal policy, we consider only flushing a committed page — an event that makes the pa
sh (see rule Flush–Fresh below).

Based on the above transitions we present a reactive algorithm that manages a stale/fresh marking of pag
indicate whether they are stale or fresh. In order to present the algorithm formally, we introduce the followi
riables and conventions:

The pipeline of log records can be efficiently mapped onto a multi-processor shared-memory architecture. rticular, the propagator and the logger tasks can be carried out by dedicated processors. This way, recover ated I/O is divorced from the main processor that executes the transactions processing activity.

The timing of discarding log records from the stable memory presents a trade-off. A log record may scarded only after it is written to the log disk by the logger. However, such an early discarding implies th the record has not yet been processed by the propagator, then its update will not be reflected in the BI nce it skipped the propagator processing stage). The propagator can fetch the missing records from the di g but this would really delay the propagation. Alternatively, the pages whose updates where skipped by t opagator can be marked stale (see below on how the marking is managed), thereby postponing handling of t ssing updates to a later time. These difficulties can be avoided when log records are not discarded from stal emory before they have been processed by the propagator. However, the trade-off arises as it is anticipat at the propagator would lag behind the logger because the former performs random access I/O whereas t tter performs sequential I/O. In [LS90] we analyze this trade-off and propose to use a RAID I/O architectu GK88] for the propagator in order to balance the I/O load between the logger and the propagator.

Independently from the log-driven activity, database pages are exchanged between the buffer and the BD dictated by the demands of the executing transactions. The *Buffer Manager* is in charge of this exchange. W nphasize that the buffer manager flushes only pages that reflect updates of already committed transactions e no-steal policy. Observe that the principle of Redo-Only BDB is implemented by both sources of updates e BDB; the buffer manager as well as the propagator.

Conceptually, the scheme could have been designed without flushing database pages at all. That is, propagati dates by the propagator would have been the sole mechanism for keeping the BDB up-to-date. The proble th such an approach is that page fetching must be delayed until the most recent committed values are appli the propagator. Such a delay of transaction processing is intolerable. Since only committed database pages a shed (no-steal buffer management), flushing can serve as a very effective means for keeping the BDB up-to-da

The fact that the BDB is updated by both the propagator and by flushing buffered pages must be consider th care. First, one should wonder whether these double updates do not interfere with the correctness of th heme. Second, since two identical updates are redundant, one of them should be avoided for performance reason egarding correctness, a problem arises when the propagator writes an older image of a page, overwriting t ost up-to-date image that was written when the page was previously flushed by the buffer manager. If the pa fetched before the up-to-date images are written to the BDB by the propagator, transactions read inconsiste ta. The problem can be solved by imposing the following *Safe–Fetch* rule:

> *The propagator applies updates only to database pages that are in the PDB. Updates pertaining to pages that are not in the PDB are ignored by the propagator.*

otice that because of this rule, a page that is fetched from the BDB was last modified when it was flushed e buffer manager. Therefore, the page is up-to-date when it is fetched to the PDB. The rule is referred to *fe–Fetch* since it ensures that a page fetched from the BDB is always up-to-date (except for following a crasl

Implementing *Safe–Fetch* implies that the propagator should know which pages are in the PDB. We assur at the propagator initially knows which pages are in the PDB, and it is notified about each page replacement e buffer manager. We assume that the propagator and the buffer manager share some memory for this purpo ternatively, since a single I/O controller serves I/O requests of both the propagator and the buffer manag forcing *Safe-Fetch* can be implemented by a smart controller. In any case, since page flushes are assumed to frequent, implementing this rule should not incur too much of an overhead.

Besides the correctness aspect, *Safe-Fetch* enables the heavily loaded propagator to avoid processing sor g records. *Safe-Fetch* deals with cases where a page was flushed to the BDB before the corresponding updat re applied to the BDB by the propagator. I/O activity can be reduced considering the opposite case too, posing the following *Single–Propagation* rule:

> *When all of the log records corresponding to a page have been applied to the BDB by the propagator, flushing that page to the BDB is useless. In such a case, the buffer manager can simply discard the page without issuing a flush to the BDB.*

his rule can be easily implemented using the log-sequence-number (LSN) mechanism [MHL$^+$90, Gra78]. Flushi the page can be avoided if the page's LSN is at most the LSN of the page that was last written by the propagat

