[SZ90] E. Shekita and M. Zwilling. Cricket: A Mapped Persistent Object Store. In *Proc. of the Persistent Object Systems Workshop*, Martha's Vineyard, MA, September 1990.

[TSP92] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1992.

[WD94] Seth J. White and David J. DeWitt. Quickstore: A high performance mapped object store. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, 1994.

[WHBM90] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 109–123, June 1990.

[LGA96] Daniel F. Lieuwen, Narain Gehani, , and Robert Arlein. The Ode active database: Trigger semantics and implementation. In *Proc. Data Engineering*, February–March 1996.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *CACM*, 34(10):50–63, October 1991.

[Lom92] D. Lomet. MLR: A recovery method for multi-level systems. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 185–194, 1992.

[LSC92] T. Lehman, E. J. Shekita, and L. Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[MHL+92] 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 Transactions on Database Systems*, 17(1):94–162, March 1992.

[ML82] U. Manber and G.D. Ladner. Concurrency control in dynamic search structures. *ACM Proc.on Database Systems*, pages 268–282, April 1982.

[ML92] C. Mohan and F. Levine. ARIES/IM an efficient and high concurrency index management method using write-ahead logging. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, June 1992.

[Moh90] C. Mohan. ARIES/KVL: A key-value locking method for concurrencty control of multiaction transactions operating on btree indexes. In *IBM Almaden Res.Ctr, Res.R. No.RJ7008, 27pp.*, March 1990.

[ÖRS+96] B. Özden, R. Rastogi, A. Silberschatz, P. S. Nararyan, and C. Martin. The Fellini multimedia storage server. In S. M. Chung, editor, *Multimedia Information Storage and Management*. Kluwer Academic Publishers, 1996.

[SGM90a] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[SGM90b] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, 1990.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *Proc. Fifth Int'l. Workshop on Persistent Object Systems*, September 1992.

[SS91] Mark Sullivan and Michael Stonebreaker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proc. of the Int'l Conf. on Very Large Databases*, pages 171–179, 1991.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[Hag86] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–847, September 1986.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[Her88] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. Technical report, CMU, TR-CS-88-140, May 1988.

[Her89] Maruice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, March 1989.

[JLR+94] H.V. Jagadish, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In *Proc. of the Int'l Conf. on Very Large Databases*, 1994.

[JSS93] H.V. Jagadish, Avi Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Proc. of the Int'l Conf. on Very Large Databases*, 1993.

[KL80] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

[KS91] H. Korth and A. Silberschatz. *Database System Concepts*. McGrawHill, (second edition), 720pp., 1991.

[LC86a] T.J. Lehman and M.J. Carey. Query processing in main memory database management system. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 239–250, 1986.

[LC86b] T.J. Lehman and M.J. Carey. A study of index structures for main memory database management systems. In *Proc. of the Int'l Conf. on Very Large Databases*, pages 294–303, August 1986.

[LC87] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 104–117, 1987.

[LE93] X. Li and M. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *Proc. IEEE CS Intl. Conf. on Data Engineering*, April 1993.

[BBG+90] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufman, 1990.

[BHM90] P.A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, May 1990.

[BLR+95] P. Bohannon, D. Leinbaugh, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. Technical Report 113880-951031-12, Lucent Technologies Bell Laboratories, Murray Hill, 1995.

[BLS+95] P. Bohannon, D. Lieuwen, A. Silbershatz, S. Sudarshan, and J. Gava. Recoverable user-level mutual exclusion. In *Proc. 7th IEEE Symposium on Parallel and Distributed Processing*, October 1995.

[BP93] Alexandros Biliris and Euthimios Panagos. EOS User's Guide, Release 2.0.0. Technical report, AT&T Bell Labs, 1993. BL011356-930505-25M.

[BPR+96] P. Bohannon, J. Parker, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Distributed multi-level recovery in main memory databases. Technical Report 1125300-96-0227-01TM, Lucent Technologies Bell Laboratories, Murray Hill, 1996.

[CDRS89] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Storage management for objects in EXODUS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts and Databases*. Addison-Wesley, 1989.

[DKO+84] D. J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 1–8, Boston, Mass., June 1984.

[Eic89] M.H. Eich. A classification and comparison of main memory database recovery techniques. In *Proc. of the IEEE Conference on Data Engineering*, page 332, Los Angeles, CA., February 1989.

[FNPS78] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing — a fast access method for dynamic files. *IBM, Res.R. RJ2305.*, July 1978.

[GL92] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. In *Proc. of the Int'l Conf. on Very Large Databases*, pages 533–544, August 1992.

[GLPT76] J.N. Gray, R.A. Lorie, G.F. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency in a shared database. In G.M. Nijssen, editor, *Modeling in Data Base Management Systems*, pages 365–394. North-Holland, Amsterdam, 1976.

9 Conclusion

We have presented a detailed overview of the architecture of the Dalí Main-Memory Storage Manager. To our knowledge, Dalí is the only main-memory storage manager tuned for fine-grained concurrency and small transactions. Also, to our knowledge, it is the only explicit implementation of multi-level recovery for main-memory, and one of very few for disk-based systems. We have described the storage architecture for Dalí, and the implementation of the T-tree and extendible hash index structures. We have presented an overview of our multi-level concurrency control and recovery services, and described how the design of these services allows for minimal conflict with running transactions – in particular through the use of fuzzy checkpoints and through techniques for physical versioning of index structures. We have also described the extensive features for detection of bad writes by processes, and for recovery from process failure. We have briefly described the two databases currently built on Dalí, the Dalí Relation Manager, and the MM-Ode main-memory object-oriented database.

With the exception of physical versioning and the extendible hash, which are actively being added to the system at this time, all features of the design described here are implemented in the current version of Dalí at Bell Laboratories. Our future work includes logical versioning at the relational level, and data-shipping distributed versions based on the shared disk or client-server model.

Acknowledgements

We would like to thank H.V. Jagadish for significant early contributions to Dalí. Steve Coomer suggested the strategy of using codewords to protect checkpoint images on disk. Dennis Leinbaugh suggested the structure of the free tree. We would like to thank Jerry Baulier for his support of the project. We would also like to thank the following talented individuals who have contributed to design and implementation of specific systems in Dalí over the last three years: Soumya Chakraborty, Ajay Deshpande, Sadanand Gogate, Chandra Gupta, Sandeep Joshi, Amit Khivesara, Sekhara Muddana, Mike Nemeth, James Parker, and Yogesh Wagle.

References

[AG89] Rakesh Agrawal and Narain Gehani. Ode (object database and environment): the language and the data model. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 36–45, Portland, OR, May 1989.

[AGGL96] R. Arlein, J. Gava, N. Gehani, and D. Lieuwen. Ode 4.2 user manual. Included in distribution at `ftp://research.att.com/dist/ode` in the `doc` directory, 1996.

[AHU74] A. Aho, J. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AP92] A. Analyti and S. Pramanik. Fast search in main memory databases. *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, 1992.

second database management system built on Dalí is a main memory version of the ODE Object-Oriented Database. MM-ODE includes a compiler (O++) which supports a small superset of the C++ syntax. We now provide a brief overview of these high-level interfaces to Dalí.

8.1 Dali Relational Manager

The Dalí Relational Manager is a C++ class library interface to a relational system with SQL support limited to definition statements. Access to data is through C++ classes, corresponding to tables, iterators, search criteria, etc. Schema information is stored in tables, and limited views (projection only) are allowed. Indices may be created on arbitrary subsets of the attributes in a table. Referential integrity is supported (foreign key constraints), as are null values. Locking strategies avoid "phantom" anomalies (see e.g. [KS91]).

Navigation is supported through iterators over a single table. A conjunctive query may be specified for the iterator, and automatic index selection is performed.

The one extension to the relational model is that inter-table joins may be stored in the schema. From one open iterator, a new iterator on the matching tuples in the other table may be easily opened. This join relationship may be *materialized* leading to underlying pointer list structures similar to a network database. This last feature is required for the relational interface to allow navigation which competes with object-oriented models without explicit pointer types.

Building this interface has served the dual purposes of providing a higher-level interface for users and serving as a validation of the storage manager functionality. The interface described here took approximately one staff-year to produce, which we feel is very reasonable for a full-featured system, and illustrates the leverage gained from a good storage manager.

8.2 MM-Ode Object Oriented Database

MM-Ode<Dali>, also known as MM-Ode, is the main-memory version of the Ode object-oriented database system. It is built using the Dali main memory storage manager. MM-Ode supports a user interface identical to that provided by Ode<EOS>, also known as Ode, an object database described in [AG89, AGGL96, LGA96] The primary interface for both database management systems is the database programming language O++, which is based on C++. A few facilities have been added to C++ to make it suitable for database applications. O++ provides facilities for creating persistent objects which are stored in the database and for querying the database by iterating over these objects. Navigation is supported through pointers to persistent objects. It also has support for versioned objects. Indexes can also be built to speed-up object access. The run-time system checks for the existence of an index relevant to each query that may benefit from index use. The most recent release of MM-Ode allows triggers to be associated with objects.[8] Use of Ode allows applications to be used on both a disk-based (Ode) and a main-memory (MM-Ode) database. Recompilation is the only porting effort required. MM-Ode programs are often significantly shorter and easier-to-understand than the corresponding Dali program. However, this convenience comes at a significant performance cost. We are looking into reducing these costs.

[8] For more information on MM-Ode, see `http://www-db.research.att.com/ode-announce.univ.html`.

the following is true.

1. Search value is bounded by N, and either the version number of N is unchanged or N does not contain a key value between search value and target value.

2. Search value is less than the smallest key in N, and the version number of N has not changed, and N has no left child (thus, no new key between search value and target value could have been inserted, and target value itself, could not have been deleted) .

3. Search value is greater than largest key in N, and the version number of N has not changed, and N has no right child.

In case validation fails, the lock on the target key is released and the search is resumed from the most recently visited node in the stack whose version number is the same as that noted in the stack. The intuition for this is based on the observation that no target key could "escape" from a subtree without modifying, and therefore changing the version number of the root of that subtree. Note that restarting as described above implies that termination of the algorithm is probabilistic, but this is true of every scheme that follows the unlatch-lock-validate model [Moh90, ML92]. Successive key values in a range are obtained by repeatedly invoking the search procedure with the key value returned by the previous invocation and the stack at the end of the previous search.

