teractions. Although a complete discussion of the AIRES algorithm is beyond the scope of this paper, we will provide a breif overview, and explain the details that are relevant to developers that wish to extend LLADD.

By documenting the interface between AIRES and higher-level primitives such as data structures, and by structuring LLADD to make this interface explicit in both the library and its extensions, we hope to make it easy to produce correct and efficient durable data structures. In existing systems (and indeed, in earlier versions of LLADD), the implementation of such structures is extremely complicated, and subject to the introduction of incredibly subtle errors that would only be evident during crash recovery or at other inconvenient times. Thus there is great value is reusing these lower layers once developed.

Finally, by approaching this problem by implementing a number of simple modules that "do one thing and do it well", we believe that LLADD can provide competitive performance while making future improvements to its core implementation significantly easier. In order to achieve this goal, LLADD has been split into a number of modules forming a 'core library', and a number of extensions called 'operations' that build upon the core library. Since each of these modules exports a stable interface, they can be independently improved.

1.1 Prior Work

An extensive amount of prior work covers the algorithms presented in this paper. Most fundamentally, systems that provide transactional consistency to their users generally include a number of common modules. A high-level overview of a typical system is given in Figure 1.

Many applications make use of transactional storage, and each is designed for a specific application, or set of applications. LLADD provides a flexible substrate that allows such applications to be developed. The complexity of existing systems varies widely, as do the applications for which these systems are designed.

On the database side of things, relational databases excel in areas where performance is important, but where the consistency and durability of the data is crucial. Often, databases significantly outlive the software that uses them, and must be able to cope with changes in business practices, system architechtures, etc.

Object-oriented databases are more focused on facilitating the development of complex applications
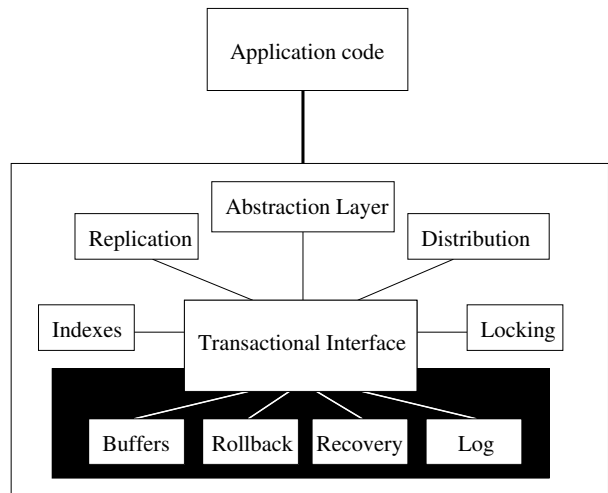

Figure 1: Conceptual view of a modern transactional application. Current systems include high level functionality, such as indices and locking, but are not designed to allow developers to replace this functionality with application specific modules.

that require reliable storage, and may take advantage of less-flexible, more efficient data models, as they often only interact with a single application, or a handful of variants of that application.

Databases are designed for circumstances where development time may dominate cost, many users must share access to the same data, and where security, scalability, and a host of other concerns are important. In many, if not most, circumstances these issues are less important, or even irrelevant. Therefore, applying a database in these situations is likely overkill, which may partially explain the popularity of MySQL, which allows some of these constraints to be relaxed at the discretion of a developer or end user.

Still, there are many applications where MySQL is still too inflexible. In order to serve these applications, a host of software solutions have been devised. Some are extremely complex, such as semantic file systems, where the file system understands the contents of the files that it contains, and is able to provide services such as rapid search, or file-type specific operations such as thumbnailing, automatic content updates, and so on. Others are simpler, such as BerkeleyDB, which provides transactional storage of data in unindexed form, in indexed form using a hash table, or a tree. LRVM is a version of malloc() that provides transacational memory, and is similar to an object-oriented database, but is much lighter weight, and more flexible.