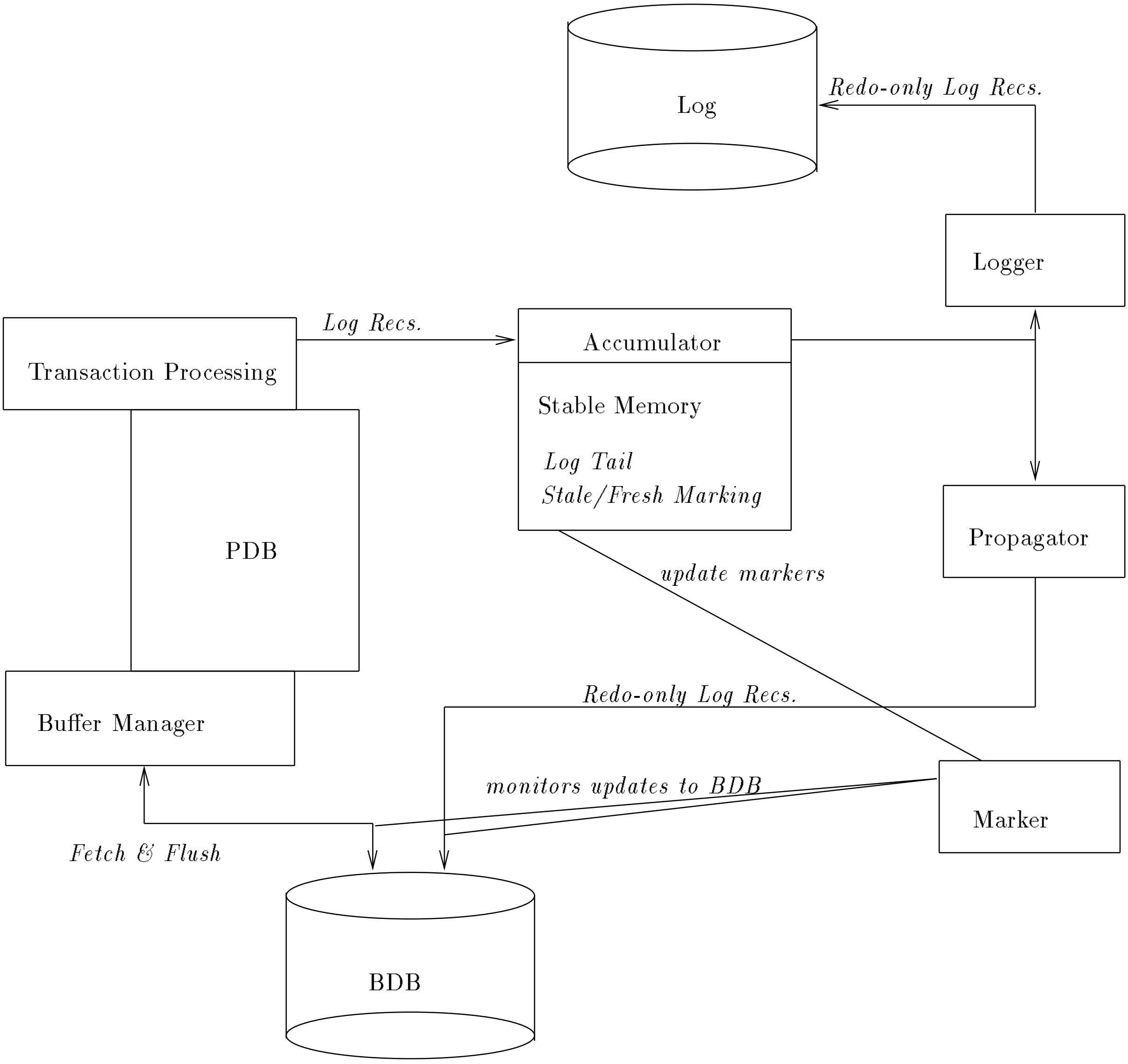

Figure 1: A Schematic View of the Architecture

We incorporate the above principles in the proposed architecture. We do not assume an entirely-reside MDB, in the spirit of the first principle. Consequently, we deal with buffer management issues. The seco nciple is enforced by insisting on using the no-steal buffer management policy. Namely, only updates mmitted transactions are propagated to the BDB. This is an explicit assumption of our design.

The preservation of the third principle is the crux of the problem. Fortunately, stable memory is the technolo at enables promoting this principle. In the architecture we propose, the log tail is stored in stable memo ommitting a transaction, thereby making its updates persistent, is guaranteed by writing the commit log reco the *log tail in stable memory*. Any further recovery activity is totally separated from transaction processi e emphasize that in the architecture we propose the log tail is kept in stable memory (i.e., non-volatile RA making the fast stable memory the only point of friction between transaction and recovery processing hieve the goal of decoupling the two as much as possible.