Inserts and deletes on the T-tree are implemented as described in the overview above. Inserts on the tree invoke the search procedure to obtain a short duration (exclusive lock) lock on the key value larger than the key being inserted to ensure that no scans are in progress (this is referred to as next-key locking [Moh90]). Once this is done, modifications to T-tree nodes and rotations are performed while holding the T-tree latch in exclusive mode. Updates also increment the version number for any node which has been changed (note, however, that changing balance information does not require changing the version number). [[[fix next sentence]]] <== Deletes, on the other hand, obtain an additional transaction duration lock on the key value larger than the key value being deleted. This lock (combined with next-key locking on inserts) ensures the ability to abort the transaction by reinserting the missing key, and also prevent scans from proceeding past the deleted key, avoiding the phantom problem. Once this next-key lock is acquired which, the key delete on the node is performed while holding the T-tree latch in exclusive mode. Both inserts and deletes on the T-tree are treated as operations and multi-level recovery techniques described in Section 4 are employed for maintaining consistency in the presence of system crashes. For example, the undo operation for an insert is a delete, but any rotations caused by the insert are not necessarily undone by the delete (though new rotations may be caused).

8 Higher level Interfaces

There are currently two database management systems built on the Dalí Storage Manager. The Dalí Relational Manager is a C++ class library interface built on the relational model. This product retains the name Dalí, following our principle of offering multiple levels of interface. The

in the left subtree into the node, or by merging the node with its right child.

In both insert and delete, allocation/de-allocation of a node may cause the tree to become unbalanced and rotations (RR, RL, LL, LR) described in [LC86b] may need to be performed. (The heights of subtrees in the following description include the effects of the insert or delete.) In the case of an insert, nodes along the path from the newly allocated node to the root are examined until either 1) a node for which the two subtrees have equal heights is found (in this case no rotation needs to be performed), or 2) a node for which the difference in heights between the left and the right subtrees is more than one is found and a single rotation involving the node is performed. In the case of delete, nodes along the path from the de-allocated node's parent to the root are examined until a node is found whose subtrees' heights now differ by one. Furthermore, every time a node whose subtrees' heights differ by more than one is encountered, a rotation is performed. Note that de-allocation of a node may result in multiple rotations.

7.3.2 Concurrency control Issues in T-trees

We now describe the features of our implementation of T-trees in Dalí. We implement T-trees with a single tree latch which is obtained in shared mode for readers and exclusive mode for updaters. In our implementation, each node contains a *version number* which is incremented whenever the node is modified. Also, a stack is used for all operations on the T-tree. The stack stores the nodes visited during a traversal, whether the left or right child was taken when leaving the node, and the version number of the node when it was first encountered. In the following, a version number is said to have *changed* if the current version number in the node is different from the version number stored in the stack.

Our T-tree implementation supports next-key-locking [ML92, Moh90] to avoid the *phantom phenomenon* (see e.g. [KS91]). Thus, in our implementation, a key in the range of an in-progress scan cannot be inserted or deleted since this could result in a non-serializable execution. This is controlled by passing lock modes to each operation, so that transactions running at lower degrees of consistency can be mixed with those running at higher degrees [GLPT76].

Even though search operations on T-trees obtain the tree latch, concurrent inserters and deleters could cause problems. For example, consider searching for a key greater than a certain value. Once the target key has been found, a lock on it needs to be obtained. However, the lock request cannot be made while holding the latch since this could result in deadlocks involving latches and locks. As a result, the latch must be released before the lock request is made – this, however, opens up a window for inserters who could insert a key value in between the value being searched for and the target key determined by the search procedure. Thus, after the lock is obtained, the search procedure needs to perform a *validation* to ensure that the target key has not been invalidated due to a concurrently executing insert or delete.

In general, the search proceeds as described in Section 7.3.1 except that each node visited is pushed onto the stack. We describe only the search for a "greater-than" value, though all comparisons are supported in the implementation. After a lock on the target key is obtained, validation is performed. Let N be the top node on the stack (which is the last node seen in the search), and let search value be the key value on which the search was initiated and target value the key value locked just prior to the validation. Then validation is said to succeed if any of

7.3 T-tree Indexes

In [LC86b], the authors proposed T-trees as a storage efficient data structure for main memory databases. T-trees are based on AVL trees proposed in [AHU74]. In this subsection, we provide an overview of T-trees as implemented in Dalí. For a detailed description, the reader is referred to [BLR+95].

7.3.1 Overview of T-trees

We now describe the T-tree from [LC86b]. Like AVL trees, the height of left and right subtrees of a T-tree may differ by at most one. Unlike AVL trees, each node in a T-tree stores multiple key values in a sorted order, rather than a single key value. The left-most and the right-most key value in a node define the range of key values contained in the node. Thus, the left subtree of a node contains only key values less than the left-most key value, while the right subtree contains key values greater than the right-most key value in the node. A key value which is falls between the smallest and largest key values in a node is said to be *bounded* by that node. Note that keys equal to the smallest or largest key in the node may or may not be considered to be bounded based on whether the index is unique and based on the search condition (e.g. "greater-than" versus "greater-than or equal-to").

A node with both a left and a right child is referred to as an *internal node*, a node with only one child is referred to as a *semi-leaf*, and a node with no children is referred to as a *leaf*. In order to keep occupancy high, every internal node has a minimum number of key values that it must contain (typically $k - 2$, if k is the maximum number of keys that can be stored in a node). However, there is no occupancy condition on the leaves or semi-leaves.

Searching for a key value in a T-tree is relatively straightforward. For every node, a check is made to see if the key value is bounded by the left-most and the right-most key value in the node; if this is the case, then the key value is returned if it is contained in the node (else, the key value is not contained in the tree). Otherwise, if the key value is less than the left-most key value, then the left child node is searched; else the right child node is searched. The process is repeated until either the key is found or the node to be searched is null.

Insertions and deletions into the T-tree are a bit more complicated. For insertions, first a variant of the search described above is used to find the node that bounds the key value to be inserted. If such a node exists, then if there is room in the node, the key value is inserted into the node. If there is no room in the node, then the key value is inserted into the node and the left-most key value in the node is inserted into the left subtree of the node (if the left subtree is empty, then a new node is allocated and the left-most key value is inserted into it). If no bounding node is found then let N be the last node encountered by the failed search and proceed as follows: If N has room, the key value is inserted into N; else, it is inserted into a new node that is either the right or left child of N depending on the key value and the left-most and right-most key values in N.

Deletion of a key value begins by determining the node containing the key value, and the key value is deleted from the node. If deleting the key value results in an empty leaf node, then the node is deleted. If the deletion results in an internal node or semi-leaf containing fewer than the minimum number of key values, then the deficit is made up by moving the largest key

Other than our work, [AP92] describes a variation of extendible hashing for main memory which is related to constructing a trie on the hash value rather than having a single large table. This more complicated scheme significantly decreases space overhead, while keeping a single key compare to determine if the correct record has been found.

In contrast, we maintain the simple directory structure of [FNPS78], but avoid the space problems of this scheme by not splitting the structure on overflow of a single bucket. In [FNPS78], fixed bucket sizes were implied by disk page sizes, and exceeding this limit triggered the doubling of the directory. In fact, we do not use a fixed size bucket at all, but have a chain of key values for each bucket. We keep space overhead lower by basing the decision to double the directory size on an approximation of occupancy rather than on the local overflow of a bucket.

The advantage of a simple directory structure arises when implementing a concurrency control mechanism. While our implementation may suffer more key comparisons (which may or may not be more expensive than index node traversals in a main memory database), the flat directory structure leads to a simple concurrency control mechanism which nevertheless provides good concurrency.

Permitting insert, delete and find operations to execute concurrently could result in several problems. For example, suppose find for a key causes it to reach a hash header which is split due to a concurrently executing insert. In such a case, the key being search for may be transferred from the current list to a different list, and find would not be able to locate the key. In order to prevent the above problem, a lock is maintained with each hash header, and two directory locks, the find-directory-lock and the split-directory-lock, are maintained. The directory locks are obtained in exclusive mode when the directory is resized; the split-directory-lock is held while the new directory is being initialized while the find-directory-lock is held only when the pointer to the directory is toggled to point to the new larger directory.

Operator find begins by first obtaining the find-directory-lock in shared mode – this ensures that the directory stays stable during the find operation. It then obtains a lock on the hash header. Furthermore, in order to ensure that the hash header has not been split between the time it reached the hash header and the time it obtained a shared lock on it, it checks to see if the directory entry that initially pointed to its hash header is still unchanged. If this is the case, find can release the find-directory-lock and proceed; else, it releases the lock on the hash header and re-traverses the directory structure to reach the correct hash header. Operator Insert, on the other hand, obtains the hash header lock in exclusive mode. If it is decided to split the header, then a shared lock is obtained on the split-directory-lock to ensure that the directory is not being resized while the split is taking place. Since splits are an optimization, an insert may choose not to wait if this lock is held, allowing the next insert to perform the split.

Note that maintaining the find and split directory locks as separate locks ensures that find operations and normal insert operations (those that do not cause splits) are only blocked for the very short time required to toggle the directory pointer when a directory is being resized. Since the split-directory-lock is acquired in exclusive mode for the duration of the doubling, it ensures that two processes do not try to double the directory at the same time. Note that if a process cannot get this lock, it simply gives up, since another process is already accomplishing the doubling.

our structure appears in Figure 7. The structure matches that of [FNPS78], except that the bucket concept from standard extendible hashing is broken into a *hash header* and a list of *key entries*. Thus, each directory entry points to a hash header, and multiple directory entries can point to a single hash header. A variable i is maintained with the hash index, and the first i bits of the computed hash value for a key are used to determine the directory entry for the key (the directory itself contains 2^i entries). Allowing multiple directory entries to point to the same bucket prevents too many hash headers from being allocated for directory entries to which very few key values have hashed. With each hash header is stored a variable j which has the property that for all the keys whose directory entries point to the hash header, the first j bits of their computed hash value are equal. In addition, the number of directory entries that point to the hash header is given by 2^{i-j}.

Searching for a key value is fairly straightforward, and involves searching for the key value in the list of key entries pointed to by the directory entry for the key. Insertions, however, are more complex and could involve splitting individual key entry lists, and in certain cases doubling the entire directory. The algorithm attempts to keep lists below some threshold in size. If an insert pushes a list over the threshold, then that list is *split*. If j in the header for the list to be split is less than i, the list is immediately split into two lists based on the value of the first $j+1$ bits. Also, the 2^{i-j} entries pointing to the original list are modified to point to one of the two lists based on the first $j+1$ bits of the hash value the entry represents. After the split, the value of the variable j in the lock headers for the two lists is incremented by 1, and the keys in each list are again equal on the first j bits. Note that j also represents the number of times that the list has split.

