Chapter 9

SYSTEM FAILURE AND RECOVERY

Rajendran M. Sivasankaran

GTE Laboratories
40 Sylvan Rd
Waltham MA 02451, USA.

raju@gte.com

Krithi Ramamritham

Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA.

krithi@cs.umass.edu

John A. Stankovic

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903, USA.

stankovic@cs.virginia.edu

1. INTRODUCTION

Current databases need logging and recovery to maintain the correctness properties of transactions and consistency of the database under failures. Also, logging and recovery have tremendous performance implications. While there has been a lot of work that deals with logging and recovery algorithms for traditional disk resident databases ([1, 2, 3, 4, 5]) and Main Memory Databases (MMDB) ([6, 7]), researchers have not systematically explored the issue of logging and recovery in Real-Time Databases (RTDB). In fact, there is a need for designing new algorithms for logging and recovery in RTDBs because the sequential nature of logging and the lack of time and priority cognizance during recovery are not in tune with the priority oriented and preemptive nature of activities in RTDBs.

This chapter motivates the need to do logging and recovery differently in Real-Time Databases (RTDBs), and presents SPLIT, a partitioned logging and recovery algorithm for RTDBs, and ARUN[1], a suite of logging and recovery algorithms for RTDBs. In order to improve performance, ARUN makes use of the Solid State Disk (SSD) technology that is referred to as Non-Volatile High Speed Store (NVHSS) in this chapter. The primary motivation for this work comes from the fact that conventional logging and recovery are not suitable for priority oriented RTDBs as illustrated in section 2. The other motivating factor is that the characteristics of NVHSS technology, which is capable of reducing unpredictability and improving i/o performance in RTDBs, must be methodically taken into account during the entire design of RTDBs.

The logging and recovery algorithms are based on dividing data into a set of equivalence classes derived from a taxonomy of data and transaction attributes. For example, data can be broadly classified into Hot and $Cold$ depending on the type of the data (like $critical$, $temporal$ etc.), and type of the transactions (like $high\ priority$, $low\ priority$ etc.), that accesses the data. The logging and recovery algorithms assume that the classes of data are disjoint and that the recovery of one class can be done without having to recover the other classes, and if one class is recovered, then the transactions accessing that class can proceed without having to wait for the system to recover the other classes. For instance, in TONICS ([8]), a network management system consists of different kind of data such as the performance, traffic, configuration, fault and security management data. The fault management data, which could potentially get updated frequently, is very critical to the operation that recovering it immediately after a crash can improve the performance of the system.

2. PRINCIPLES UNDERLYING RTDB LOGGING AND RECOVERY

Conventional databases have a sequential log, use the Steal buffer management policy (allowing dirty data from data buffers to migrate to disk), and No Force commit policy (allowing a transaction to commit before the disk resident data is updated). Sequential logging and No Force policies are motivated by the slow disk access times (due to random seeks), and Steal is motivated by the

[1]ARUN - Algorithms for Recovery Using Non-Volatile High Speed Store.

potentially inadequate buffer space. In short, conventional logging and recovery are optimized for databases where both the data and log are disk resident.

This section systematically shows why conventional logging and recovery algorithms will not work well in RTDBs where transactions have timing constraints such as deadlines and some of the data, referred to as temporal data, have validity intervals associated with them after which they become invalid. It would suffice to say that the RTDBs that are considered in this chapter are rich in terms of data and transaction characteristics. In this chapter, *data class* is a set of data items that have certain properties and these properties can be derived from data types (like critical, temporal etc.) and/or transaction types (like critical, high priority etc.) of transactions that access the data items. Also, temporal data is not explicitly addressed in this chapter.

Following are some of the desirable properties of logging and recovery algorithms for RTDBs.

- Reduced log traffic
- Speedy logging and recovery
- Transaction priority oriented logging and recovery
- Data class oriented logging and recovery

The first two properties imply that logging and recovery should be *fast* which is a desirable property for conventional databases too. But, *real-time is not equivalent to fast* ([9]). The last two properties are key to the design of logging and recovery algorithms for RTDBs where the transaction priorities and data properties are taken into account.

To improve logging and recovery performance in RTDBs this chapter combines the following techniques:

- Partitioned and parallel logging and recovery.
- Ephemeral Logging.
- Using NVHSS as a fast persistent backup store.
- Data and transaction class based logging.

Each of the above is discussed in detail.