Finally, some applications require incredibly sim-

ple, but extremely scalable storage mechanisms. Cluster Hash Tables are a good example of the type of system that serves these applications well, due to their relative simplicity, and extremely good scalability characteristics. Depending on the fault model on which a cluster hash table is implemented, it is quite plausible that key portions of the transactional mechanism, such as forcing log entries to disk, will be replaced with other durability schemes, such as in-memory replication across many nodes, or multiplexing log entries across multiple systems. This level of flexibility would be difficult to retrofit into existing transactional applications, but is appropriate in many environments.

We have only provided a small sampling of the many applications that make use of transactional storage. Unfortunately, it is extremely difficult to implement a correct, efficient and scalable transactional data store, and we know of no library that provides low level access to the primitives of such a durability algorithm. These algorithms have a reputation of being complex, with many intricate interactions, which prevent them from being implemented in a modular, easily understandable, and extensible way.

Because of this, many applications that would benefit from transactional storage, such as CVS, and many implementations of IMAP either ignore the problem, leaving the burden of recovery to system administrators or users, or implement ad-hoc solutions that employ complex, application specific consistency protocols in order to ensure the consistency of their data. This increases the complexity of such applications, and often provides only a partial solution to the transactional storage problem, resulting in erratic and unpredictable application behavior.

In addition to describing such an implementation of ARIES, a well-tested "industrial strength" algorithm for transactional storage, this paper outlines the most important interactions that we discovered (that is, the ones that could not be encapsulated within our implementation), and gives the reader a sense of how to use the primitives the library provides.

2 ARIES from an Operation's Perspective

Instead of providing a comprehensive discussion of ARIES, we will focus upon those features of the algorithm that are most relevant to a developer attempting to add a new set of operations. Correctly implementing such extensions is complicated by concerns regarding concurrency, recovery, and the possibility that any operation may be rolled back at runtime.

We first sketch the constraints placed upon operation implementations, and then describe the properties of our implementation of ARIES that make these constraints necessary. Because comprehensive discussions of write ahead logging protocols and ARIES are available elsewhere, we only discuss those details relevant to the implementation of new operations in LLADD.

2.1 Properties of an Operation

A LLADD operation consists of some code that performs some action on the developer's behalf. These operations implement the high-level actions that are composed into transactions. They are implemented at a relatively low level, and have full access to the ARIES algorithm. We expect the majority of an application to reason in terms of the interface provided by custom operations, allowing the the application, the operation, and LLADD itself to be independently improved.

Since transactions may be aborted, the effects of an operation must be reversible. Furthermore, aborting and comitting transactions may be interleaved, and LLADD does not allow cascading aborts,[3] so in order to implement an operation, we must implement some sort of locking, or other concurrency mechanism that isolates transactions from each other. LLADD only provides physical consistency; we leave it to the application to decide what sort of transaction isolation is appropriate. For example, it is relatively easy to build a strict two-phase locking lock manager on top of LLADD, as needed by a DBMS, or a simpler lock-per-folder approach that would suffice for an IMAP server. Thus, data dependencies among transactions are allowed, but we still must ensure the physical consistency of our data structures, such as operations on pages or locks.

Also, all actions performed by a transaction that committed must be restored in the case of a crash, and all actions performed by aborting transactions must be undone. In order for LLADD to arrange for this to happen at recovery, operations must produce log entries that contain all information necessary for undo and redo.

An important concept in ARIES is the "log sequence number" or LSN. An LSN is essentially a

---

[3]That is, by aborting, one transaction may not cause other transactions to abort. To understand why operation implementors must worry about this, imagine that transaction A split a node in a tree, transaction B added some data to the node that A just created, and then A aborted. When A was undone, what would become of the data that B inserted?

virtual timestamp that goes on every page; it tells you the last log entry that is reflect on the page, which implies that all previous log entries are also reflected. Given the LSN, you can tell where to start playing back the log to bring a page up to date. The LSN goes on the page so that it is always written to disk atomically with the data of the page.