The description of splitting above assumed that j in the hash header is less than i for the table. If it is found that j in that list's hash header is equal to i for the hash table then only one entry in the directory points to this list. In this case, a split requires that the directory structure be doubled in size. Our algorithm will forego this split, tolerating somewhat longer lists, until the *utilization factor* (keys/lists) of the hash table has exceeded a certain threshold. Once the threshold has been exceeded, the first attempted split that finds $j = i$ will double the directory structure, and then proceed with the split as above. Other lists whose lenght exceeds the threshold will be split by the next insert on that list.

The first step in doubling the directory is to allocate a new directory structure twice the size of the original. Directory entries in the new directory that are equal for the first i bits are set to point to the hash header pointed to by the directory entry in the old directory with the same value for the first i bits. The variable i is then incremented (and the original split is carried out). Deletes could similarly cause lists to be merged and possibly the directory size to be halved.

In the original proposal, the goal of the work was to provide fast location of records in a disk file. The "directory" size varied linearly with the number of keys in the table, while providing a guaranteed maximum of two disk reads. A similar approach may be worthwhile in main-memory, since keeping space overhead in line with usage is very important, as is speed of access. The cost of accessing a record consists of the cost of navigating the directory structure, followed by key comparisons between the search key and any keys on the overflow chain pointed to by the directory.

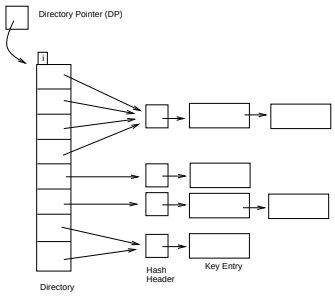

Figure 7: Extendible Hashing in Dalí

7.1 Heap File

The *heap file* is Dalí's abstraction for handling a large number of fixed-length data items. It is a thin layer provided on top of the power-of-two allocator and the segment headers provided by the allocation system. A separate chunk at the storage allocator level is created for each heap file, and fixed-length items are allocated from segments of the chunk. When creating a heap file, its itemsize – the length of objects in that heap file – is specified. All allocation is done by the underlying allocator, so that the associated logic and concurrency mechanisms are not disturbed. However, it is assumed by the heap file that the underlying allocator will allocate items on exactly itemsize-byte boundries. This is the case with the power-of-two allocator when the minimum allocation size is equal to the fixed size of the items. In addition to insertion and deletion, the heap file supports locking and an unordered scan of items.

Item locking is supported either via lock names (the lock name is derived from the database file and the offset of the item in the database file) or by allocating an array of lock headers for items in each segment. A pointer to this array of lock headers is stored in the segment header, and the page table (described in Section 3.3.2) is used to determine the lock header for an item. Item locks are obtained transparently when items are inserted, deleted, updated or scanned. Responsibility for implementing a lock which covers the entire heap file is *not* implemented in the heap file itself. This corresponds to the principle, which we have encountered repeatedly, that locking for a structure is best left to the encapsulating structure. This has been borne out by very significant difference in the locking needs of the relational and object-oriented databases implemented on Dalí, while the item locking needs are uniformly served by the heap file.

Scans are supported through a bitmap stored in the segment header for items in the segment. Bitmap entries for items that have been deleted from the heap file are 0. Heap file scans thus simply return items in the chunk for which the corresponding bitmap entry is 1. Note that zero entries in the bitmap mirror the allocator's free lists for that segment. The bitmap makes the process of determining valid records very efficient, and is necessary because information about allocated data is not stored by the allocator due to the implied space overhead for all allocated data.

7.2 Extendible Hash

Dalí includes a variation on Extendible Hashing as described in [FNPS78]. An overview of

a process from the list of potential owners when it dies or sets its wants variable to point to null or another lock. In all cases, a process must only advance by a few instructions to either register ownership, or notice that the cleanup-in-progress flag is set, and relinquish its interest in the latch. Should our list of potential owners become empty, we can conclude that the owner is dead. Further, if we reach this conclusion, and yet there is no *registered* owner for the latch, we may conclude that the owner had just acquired it or was about to release it, and thus the system structure guarded by the latch could not be in an inconsistent state. However, if during this cleanup process we find that a dead process is the registered owner, then the cleanup function associated with this latch must be called, as described in Section 5.1.

6.2 Locking System

Having discussed latching, we turn to locking, usually used as the mechanism for concurrency control at the level of a transaction. Locks are normally used to guard accesses to persistent data, and can support modes richer than shared and exclusive. At the lowest level, lock requests in Dalí are made on a lock header structure which simply stores a pointer to a list of locks that have been requested (but not yet been released) by transactions. A lock request that does not conflict with outstanding locks on the lock header is granted. Lock requests that conflict are handled by adding the requested lock to the list of locks for the lock header, and are granted when the conflicting locks are released. If a lock request cannot be granted within a certain interval (specified at system startup time), then the request simply times out. Users can thus pre-allocate lock headers and associate each lock header with a data structure to be locked.

Lock requests in Dalí can also be made on 64-bit lock names which are mapped internally by the Dalí system to lock headers via a hash table. The lock names save space at the cost of time since lock headers are dynamically allocated when locks are requested on a lock name. Blocks of lock nodes are allocated to a transaction when it begins, and locks requested by the transaction are quickly allocated from this set.

Lock modes are easily added to the system through the use of two boolean tables, conflicts and covers. The former is obvious, and the second is an optimization which indicates whether a holder of lock type A needs to check for conflicts when requesting a new lock of type B. If A covers B, then this check is unnecessary, since A conflicts with any lock mode B would conflict with.

Locks default to transaction duration, but may be requested for instantaneous duration, operation duration, or *post-commit* duration, in which case they ensure the ability to carry out a post-commit action.

7 Collections and Indexing

The storage allocator provides a low-level interface for allocating and freeing data items. Dalí also provides higher-level interfaces for grouping related data items, and performing scans as well as associative access on data items in a group.

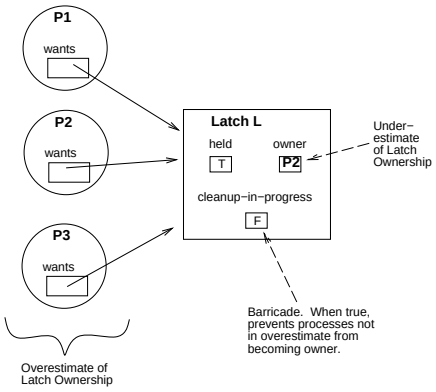

Figure 6: Overview of Latch Control Information

hold the latch.

As described in Section 5.1, a *cleanup server* handles recovery from process failure. The first step it takes on detecting the death of a process is to determine if that process held the latch, if any, pointed to by its wants field. If that process is also the owner of the latch, then the work is done. If it is not, the ownership is ambiguous, and the cleanup server must examine the wants field for all processes,

The set of processes that want the latch may, however, change even as the cleanup server attempts to determine which processes have set their "wants" field. To solve this problem, we introduce a flag associated with the latch called cleanup-in-progress, and forbid processes to attempt to get the latch if this flag is set. This flag provides a barrier which, when "raised" (set to True), prevents any new processes from becoming owners of the latch. The cleanup-in-progress flag for a particular latch is set by the cleanup process while it attempts to resolve the ownership of that latch. Without this "barricade," the (remote) possibility exists that one or more processes can repeatedly acquire and release the latch, always leaving the latch acquired but unregistered while its status is tested by the cleanup process. We cannot distinguish between this case and the death of a single process in an indeterminate state. The data structures used by our system latches are summarized in Figure 6. The method described in the rest of this section avoids these problems, and guarantees resolution of the latch if all live processes receive CPU time. (Actually, extensions to handle starved processes have also been developed and are used in Dalí.)

The acquisition protocol of a simple spin-lock implementation is modified so that the process starts by setting its wants field to point to the latch. It then checks cleanup-in-progress for the latch. If a cleanup is found to be in progress, wants is set back to null, and the process waits for the cleanup to end before retrying. Otherwise, it attempts to acquire the lock with a test-and-set instruction.

Given these additional tools, how does the cleanup server determine whether a dead process holds a latch? It starts by setting the cleanup-in-progress flag to True, then gathering a list of potential owners from the wants variables for each process. Now it becomes reasonable to wait until the situation resolves itself, as we must only wait for a finite number of processes to give up their interest in, or register their ownership of, the latch. We do this by removing

First, regardless of the type of atomic instruction available, the fact that the process holds or may hold a latch must be observable by the cleanup server (see Section 5.1) so that it can determine that a dead process held a system resource. This is implemented by maintaining an array of pointers to held (or possibly held) latches from a process's entry in the Active Process List.

However, if the target architecture provides only test-and-set or register-memory-swap as atomic instructions (as opposed to compare-and-swap or load-linked/store-conditional), then extra care must be taken to determine if the process did in fact own the latch. In this section, we provide an overview of the technique used to handle this in Dalí, which is based on [BLS+95].

To allow the cleanup server to determine ownership of the latch, a process having acquired a latch must also register itself as the owner, for example by writing its process identifier to a known location. Unfortunately, the act of acquiring the latch using the basic hardware instruction test-and-set (or register-memory-swap) cannot be used to also atomically register ownership. At best, the atomic instruction can be followed by a conditional branch testing for a successful acquisition, which can be followed by an instruction writing the process id of the new owner. If the process that is trying to acquire a latch is interrupted between the test-and-set and the write, the ownership of a latch is left in doubt until the process gets to execute the write. If the process fails in this interval, the ownership of the latch will never become clear. Worse still, it is impossible to distinguish between a process that has failed at this step and a process that has not failed, but has not yet carried out the write, either because it is servicing an interrupt, or because it has not been allocated CPU cycles. A symmetric problem can also arise when releasing the semaphore, since the de-registration and release may have to be accomplished using separate instructions (depending on the exact atomic instruction used).

Although acquisition and registration are separate operations and cannot be executed atomically, we now present an overview of the technique used in Dalí to examine the status of *other* processes that are attempting acquisition of a latch and thereby determine ownership.

We now present the intuition behind our approach. Details of our algorithms are presented in [BLS+95].

Consider an atomic test-and-set based implementation of a latch. The first and most obvious step in tracking ownership of such a latch is to require that a successful attempt to acquire the test-and-set latch be immediately followed by a write which stores the new owner's identifier (process or thread identifier, which we abbreviate to *process id*) in an owner field associated with the latch. Clearly, if these two steps were atomic, we could always find out which process currently owns the latch. (Note that if a compare-and-swap instruction is available, the two steps *can* be made atomic.)