The Incremental Techniques

here are two techniques that are integrated in our architecture:

- *Log-driven backups*: The key idea is to use log records as the means for propagating updates to the BI rather than relying on page flushes.
- *Fresh/Stale Marking*: Maintaining in stable memory a "freshness" status of each database page. Con quently, restart processing is simplified and made very fast.

e first review each of these techniques separately.

1 Log-Driven Backups

he flow of log records in our architecture is a central element to the understanding the log-driven techniq he abstraction we are using here is that of a stream of log records that continuously flows from a compone its successor in a pipelined fashion. These components manipulate the log records and pass them along to t xt component down the pipeline. The flow of log records is depicted schematically in Figure 1.

Log records are produced by active transactions as they access the PDB, and are appended to the log ta here, a component referred to as the *accumulator* processes the stream of log records as follows before rwards them to the next stage in the pipeline. Log records of active transactions are queued and delayed u e transaction either commits or aborts. If a transaction aborts, its log records are used for the undoing of t rresponding updates on the relevant PDB pages and then discarded. Log records of committed transactio e grouped together on a page-basis and then transferred to the next stage in the pipeline. That is, all recor cumenting updates to a certain database page are grouped together. Thus, the accumulator filters out cords of active and aborted transactions and forwards only log records of committed transactions grouped o tabase page basis. The accumulator operates entirely within the non-volatile stable memory. Observe that cords that pass the accumulator are Redo-Only log records, and have no before-image information since th cument only committed updates.

Next in the pipeline are two parallel components: the *logger* and the *propagator*. The logger flushes log recor the log disk in order to make room in the (limited-size) stable memory.

The task of the propagator is to update the BDB pages to reflect the modifications specified by the log recor order to amortize page I/O, the accumulator groups log records that belongs to the same page together, so th e propagator will apply them all in a single I/O. Since the updates of the BDB are driven by the log recor coin the name *log-driven backups* accordingly.

Notice that the propagator applies to the BDB updates of only committed transactions. In effect, following t cumulator, there are only Redo log records. These log records are grouped on a database page basis. They a itten to the log on disk by the logger, and are used to guide a continuous update of the BDB by the propagat hen rearranging the log records, the accumulator can also reorder the records to minimize seek-time when t opagator applies the corresponding updates to the BDB.

- *Backup image.* The image of x as found in secondary memory at this particular instance, regardless relevant log information. The backup image of x is denoted $backup[x]$.

- *Committed image.* The image that reflects the updates performed by the last committed transaction so f The committed image of x is denoted $committed[x]$.

The committed image of a page may not be realized directly on either secondary or main memory. Howev should always possible to restore the committed image by applying log records to the backup image. Followi crash, the backup image of the database pages is available on secondary storage. It may not reflect updates mmitted transactions (depending on the buffer management policy) may reflect updates of aborted ones. Th it differs from the committed image.

We use the term *Primary Database* (PDB) to denote the set of database pages that reside in main memo e set of backup pages stored on secondary storage is referred to as the *Backup Database* (BDB). The BDB instance of the entire database, and the PDB is just a subset of the database pages.

Following a crash, the restart procedure brings the database up-to-date based on the BDB and the log. Duri rmal operation, updates to the PDB are propagated to the BDB keeping it close to being up-to-date (an activi refer to as checkpointing).

We use the following terminology to denote the properties of a page x. We say that:

- page x is <u>dirty</u> iff $backup[x] \neq current[x]$

- page x is <u>stale</u> iff $backup[x] \neq committed[x]$

- page x is <u>up-to-date</u> iff $current[x] = committed[x]$

nversely, when x is *not* dirty, we say that x is *clean*, and similarly we say that x is *fresh* when it is *not* stale. lows from our definitions that a page that does not reside in main memory is clean. These three notions (dir le, up-to-date) are central to recovery management.

We use the variable $x.dirty$ to denote the clean/dirty status of page x. Whenever a PDB page is updated tl riable is set. Conversely, once a page x is flushed, $x.dirty$ is cleared and we say that $x.clean$ holds.

In the sequel, $x.dirty$ is interchangeable with the phrase "x is dirty", and similarly for $x.clean$ and "x an". Formally: $(\forall x : (backup[x] \neq current[x]) \equiv x.dirty)$. Notice that using our terminology, a page may ty and up-to-date. Such a situation arises when the committed image of the page has not been propagated BDB.