ARIES (and thus LLADD) allows pages to be *stolen*, i.e. written back to disk while they still contain uncommitted data. It is tempting to disallow this, but to do so has serious consequences such as a increased need for buffer memory (to hold all dirty pages). Worse, as we allow multiple transactions to run concurrently on the same page (but not typically the same item), it may be that a given page *always* contains some uncommitted data and thus could never be written back to disk. To handle stolen pages, we log UNDO records that we can use to undo the uncommitted changes in case we crash. LLADD ensures that the UNDO record is be durable in the log before the page is written back to disk, and that the page LSN reflects this log entry.

Similarly, we do not force pages out to disk every time a transaction commits, as this limits performance. Instead, we log REDO records that we can use to redo the change in case the committed version never makes it to disk. LLADD ensures that the REDO entry is durable in the log before the transaction commits. REDO entries are physical changes to a single page ("page-oriented redo"), and thus must be redone in the exact order.

One unique aspect of LLADD, which is not true for ARIES, is that *normal* operations use the REDO function; i.e. there is no way to modify the page except via the REDO operation. This has the great property that the REDO code is known to work, since even the original update is a "redo". In general, the LLADD philosophy is that you define operations in terms of their REDO/UNDO behavior, and then build the actual update methods around those.

Eventually, the page makes it to disk, but the REDO entry is still useful: we can use it to roll forward a single page from an archived copy. Thus one of the nice properties of LLADD, which has been tested, is that we can handle media failures very gracefully: lost disk blocks or even whole files can be recovered given an old version and the log.

2.2 Normal Processing

Operation implementors follow the pattern in Figure 2, and need only implement a wrapper function ("Tset()" in the figure, and a pair of redo and undo functions will be registered with LLADD. The Tup-

date function, which is built into LLADD, handles most of the runtime complexity. LLADD also uses the undo and redo functions during recovery, in the same way that they are used during normal processing.

2.2.1 The buffer manager

LLADD manages memory on behalf of the application and prevents pages from being stolen prematurely. Although LLADD uses the STEAL policy and may write buffer pages to disk before transaction commit, it still must make sure that the undo log entries have been forced to disk before the page is written to disk. Therefore, operations must inform the buffer manager when they write to a page, and update the LSN of the page. This is handled automatically by many of the write methods provided to operation implementors (such as writeRecord()), but the low-level page manipulation calls (which allow byte-level page manipulation) leave it to their callers to update the page metadata appropriately.

2.2.2 Log entries and forward operation (the Tupdate() function)

[TODO...need to make this clearer... I think we need to say that we define a function to do redo, and then we define an update that use it. Recovery uses the same function the same way.]

In order to handle crashes correctly, and in order to the undo the effects of aborted transactions, LLADD provides operation implementors with a mechanism to log undo and redo information for their actions. This takes the form of the log entry interface, which works as follows. Operations consist of a wrapper function that performs some pre-calculations and perhaps acquires latches. The wrapper function then passes a log entry to LLADD. LLADD passes this entry to the logger, and then processes it as though it were redoing the action during recovery, calling a function that the operation implementor registered with LLADD. When the function returns, control is passed back to the wrapper function, which performs any post processing (such as generating return values), and releases any latches that it acquired.