Unfortunately, detection of ownership is complicated by the fact that processes may get interrupted, and may even be killed, in between acquisition and registration, and thus a latch may be in a state where it has been acquired by some process, but we do not know which one.

As a first step toward solving these problems, we require that all processes that are trying to acquire a latch keep a pointer to that latch in a per-process shared location. We call this location the process's wants field. The collection of all processes' wants fields provides us with an overestimate of the set of possible owners of the latch (there are zero or one owners, but an arbitrary number of "interested" parties). This helps establish a set of all processes that might

5.2.2 Codewords

The codeword strategy of error detection is to associate a logical parity word with each page of data. Whenever data is updated using valid Dalí system calls, the codeword is updated accordingly. An erroneous write will update only the physical data and not the codeword. We then use the strategy of protecting the checkpoint image on disk. Before writing a page to a checkpoint, its contents are verified against the codeword for that page. Should a mismatch be found, a simulated system crash is caused, and the database is recovered from the last checkpoint.

Our current implementation of codewords, based on page-level latching, is as follows. Each page has an associated latch and codeword. While updating a page, the latch for the page is held in shared mode by the updater. At the end, the change to the codeword for the page is computed from the current contents of the updated region, and the contents of the region before the update (this is determined from the physical undo log record for the update). A short term exclusive latch on the codeword table is then obtained to actually apply the computed change to the codeword value for the page. The latch ensures that concurrent updaters to different regions on a page do not install an incorrect value for the codeword. The checkpointer, on attempting to checkpoint a page, obtains an exclusive latch on the page long enough to copy it and the codeword associated with it to a separate area. The codeword for the copy is then computed and compared with the value from the table.

This implementation is the only use of page-level latching in Dalí. Note that we are currently designing a new scheme which uses the global redo log to avoid this page latching. We expect this will reduce blocking as well as the performance cost associated with the codeword scheme.

The advantages of codewords over memory protection are that lower overhead is incurred during normal updates, and it is less likely that an application error will escape detection. For example, an erroneous write to a page which has just been updated *will* be caught. The disadvantage is that erroneous writes are not detected immediately, making debugging based on this information difficult. (We do allow user-driven codeword audits to make debugging with codewords possible.)

6 Concurrency Control

In this section we describe the concurrency control facilities available in Dalí. These facilities include latches (low-level locks for mutual exclusion) and queueing locks. Our latch implementation is novel in its contribution to recovery from process failure, and our lock implementation makes the task of adding new locking modes trivially easy.

6.1 Latch Implementation

Latches in Dalí are implemented using the atomic instructions supplied by the underlying architecture. The decision to implement latches in Dalí was made since operating system semaphores are too expensive due to the overhead of making system calls. However, to support the process failure scenarios described above, there are certain issues that must be dealt with.

spawn multiple cleanup agents concurrently. This is required since a transaction for one dead process may have to wait for a resource held by another dead process in order for it to complete its abort. (Note that these locks must be at lower levels of abstraction than the transaction level, as a transaction level lock would have been held by the transaction until commit and thus not need to be reacquired.) If the dead process were threaded, the cleanup agent must include a separate thread for each active transaction for the same reason.

5.2 Protection from Application Errors

The direct access principle of Dalí implies that at least some user application code will be linked directly with Dalí libraries and access the data stored in the database through shared memory. With this comes the inevitable chance that an application error will lead to writes which can cause persistent data to become corrupted. Dalí provides two mechanisms for minimizing the probability that such errors will lead to corrupting persistent data. The first mechanism is *memory protection* and the second is *codewords*.

Both mechanisms are designed only to prevent updates which are not correctly logged from becoming reflected in the permanent database. This would occur if a database pointer used for direct memory access were used later for an update as if it were a normal pointer. Similarly, such a problem might occur due to a random garbage value in a pointer. This second case becomes much more probable as database sizes approach virtual memory address range sizes. This is very conceivable in modern 32-bit systems, where it is very reasonable to find multiple gigabytes of RAM.

These mechanisms do not protect from erroneous updates which follow the proper database conventions. Nor are they capable of protecting from *all* erroneous updates which do not use database conventions, and such limits are described for each scheme. However, these schemes vastly reduce the probability that a programming error can corrupt the persistent database. The codeword scheme also protects against bit errors in memory, which are a significant worry with gigabytes of RAM.

The schemes are independent, and can be used in conjunction.

5.2.1 Memory Protection

Applications wishing to prevent corruption due to stray pointers can map a database file in a special protected mode. For such database files, Dalí uses the mprotect system call to disable updates to the file by the process. Before a page is updated, when an undo log record for the update is generated, munprotect is called on the page. At the end of the transaction, all the unprotected pages are re-protected. Thus, an erroneous update would attempt to update a protected page, thus resulting in a protection violation. The advantage of this scheme is that erroneous writes are detected immediately, and can be traced to their source using a debugger. The disadvantage is that the system calls are a significant performance hit. As a result, this scheme is more beneficial when debugging applications, and for read-only databases or databases with low update rates. Note also that since unprotected pages stay unprotected until the end of transaction, erroneous writes in a transaction following a correct write to the same page will not be detected with this scheme.

5.1.1 Detecting Process Death

The first step in cleanup of crashed processes is to detect that a process crashed. When a process connects to the Dalí system, information about the process such as its operating system process identifier are noted in an Active Process Table in the system database. Dalí server processes also register themselves in the same table. When a process terminates its connection via DaliSys::close(), it is deregistered from the table. The cleanup process periodically goes through the table and checks (via the operating system) if each registered process is still alive.

If a registered process is found not to exist, cleanup actions have to be taken.

5.1.2 Low Level Cleanup

Once a dead process has been found, the cleanup process determines what low-level latches, if any, were held by the crashed process. Whenever a process acquires a latch, an entry is made in the Active Process Table.[7] This table is consulted by the cleanup process to determine if the process was holding a latch. The detection is complicated significantly by the fact that in many machine architectures it is not possible to atomically acquire a spin-lock and register ownership. A technique for getting around the problem is presented in Section 6.1.

If any system latches are held by the process, then a cleanup function associated with the latch is called. This is an example of a "functional recovery procedure" and may require that a few words of undo information be stored in the system structure. A special return code is reserved to indicate that the structure could not be repaired by this function. In this case, the system must simulate a full crash, causing a recovery which does not depend on the corrupted structure (or any other transient system data).

However, if the dead process did not hold any such latches, or if the latches it held were successfully cleaned up, then the cleanup server may proceed to the next phase which involves cleaning up transactions owned by the dead process.

5.1.3 Cleaning Up Transactions

Once low-level structures have been restored to a consistent state, the cleanup server spawns a new process, called a *cleanup agent*, to take care of cleaning up any transactions still running on behalf of this process. This amounts to scanning the transaction table, and aborting any in-progress transactions owned by the dead process, or executing any post-commit actions for a committed transaction which had not been executed.

Two subtle points arise here. First, the in-progress transaction may have already started an abort. Similarly, the dead process may have been executing its post-commit actions. Thus, the transaction must indicate during these activities (and in fact during all activities) whether the transaction table entry for that transaction is in a consistent state. If not, then that is handled as if the process held a low-level latch, which is described above.

The subtle point concerns the case that multiple processes have died, or the case that a new process dies while the old one was being cleaned up. In these cases, the cleanup server must

[7] These are stored with the process rather than the transaction to handle very low level latches, such as the one used to allocate transaction table entries.

A separate post-commit log is maintained for each transaction – every log record contains the description of a post-commit operation to be executed. These are appended to the system log immediately before the commit record for a transaction (when it pre-commits) or immediately before the operation commit record when the operation pre-commits. Once transaction commit/operation pre-commit completes, the post-commit operations are executed. Furthermore, the checkpoint of the ATT writes out post-commit log records along with undo log records for the transaction. Thus, for every committed transaction, post-commit log records for the transaction are contained on disk, in the log, and possibly also in the checkpointed ATT (in cases where post-commit log records in the log precede the begin-recovery point). As a result, during restart recovery, the post-commit operations can be determined and executed for transactions that were in the process of executing post-commit operations when the system crashed.

5 Fault Tolerance

In this section, we present features for fault tolerant programming in Dalí, other than those provided directly by transaction management. These techniques help cope with process failure scenarios. The first technique returns the system to a fully available state if a process dies with transactions in progress. The second and third techniques help detect and recover from user programs with "stray pointers" which might corrupt persistent data stored in shared memory.

5.1 Process Death

In this section, we discuss how Dalí handles "untimely" process death. This may be caused by the process violating hardware protection such as attempting to access invalid memory, or by a process being killed by an operator. In either case, we assume that the process did not corrupt any system control structures. Recovering from process death primarily consists of returning any shared data partially updated by the process to a consistent state. Since no volatile memory has been lost, this is in some ways easier than crash recovery. However, during crash recovery, one can assume that internal system structures (such as the transaction table and lock tables) are in a consistent state, as they are recreated on recovery. It is the lack of this low-level consistency which complicates process recovery.

Obviously, the main approach to handling a dead process is to abort any uncommitted transactions owned by that process. Also, in our system, for committed transactions and pre-committed operations, post-commit actions must also be executed on behalf of the process. However, this cannot be begun immediately upon determining that a process died. That process may hold latches on low-level system structures, such as the system log. Any attempt to abort a transaction would attempt to get that latch, causing the process attempting to clean up the resource to wait on the (dead) process it is trying to clean up. Thus, process recovery must be done carefully.

A Dalí system process known as the *cleanup server* is primarily responsible for handling the cleanup of dead processes. We now describe the actions taken by this server.

records in the transaction's undo log for the operation are replaced by a higher-level undo description.

Once all the redo log records have been applied, the active transactions are rolled back. To do this, all completed operations that have been invoked directly by the transaction, or have been directly invoked by an incomplete operation have to be rolled back. However, the order in which operations of different transactions are rolled back is very important, so that an undo at level L_i sees data structures that are consistent [Lom92]. First, all operations (across all transactions) at L_0 that must be rolled back are rolled back, followed by all operations at level L_1, then L_2 and so on.

Note that for certain uncommitted updates present in the redo log, undo log records may not have been recorded during the checkpoint – this could happen for instance when an operation executes and commits after the checkpoint, and the containing transaction has not committed. However, this is not a problem since the undo description for the operation would have been found in operation commit log records during the forward pass over the system log earlier during recovery. Any redo log records for updates performed by an operation whose commit log record is not found in the system log are ignored (since these must be due to a crash during flush and are at the tail of the system log).