Principles Underlying the Architecture

st, we list the principles that should constitute a good design of a recovery component for a MMDB.

- *Large memory and larger database.* The database systems for which we target our study are characterized having a very large database buffer, and an *even larger* physical database. It is assumed that by exploiti the size of the buffer, the disk-resident portion of the database is accessed infrequently. By adhering to tl principle, we guarantee that the approach capitalizes on the performance advantages offered by MMDE without precluding the possibility of having some portions of the database on secondary storage.

- *Redo-only BDB.* Having a very large buffer, it is anticipated that page replacements are not going to be ve frequent or very urgent. Therefore, there is no need to complicate recovery by propagating uncommitt updates to the BDB (i.e., the steal policy [HR83] should not be used). By enforcing this principle, a st page is brought up-to-date by only redoing missing updates; there are no updates to undo. This princi will contribute to fast and simple recovery management.

- *Decoupling of transaction and recovery processing.* Transaction processing should be interrupted as lit as possible by recovery-related overhead. Otherwise, as noted earlier, the performance opportunities MMDBs would remain unexploited. This principle can be satisfied only by virtually separating recove and transaction processing.

2 Restart Processing

ne notion of a *restart procedure* is common to a variety of transaction processing systems that rely on loggi a recovery mechanism. After a system crash, the restart procedure is invoked in order to restore the databa its most recent consistent state. Restart has to undo the effects of all incomplete transactions, and to redo t mmitted transactions, whose effects are not reflected in the database. Restart performs its task by scanni suffix of the log. In some cases there are up to three sweeps of the suffix of the log (analysis, forward, a ckward sweeps [Lin80, MHL$^+$90]).

There are two major activities that contribute to the delay associated with restart processing. First, t g suffix must be read from disk to facilitate the undoing and redoing of transactions. Second, bringing t tire database up-to-date triggers a significant amount of updates that translate to substantial I/O activity. T erval between consecutive checkpoints largely determines how long performing these two activities would ta eu84, CBDU75]. The longer the interval, more log records are generated and accordingly more transactions a be undone and redone by restart. The key point is that normal transaction processing is resumed only *af* start's termination. That is, standard restart processing is accounted as part of the down-time of the systen

The maximum tolerable down-time is a very important parameter, and in certain cases the delay caused ecuting restart is intolerable. In systems featuring high transaction rates, for instance, restart has to be fa ice even a short outage can cause a severe disruption in the service the system provides [Moh87]. We arg at the standard approach to restart is not appropriate in an advanced database management systems featuri ge storage capacity and high transaction rates, since recovering the entire database by replaying the executi uld contribute significantly (in the order of minutes) to the down-time of the system.

A Page-Based Recovery Model

the sequel we use the following terms and assumptions to define our model. The model is simplified for ea exposition.

On the lowest level, a database can be viewed as a collection of data *pages* that are accessed by transactic uing **read** and **write** operations. Pages are stored in secondary storage and are transferred to main memc ffer to accommodate reading and writing. A *buffer manager* controls the transfer of individual pages betwe condary and main memory by issuing *flush* and *fetch* operations to satisfy the reading and writing requests ecuting transactions. Flush transfers and writes a page from the buffer to secondary storage. Flushing a pa secondary storage is made atomic by stable storage techniques (e.g, [Lam81]). A page is brought to the buf om secondary storage by issuing the fetch operation. If the buffer is full, a page is selected and flushed, there aking room for the fetched page. It is assumed that executing a **read** or a **write** is not interrupted by pa shes.

Abstractly, a *log* is an infinite sequence (in one direction) of *log records* which document changes in the databa ite. A suffix of the sequence of log records is stored in a log buffer in memory, and is occasionally forced condary storage, where the rest of the log is safely stored. We refer to the portion of the log in main memc the *log's tail.* Whenever a page is updated by an active transaction, a record that describes this update pended to the log tail. In order to save log space, each update log record includes only the old and new sta so called the before and after images) of the affected portion of the updated page, along with an indication at portion (e.g., an offset and length of affected portion) [Lin80]. Such a logging method is called *entry loggi* partial physical logging).

Concurrency control is achieved through the use of a *locking* protocol. Appropriate locks must be acquir ior to any access to the database pages. We emphasize that (at this stage) locking granularity is *entire pag* d that the protocol produces *strict* schedules with respect to pages [BHG87]. Granularity of locking is refin Section 8. Strict locking means that only one active transaction can update a page, at any given instance.