2.1 PARTITIONED AND PARALLEL LOGGING AND RECOVERY

The following example motivates the need for partitioning the log in RTDBs. In RTDBs, usually, cpu scheduling, i/o scheduling and concurrency control conflict resolution are based on transaction priorities ([10, 11, 12, 13]). But, conventional (non-partitioned) logging and recovery techniques are sequential activities that do not work well in such a priority oriented and preemptive RTDB setting. The following example illustrates that it is difficult to attain the key (last two) desirable real-time properties for RTDB logging and recovery algorithms using conventional sequential log. Let T_1 be a high priority transaction updating data items H_1 and H_2. Let T_2 be a low priority transaction updating data items C_1, C_2, C_3 and C_4. Let the sequence of updates be $W_1(H_1), W_2(C_1), W_2(C_2)$, $W_2(C_3)$, $W_2(C_4)$, and $W_1(H_2)$. This situation can arise in a multiprocessor with combined logs where T_1 and T_2 are executing on different processors[2].

If transaction T_1 wants to commit, then it has to flush the log until $W_1(H_2)$ which means it has to flush the log records of a low priority transaction T_2. In this chapter, this phenomenon of high priority requests doing work for low priority requests is referred to as *Priority Diversion*[3]. Priority diversion will increase the response times of high priority transactions thus making the conventional sequential logs unsuitable for RTDBs. Similarly, recovery of important data items cannot be done separately, but has to be integrated with the reading of log records of unimportant data items (and recovery of unimportant data items). Let H_1 and H_2 be hot (frequently accessed) data items and C_1, C_2, C_3 and C_4 be cold (infrequently accessed) data items. During recovery, in order to recover hot data items H_1 and H_2 the log records of cold items C_1, C_2, C_3 and C_4 have to be read (and possibly recovered). This example shows that a sequential log, and recovery using that log will pose problems in a priority based, preemptive RTDB setting.

Partitioned and parallel logging and recovery in itself is not a new idea. Parallel logging in conventional databases was explored in [14]. In [15], data placement, logging and recovery issues in real-time active databases are discussed which includes the idea of partitioned logs in a real-time context. In [16], log partitioning and related performance evaluation for real-time MMDB are presented. In all this work, there are no details of partitioned logging and recovery algorithms together with details of required data structures.

2.2 EPHEMERAL LOGGING

While the last subsection motivated the need to partition the log in RTDBs, this section presents ephemeral logging technique which helps to partition the log, and reduce the amount of log to process at recovery. We claim that ephemeral logging is suitable for RTDBs and substantiate this claim by presenting novel RTDB logging and recovery algorithms that take advantage of ephemeral logging.

---

[2] This can also happen in a single processor case if T_2 is waiting for a disk I/O or there is priority inversion.
[3] to indicate that high priority requests are *diverted* from what they are supposed to be doing.

An extreme case of partitioning would be to write separate logs for each transaction which will facilitate logging based on transactions' priorities for RTDBs. Using example 1, separate logs would imply that transactions T_1 and T_2 will have separate logs.

Now, log of T_1 can be flushed to the stable storage without having to flush log of T_2 thus avoiding priority diversion. Also, with transaction based logs, the log of a particular transaction could be discarded at the end (commit or abort) of that transaction, if the data pages accessed by that transaction are forced at commit. Transaction based logs that can be discarded at the end of the transaction are referred to as *ephemeral transaction logs* (as in [17]). Transaction based logs that cannot be discarded are referred to as *non-ephemeral transaction logs*.

The advantages of ephemeral logging are

- Transaction priority based logging and recovery is possible making it suitable for RTDBs.
- Checkpointing becomes unnecessary, since data items are forced at commit.
- After a system crash, potentially, any data item can be undone on demand since there is not much log to process.

The main disadvantages of ephemeral logging are

- The lack of permanent system-wide log that can be used as an audit trail.
- There is a need for a high speed stable storage to force data items and logs because ephemeral logging does not work well with disk-resident log or data ([18]).

2.3 USING NVHSS

NVHSS technology is imminent. The presence of database accelerators and log accelerators using SSD only confirm the hypothesis that SSD will become an integral part of future databases ([19]). However, we strongly believe that database designers should take Non-Volatile High Speed Store (NVHSS) such as SSD into account in their initial designs and not just use them in a brute force and adhoc manner, as it is used now, to improve performance.

