


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

The above comparison favors the log-driven approach. Among the rest, fuzzy algorithms seem to be close competitors. We note that fuzzy algorithms stand out (considering CPU overhead during normal operation) according to 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 is interesting to note that in [Eic86], log records of a transaction are marked after the transaction has committed, so that only log records of committed transactions would affect the BDB. It should also be mentioned that a log-driven approach is often used to manage remote backups for disaster recovery purposes (e.g., [KGMHP87, Tan87]).

10 Conclusions

The increasing size of contemporary databases, and the availability of stable memory and very large physical memories are bound to impact the requirements from, and the design of recovery components. In particular, for checkpointing and restart processing, the traditional approach becomes inappropriate for high rates of transactions and very large databases. An incremental approach, that exploits the new technological advances, is a natural solution. 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 interleaving with normal transaction processing. The techniques exploit stable memory and are geared to meet the demands of systems that incorporate large main memories. We have proposed both restart algorithm (called incremental restart) and a checkpointing-like technique (called log-driven backups) that operate in an increment manner, in parallel with transaction processing. The prominent original concepts motivating our design are as follows:

- Associating restoration activities with individual data objects, and assigning priorities to these activities according to the demand for these objects. Consequently, recovery processing is interleaved with normal transaction processing. By contrast, the conventional restart procedure for example, treats the database as 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 records) on data objects basis. This structuring is aimed to facilitate the efficient restoration of individual data objects.

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

References


Delaying restart activities was first described in [Rap75]. There, restart does not perform any recovery activity until a transaction is read. Instead, reading a data item triggers a validity check that finds the committed version of the data item that should be read. The incremental restart procedure we propose resembles this early work in that data items are recovered only once they are read.

A more conventional approach to speeding up restart is proposed in [MP91] in the context of the ARIES transaction processing method. The idea there is to shorten the redo pass of conventional restart by performing a selective redo. Instead of repeating the history by redoing all the actions specified in the log, only those actions specified in winner log records are redone. It is also mentioned there that undo of loser transactions can be interleaved with the processing of new transactions if locks (similar to RS-locks) protect the uncommitted data items updated by the loser transactions. During the analysis pass of restart, the identity of these data items is recovered, 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 the context of ARIES. It is mentioned that in IBM’s DB2 redo/undo for objects that are off-line can be deferred if the system remembers the LSN ranges for those objects and makes sure that they are recovered once they are brought on-line and before they are made accessible to other transactions. DB2 employs physical, page-level logging. Problems related to logical undoing and deferred restart are also discussed in [MHL+90]. Our work differs from the ARIES work in exploiting stable memory and as it presents a simple algorithmic description of the 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 transactions are never flushed to the backup database. At restart, the committed state is constructed immediately by loading the recently committed pages from a log device (called safe there). The main disadvantages of this approach are that locking is supported only at the granularity of pages, full-page physical logging is used, and in contrast to our entry logging, update-intensive transactions need to be treated specially, and that committing includes a synchronous I/O. The DB cache idea is refined to accommodate finer granularity locking [MLC87], however this extension does not deal with operation logging and concurrency among semantically compatible operations.

Work on improving restart processing is reported in [Moh94]. The approach there is to adapt the pass through traditional restart and admit new transactions during these passes. Also, associating freshness status with committed pages is discussed there and in [Moh90].

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

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

- Checkpointing interferes in one way or another with transaction processing, since both activities compete for the PDB and the main CPU. Taking a consistent checkpoint requires bringing transaction activity to a quiescent state, since a transaction-consistent checkpoint reflects a state of the database as produced by completed transactions. In the extreme case, transactions have to be aborted to guarantee the consistency of the checkpoint [Pu86]. Even in fuzzy algorithms, which do not produce consistent checkpoints [Hag86], memory contention is inevitable since both normal transactions and the checkpoint must access the very same memory. By contrast, in the log-driven backups scheme, transaction processing and propagation to the BDB do not use the same memory and may use different processors. This separation is the key advantage of the scheme.

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

- It is not clear how checkpointing algorithms can be adjusted to support our assumption of a partially resident database. The correctness of these algorithms may be jeopardized by arbitrary fetching and flushing of database pages. It seems that fuzzy checkpointing, which is the simplest type of checkpointing algorithm,
[HL+90, Gra78]. In our case, since updates are not propagated before the commit of an operation, the W2C
matrix means that the high-level undo record should be written to the high-level log prior to the commit point of the corresponding operation.