This way, the operation's behavior during recovery's redo phase (an uncommon case) will be identical to the behavior during normal processing, making it easier to spot bugs. Similarly, undo and redo operations take an identical set of parameters, and undo during recovery is the same as undo during normal processing. This makes recovery bugs more

```
┌─────────────────────────────────────┐
│ TSet(Record, Value)                  │
│   Serialize arguments for log        │
│  ┌────────────────────────────────┐  │
│  │ TUpdate(Record, Args)          │  │
│  │   Load page (pin in memory)    │  │
│  │   Read pre−image               │  │
│  │   Allocate log entry           │  │
│  │   Write log entry              │  │
│  │   Invoke redo operation        │  │
│  │  ┌──────────────────────────┐  │  │
│  │  │ do_set(LogEntry)         │  │  │
│  │  │                          │  │  │
│  │  │   Parse arguments        │  │  │
│  │  │   Write data to page     │  │  │
│  │  │   Update lsn             │  │  │
│  │  └──────────────────────────┘  │  │
│  │                                │  │
│  │   Release page                 │  │
│  └────────────────────────────────┘  │
│                                      │
└─────────────────────────────────────┘
```
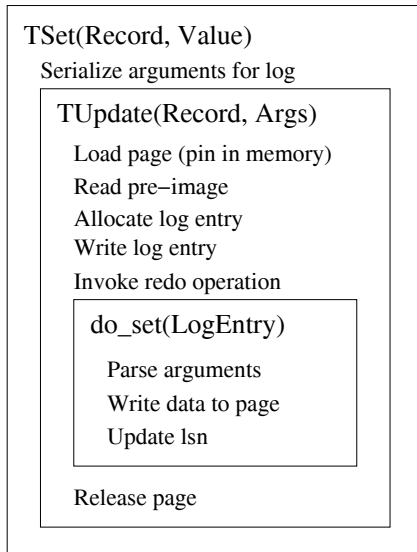
Figure 2: Runtime behavior of a simple operation. Tset() and do_set() are implemented as extensions, while Tupdate() is built in. New operations need not be aware of the complexities of LLADD.

obvious and allows redo functions to be reused to implement undo.

Although any latches acquired by the wrapper function will not be reacquired during recovery, the redo phase of the recovery process is single threaded. Since latches acquired by the wrapper function are held while the log entry and page are updated, the ordering of the log entries and page updates associated with a particular latch must be consistent. Because undo occurs during normal operation, some care must be taken to ensure that undo operations obtain the proper latches.

2.2.3 Concurrency and Aborted Transactions

[move to later?]

Section 2.1 states that LLADD does not allow cascading aborts, implying that operation implementors must protect transactions from any structural changes made to data structures by uncomitted transactions, but LLADD does not provide any mechanisms designed for long-term locking. However, one of LLADD's goals is to make it easy to implement custom data structures for use within safe, multi-threaded transactions. Clearly, an additional mechanism is needed.

The solution is to allow portions of an operation to 'commit' before the operation returns.[4] An op-

eration's wrapper is just a normal function, and therefore may generate multiple log entries. First, it writes an undo-only entry to the log. This entry will cause the *logical* inverse of the current operation to be performed at recovery or abort, must be idempotent, and must fail gracefully if applied to a version of the database that does not contain the results of the current operation. Also, it must behave correctly even if an arbitrary number of intervening operations are performed on the data structure.

[TODO...this next paragraph doesn't make sense; also maybe move this whole subsection to later, since it is complicated] The remaining log entries are redo-only, and may perform structural modifications to the data structure. They should not make any assumptions about the consistency of the current version of the database. Finally, any prefix of the sequence of the redo-only operations performed by this operation must leave the database in a consistent state. The B^{LINK} tree [...] is an example of a B-Tree implementation that behaves in this way, as is the linear hash table implementation discussed in Section 4.1.

Some of the logging constraints introduced in this section may seem strange at this point, but are motivated by the recovery process.

[TODO...need to explain this...]

2.3 Recovery

2.3.1 ANALYSIS / REDO / UNDO

Recovery in AIRES consists of three stages, analysis, redo and undo . The first, analysis, is implemented by LLADD, but will not be discussed in this paper. The second, redo, ensures that each redo entry in the log will have been applied each page in the page file exactly once. The third phase, undo, rolls back any transactions that were active when the crash occured, as though the application manually aborted them with the "abort()" call.

After the analysis phase, the on-disk version of the page file is in the same state it was in when LLADD crashed. This means that some subset of the page updates performed during normal operation have made it to disk, and that the log contains full redo and undo information for the version of each page present in the page file.[5] However, we

---

[4]We considered the use of nested top actions, which LLADD could easily support. However, we currently use the slightly simpler (and lighter-weight) mechanism described here. If the need arises, we will add support for nested top actions.