4.8 Post-commit Operations

Some types of operations that a transaction may need to execute cannot be rolled back. For example, consider the deletion of a record from the database when physical pointers are employed. If the space for the record were de-allocated as part of the delete, then problems may be encountered during rollback if the transaction were to abort. The reason for this is that, for high concurrency, we need to permit storage allocation and de-allocation to continue once the space for the record was de-allocated but before the transaction (or operation) that de-allocated the space committed. As a result, the space may potentially be allocated by another transaction, making it impossible for the transaction that freed it to re-obtain it in case it were to abort. Thus, new storage space would need to be allocated for the record and old references/pointers to the record (e.g., in the index) may no longer be valid.

The above problem can be avoided by using the notion of *post commit operations*, that is, operations that are guaranteed to be carried out after the commit of a transaction or operation, even in the face of system/process failure. (Recoverable queues are used for the same purpose in other systems [BHM90].) Transaction operations that cannot be undone can be performed as post-commit operations, preserving the all-or-nothing property of the transaction. Thus, by executing the de-allocation of storage space for the record as a post-commit operation, we can permit high concurrency on the storage allocator (no transaction duration locks on the allocator are required), and at the same time, ensure that space gets de-allocated if the transaction commits, whether or not the system fails after the commit. This facility is also valuable for implementing externally visible writes, such as sending a message on commit of a transaction, which are especially important in work-flow situations. Similarly, the notion of post-commit operations can be extended to operations by permitting an operation at level L_i to require post-commit operations at level L_{i-1} to be executed once it pre-commits.

For any uncommitted update whose effects have made it to the checkpoint image, undo log records would be written out to disk after the database image has been written. This is performed by checkpointing the ATT after checkpointing the data; the checkpoint of the ATT writes out undo log records, as well as some other status information.

At the end of checkpointing, a log flush must be done before declaring the checkpoint completed (and consistent) by toggling cur_ckpt to point to the new checkpoint, for the following reason. Undo logs are deleted on transaction/operation pre-commit, which may happen before the checkpoint of the ATT. If the checkpoint completes, and the system then fails before a log flush, then the checkpoint may contain uncommitted updates for which there is no undo information. The log flush ensures that the transaction/operation has committed, and so the updates will not have to be undone (except perhaps by a compensating operation, for which undo information will be present in the log).

4.6 Abort Processing

When a transaction aborts, that is, does not successfully complete execution, updates/operations described by log records in the transaction's undo log are undone by traversing the undo log sequentially from the end. Transaction abort is carried out by executing, in reverse order, every undo record just as if the execution were part of the transaction.

Following the philosophy of *repeating history* [MHL+92], new physical-redo log records are created for each physical-undo record encountered during the abort. Similarly, for each logical-undo record encountered, a new "compensation" or "proxy" operation is executed based on the undo description. Log records for updates performed by the operation are generated as during normal processing. Furthermore, when the proxy operation commits, all its undo log records are deleted along with the logical-undo record for the operation that was undone. The commit record for the proxy operation serves a purpose similar to that served by *compensation log records* (CLRs) in ARIES – during restart recovery, when it is encountered, the logical-undo log record for the operation that was undone is deleted from the transaction's undo log, thus preventing it from being undone again.

4.7 Recovery

As part of the checkpoint operation, the end-of-the-system-log on disk is noted before the database image is checkpointed, and becomes the "begin-recovery-point" for this checkpoint once the checkpoint has completed. All updates described by log records preceding this point are guaranteed to be reflected in the checkpointed database image. Thus, restart recovery, after initializing the ATT and transaction undo logs with the copy of the ATT and undo logs stored in the most recent checkpoint, loads the database image and sets dpt to zero. It then applies all redo log records following the begin-recovery-point for the last completed checkpoint of the database (appropriate pages in dpt are set to dirty for each log record). During the application of redo log records, necessary actions are taken to keep the checkpointed image of the ATT consistent with the log applied so far. These actions mirror the actions taken during normal processing. For example, when an operation commit log record is encountered, lower-level log

a single unifying resource to coordinate the applications interaction with the recovery system, and this approach has proven very useful.

4.5 Ping-pong Checkpointing

Consistent with the terminology in main-memory databases, we use the term *checkpoint* to mean a copy of main-memory, stored on disk, and *checkpointing* refers to the action of creating a checkpoint. This terminology differs slightly from the terminology used, for example, in ARIES [MHL+92].

Traditional recovery schemes implement write-ahead logging (WAL), whereby all undo logs for updates on a page are flushed to disk before the page is flushed to disk. To guarantee the WAL property, a latch on the page (or possibly on the system log) is held while copying the page to disk. In our recovery scheme, we eliminate latches on pages during updates, since latching can significantly increase access costs in main memory and interferes with normal processing, as well as increasing programming complexity. However, as a result, it is not possible to enforce the WAL policy, since pages may be updated even as they are being written out.

For correctness, in the absence of write-ahead logging, two copies of the database image are stored on disk, and alternate checkpoints write dirty pages to alternate copies. This strategy is called *ping-pong* checkpointing (see, e.g., [SGM90b]). The ping-pong checkpointing strategy permits a checkpoint that is being created to be temporarily inconsistent; i.e., updates may have been written out without corresponding undo records having been written. However, after writing out dirty pages, sufficient redo and undo log information is written out to bring the checkpoint to a consistent state. Even if a failure occurs while creating one checkpoint, the other checkpoint is still consistent and can be used for recovery.

Keeping two copies of a main-memory database on disk for ping-pong checkpointing does not have a very high space penalty, since disk space is much cheaper than main-memory. As we shall see later, there is an I/O penalty in that dirty pages have to be written out to both checkpoints even if there was only one update on the page. However, this penalty is small for hot pages, and the benefits outweigh the I/O cost for typical main-memory database applications.

Before writing any dirty data to disk, the checkpoint notes the current end of the stable log in the variable end_of_stable_log, which will be stored with the checkpoint. This is the start point for scanning the system log when recovering from a crash using this checkpoint. Next, the contents of the (in-memory) ckpt_dpt are set to those of the dpt and the dpt is zeroed (noting of end_of_stable_log and zeroing of dpt are done atomically with respect to flushing). The pages written out are the pages that were either dirty in the ckpt_dpt of the last completed checkpoint, or dirty in the current (in-memory) ckpt_dpt, or in both. In other words, all pages that were modified since the current checkpoint image was last written, namely, pages that were dirtied since the last-but-one checkpoint, are written out. This is necessary to ensure that updates described by log records preceding the current checkpoint's end_of_stable_log have made it in the database image in the current checkpoint.

Checkpoints write out dirty pages without obtaining any latches and thus without interfering with normal operations. This *fuzzy* checkpointing is possible since physical-redo log records are generated by all updates; these are used during restart recovery and their effects are idempotent.

4.3 Transactions and Operations

Transactions, in our model, consist of a sequence of operations. Similar to [Lom92], we assume that each operation has a level L_i associated with it. An operation at level L_i can consist of a sequence of operations at level L_{i-1}. Transactions, assumed to be at level L_n, call operations at level L_{n-1}. Physical updates to regions are level L_0 operations. For transactions, we distinguish between *pre-commit*, when the commit record enters the system log in memory establishing a point in the serialization order, and *commit* when the commit record hits the stable log. We use the same terminology for operations, where only the pre-commit point is meaningful, though this is sometimes referred to as "operation commit" in the paper.

Each transaction obtains an *operation* lock before an operation executes (the lock is granted to the operation if it commutes with other operation locks held by active transactions), and L_0 operations must obtain region locks. The locks on the region are released once the L_1 operation pre-commits; however, an operation lock at level L_i is held until the transaction or the containing operation (at level L_{i+1}) pre-commits. Thus, all the locks acquired by a transaction are released once it pre-commits. The notion of pre-commit for transactions and a locking optimization related to the one implemented in Dalí is described in [DKO+84].

4.4 Logging Model

The recovery algorithm maintains separate undo and redo logs in memory for each transaction. These are stored as linked lists off the entry for the transaction in the ATT. Each update (to a part of a region) generates physical-undo and redo log records that are appended to the transaction's undo and redo logs respectively. When a transaction/operation pre-commits, all the redo log records for the transaction in its redo log are appended to the system log, and the logical-undo description for the operation is included in the operation commit log record in the system log. Thus, with the exception of logical-undo descriptors, only redo records are written to the system log during normal processing.

Also, when an operation pre-commits, the undo log records for its sub-operations/updates are deleted (from the transaction's undo log) and a logical-undo log record containing the undo description for the operation is appended to the transaction's undo log. In-memory undo logs of transactions that have pre-committed are deleted since they are not required again. Locks acquired by an operation/transaction are released once they pre-commit.

The system log is flushed to disk when a transaction decides to commit. Pages updated by a redo log record written to disk are marked dirty in the dirty page table, `dpt`, by the flushing procedure. In our recovery scheme, update actions do not obtain latches on pages – instead region locks ensure that updates do not interfere with each other.[6] In addition, actions that are normally taken on page latching, such as setting of dirty bits for the page, are now performed based on log records written to the redo log. For example, the flusher uses physical log records to set per-page dirty bits, avoiding contention on the dirty page table. The redo log is used as

[6] In cases when region sizes change, certain additional region locks on storage allocation structures may need to be obtained. For example, in a page-based system, if an update causes the size of a tuple to change, then in addition to a region lock on the tuple, an X mode region lock on the storage allocation structures on the page must be obtained.

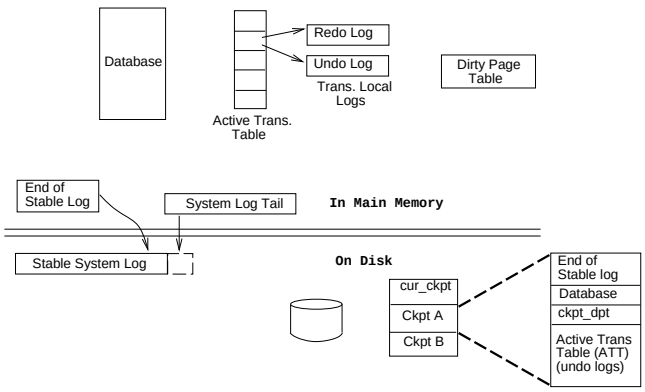

Figure 5: Overview of Recovery Structures

value, record pointer> pair. Also, the X region locks on the free list and the bucket are no longer required, and are released. Only the insert operation lock is held till end of transaction.