By structuring the high-level recovery on top of our incremental restart method, we intend to give the over-
all recovery scheme incremental flavor. The major challenge in making this multi-level recovery scheme increment-
al is the fact that we can no longer treat single pages as the individual unit for recovery, since operations affect-
many pages. Had we used single pages, we would have violated the high-level action atomicity requirement men-
tioned above. For this reason we devise the notion of a recovery unit (RU). An RU is a set of pages, such that
it is not possible for any high-level operation to affect more than one RU. For instance, if an INSERT opera-
tion is used for updating both index and data files, then the index and the corresponding data file constitute an RU.
The responsibility of the base recovery is bringing a RU to an operation-consistent state before any high-level undo can be applied to it.

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

By partitioning the database to RUs the incremental effect is obtained. RUs can be of coarse granularity
thereby diminishing the benefits of incremental restart. For example, an entire relation and the corresponding
index structure must be recovered before a post-crash transaction may read any of the tuples. This observa-
tion holds for as small RUs as possible.

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

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

In terms of transactions, $T_1$ is a loser, whereas $T_2$ is a winner. Base recovery for the three pages takes place
exactly as in Example 1 (i.e., only page $a$ is recovered). In the high-level recovery phase, only $T_{11}$ is undone, since
$T_{12}$ was a loser in the base-level.

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

Related Work

The work reported in this paper is a continuation of our earlier work in this area [Lev91, LS90]. A general
stale/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
memory is used extensively to implement this approach. There are several aspects that distinguish our work from the work on [LC87]. Some aspects there are peculiar to entirely-resident MMDs. Namely, there is no
consideration of paging activity. Integrating full-fledged operation logging is not discussed in [LC87] at all. Aside from
stale/fresh partition and the improvements it entails are lacking from the work in [LC87].
Incremental Recovery for High-Level Recovery Management

A common way of enhancing concurrency is the use of semantically-rich operations instead of the more primitive read and write operations. Having semantically-rich operations allows refining the notion of conflicting versus commutative operations [BR87, Wei88]. It is possible to examine whether two operations commute (i.e., do not conflict); such operations have the nice property that they can be executed concurrently. Semantics-based concurrency control is often cited as a very attractive method for handling high contention to data (i.e., 'hot spots') [MHL+90, BR87, Wei88]. The problem, however, is that the simple state-based (i.e., physical) recovery methods no longer work correctly in conjunction with these operations. Only operation logging, referred to as logical-transition logging [HR83], can support this type of enhanced concurrency. For instance, consider the increment and decrement operations which commute with each other and among themselves. A data item can be incremented concurrently by two uncommitted transactions. If one of the transactions aborts, its effect can be undone by decrementing the item appropriately. However, reverting to the before image may erase the effects of the 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 implemented as a set of lower-level operations, and hence their atomicity is not guaranteed. Therefore, when logged operations are undone or redone after a crash, they should not be applied to a backup database that reflects partial effects of the operations. Therefore, a key assumption in any operation logging scheme is that operations must appear as though they were executed atomically. This requirement is a prerequisite to any correct application of operation logs to the BDB at restart time, and is referred to as high-level action atomicity in [WHBM90]. As an illustration, we mention System R [G+81] which employs operation logging. There, at all times, the BDB is in a consistent state — a state that reflects the effects of only completed operations, and no partial effects of operations. This property is obtained by updating the BDB atomically, and only at checkpoint time, using a shadowing technique [Lor77]. At restart, the operation log is applied to the consistent shadow version of the database.

The problem of implementing operation logging is best viewed as a multi-level recovery problem. A very elegant and simple model of (standard, non-incremental) multi-level recovery is introduced in [WHBM90]. What 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-granularity items (e.g., tuples, records), and is implemented by several base-level primitives that collectively may affect more than a single page. The base level primitives are read and write that affect single pages—primitives that are consistent with our page-level model.