[5]Although this discussion assumes that the entire log is present, the ARIES algorithm supports log truncation, which allows us to discard old portions of the log, bounding its size

make no further assumptions regarding the order in which pages were propogated to disk. Therefore, redo must assume that any data structures, lookup tables, etc. that span more than a single page are in an inconsistent state. Therefore, as the redo phase re-applies the information in the log to the page file, it must address all pages directly.

Therefore, the redo information for each operation in the log must contain the physical address (page number) of the information that it modifies, and the portion of the operation executed by a single log entry must only rely upon the contents of the page that the log entry refers to. Since we assume that pages are propagated to disk atomically, the REDO phase may rely upon information contained within a single page.

Once redo completes, we have applied some prefix of the run-time log that contains complete entries for all committed transactions. Therefore, we know that the page file is in a physically consistent state, although it contains portions of the results of un-comitted transactions. The final stage of recovery is the undo phase, which simply aborts all uncomitted transactions. Since the page file is physically consistent, the transactions are aborted exactly as they would be during normal operation.

2.3.2 Physical, Logical and Phisiological Logging.

The above discussion avoided the use of some terminology that is common in the database literature and which should be presented here. "Physical loggging" is the practice of logging physical (byte level) upates and the physical (page number) addresses that they are applied to.

It is subtly different than "physiological logging," which is what LLADD recommends for its redo records. In physiological logging, the physical (page number) address is stored, but the byte offset and the actual difference are stored implicitly in the parameters of some function. When the parameters are applied to the function, it will update the page in a way that preserves application semantics. This allows for some convenient optimizations. For example, data within a single page can be re-arranged at runtime to produce contiguous regions of free space, or the parameters passed to the function may be significantly smaller than the physical change made to the page.

"Logical logging" can only be used for undo entries in LLADD, and is identical to physiological logging, except that it stores a logical address (the key of a

hash table, for instance) instead of a physical address. This allows the location of data in the page file to change, even if outstanding transactions may have to roll back changes made to that data. Clearly, for LLADD to be able to apply logical log entries, the page file must be physically consistent, ruling out use of logical logging for redo operations.

LLADD supports all three types of logging, and allows developers to register new operations, which is the key to its extensibility. After discussing LLADD's architecture, we will revisit this topic with a concrete example.

2.4 Summary

This section presented a relatively simple set of rules and patterns that a developer must follow in order to implement a durable, transactional and highly-concurrent data structure using LLADD:

- Pages should only be updated inside of a redo or undo function.

- An update to a page should update the LSN.

- If the data read by the wrapper function must match the state of the page that the redo function sees, then the wrapper should latch the relevant data.

- Redo operations should address pages by their physical offset, while Undo operations should use a more permenant address (such as index key) if the data may move between pages over time.

- An undo operation must correctly update a data structure if any prefix of its corresponding redo operations are applied to the structure, and if any number of intervening operations are applied to the structure.

Because undo and redo operations during normal operation and recovery are similar, most bugs will be found with conventional testing strategies. It is difficult to verify the final property, although a number of tools could be written to simulate various crash scenarios, and check the behavior of operations under these scenarios.

Note that the ARIES algorithm is extremely complex, and we have left out most of the details needed to understand how ARIES works, or to implement it correctly.[6] Yet, we believe we have covered everything that a programmer needs to know in order to

---

on disk.

[6]The original ARIES paper was around 70 pages, and the ARIES/IM paper, which covered index implementation is roughly the same length.
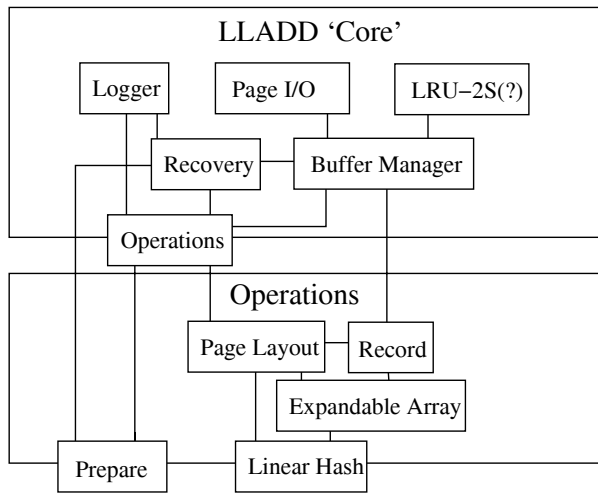
Figure 3: Simplified LLADD Architecture: The core of the library places as few restrictions on the application's data layout as possible. Custom "operations" implement the client's desired data layout.