In order to present our algorithms formally and precisely we introduce the following terminology and notatic any given instance there are three *images* (or states) associated with each page x:

- *Current image.* If x is currently in main memory then its image there is its current image; otherwise current image is found in secondary memory. The current image of x is denoted *current*$[x]$.

an *incremental* fashion, concurrently with, and without impeding, transaction processing. The algorith propose are motivated by the characteristics of an MMDB and exploit the technology of stable memory in uine manner that differs from the numerous proposals for using these devices in transaction processing syste g., [Eic87, DKO$^+$84, LC87, CKKS89]).

The techniques we propose concentrate on incremental approach to restart processing and checkpointing MDBs. We devise a scheme in which transaction processing resumes at once after a crash. Restoring da jects is done *incrementally* and is guided by the *demand* of the new transactions. Our checkpointing scher pitalizes on the performance advantages of MMDBs without precluding the possibility of having some portic the database on secondary storage. The scheme's main feature is decoupling of recovery processing a nsaction execution, thereby almost eliminating the common effect of the former delaying the latter. The wo orted in this paper is a continuation of our earlier work in this area [Lev91, LS90].

Our intention in this paper is to emphasize the principles of an incremental approach to recovery processi her than present an involved implementation. We first develop incremental recovery techniques that a sed on physical entry logging for a simple page-based model. Then we use this algorithm as a module in t nstruction an incremental restart algorithm based on operation logging and multi-level transactions.

The paper is organized as follows. In Section 2 we briefly survey why conventional recovery techniques a t suitable for MMDBs. Section 3 outlines a page-based recovery model that is used in the construction of t wer layer of our architecture. The model and terminology established in this section are used in the rest of t per. The principles that should underlie a sound design are presented in Section 4. The incremental techniqu propose are described in Section 5, and proved correct in Section 6. Several improvements to the architectu e proposed in Section 7. The applicability of our methods for high-level recovery management, which is n ge-based, is elaborated in Section 8. Related work is reviewed in Section 9. We sum up with conclusions ction 10.

The Deficiencies of Conventional Approaches

e concentrate on the subjects of checkpointing a large buffer, and restart processing. Later, we propose egrated solution for these problems that does not possess the deficiencies outlined in this section and thus ore suitable for MMDBs.

1 Checkpointing a Large Buffer

illustrate the problem of checkpointing large buffers, consider the *direct checkpointing* technique, variants hich are offered as the checkpoint mechanism for MMDBs [Eic87, SGM87a]. A direct checkpoint is a perio mp of the main memory database to disk, and is essential for the purposes of recovering from a system cra nsider a naive checkpointing algorithm which simply halts transaction processing and dumps the main memo tabase to disk. For a database size in the order of Gbytes, execution of this algorithm takes hundreds of secon ring which no transactions are processed! Moreover, as sizes of databases and memory chips are increasi pidly, the problem will become more severe. Indeed, contemporary direct checkpointing algorithms are mu ore sophisticated and efficient than this naive algorithm, but still the periodic sweep of the main memory th ides the dumping to the disk is the basis to all of them. Therefore, any variation of direct checkpointing und to delay transaction processing to a considerable extent.

Many of the proposed algorithms and schemes for MMDBs rely on the explicit assumption that the ent tabase is memory-resident [GMS90, LN88, SGM87b, DKO$^+$84]. Although other proposals acknowledge th s assumption is not valid for practical reasons, the issue is not addressed directly in their designs [LC87, Hag c86]. Even though the size of main memory is increasing very rapidly the size of future databases is expected crease even more rapidly. Indeed, there are a number of commercial database management systems in existen th a Tera byte or more of active data. We stress that the assumption that the database is only partia emory-resident *must* underlie a practical design of a practical database system.

Introduction

e task of a *recovery manager* in a transaction processing system is to ensure that, despite system and tra tion failures, the consistency of the database is maintained. To perform this task, book-keeping activities rformed during the normal operation of the system and restoration activities take place following the failu aditionally, the recovery activities are performed in a quiescent state where no transactions are being process r instance, following a crash, transaction processing is resumed only once the database is brought up-to-d d its consistency is restored by a *restart* procedure. Essentially, restart processing is accounted as part of wn-time of the system, since no transactions are processed until it terminates. A similar effect of halting, erfering with, transaction processing in order to perform a recovery-related activity is observed in connect th certain checkpointing techniques. To checkpoint a consistent snapshot of the database, transaction p ssing has to halt. The appealing alternative is to perform these activities *incrementally* and *in parallel* w nsaction execution.