In other words, transactions are nested in two levels. Serializability of transactions is enforced by a multi-level concurrency 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 recovery, which ensures persistence and atomicity of higher-level operations and not of complete transactions. That is, the high-level operations are regarded as transactions as far as the base recovery module is concerned. Persistence of committed transaction is obtained as a by-product of the persistence of its operations (i.e., if all operations of a transaction have committed, then the transaction itself has committed). Observe that both the log-driven backup and marking algorithms refer to operations rather than transactions in the current context. Any occurrence of an operation the transaction 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.e., -seal policy with respect to operations is enforced). This is not a major restriction since operations update a small number of pages. Imposing this restriction also helps avoiding the extra overhead due to the hierarchical store. Consequently, the log of the base recovery, called the base log, is a redo log and there is no need to perform base-level undo at restart.

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

Since the high-level log deals with Undo log records, it should obey the Write-Ahead-Log (WAL) rules.
Figure 2: Lock compatibility matrix.

<table>
<thead>
<tr>
<th>requested held</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RS</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

allowed even greater flexibility. Indeed, stale pages cannot be read by post-crash transactions; however, writing data 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 stale pages, and no other pages, after a crash. An imaginary restart transaction acquires these locks as soon as the system is rebooted and before post-crash transactions are processed. In Figure 2, we present the lock compatibility matrix for the three lock modes read (R), write (W), and restart (RS). Since restart locks are not requested or held rather are held by convention by the restart transaction, the compatibility matrix lacks the request column for 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 observing that the imaginary restart transaction is a two-phased transaction that is serialized before any post-crash transaction that attempts to access a stale page. Also, restart locking cannot introduce deadlocks, since the 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 when it is explicitly brought up-to-date by the incremental restart procedure, by applying log records to the back page.

A write of a stale page results in an update log record containing only the after image of the update, since the page has not been recovered yet. Such a log record will actually affect the relevant page once the page is recovered 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 incremental restart processing. Some transactions are scheduled without being delayed by the recovery activity at all, and some are delayed only as a result of recovering data items they need.

2 Further Improvements

In this subsection, we briefly mention several points that can further improve an implementation of the incremental restart algorithm.

- RS-locking can be used to combine incremental and standard restart for different sets of pages, thereby avoiding the need to maintain stale/fresh marking for too many pages. The set of pages that are recovered using standard restart should be RS-locked until they are made consistent. Only predicted ‘hot spot’ data can be supported by incremental restart (and the stale/fresh marking). This improvement allows a very 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) recover pages demanded by executing transactions. Once a page is recovered and made consistent, the RS lock can be released. This technique provides even greater concurrency between restart and transaction processing.

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

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

We prove two claims that underlie the correctness of our integrated architecture. The correctness of the marking algorithm 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 $x$'s stale holds. Formally: $x : (\text{backup}[x] \neq \text{committed}[x]) \Rightarrow x$.stale is invariant.

**Proof.** Consider the state space formed by the variables we have introduced. We model the execution of transactions and fetching and flushing of pages, as transitions over that state space. We prove the claim by showing 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. Flushing a page is modeled as an assignment to $\text{backup}[x]$, and committing a page is modeled as an assignment to $\text{committed}[x]$. There are four state transitions that may affect the validity of the invariant: an execution of the assignment statement specified in one of the rules Dirty–Stale, Flush–Fresh, the commitment of an update transaction, and the flushing of a page. We prove that the invariant holds by showing that each of these state transitions 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., $x \in T$.writeset). Since a strict concurrency protocol is employed at a page level, we are assured that no other transaction has updated $x$ subsequently to $T$’s update and before $T$’s commitment. If $x$ is dirty, then $T$’s commitment renders it stale. However, since the assignment in Dirty–Stale is executed prior to the commitment of $T$, $x$.stale holds, and the invariant still holds.
- **Flushing $x$:** According to our assumptions regarding buffer management policy, flushing a page $x$ always 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 the flush, and hence falsifying $x$.stale preserves the invariant.

Thus, the invariant holds.

It should be realized that if $x$.stale holds it does not necessarily mean that $x$ is indeed stale, however the converse implication does hold, as stated in Lemma 1. Hence, notice that $x$.stale and “$x$ is stale” are perfectly interchangeable.

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

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

Improvements

In this section we present several possible enhancements and refinements to the techniques we have presented earlier.