The following reasons make NVHSS a suitable technology for RTDBs:

- Log write time becomes low, if the logs are written to NVHSS[4].
- Force time becomes low, if the data items are forced to NVHSS.
- Unlike disks, NVHSS access time is a constant. This improves the predictability of the system.

[4]A page (4K) write that takes 20 ms (5 ms) in a random (sequential) disk setting takes $100mus$ in NVHSS.

- Ephemeral logging, that is suitable for RTDBs, is facilitated because all the data pages can be forced to NVHSS and the logs can be written to NVHSS.

The fact that adding NVHSS to a database will improve the performance is obvious ([20]). But, NVHSS could be used as a general second level buffer, a data class specific second level buffer, a logging device or some meaningful combinations of the above. For instance in [21], NVHSS is used just a fast stable write buffer. Also, whereas in [17] logging and recovery algorithms for databases with non-volatile main memory are presented certain subtle, practical issues like the possibility of corruption of main memory by system failures are not considered. In [17], the study considers an integrated safe where the approach is to write page based before images, which can be memory consuming. Also, [17] does not deal with priority based, preemptive RTDBs. Basically, these studies don't consider the requirements of RTDBs to design the logging and recovery algorithms in a systematic way.

2.4 DATA AND TRANSACTION CLASS BASED LOGGING

In this section, we argue that there are advantages to maintaining separate logs, but not one per transaction, but one per transaction class. Transaction class oriented logging is a coarser granularity of logging than individual transaction oriented logging, i.e., each transaction class has a sequential log associated with it. This can work well if the transactions from different classes do not access the same data pages/objects at the same time. Otherwise, with transaction class based logging the log records of a data item can be in different logs and all these log records (and hence the logs) might have to be processed to recover the data item.

On the other hand, if data classes are disjoint (a reasonable assumption), then each data class can have its own log. For instance, in TONICS, the fault management data is independent of the traffic management data and it is desirable that the fault management data is recovered first. Also, the disjoint logs of related data can work well with the notion of *eventual consistency* ([22]) where even the related data can be inconsistent for some time and then they are made consistent at some later point in time. This data class oriented logging will facilitate data class oriented recovery which is a desirable feature for logging and recovery in RTDBs. This will facilitate partitioned and parallel logging and recovery of different classes of data items. The logging and recovery can be prioritized depending on the data class. Classes of data which are immediately needed by transactions can be recovered first and the remaining classes of data can be recovered in the background. While data class oriented logging

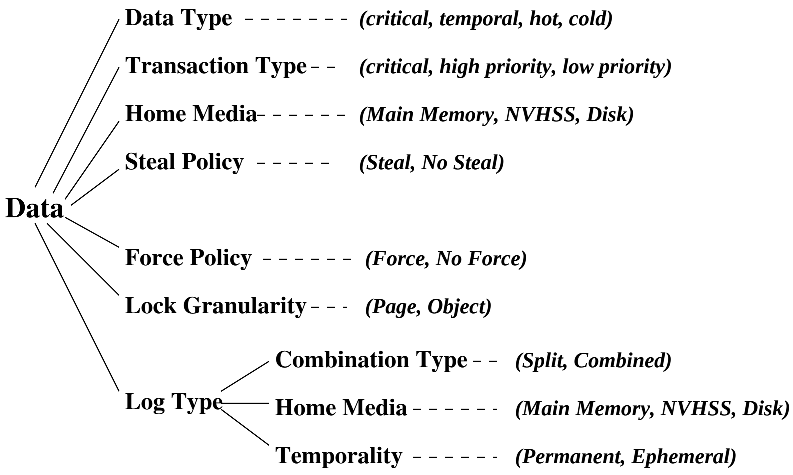

Figure 9.1 Taxonomy

is suitable for systems with or without NVHSS, transaction based logs work well only if NVHSS is present. Of course, NVHSS will reduce the overheads of logging and commitment.

Having introduced the ingredients of partitioned logging, ephemeral logging, NVHSS, and transaction/data classes, we now discuss the attributes of data in detail to pave the way for the development of a suite of algorithms that mix the ingredients depending on the data attributes of interest.

3. DATA, TRANSACTION AND LOG TYPES IN RTDBS: A TAXONOMY

Figure 9.1 gives a taxonomy of data attributes along seven dimensions. The possible values that an attribute can have are given along with the attribute in brackets. The data type attribute is the type of the data which could be $critical$, $temporal$, hot or $cold$. Data types themselves are not completely orthogonal. For instance, there could be a $cold$ data item that is $critical$[5].