Note that if the physical-undo log records were not replaced by the logical-undo log record, it would not have been possible to release locks on the free list and the bucket. Once the region locks are released, other operations can update the same regions (bucket and free list), and attempting to roll back the first operation using physical-undo records would damage the effects of the later actions. Once the region locks are released, only a compensating undo operation can be used to undo the operation. The replacement of lower-level undo operations by higher-level undo operations is, in a nutshell, the idea underlying multi-level recovery. Without multi-level recovery, other allocation operations on the storage allocator would have been blocked until the end of the transaction – resulting in a lower degree of concurrency. Multi-level recovery is supported in, for example, ARIES [MHL+92].

4.2 System Overview

Figure 5 gives an overview of the structures used for recovery. The database[5] is mapped into the address space of each process as described in Section 3. Two checkpoint images of the database, Ckpt_A and Ckpt_B, reside on disk. Also stored on disk are 1) cur_ckpt, an "anchor" pointing to the most recent valid checkpoint image for the database, and 2) a single system log containing redo information, with its tail in memory. The variable end_of_stable_log stores a pointer into the system log such that all records prior to the pointer are known to have been flushed to the stable system log.

There is a single *active transaction table* (ATT), stored in the system database, that stores separate redo and undo logs for each active transaction. A dirty page table, dpt, is maintained for the database (also in the system database) which records the pages that have been updated since the last checkpoint. The ATT (with undo logs) and the dirty page table are also stored with each checkpoint. The dirty page table in a checkpoint is referred to as ckpt_dpt.

[5]The database here represents a single database file. In fact, different database files can be checkpointed at different times, and transactions can span database files arbitrarily. The generalization for multiple database files is straightforward, but is omitted for clarity and space.

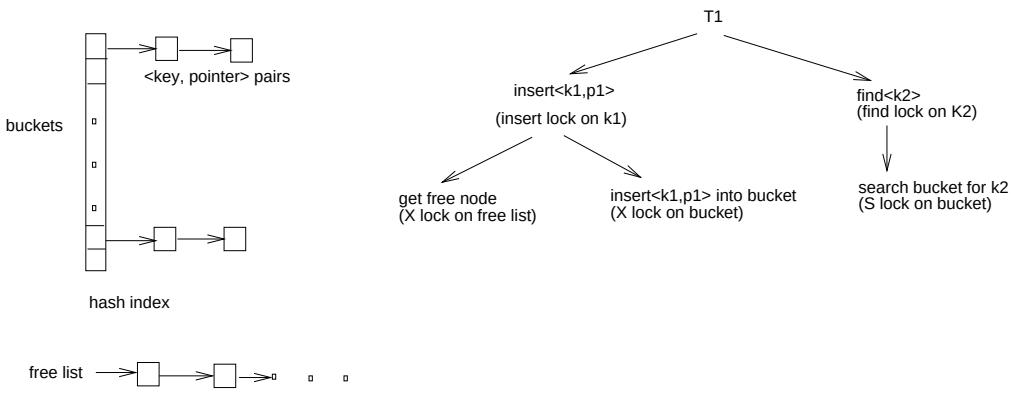

Figure 4: Overview of Multi-level Recovery

region locks on the particular nodes involved in an insert can be released, while an operation lock on the newly inserted key that prevents the key from being accessed or deleted is retained.

We illustrate multi-level recovery under this model. Consider a unique hash index that stores a <key value, pointer to record> pair for every record in a database. Let the hash index support operations insert, delete and find with the obvious meanings.

Note that each operation takes a key value as a parameter; it is on this key value that the operation gets an operation lock when it begins. Operation locks here are of three kinds: insert, delete and find locks. Operation locks on different key values do not conflict with each other. Furthermore, insert and delete locks conflict with every other operation lock (find/insert/delete) on the same key value; however, find locks on the same key value do not conflict with each other.

In order to see how operation locking can help enhance concurrency, consider an implementation of the hash index with buckets, (see Figure 4), where each bucket points to a linked list of nodes, each node containing a single <key value, record pointer> pair. In this implementation, it was decided that each bucket including the linked list constitute a region, and thus one region lock is associated with one bucket. In addition, there is also a free list of nodes from which nodes are obtained when inserting into the hash index. The free list is a separate region with its own lock.

A find operation obtains a find operation lock on the key value, and then an S region lock on the bucket containing the key value and releases the lock on the bucket once the node in the bucket chain containing the key value has been found. However, the find lock on the key-value is held for the duration of the transaction.

An insert operation first obtains an insert operation lock on the key value, and then obtains a region lock on the free list in X mode, and deletes a node from the free list. It then determines the bucket into which the key value is to be inserted and obtains an X region lock on the bucket. It then copies the <key value, pointer> pair into the free node and links the node into the bucket chain. Obtaining the free node and linking it into the bucket chain result in several updates, which are all *physically* logged. Once the node has been added to the chain, the insert operation is complete. At this point, the physical-undo log records are deleted and replaced by a logical-undo log record, which if executed, would call the delete operation on the new <key

segment is updated to point to the segment header for the segment. The segment header, in addition to containing the start address for the segment and the chunk containing the segment, can also contain additional information about data in the segment to support higher-level abstractions (e.g., lock and type information). This last facility is used by the heap file described in Section 7.1.

4 Transaction Management in Dalí

In this section we present how transaction atomicity, isolation and durability are achieved in Dalí. Transaction management in Dalí is based on principles of multi-level recovery [WHBM90, MHL+92, Lom92]. To our knowledge, Dalí is the only implementation of multi-level recovery for main-memory, and one of the few implementations of explicit multi-level recovery reported to date (Weikum [WHBM90] reports use of explicit MLR in a prototype database management system).

We begin with a review of multi-level recovery concepts, followed by a description of the structures used in Dalí for transactions, logging and other recovery support mechanisms. Our implementation extends the scheme presented in [JSS93] with multiple levels of abstraction, and a fuzzy checkpointing scheme that only writes dirty pages. Low-level details of our scheme are described in [BPR+96].

In our scheme, data is logically organized into *regions*. A region can be a tuple, an object, or an arbitrary data structure like a list or a tree. Each region has a single associated lock with exclusive (X) and shared (S) modes, referred to as the *region lock*, that guards accesses and updates to the region.

4.1 Multi-Level Recovery

Multi-level recovery provides recovery support for enhanced concurrency based on the semantics of operations. Specifically, it permits the use of weaker *operation* locks in place of stronger shared/exclusive region locks.

A common example is index management, where holding physical locks until transaction commit leads to unacceptably low levels of concurrency. If undo logging has been done physically (e.g. recording exactly which bytes were modified to insert a key into the index) then the transaction management system must ensure that these physical-undo descriptions are valid until transaction commit. Since the descriptions refer to specific updates at specific positions, this typically implies that the region locks on the updated index nodes are retained to ensure correct *recovery*, even though they are no longer needed for correct concurrent access to the index.

The multi-level recovery approach is to replace these low-level physical-undo log records with higher-level logical-undo log records containing undo descriptions at the operation level. Thus, for an insert operation, physical-undo records would be replaced by a logical-undo record indicating that the inserted key must be deleted. Once this replacement is made, the region locks may be released, and only (less restrictive) operation locks are retained. For example,

- The *inline power-of-two allocator* is the same except that the free space list uses the first few bytes of each free block to implement the list.
- The *coalescing allocator* merges adjacent free space and uses a *free tree*, described below.

In both power-of-two allocators, space requests are rounded up to a bucket size. Free-space lists are maintained per-bucket, where a bucket represents an allocation size, with a maximum item size of $2^{31} * m$. Free-space lists are stored in a *separate* chunk, making it much harder for simple programming errors to corrupt the system-wide free space tables. Requested (freed) space that is rounded up to size $2^i * m$ is allocated (freed) from (to) the i^{th} bucket. The inline allocator is faster and more space efficient, but susceptible to corruption from simple off-by-one programming errors. This allocator is mostly used for system-maintained data (such as the free space lists for the power-of-two allocator). Since the power-of-two allocators do not coalesce adjacent free space, they are subject to fragmentation and are thus primarily used for fixed size data.

The coalescing allocator provided by Dalí is implemented using a *free tree*. Our implementation of this structure is based on the T-tree described in Section 7.3. It consists of a T-tree of free space which uses the starting address of free blocks as the key. Thus, any two free blocks which are candidates to be merged will be adjacent in the tree. Each node is annotated with the largest free block in the subtree rooted at that node. This information is used during allocation to traverse the free tree – at each node the subtree chosen is one which contains a free block larger than the requested size. Traversal halts once a sufficiently large free block is found. Each time space is freed, it is inserted into the free tree and an attempt is made to merge it with its in-order successor and predecessor in the tree. In the case that allocation or freeing of space causes the sizes of free blocks in the free tree to change, this information is propagated upwards to all the ancestors of the node in the free tree, if necessary. This structure provides logarithmic time for both allocation and freeing, while keeping all adjacent free space coalesced and providing exact allocation.

In the allocation schemes described above, contiguous unallocated space at the end of the last segment for the chunk is not contained in the free lists and the free tree. Thus, if no free blocks are found in the free lists or the free tree, space is allocated from the end of the last segment in the chunk if possible. If sufficient space is not available at the end of the last segment, then a new segment is allocated for the chunk from the database file and space is allocated from it (the new segment is also appended to the list of segments for the chunk, and the insufficient free space in the former last segment is added to free space list).

3.3.2 The Page Table and Segment Headers

Database systems which use physical addressing may need to associate some information about the segment or the chunk with a physical database pointer. For this reason, we have implemented *segment headers* in Dalí, and use a *page table* to map pages to segment headers. The page table is pre-allocated based on the maximum number of pages in the database (and reallocated if the database is resized). Segment headers are allocated when a new segment is added to a chunk. Furthermore, each page table entry corresponding to a page in the newly-allocated

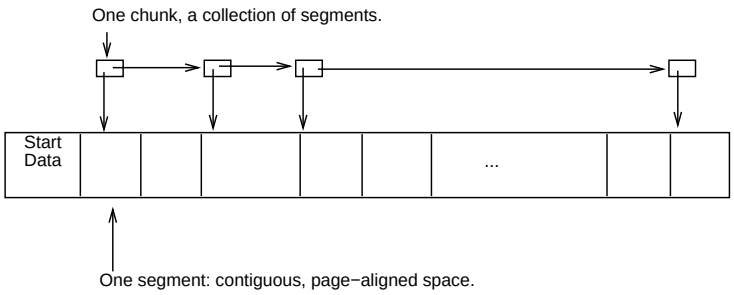

Figure 3: Segments and Chunks

negate that advantage. Thus, Dalí items can be of arbitrary size.