1. **Improving Restart Processing**

Using the fresh/stale marking post-crash transactions can access fresh pages as soon as the system is up. Attempting to access a stale page triggers the recovery of that individual page. The transaction that requested the access 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 of variables is the only data structure that is maintained in stable memory. All other data structures are kept in volatile memory and are lost in a crash. We stress that the boolean variables are introduced only to 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 modifies.

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

- **Dirty & Stale.** Prior to the commit point of \( T \): 
  \[
  (x \in T.writeSet \land x.dirty) \quad \text{then} \quad x.stale := \text{true}
  \]

- **Flush & Fresh.** After flushing a dirty page \( x \): 
  \[
  x.stale := \text{false}
  \]

An assignment and its triggering event need not be executed as an atomic action. All that is required is that the events that affect the variables we have introduced occur between the triggering event and the corresponding assignment. The key idea in the algorithm is to always set \( x.stale \) to true just prior to the event that actually causes \( x \) to become stale. As a consequence, a situation where \( x.stale \) holds but \( x \) is still fresh is possible, and likewise, 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 \( c \):

\[
T_{11} = R/W(a), R/W(b) \\
T_{12} = R/W(a), R/W(b) \\
T_{21} = R/W(c), R/W(a)
\]

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

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

After the crash, \( a.stale \) holds (by Dirty-Stale prior to \( c_{21} \)), \( b.stale \) does not hold (by Flush-Fresh after \( flush(b) \)), and \( c.stale \) also does not hold (by Flush-Fresh after \( flush(c) \)). Note that \( T_{11} \) and \( T_{21} \) are committed whereas \( T_{12} \) is to be aborted. We say that \( T_{11} \) and \( T_{21} \) are winner transactions, whereas \( T_{12} \) is a loser transaction. Using the marking, only the updates of the winner transactions to page \( a \) need to be redone, since only \( a \) is marked stale.

3 The Integrated Architecture

In summarize the integrated architecture we list the five components we have introduced and their corresponding functionality. 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 produced by transactions and forwards log records of committed transactions. In order to amortize page I/O, the accumulator groups log records that belongs to the same page together, so that the propagator will append 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 maintains 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.
In this case, we are assured that all the updates have been applied to the BDB already (by the propagator) and there is no need to flush the page.