The other data attributes are straightforward. The transaction type attribute is the type of the transaction that accesses the data. The Home Media is the media where the data resides. The steal and force attributes stand for the steal and force policies that are used to manage the data. Lock granularity stands for the granularity of locking on the data. The log type attribute stands for the type

[5]The data type of this item would be $critical$.

of the log that is used to manage the data. The log type has its own attributes. The home media attribute indicates where the log resides and the temporality attribute indicates if the log is ephemeral or permanent.

The data type and the type of the accessing transaction are application dependent attributes. All other attributes are policy decisions that the RTDB makes to process the data. These attributes, i.e., the home media, steal policy, force policy, locking granularity and log type are closely related attributes and are not totally orthogonal. For instance, a data item can not have ephemeral log and not be forced at commit time.

Data class, as defined in section 2., is a set of data items that have certain properties and these properties can be derived from data types and/or transaction types of transactions that access the data items. Log class is derived from the temporality of the log, home media of the log, home media of the data, lock granularity, and Steal and Force policies.

Let $L_1, L_2, ..., L_n$ be the classes of log. Let $C_1, C_2, ..., C_m$ be the classes of data. Now, a general logging scheme for the different classes of data can be expressed as a set of 2-tuples $\{(c_1, l_1), (c_2, l_2), ..., (c_q, l_q)\}$ where c_i is a set of data classes $\{C_i, C_j, ..., C_k\}$ such that if $\exists C_k \in c_i$ then $C_k \notin c_j$, $\forall i, j,\ i \neq j, 1 \leq i, j \leq q$ and l_i is a log class L_k where $1 \leq k \leq n$ Also, if $l_i = l_j$, $i \neq j$, then the *physical* logs of c_i and c_j are separate. Informally,

- A set of data classes (c_i) is associated with one log class (L_k).
- One data class cannot belong to two different sets c_i and c_j, i.e., a data class cannot be logged in two different ways.
- If two different sets of data classes, say c_i and c_j are associated with the same class of log ($l_i = l_j$), say L_k, then the *physical* logs in which the log records for c_i and c_j are written are different. Also, the physical log to which the data classes that belong to a particular set of data classes c_i is written is the same.

For the sake of simplicity and without loss of generality, the following specific log and data classes are considered in the rest of this chapter. The first class, referred to as EL, is ephemeral logging where the data and log are NVHSS resident, with Force policy and object level locking. There are two subclasses of EL, namely, EL_s and EL_n that refer to EL with Steal and EL with No Steal, respectively. The second class, referred to as PL, is permanent logging where the data and log are disk resident with No Force/Steal policies and object level locking. Adding NVHSS to the second class can give rise to other subclasses of logging and recovery algorithms where either the data is NVHSS resident or the log is NVHSS resident or both are NVHSS resident.

A system can have combinations of the EL and PL algorithms where some classes of data have EL and some PL. Also, two primary equivalence classes of data with respect to logging and recovery are proposed. The first class consists of $critical$, $temporal$, *hot* and $cold$ items that are accessed by $critical$ or $high\ priority$ transactions. The second class consists of $cold$ data items accessed by $low\ priority$ transactions. It is possible to have different sets of equivalence classes in practice. This proposed classification, apart from being a pedagogical aid, is also a reasonable way to classify data in RTDBs. The claim that the classification is reasonable will be strengthened, if the first class of data items comprises a small percentage of the total data items in the database, which is a reasonable assumption and a necessary one for the ARUN suite of algorithms to perform well with limited amount of NVHSS. For instance, the access skew in TPC-C benchmarks is such that 84% of the access go to about 20% of the tuples and in terms of page accesses 75% of the page accesses go to 25% of the pages ([23]). Also, given the application characteristics, data types and transaction types, coming up with these equivalence classes in itself is a non-trivial topic of research. For ease of exposition, the first data equivalence class is referred to as Hot and the second equivalence class is referred to as $Cold$.

Traditional logging scheme can be expressed as $\{(\{Hot, Cold\}, PL)\}$, i.e., permanent, combined disk based log, with Steal No Force policies and object level locks for disk resident data items for any data type and transaction type.

The next three sections present the RTDB logging and recovery algorithms that have been developed with the principles discussed in section 2. as the guidelines. The algorithms are SPLIT, a logging and recovery algorithm for RTDBs without NVHSS, and General ARUN (g-ARUN), Real-Time ARUN (rt-ARUN) and Hybrid Real-Time ARUN (hrt-ARUN) that belong to ARUN, a suite of logging and recovery algorithms for RTDBs with NVHSS.