This fundamental trade-off between recovery activities and forward transaction processing is underlined i tabase system incorporating very large semiconductor memory (in the order of Gbytes). Such Main Mem tabase systems are subsequently referred to as an MMDBs (see [Bit86], and the references there for an overvi different aspects of MMDBs). The potential for substantial performance improvement in an MMDB is prom , since I/O activity is kept at minimum On the other hand, because of the volatility of main memory, ue of failure recovery becomes more complex in this setting than in traditional, disk-resident database syste oreover, since recovery processing is the only component in a MMDB that must deal with I/O, this compon ust be designed with care so that it would not impede the overall performance.

Another advancement in semiconductor memory technology is that of non-volatile RAM, which is refer hereafter, as *stable memory*. An example of stable memory technology is battery-backup CMOS memo at are widely available [CKKS89]. In case of a power failure, the contents of this memory are not lost. Sta emories are available in sizes on the order of tens of megabytes and have read/write performances two to f nes slower than regular RAMs, depending on the hardware. The reader is referred to [CKKS89] for more det this technology.

The traditional approach to recovery has to be revisited in light of he availability of large main memo d stable memories. On the one hand, traditional recovery techniques fall short of meeting the requirements gh-performance databases systems that incorporate very large volatile buffers. In such systems, the trade tween recovery and forward processing is sharpened and made more critical. On the other hand, by th ture, stable memory devices are bound to advance the design of a recovery management subsystem.

The following points explain the impact of large main memories and stable memories on the approach covery:

- The larger the database buffer, the less page replacement occurs. Therefore, in database systems where database buffer is huge, paging cannot be relied upon as the primary mechanism for propagating upda to backup database on disk, since paging is expected to be a relatively rare activity. Many recent resea efforts go to the extreme with this trend arguing that there are cases where the entire database can in memory, thus eliminating paging entirely (e.g., [DKO$^+$84, LN88, GMS90, SGM87a]). With infreque page replacements, checkpointing and keeping a stable copy of the database may become a very disrupt function.

- Typically, persistence and atomicity of transactions is guaranteed by performing disk I/O at certain criti points (e.g., flushing a commit log record at the end of transaction). Stable memory enables divorc atomicity and persistence concerns from slow disk I/O. This simple, yet promising, approach was explor in [CKKS89, DKO$^+$84].

- Traditionally, a sequential I/O method, namely logging, is used to accommodate efficiently the book-keep needs of the recovery management system. Consequently, this information is a sequence of log reco lacking any helpful structure or organization. The availability of a stable memory provides the means maintaining some of the recovery book-keeping information in randomly accessible and fast memory.

In light of the above factors, we propose an alternative to the traditional approach to recovery managem database systems. Our approach is based upon the principle that recovery activities should be perfor

Incremental Recovery In Main Memory Database Systems[*]

Eliezer Levy

Avi Silberschatz
avi@cs.utexas.edu (512) 471-9706

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Abstract

In traditional database management systems, recovery activities, like checkpointing and restart, are performed in a quiescent state where no transactions are active. This approach impairs the performance of on-line transaction processing systems. Recovery related overhead is particularly troublesome in an environment where a large volatile memory is used. The appealing alternative is to perform recovery activities *incrementally* and *in parallel* with transaction execution. An incremental scheme for recovery in main memory database systems is presented in this paper. We propose a page-based incremental restart algorithm that enables the resumption of transaction processing as soon as the system is up. Pages are recovered individually and according to the demands of the post-crash transactions. In addition, an incremental method for propagating updates from main memory to the backup database on disk is also provided. Here the emphasis is on decoupling the I/O activities related to the propagation to disk from the forward transaction execution in memory. Finally, we construct a high-level recovery manager based on operation logging on top of the low-level page-based algorithms. The algorithms we propose are motivated by the characteristics of main memory database systems, and exploit the technology of non-volatile RAM.

Keywords

Transaction management; Recovery; Main-Memory Databases

[*] Work partially supported by grants from Unisys Roseville Operations, the National Science Foundation (IRI-8805215, I
03341, IRI-9106450), and the IBM corporation.

INCREMENTAL RECOVERY IN MAIN MEMORY DATABASE SYSTEMS

Eliezer Levy and Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-01 January 1992

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712