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

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

2 Stale/Fresh Marking

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

```plaintext
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) stale needs to be implemented. The updates needed to bring a stale page up-to-date are always Redo updates because, according to our assumptions. The log records with the missing updates can be found either in the log tail or on the log disk. According to the trade-off presented earlier regarding the timing of discarding a log record from stable memory, [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 transaction processing immediately after a crash, while preserving the consistency of the database. Typically, the log stores enough information to deduce the stale/fresh status of pages. However, this information is not available immediately. The marking also controls the recovery of data pages one by one according to the transaction demands. In order for the algorithm to be practical, it is critical to both maintain the stale/fresh marking in main memory, as well as have it survive a crash. Therefore, we underline the decision to maintain the stale/fresh marking in stable memory. We do not elaborate on how to manage the marking efficiently. However, in light of the scale of current databases, an appropriate data structure holding page IDs that supports efficiently inserts, deletes, and searches is deemed crucial. Observe that the functions of the analysis pass [MHL+90] in standard start procedures are captured by the stale/fresh marking, and are ready for use by restart without the need to analyze 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 dynamically as 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.

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

Based on the above transitions we present a reactive algorithm that manages a stale/fresh marking of pages and indicate whether they are stale or fresh. In order to present the algorithm formally, we introduce the following variables and conventions:
The pipeline of log records can be efficiently mapped onto a multi-processor shared-memory architecture. In particular, the propagator and the logger tasks can be carried out by dedicated processors. This way, recovery associated 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 be discarded only after it is written to the log disk by the logger. However, such an early discarding means that the record has not yet been processed by the propagator, then its update will not be reflected in the BDB (since it skipped the propagator processing stage). The propagator can fetch the missing records from the disk but this would really delay the propagation. Alternatively, the pages whose updates were skipped by the propagator can be marked stale (see below on how the marking is managed), thereby postponing handling of the missing updates to a later time. These difficulties can be avoided when log records are not discarded from stable memory before they have been processed by the propagator. However, the trade-off arises as it is anticipated that the propagator would lag behind the logger because the former performs random access I/O whereas the latter performs sequential I/O. In [LS90] we analyze this trade-off and propose to use a RAID I/O architecture (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 BDB dictated by the demands of the executing transactions. The Buffer Manager is in charge of this exchange. We emphasize that the buffer manager flushed only pages that reflect updates of already committed transactions under the no-stale policy. Observe that the principle of Redo-Only BDB is implemented by both sources of updates to the BDB; the buffer manager as well as the propagator.

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

The fact that the BDB is updated by both the propagator and by flushing buffered pages must be considered with care. First, one should wonder whether these double updates do not interfere with the correctness of the scheme. Second, since two identical updates are redundant, one of them should be avoided for performance reasons. Regarding correctness, a problem arises when the propagator writes an older image of a page, overwriting the last up-to-date image that was written when the page was previously flushed by the buffer manager. If the page was fetched before the up-to-date images are written to the BDB by the propagator, transactions read inconsistent data. 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.

Observe that because of this rule, a page that is fetched from the BDB was last modified when it was flushed by the buffer manager. Therefore, the page is up-to-date when it is fetched to the PDB. The rule is referred to as Safe-Fetch since it ensures that a page fetched from the BDB is always up-to-date (except for following a crash).

Implementing Safe-Fetch implies that the propagator should know which pages are in the PDB. We assume that the propagator initially knows which pages are in the PDB, and it is notified about each page replacement by the buffer manager. We assume that the propagator and the buffer manager share some memory for this purpose. Alternatively, since a single I/O controller serves I/O requests of both the propagator and the buffer manager, forcing Safe-Fetch can be implemented by a smart controller. In any case, since page fetches are assumed to be 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 some log records. Safe-Fetch deals with cases where a page was flushed to the BDB before the corresponding updates were applied to the BDB by the propagator. I/O activity can be reduced considering the opposite case too, imposing 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.

This rule can be easily implemented using the log-sequence-number (LSN) mechanism [MHL90, Gra78]. Flushing the page can be avoided if the page’s LSN is at most the LSN of the page that was last written by the propagator.
Figure 1: A Schematic View of the Architecture
We incorporate the above principles in the proposed architecture. We do not assume an entirely-resident MDB, in the spirit of the first principle. Consequently, we deal with buffer management issues. The second principle is enforced by insisting on using the no-steal buffer management policy. Namely, only updates of committed 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 technology that enables promoting this principle. In the architecture we propose, the log tail is stored in stable memory. By committing a transaction, thereby making its updates persistent, is guaranteed by writing the commit log records to the log tail in stable memory. Any further recovery activity is totally separated from transaction processing and emphasize that in the architecture we propose the log tail is kept in stable memory (i.e., non-volatile RAM). For making the fast stable memory the only point of friction between transaction and recovery processing to achieve 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 BDB, rather than relying on page flushes.