Different recovery characteristics should be available for different regions of the database. Not all data needs to be recovered in case of a system crash. For example, indexes could be recovered by recreating the index (at a substantial cost in recovery time). Similarly, lock and semaphore contents do not need to persist across system crashes – they simply need to be re-initialized at recovery time. We distinguish two levels of non-recovered data: *zeroed memory* and *transient memory*. Zeroed memory remains allocated upon recovery but each byte is set to zero. With transient memory, the data is no longer allocated upon recovery. These characteristics can be applied at the database level, and at the sub-database level as described in the next section.

3.3.1 Segments and Chunks

We now describe the storage allocation mechanism in Dalí, and show how it meets the requirements described above. Each database file in Dalí is comprised of *segments*, which are contiguous page-aligned units of allocation, similar to clusters in a file system. As illustrated in Figure 3, a *chunk* is a collection of segments. Recovery characteristics (transient memory, zeroed memory, or persistent memory) are specified on a per-chunk basis at the time of chunk creation. Users allocate within a chunk, and do not specify a particular segment. Since segments can be arbitrarily large (within the size of the database), arbitrarily large objects can be stored contiguously. Upon allocation within a chunk, the system returns a standard Dalí pointer to the space, which specifies the offset within the file. Thus, indirection is not imposed at the storage manager level. The elements shown linking together segments in a chunk are themselves stored in a special chunk used for control information.

Within a chunk, different allocators are available which trade off speed, safety and size. In all allocators, no record of allocated space is retained, and the user must remember the size of the allocated data. This is required to avoid excessive overhead for small items. A layer above the allocator can be implemented to store this data above the allocated space, if required. The currently defined and implemented allocators in Dalí are

- The *power-of-two allocator* allocates storage in buckets of size $2^i * m$ where m is some minimum item size.

The primary kind of database pointer in Dalí contains a database file local-identifier and an offset within the database file. Dereferencing a database pointer `p` simply involves adding the offset contained in `p` to the virtual memory address at which the database file is mapped, looked up from the offset table. A second form of database pointer is available for cases where the database file is known from context. For example, all pointers out of a certain index might reside in a particular database file. In this case, we may store just the offset within the database file as the pointer. Both offsets and full pointers are implemented as simple C++ template classes which allow them to be used as "smart pointers".

3.3 Storage Allocation

We next describe how storage for data is allocated within a database file. Designing storage allocation structures consists primarily of trading speed for generality and flexibility. Our particular choices are motivated by the following requirements: 1) control data should be stored separately from user data, 2) indirection should not exist at the lowest level, 3) large objects should be stored contiguously, and 4) different recovery characteristics should be available for different areas. We now describe each requirement in more detail.

Control data should be stored separately from user data. Since processes map the entire database file into their address space, stray pointers in applications can easily corrupt the database. Maintaining the integrity of control data (e.g., information about free storage space) is crucial since it's corruption implies the corruption of the entire database file. If control data is stored with the data itself, then the control data would be very susceptible to corruption by simple errors such as improper bounds checking or accessing recently freed memory. However, if control information is stored separately from the data, then stray application pointers are more likely to corrupt other user data rather than control data. Thus, the corruption of the entire database file can be avoided.

Indirection should not exist at the lowest level. Most disk-based storage managers (e.g., Exodus, EOS, System R) have a slotted page architecture in which data is allocated in pages and a slot array at the bottom of the page contains pointers to allocated data. Each data item is then identified by the page containing it and the index of the slot containing a pointer to it. This indirection has the advantage that data items in a page can be moved around in order to reclaim space, and only the pointers in the slots need to be updated. However, the indirection almost certainly adds a level of latching to each data access, as well as adding path length for the dereference itself. Finally, there is an additional storage cost for the extra pointers. In Dalí, since the database is main-memory resident, these costs are proportionally much higher than in a disk resident system. Further, for an object-oriented database, a level of indirection may already exist in the mapping from object identifier to objects which offers many of these advantages, such as the ability to move and resize objects, making the overhead of slotted pages redundant. For these reasons, we did not adopt the slotted page architecture in Dalí. Instead, the storage allocator exposes direct pointers to allocated data providing both time and space efficiency.

Large objects should be stored contiguously. If large objects are stored in main memory, the advantage is obviously speed. Having to reconstruct them from smaller objects will serve to

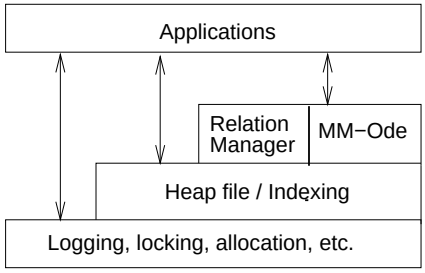

Figure 2: Layers of Abstraction in Dalí

files, and may map the same database file to different locations in their address space. This feature precludes using virtual memory addresses as physical pointers to data (in database files), but provides two important benefits. First, a database file may be easily resized. Second the total active database space on the system may exceed the addressing space of a single process. This is useful on machines with 32-bit addressing (e.g., the SPARCCenter) in which physical memory can significantly exceed the amount of memory addressable by a single process.

However, in a 64-bit machine, both of these considerations may be significantly mitigated, leading us to consider using physical addressing. If a single database file can be limited to something like 64 Gigabytes, then each process could still map close to a billion database files (which can be expected to far exceed the total database space).

3.1 Layers of Abstraction

An important feature of Dalí's architecture is that it is organized in multiple layers of abstraction to support the toolkit approach discussed earlier. Figure 2 illustrates this architecture. At the highest level, users can interact with either Dalí's relational manager or the Main-Memory Ode object database. These two layers are described later in sections 8.1 and 8.2. Below that level is what we call the "heap-file/indexing layer," which provides support for fixed-length and variable-length collections, as well as template-based indexing abstractions. In general, at this level, one does not need to interact with individual locks or latches. Instead, one specifies a policy to the lower level, such as "no locking" or "lock-plus-handle-phantoms".

Services for logging, locking, latching, multi-level recovery and storage allocation are exposed at the lowest level. New indexing methods can be built on this layer, as can special-purpose data structures for either an application or a database management system. Of course, this level has the most complex user-interface, but it has proven itself during the creation of the higher-level interfaces and database systems described above.

3.2 Pointers and Offsets

It is crucial for performance that mapping from database pointers to virtual memory addresses be done efficiently. In Dalí, each process maintains a database-offset table, which specifies where in memory each database file is mapped. The table is currently implemented as an array indexed by the (integer) database file identifier.

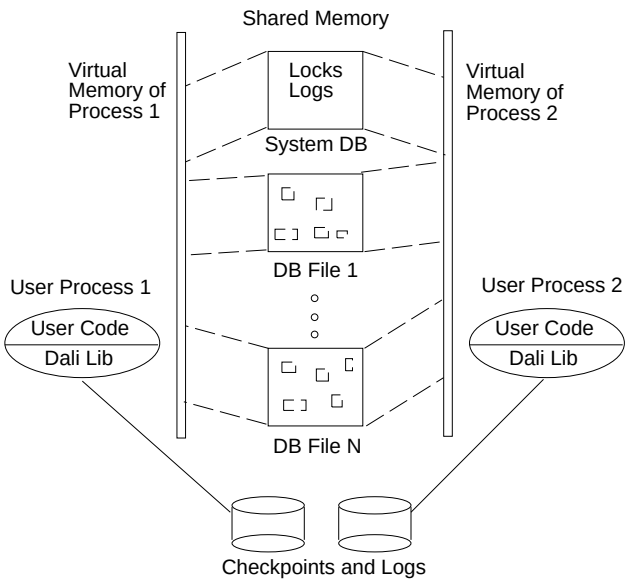

Figure 1: Architecture of the Dalí System

considerations. To our knowledge, no prior work addresses concurrency control and recovery issues for T-trees in particular. Further details of our scheme can be found in [BLR+95]. A structure for hashing in main-memory was proposed in [AP92]. Our hashing structure is much simpler at the cost of more-than-one compare for some searches (no concurrency control scheme was given in [AP92]). Our concurrency control and recovery mechanisms take advantage of its simplicity; these mechanisms are described in Section 7.2.

The latch recovery techniques are based on work in [BLS+95]. A significant body of work on making data structures tolerant to process slow-downs and failures exists under the title of "wait free" data structures (see e.g. [Her89, Her88, TSP92]). This work is not designed for transaction processing systems, however, and depends on the presence of a compare-and-swap instruction, which is not available in many architectures (such as SPARC). Sullivan and Stonebraker address the problem of protection from erroneous writes in [SS91], but they assume inexpensive operating system support for protecting and unprotecting data.

3 Architecture

In the Dalí architecture, the database consists of one or more *database files*, along with a special *system database file*. User data itself is stored in database files while all data related to database support, such as log and lock data, is stored in the system database file. This enables storage allocation routines to be uniformly used for (persistent) user data as well as (non persistent) system data like locks and logs. The system database file also persistently stores information about the database files in the system.

As shown in Figure 1, database files opened by a process are directly mapped into the address space of that process. In Dalí, either memory-mapped files or shared-memory segments can be used to provide this mapping. Different processes may map different sets of database

transaction processing applications but with much lower latency and higher throughput requirements. In a typical Dalí application, transactions are small, multiple processes may access shared data, and high concurrency – especially on index structures – is important. As a result, Dalí supports item level locking. Also, the recovery algorithm used in Dalí is designed to work well with small transactions.

There has been a good deal of prior work in the area of main-memory databases. Much of this work has concentrated on recovery schemes, and will be discussed in the context of other recovery research below. An early paper by DeWitt *et al*, [DKO+84], covered a number of topics including query processing, recovery and data organization. A later work by Lehman and Carey on indexing and query evaluation issues, [LC86a], introduced the T-Tree. (A concurrent version of T-Trees is implemented in Dalí.) Garcia-Molina and Salem [GMS92] provide an excellent overview of research on main-memory databases. Lehman *et al.* [LSC92] and Gottemukkala and Lehman [GL92] discuss the relative costs of operations such as locking and latching in the main-memory storage component of the Starburst extensible database system. They demonstrate that once the I/O bottlenecks of paging data into and out of the database are removed, other factors such as latching and locking dominate the cost of database access, and they provide techniques for reducing such costs. Thus, they provide an excellent motivation for closely examining the system design of a main-memory database and tuning it to remove bottlenecks, and have thereby influenced our work significantly.