4. SPLIT ALGORITHM

Logging in SPLIT can be expressed as $\{(\{Hot\}, PL), (\{Cold\}, PL)\}$. Essentially, SPLIT uses separate permanent sequential logs for Hot and $Cold$ items. In SPLIT, there are separate, permanent sequential logs for *Hot* and *Cold* data items. The log for *Hot* items is physically realized by having two logs, namely, *HotUndoLog* and *HotRedoLog*. The reasons behind this split are:

- The amount of log generated by *Hot* items will be usually higher than the amount of log generated for *Cold* items. Hence, splitting the *Hot* log further will enhance partitioned and parallel logging.
- To show how a log can be split into undo and redo parts could prove useful. For instance, if page level locks are used with No Steal policy for Hot items, then *HotUndoLogs* need not be flushed but used only for transaction aborts.

The logging technique for PL class log assumes page level redos and operation level undos with compensation logs. This is similar to ARIES, a logging and recovery algorithm used in commercial databases ([4]).

4.1 DATA STRUCTURES

In this subsection, details of the data structures that are used in the SPLIT algorithm are presented. It should be noted that some of the data structures that are presented here are also used in the ARUN suite of algorithms. Detailed pseudo-code of all the algorithms are presented in [18].

Following are some of the important attributes of a permanent, sequential log:

- $currLSN$ - The Log Sequence Number (LSN) of the log record that is currently being used.
- $stableLSN$ - The LSN of the log record until which the log has been stabilized.

The important fields of a log record are as follows:

- $Type$ - Type of the log record. It can be $compensation$, $update$, or commit related such as $commit$, $abort$ or $prepare$.
- t - Identifier of the transaction that wrote the log record.
- ob - Identifier of the object to which the updates of this log record were applied.
- $updt$ - The operation responsible for the log record.
- $Undo\_Details$ - Details of the undo operation to be performed. Present only in $update$ type log records.
- $Redo\_Details$ - Details of the redo operation to be performed. Present only in $update$ type log records.

The $XTable$ entry consists of the following fields:

- t - Identifier of the transaction that is represented by the entry.
- $State$ - State of the transaction that could be $active$, $prepare$ or $commit$.

Some of the important fields in the transaction data structure are

- $hotUndoLastLSN$ - The LSN of the latest log record written by the transaction in the $HotUndoLog$.
- $hotRedoLastLSN$ - The LSN of the latest log record written by the transaction in the $HotRedoLog$.
- $coldLastLSN$ - The LSN of the latest log record written by the transaction in the $ColdLog$.
- $hotUndoNextLSN$ - The LSN of the next record to be processed during rollback in the $HotUndoLog$.
- $coldUndoNextLSN$ - The LSN of the next record to be processed during rollback in the $ColdLog$.

The commit log record for SPLIT consists of the following important fields:

- $hotRedoPrevLSN$ - The LSN of the previous log record written by the transaction in the $HotRedoLog$.
- $hotUndoLastLSN$ - The LSN of the last log record written by the transaction in the $HotUndoLog$.

- $coldLastLSN$ - The LSN of the last log record written by the transaction in the $ColdLog$.

4.2 NORMAL PROCESSING

Below is a description of how different events are processed in SPLIT. When a transaction begins execution, the transaction is added to the $XTable$. When a transaction updates an object, depending on whether the object is *Hot* or *Cold*, the corresponding undo and redo log records are added to $HotUndoLog$ and $HotRedoLog$ or $ColdLog$, respectively. On commit of a transaction, a commit record is written to the $HotRedoLog$ and the Write Ahead Log (WAL) protocol is followed where the log records are forced to stable $HotRedoLog$ and $ColdLog$. If there are *Hot* objects whose uncommitted changes could migrate to disk then the $HotUndoLog$ is also flushed. In case of an abort of a transaction, the data updated by the transaction is undone, compensation log records (like in ARIES) are written, an abort record is written and finally the log is flushed. The compensation log Record (CLR) will be written both in the $HotRedoLog$ and *HotUndoLog*. The redo part will be written in the $HotRedoLog$. The CLR written in $HotUndoLog$ is similar to that of ARIES. When there is a disk flush of an object then WAL protocol is followed where the log records are flushed to the log depending on the class of the object.