- **Fresh/Stale Marking**: Maintaining in stable memory a “freshness” status of each database page. Consequently, restart processing is simplified and made very fast.

We first review each of these techniques separately.

1 Log-Driven Backups

The flow of log records in our architecture is a central element to the understanding the log-driven technique. The abstraction we are using here is that of a stream of log records that continuously flows from a component to its successor in a pipelined fashion. These components manipulate the log records and pass them along to the next 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 tail. A component referred to as the accumulator processes the stream of log records as follows before passing them towards them to the next stage in the pipeline. Log records of active transactions are queued and delayed until the transaction either commits or aborts. If a transaction aborts, its log records are used for the undoing of the corresponding updates on the relevant PDB pages and then discarded. Log records of committed transactions are grouped together on a page-basis and then transferred to the next stage in the pipeline. That is, all records documenting updates to a certain database page are grouped together. Thus, the accumulator filters out records of active and aborted transactions and forwards only log records of committed transactions grouped on a database page basis. The accumulator operates entirely within the non-volatile stable memory. Observe that log records that pass the accumulator are Redo-Only log records, and have no before-image information since they document only committed updates.

Next in the pipeline are two parallel components: the logger and the propagator. The logger flushes log records to 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 records. In order to amortize page I/O, the accumulator groups log records that belongs to the same page together, so that the propagator will apply them all in a single I/O. Since the updates of the BDB are driven by the log records, we coin the name log-driven backups accordingly.

Notice that the propagator applies to the BDB updates of only committed transactions. In effect, following the accumulator, there are only Redo log records. These log records are grouped on a database page basis. They are written to the log on disk by the logger, and are used to guide a continuous update of the BDB by the propagator. When rearranging the log records, the accumulator can also reorder the records to minimize seek-time when the propagator applies the corresponding updates to the BDB.
• *Backup image.* The image of $x$ as found in secondary memory at this particular instance, regardless of relevant log information. The backup image of $x$ is denoted $\text{backup}[x]$.

• *Committed image.* The image that reflects the updates performed by the last committed transaction so far. The committed image of $x$ is denoted $\text{committed}[x]$.

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

We use the term Primary Database (PDB) to denote the set of database pages that reside in main memory. The 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. During normal operation, updates to the PDB are propagated to the BDB keeping it close to being up-to-date (an activity we refer to as checkpointing).

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

- page $x$ is **dirty** iff $\text{backup}[x] \neq \text{current}[x]$
- page $x$ is **stale** iff $\text{backup}[x] \neq \text{committed}[x]$
- page $x$ is **up-to-date** iff $\text{current}[x] = \text{committed}[x]$

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

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

In the sequel, $x.\text{dirty}$ is interchangeable with the phrase “$x$ is dirty”, and similarly for $x.\text{clean}$ and “$x$ stale”. Formally: $(\forall x : \neg (x.\text{dirty}) \equiv \text{backup}[x] = \text{current}[x])$. Notice that using our terminology, a page may be dirty and up-to-date. Such a situation arises when the committed image of the page has not been propagated to the BDB.

### Principles Underlying the Architecture

First, 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 by having a very large database buffer, and an even larger physical database. It is assumed that by exploiting the size of the buffer, the disk-resident portion of the database is accessed infrequently. By adhering to this principle, we guarantee that the approach capitalizes on the performance advantages offered by MMDB 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 very frequent or very urgent. Therefore, there is no need to complicate recovery by propagating uncommitted updates to the BDB (i.e., the steal policy [HR83] should not be used). By enforcing this principle, a page is brought up-to-date by only redoing missing updates; there are no updates to undo. This principle will contribute to fast and simple recovery management.

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

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

There are two major activities that contribute to the delay associated with restart processing. First, the log suffix must be read from disk to facilitate the undoing and redoing of transactions. Second, bringing the database up-to-date triggers a significant amount of updates that translate to substantial I/O activity. The interval between consecutive checkpoints largely determines how long performing these two activities would take [eu84, CBDU75]. The longer the interval, more log records are generated and accordingly more transactions are undone and redone by restart. The key point is that normal transaction processing is resumed only after the system restart's termination. That is, standard restart processing is accounted as part of the down-time of the system.

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

A Page-Based Recovery Model

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

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

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

Concurrency control is achieved through the use of a locking protocol. Appropriate locks must be acquired or to any access to the database pages. We emphasize that (at this stage) locking granularity is entire page, and that the protocol produces strict schedules with respect to pages [BHGS87]. Granularity of locking is refined in 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 notation:

- **Current image**. If \( x \) is currently in main memory then its image there is its current image; otherwise the current image is found in secondary memory. The current image of \( x \) is denoted \( \text{current}[x] \).

an incremental fashion, concurrently with, and without impeding, transaction processing. The algorithms and
proposals are motivated by the characteristics of an MMDB and exploit the technology of stable memory in a
unique manner that differs from the numerous proposals for using these devices in transaction processing systems
(e.g., [Eic87, DKO'84, LC87, CKKS89]).

The techniques we propose concentrate on incremental approach to restart processing and checkpointing