Much of the work on main-memory databases has concentrated on recovery [Eic89, Hag86, LC87, LE93, SGM90a]. The work by Eich [Eic89] provides a survey, and the performance studies using System M by Salem and Garcia-Molina [SGM90a] provide both a good review and performance comparison of many of the schemes suggested by earlier work. Our recovery algorithm is in many ways similar to the "fuzzy" checkpointing schemes of [SGM90a], including use of ping-pong checkpointing and dirty page bits. One difference is that updates in Dalí are in-place, requiring that undo information is sometimes necessary. We use the techniques developed for Dalí in [JSS93] to limit this undo logging to during checkpointing so that the majority of undo log information is never written to disk. Our main contribution to this earlier scheme is integration with multi-level recovery, which allows early release of low-level locks for indexing and storage allocation, while retaining the benefits of fuzzy checkpointing for consistency of response times. Multi-level recovery (MLR) schemes have been proposed in the literature [WHBM90, Lom92, MHL+92]. Like these schemes, our scheme repeats history, generates log records during undo processing and logs operation commits when undo operations complete (similar to CLRs described in [MHL+92]). Also, as in [Lom92], transaction rollback at crash recovery is performed level-by-level. By integrating MLR with the main-memory recovery techniques described above and in [JSS93], we have produced a significantly optimized multi-level recovery algorithm for main memory database management systems.

As mentioned above, T-trees were proposed by Lehman and Carey in [LC86a]. Our implementation supports concurrent access, scans, and addresses recovery issues. Earlier work on concurrency for binary and AVL trees relates to our work due to the similarity of the structures. The index techniques of [KL80] do not address all the concurrency control issues needed to implement transaction semantics, while the treatment of [ML82] requires pre-ordering all accesses to a tree by a given transaction by key value. Neither of [KL80, ML82] address recovery

the architecture and the storage structures used in Dalí. Details of how Dalí recovers data from system crashes and process failures are given in Sections 4 and 5. Section 6 covers the implementation of latches (semaphores) and locks. Section 7 describes support for collections of data items and indexes, while higher-level relational and object-oriented interfaces are described in Section 8. Finally, concluding remarks are offered in Section 9.

2 Related Work

A storage manager provides the core functionality of a database system, such as concurrency control, recovery mechanisms, storage allocation/free space management, and transaction management. There have been numerous implementations of storage managers for disk resident data. These include the storage managers of Exodus [CDRS89], Starburst [HCL+90], ObjectStore [LLOW91], EOS [BP93], Texas [SKW92], Cricket [SZ90], and QuickStore [WD94]. After describing how Dalí relates to other storage managers and main-memory database research in general, we will briefly address related work for the various novel aspects of Dalí's implementation.

With the exception of the Starburst main-memory storage component [LSC92] we are not aware of any storage manager that is tailored for main-memory resident data.[4] The Starburst main-memory storage component is a relational storage manager used as a component of the Starburst database system. Its emphasis is on data allocation and structuring issues; the Starburst main-memory storage component described in [LSC92] uses the recovery manager of Starburst rather than implementing its own recovery manager. In contrast, the recovery mechanisms of Dalí are based on a recovery algorithm tailored to main memory, evolved from those proposed in [JSS93].

Unlike Dalí and the Starburst main-memory storage component, the other (existing or proposed) storage managers of which we are aware are not tailored for memory-resident data. The storage managers for disk-resident data can be divided into two groups. The first group consists of traditional storage managers, such as Exodus and EOS, that provide their own buffer management facilities. The second category consists of storage managers that map the database into virtual memory. Included in this category are the storage manager of ObjectStore, the Texas system, Cricket, and QuickStore.

Storage managers in this second category are more closely related to Dalí, since Dalí also uses a memory-mapped architecture. However, the architecture of existing memory-mapped storage managers, in particular their recovery mechanism, does not take advantage of the database being resident in main memory. For instance, ObjectStore uses page-wise checkpointing, and Texas uses a shadow paging architectures which, while providing support for old versions of data, results in slow commit processing. Also, the storage managers were designed for CAD environments where transactions are long, concurrency control at the level of pages is sufficient, and fast sharing of data is not a primary concern.

Dalí, on the other hand, is designed for high performance applications similar to traditional

[4] System M [SGM90a] is a transaction processing test-bed for memory resident data, but is not a full feature storage manager.

the direct access principle by allowing the user access to the data without a copy.[2] A related principle is *no interprocess communication* for basic system services. All concurrency control and logging services are provided via shared memory rather than communication with a server. While Dalí does provide servers to orchestrate system activity, take checkpoints, cope with process failure, etc., a typical user process only communicates with them when connecting to and disconnecting from the database.

The next guiding principle of Dalí is that it enables the creation of *fault-tolerant* applications. The primary expression of this principle is the use of the transactional paradigm, the dominant technology for providing fault-tolerance to critical applications. In fact, Dalí provides an advanced, explicitly multi-level transaction model which has facilitated the production of high-concurrency indexing and storage structures, and the description of this transaction management facility and these storage structures is the primary focus of this paper. The Dalí system also includes other features supporting the principle of fault-tolerance. One is support for recovery from *process failure* in addition to system failure. Another is the use of *codewords* and *memory protection* to help ensure the integrity of data stored in shared memory. Describing these features is a secondary focus of this paper.

Another key requirement for applications which expect to store all their data in main memory is *consistency of response time*. Support for fine-grained concurrency control and minimal interference with the checkpointer due to latching help provide this consistency in Dalí. Other principles that have guided Dalí's implementation have been *a toolkit approach* and support for *multiple interface levels*. The former implies, for example, that logging facilities can be turned off for data which need not be persistent, and locking can be turned off if data is private to a process. The second principle means that low-level components are exposed to the user so that critical system components can be optimized with special implementations. Most applications will prefer the high-level relational and object-oriented interfaces, however.

As a storage manager, Dalí is intended to support a variety of data models – for example, relational and object-oriented models have been implemented in Dalí already. Our intention, like that of the Genesis system [BBG+90] and the Exodus Storage Manager [CDRS89], is to provide the implementor of a database management system flexible tools for storage management, concurrency control and recovery, without dictating a particular storage model or precluding optimizations. Some of the aspects of Dalí which support this goal are subtle, including a flexible recovery model and storage abstractions which do not build in significant per-item overheads. This last point is particularly important in a main-memory system.

While Dalí can be used in systems where the database is larger than main-memory (as long as the database fits in the virtual address space of the process), the architecture of Dalí, from storage allocation and indexing to its recovery facilities, has been designed to deliver high performance when the database fits into main memory. With minor variations, the version of Dalí described in this paper is currently implemented as a research prototype in Bell Laboratories.[3]

The remainder of the paper is organized as follows. Related work on storage managers and recovery techniques is surveyed in Section 2. In Section 3, we present an overview of

[2] For some of the interfaces, a copying mode is also supported.

[3] For more information, see `http://www.bell-labs.com/org/1123/what/dali/`

1 Introduction

There are a number of database applications, particularly in the telecommunications industry (and other industries involved in real-time content delivery), where very high performance access to data is required. Such applications typically need high transaction rates, coupled with very low latency for transactions, and impose stringent durability and availability requirements. As as example, consider a real phone-company application where phone call data is recorded, and queries against the data can be issued. The application requires several thousand (albeit small) requests (lookups/updates) to be processed per second, with less than 50 milliseconds latency for lookups, and less than a few minutes of down-time a year. Such applications have been previously implemented as stand-alone programs that run in main memory and provide their own (usually limited) forms of sharing and persistence mechanisms. It is increasingly being realized that the storage needs of these types of applications would best be met by using an underlying main-memory storage manager that supports an array of functionality such as transaction management, data organization, concurrency control and recovery services. Using the same storage manager across multiple applications can greatly reduce development costs.

The increasing availability of large and relatively cheap memory also suggests that more database applications could reside entirely or almost entirely in main memory. Such applications will experience performance benefits by having data cached in main memory. However, if the storage manager supporting such applications is tailored to main memory, significant additional performance benefits can be achieved, as shown in [LSC92]. Thus, storage managers tailored to main memory would also be ideally suited for such databases.

The Dalí[1] system [JLR+94], implemented at Bell Laboratories, is a storage manager for persistent data whose architecture has been optimized for environments in which the database is main-memory resident. While not directly suitable for storing large multimedia objects, Dalí can be used in a number of ways to facilitate delivery of multimedia content. First, it can be used to store *meta-data* such as allocation information about multimedia objects, and has been used in this manner in a prototype of the Fellini continuous media storage server [ÖRS+96]. In such an environment, Dalí may also be used to coordinate shared access to main memory buffers, providing concurrency control and allowing recovery from process failure. Another increasingly significant target application for Dalí is real-time billing and control of multimedia content delivery. In fact, Dalí's original target application, aiding in control and billing for real-time voice data in telephony applications, is just one example of the need for high-speed transactional access to data in multimedia content delivery. We expect a number of new applications in this category to arise from novel content delivery services provided over the World Wide Web.

A number of principles have evolved with Dalí over the past three years and now guide its design and evolution. The first of these principles is *direct access to data.* As described above, we have found that this requirement has already been imposed by the architects of high performance applications. Dalí uses a memory-mapped architecture, where the database is mapped into the virtual address space of the process, allowing the user to acquire pointers directly to information stored in the database. The various interface levels further support

[1] Named in honor of Salvadore Dalí, for his famous painting, "The Persistence of Memory".

The Architecture of the Dalí Main-Memory Storage Manager

Philip Bohannon[1,*]
Daniel Lieuwen[1]
Rajeev Rastogi[1]
S. Seshadri[2]
Avi Silberschatz[1]
S. Sudarshan[2]

[1]Bell Laboratories
700 Mountain Ave., Murray Hill, NJ 07974
{bohannon,lieuwen,rastogi,silber}@research.bell-labs.com

[2]Indian Institute of Technology, Bombay
{seshad,sudarsha}@cse.iitb.ernet.in

Abstract

Performance needs of many database applications dictate that the entire database be stored in main memory. The Dalí system is a main memory storage manager designed to provide the persistence, availability and safety guarantees one typically expects from a disk-resident database, while at the same time providing very high performance by virtue of being tuned to support in-memory data. User processes map the entire database into their address space and access data directly, thus avoiding expensive remote procedure calls and buffer manager interactions typical of accesses in disk-resident commercial systems available today.

Dalí recovers the database to a consistent state in the case of system as well as process failures. It also provides unique concurrency control and memory protection features, as well as ordered and unordered index structures. Both object-oriented and relational database management systems have been implemented on top of Dalí. Dalí provides access to multiple layers of application programming interface, including its low-level recovery, concurrency control and indexing components as well as its high-level relational component. Finally, various features of Dalí can be tailored to the needs of an application to achieve high performance – for example, concurrency control and logging can be turned off if not desired, enabling Dalí to efficiently support applications that require non-persistent memory-resident data to be shared by multiple processes.

*A Ph.D. candidate in the Department of Computer Science at Rutgers University.