4.3 CRASH RECOVERY

In SPLIT, crash recovery is done in two phases where each phase has a forward and backward pass. In the first phase, the data items belonging to *Hot* are recovered by processing the redo and undo logs (*HotRedoLog*, *HotUndoLog*). In the second phase, $ColdLog$ is processed to recover data items belonging to $Cold$. The forward pass of the first phase consists of analysis and redo of $HotRedoLog$. The backward pass consists of undoing the Hot data items using the $HotUndoLog$. The second phase consists of two passes (like ARIES), a forward pass of $ColdLog$ where all updates are redone and a backward pass where the losers are undone.

Two transaction lists that are maintained during recovery, namely, *Winners*, *Losers*. The $Winners$ consists of all transactions that are committed and have to be redone, and the $Losers$ consists of transactions that have to be undone, i.e., active transactions that did not have a chance to commit (or abort) at the time of crash. Before the first phase of recovery starts, $Winners = Losers = \phi$. During phase 1, if a transaction begin record is encountered, an entry is made in the $XTable$, and the transaction is added to the $Losers$. If a redo record of an update on an object is encountered that redo is done. If a commit or abort record of a transaction is encountered then the transaction is deleted from the $Losers$. This $Losers$ list is used in the backward passes of both the phases.

SPLIT algorithm can be easily extended to accommodate more data classes. For instance, say, if $critical$ data has to be a separate class, then the data structures will have to be augmented with $criticalLSN$ and $CriticalLog$. The order of recovery will be to first process the $CriticalLog$, then process the $HotLog$ and finally process the $ColdLog$. The only constraint is that the commit record should be written to the log that will be recovered first because the $Losers$ list is necessary for the backward pass (undo recovery). Also, SPLIT algorithm will perform well only if there are multiple log disks because the advantage of sequential disk access might be lost if there are multiple logs and a single disk.

5. GENERAL ARUN (G-ARUN)

The previous section discussed SPLIT, a logging and recovery algorithm for disk resident databases that permits parallel and partitioned logging and recovery. This section presents g-ARUN, an algorithm that belongs ARUN suite of algorithms with ephemeral logging that is facilitated by the presence of NVHSS. In all the ARUN algorithms that are presented, the $XTable$ is also made NVHSS resident, thus making it easier to find out during crash recovery which transactions were active before the crash. The $XTable$ entry can be used as the commit record, if necessary.

5.1 NORMAL PROCESSING

As seen in section 3., logging in g-ARUN can be expressed as $\{(\{Hot, Cold\}, EL_s)\}$. In g-ARUN, there is one combined ephemeral transaction undo log ($undoLog$) per transaction for both $Cold$ and Hot items. When a data item is flushed, the undo log of all transactions that accessed this data item will be flushed to NVHSS.

All the functions in the ARUN suite take the log combination type as a parameter. For g-ARUN the log combination type is *combined* and for rt-ARUN and hrt-ARUN the log combination type is *split*. In g-ARUN, like in SPLIT, when a transaction begins execution, the transaction is added to the $XTable$. When a transaction updates an object, since the log combination type is *combined*, an undo log record is written to the transaction's ephemeral undo log, i.e., $undoLog$. At commit, the objects updated by the transaction are atomically flushed to NVHSS. In case of an abort of a transaction, the data updated by the transaction is undone using the $undoLog$.

5.2 CRASH RECOVERY

In ARUN, crash recovery is done by loading the $XTable$ from NVHSS and undoing all transactions that were $active$ at the time of crash . The $active$ transactions are undone using the transaction specific $UndoLog$.

The problems with g-ARUN are

- While recovery can be made cognizant of transaction priority, i.e., undoing of transactions can be according to transactions' priorities, it is very difficult to make the recovery cognizant of data priority. For instance, to recover the *Hot* items first, one has to process all the log records of a transaction.
- Hot items could be stolen to disk. This implies that at the time of recovery Hot items might have to be read from disk to be undone.
- Ephemeral logging means that there is no system-wide log to keep track of what happened.
- Forcing all the data items to NVHSS implies a large NVHSS. Also, there has to be a flusher daemon process to flush the pages from NVHSS to Disk. If this flusher is not aware of *Hot* and *Cold* classes then during recovery the disk might have to be accessed to recover *Hot* data items that the flusher might have flushed to make space in the NVHSS.

These considerations motivate rt-ARUN.