of MMDBs. We devise a scheme in which transaction processing resumes at once after a crash. Restoring data
objects is done incrementally and is guided by the demand of the new transactions. Our checkpointing scheme
maximizes the performance advantages of MMDBs without precluding the possibility of having some portion of
the database on secondary storage. The scheme’s main feature is decoupling of recovery processing from
transaction execution, thereby almost eliminating the common effect of the former delaying the latter. The work
reported 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 processing rather than present an involved implementation. We first develop incremental recovery techniques that are based on physical entry logging for a simple page-based model. Then we use this algorithm as a module in the construction of 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 are not suitable for MMDBs. Section 3 outlines a page-based recovery model that is used in the construction of the lower layer of our architecture. The model and terminology established in this section are used in the rest of the paper. The principles that should underlie a sound design are presented in Section 4. The incremental techniques we propose are described in Section 5, and proved correct in Section 6. Several improvements to the architecture are proposed in Section 7. The applicability of our methods for high-level recovery management, which is model-page-based, is elaborated in Section 8. Related work is reviewed in Section 9. We sum up with conclusions in Section 10.

The Deficiencies of Conventional Approaches

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

1 Checkpointing a Large Buffer

To illustrate the problem of checkpointing large buffers, consider the direct checkpointing technique, variants of which are offered as the checkpoint mechanism for MMDBs [Eic87, SGM87a]. A direct checkpoint is a period of the main memory database to disk, and is essential for the purposes of recovering from a system crash. We consider a naive checkpointing algorithm which simply halts transaction processing and dumps the main memory database to disk. For a database size in the order of Gbytes, execution of this algorithm takes hundreds of seconds during which no transactions are processed! Moreover, as sizes of databases and memory chips are increasing, the problem will become more severe. Indeed, contemporary direct checkpointing algorithms are much more sophisticated and efficient than this naive algorithm, but still the periodic sweep of the main memory that precedes the dumping to the disk is the basis to all of them. Therefore, any variation of direct checkpointing will tend to delay transaction processing to a considerable extent.

Many of the proposed algorithms and schemes for MMDBs rely on the explicit assumption that the entire database is memory-resident [GMS90, LN88, SGM87b, DKO'84]. Although other proposals acknowledge this assumption is not valid for practical reasons, the issue is not addressed directly in their designs [LC87, Hag86]. Even though the size of main memory is increasing very rapidly the size of future databases is expected to increase even more rapidly. Indeed, there are a number of commercial database management systems in existence with a Terabyte or more of active data. We stress that the assumption that the database is only partially memory-resident must underlie a practical design of a practical database system.
Introduction

The task of a recovery manager in a transaction processing system is to ensure that, despite system and transaction failures, the consistency of the database is maintained. To perform this task, book-keeping activities are performed during the normal operation of the system and restoration activities take place following the failure. Traditionally, the recovery activities are performed in a quiescent state where no transactions are being processed. For instance, following a crash, transaction processing is resumed only once the database is brought up-to-date and its consistency is restored by a restart procedure. Essentially, restart processing is accounted as part of the down-time of the system, since no transactions are processed until it terminates. A similar effect of halting, or interfering with, transaction processing in order to perform a recovery-related activity is observed in connection with certain checkpointing techniques. To checkpoint a consistent snapshot of the database, transaction processing has to halt. The appealing alternative is to perform these activities incrementally and in parallel with transaction execution.

This fundamental trade-off between recovery activities and forward transaction processing is underlined in database systems incorporating very large semiconductor memory (in the order of Gbytes). Such Main Memory Database systems are subsequently referred to as MMDBs (see [Bit86], and the references there for an overview of different aspects of MMDBs). The potential for substantial performance improvement in an MMDB is promising, since I/O activity is kept at minimum. On the other hand, because of the volatility of main memory, the time of failure recovery becomes more complex in this setting than in traditional, disk-resident database systems. Moreover, since recovery processing is the only component in a MMDB that must deal with I/O, this component must 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 referred to hereafter, as stable memory. An example of stable memory technology is battery-backup CMOS memory that are widely available [CKKS89]. In case of a power failure, the contents of this memory are not lost. Stable memories are available in sizes on the order of tens of megabytes and have read/write performances two to four times slower than regular RAMs, depending on the hardware. The reader is referred to [CKKS89] for more details on this technology.

The traditional approach to recovery has to be revisited in light of the availability of large main memory and stable memories. On the one hand, traditional recovery techniques fall short of meeting the requirements of high-performance databases systems that incorporate very large volatile buffers. In such systems, the trade-off between recovery and forward processing is sharpened and made more critical. On the other hand, by the nature, 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 to recovery:

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

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

- Traditionally, a sequential I/O method, namely logging, is used to accommodate efficiently the book-keeping needs of the recovery management system. Consequently, this information is a sequence of log records lacking any helpful structure or organization. The availability of a stable memory provides the means for 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 management in database systems. Our approach is based upon the principle that recovery activities should be performed...
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, IRI-93341, 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