6. REAL-TIME ARUN (RT-ARUN)

Logging in rt-ARUN can be expressed as $\{(\{Hot\}, EL_n), (\{Cold\}, EL_s)\}$. There are two ephemeral transaction based undo logs for each transaction, namely, $hotUndoLog$ and $undoLog$ for *Hot* and *Cold* items, respectively. Basically, the $undoLog$ that was used in g-ARUN is used for *Cold* data items and there is a separate $hotUndoLog$ for *Hot* data items. When a data item is flushed, depending on its type, the corresponding undo log of all the transactions that accessed this data item will be flushed to NVHSS.

The log combination type for rt-ARUN is *split*. In rt-ARUN, like in SPLIT, when a transaction begins execution, the transaction is added to the $XTable$. When a transaction updates an object, since the log combination type is *split*, depending on the class of the object, an undo log record is written either to $undoLog$ or $hotUndoLog$. In case of a commit, the objects updated by the transaction are atomically flushed to NVHSS. In case of an abort of a transaction, the data updated by the transaction is undone using the $undoLog$ and $hotUndoLog$.

Algorithm rt-ARUN, like g-ARUN, suffers from the problem of not having a system-wide log to keep track of what happened to the system and the problem of requiring a large NVHSS to force all the data items.

7. HYBRID-REAL-TIME ARUN (HRT-ARUN)

Logging in hrt-ARUN can be expressed as $\{(\{Hot\}, EL_n), (\{Cold\}, PL)\}$. In this case, the NVHSS need not be as large as in the case of g-ARUN or rt-ARUN. If the *Hot* items form a reasonably small subset of the database, then the amount of NVHSS to force the data items need not be very large. Also, the permanent, sequential $ColdLog$ can act as a system-wide log where the important details of the high priority transactions can also be logged. This will be an approximate log for *Hot* data items in the sense that not all committed details about them will be available, because this $ColdLog$ will not be flushed when *Hot* data items are committed or when high priority transactions commit.

7.1 NORMAL PROCESSING

The log combination type for hrt-ARUN is *split*. In hrt-ARUN, like in SPLIT, when a transaction begins execution, the transaction is added to the $XTable$. When a transaction updates an object, the Hot items are processed as in rt-ARUN and the $Cold$ items are processed as in SPLIT. In case of a commit, the $ColdLog$ is flushed as in SPLIT and the Hot objects are atomically flushed to NVHSS as in rt-ARUN. In case of an abort (Hybrid-ProcessAbort), the transaction based $hotUndoLog$ is used to undo the Hot items and the system-wide $ColdLog$ is used to undo $Cold$ items.

7.2 CRASH RECOVERY

Crash recovery in hrt-ARUN is done in two phases. The first phase consists of recovering the Hot items using the transaction based $hotUndoLog$s of all active transactions. The second phase, that is similar to the second phase of SPLIT, consists of two passes, a forward and a backward pass, where the $ColdLog$ is used recover the $Cold$ items.

8. SUMMARY AND FUTURE WORK

In many real-time applications, a considerable amount of data is kept in main memory. However, even though main memory costs are dropping and sizes are increasing, the sizes of the databases are also increasing very rapidly. As in [24], we believe that the assumption that the database is only partially main memory resident must underlie the design of a practical database system.

Motivated by this assumption, this chapter showed the need to design novel logging and recovery algorithms for disk resident RTDBs by showing that conventional logging and recovery algorithms are not suitable in a priority oriented RTDB setting. We presented a taxonomy of data characteristics and proposed two data classes that are derived from data types and transaction types. Based on a set of driving principles, we methodically developed a suite of algorithms targeted at real-time databases.

SPLIT, a logging and recovery algorithm for disk-resident RTDBs without NVHSS, was also presented. The main idea behind SPLIT is to partition the log based on data classes thus enhancing parallel and partitioned logging and recovery. Also, we showed the need to design RTDBs with NVHSS which is capable of reducing unpredictability and improving the i/o performance in RTDBs. This was shown by exploring how NVHSS impacts logging and by presenting ARUN, a suite of novel logging and recovery algorithms that uses NVHSS to facilitate ephemeral logging where there are separate logs for each transaction that can be logged and recovered in priority order and discarded when the transactions complete. Three specific algorithms were presented, namely, g-ARUN, rt-ARUN and hrt-ARUN. While g-ARUN and rt-ARUN use pure ephemeral logging, hrt-ARUN uses both ephemeral logging and permanent, sequential logging. Details of correctness and performance analysis of the above algorithms are presented in [18].

There are a number of research problems to be investigated:

- Performance Evaluation: Currently, the proposed algorithms are being implemented on RADEx (Real-Time Active Database Experimental System) for the purpose of experimental evaluation.
- Dynamic Classes: The data classification is static and hence the algorithms based on this classification is also static. What happens if a data item migrates from one class to another is an issue to be considered for future work.

References

[1] L. Cabrere, J. A. McPherson, P. M. Schwarz, and J. C. Wylie, "Implementing atomicity in two systems: Techniques, tradeoffs and experience," *IEEE Transactions on Software Engineering*, vol. 19, Oct. 1993.

[2] K. Elhardt and R. Bayer, "A database cache for high performance and fast restart in database systems," *ACM TODS*, Dec. 1984.

[3] T. Haerder and A. Reuter, “Principles of transaction oriented database recovery,” *ACM Computing Surveys*, vol. 15, Dec. 1983.

[4] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, “Aries: A transaction recovery method supporting fine granularity locking and partial rollbacks using write-ahead logging,” *ACM TODS*, vol. 17, Mar. 1992.

[5] A. Reuter, “Performance analysis of recovery techniques,” *ACM TODS*, Dec. 1984.

[6] A. Amman, M. Hanrahan, and R. Krishnamurthy, “Design of a memory resident dbms,” in *Proceedings of IEEE Compcon*, Feb. 1985.

[7] R. Hagmann, “A crash recovery scheme for a memory resident database system,” *IEEE Transactions on Computers*, Sep. 1986.

[8] GTE-Laboratories, “White paper: Network management on the web (nemow) with tonics,” tech. rep., GTE-Laboratories, 1998.

[9] J. A. Stankovic, “Misconceptions about real-time computing,” *IEEE Computer*, vol. 21, Oct. 1988.

[10] R. K. Abbott and H. Garcia-Molina, “Scheduling real-time transactions: A performance evaluation,” *ACM Trans. on Database Systems*, Sep. 1992.

[11] J. R. Haritsa, M. J. Carey, and M. Livny, “Earliest deadline scheduling for real-time database systems,” in *Proceedings of the Real-Time Systems Symposium*, Dec 1991.

[12] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, “Experimental evaluation of real-time transaction processing,” in *Proceedings of the Real-Time Systems Symposium*, pp. 144–153, Dec. 1989.

[13] R. M. Sivasankaran, J. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham, “Priority assignment in real-time active databases,” *VLDB Journal*, vol. 5, Jan. 1996.

[14] R. Agrawal and D. J. Dewitt, “Recovery architectures for multiprocessor database machines,” in *ACM SIGMOD, International Conference on Management of Data*, 1985.

[15] R. M. Sivasankaran, K. Ramamritham, J. Stankovic, and D. Towsley, “Data placement, logging and recovery in real-time active databases,” in *International Workshop on Active and Real-Time Database Systems*, Jun. 1995.

[16] J. Huang and L. Gruenwald, “Impact of timing constraints on real-time database recovery,” in *Workshop on Databases: Active and Real-time*, Nov. 1996.

[17] R. Agrawal and H. V. Jagadish, “Recovery algorithms for database machines with non-volatile memory,” in *Database Machines, Proceedings of Sixth International Workshop IWDM*, June 1989.

[18] R. M. Sivasankaran, K. Ramamritham, and J. Stankovic, "Logging and recovery alogorithms for real-time databases (being written)," tech. rep., University of Massachusetts, Technical Report, 2000.

[19] J. Boak, "Brute force: Improving oracle performance using solid state disk," tech. rep., Database Excelleration Systems, 1996.

[20] M. H. Eich, "Main memory database research directions," in *Data base Machines, Proceedings of Sixth International Workshop IWDM*, June 1989.

[21] E. Rahm, "Performance evaluation of extended storage architectures," in *Proc. ACM SIGMOD Conference*, 1992.

[22] A. P. Sheth, M. Rusinkiewicz, and G. Karabatis, "Using polytransactions to manage interdependent data," in *Database Transaction Models for Advanced Applications*, 1995.

[23] T. Tsuei, A. N. Packer, and K. Ko, "Databse buffer size investigation for oltp workloads," in *ACM SIGMOD, International Conference on Management of Data*, 1997.

[24] E. Levy and A. Silberschatz, "Incremental recovery in main memory database systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, Dec. 1992.