systems independently, and have documented both external and internal interfaces, making it easy to add new tests and debug old ones. Furthermore, by adding a 'simulate crash' operation to a few of the key components, we can simulate application level crashes by clearing LLADD's internal state, re-initializing the library and verifying that recovery was successful. These tests currently cover approximately 90% of the code. We have not yet developed a mechanism that will allow us to accurately model hardware failures, which is an area where futher work is needed. However, the basis for this work will be the development of test harnesses that verify operation behavior in exceptional circumstances.

LLADD's performance requirements vary wildly depending on the workload with which it is presented. Its performance on a large number of small, sequential transactions will always be limited by the amount time required to flush a page to disk. To some extent, compact logical and physiological log entries improve this situation. On the other hand, long running transactions only rarely force-write to disk and become CPU bound. Standard profiling techniques of the overall library's performance and microbenchmarks of crucial modules handle such situations nicely.

A more interesting set of performance requirements are imposed by multithreaded workloads. Each module of LLADD is reentrant, and a C preprocessor directive allows the entire library to be instrumented in order to profile latching behavior, which is useful both for perfomance tuning and for debugging purposes. A thread that is not involved in an I/O request never needs to wait for a latch held by a thread that is waiting for I/O.[8]

There are a number of performance optimizations that are specific to multithreaded operations that we do not perform. The most glaring omission is log bundling; if multiple transactions commit at once, LLADD must force the log to disk one time per transaction. This problem is not fundamental, but simply has not made it into the current code base. Similarly, since page eviction requires a force-write if the full ARIES recovery algorithm is in use, we could implement a thread that asynchronously maintained a set of free buffer pages. We plan to implement such optimizations, but they are not reflected in this paper's performance figures.

4   Sample Operations

In order to validate LLADD's architecture, and to show that it simplifies the creation of efficient data structures, we have have implemented a number of simple extensions. In this section, we describe their design, and provide some concrete examples of our experiences extending LLADD.

4.1   Linear Hash Table

Linear hash tables are hash tables that are able to extend their bucket list incrementally at runtime. They work as follows. Imagine that we want to double the size of a hash table of size 2^n, and that the hash table has been constructed with some hash function $h_n(x) = h(x) \bmod 2^n$ Choose $h_{n+1}(x) = h(x) \bmod 2^{n+1}$ as the hash function for the new table. Conceptually we are simply prepending a random bit to the old value of the hash function, so all lower order bits remain the same. At this point, we could simply block all concurrent access and iterate over the entire hash table, reinserting values according to the new hash function.

However, because of the way we chose $h_{n+1}(x)$, we know that the contents of each bucket, m, will be split betwen bucket m and bucket $m+2^n$. Therefore, if we keep track of the last bucket that was split, we can split a few buckets at a time, resizing the hash table without introducing long pauses while we reorganize the hash table.[...] We can handle overflow using standard techniques. LLADD's linear hash table uses linked lists of overflow buckets.

For this scheme to work, we must be able to address a portion of the page file as though it were an expandable array. We have implemented this functionality as a separate module, but will not discuss it here.

For the purposes of comparison, we provide two linear hash implementations. The first is straightforward, and is layered on top of LLADD's standard record setting operation, Tset(), and therefore performs physical undo. This implementation provided a stepping stone to the more sophisticated version which employs logical undo, and uses an identical on-disk layout. As we discussed earlier, logical undo provides more opportunities for concurrency, while decreasing the size of log entries. In fact, the physical-undo implementation of the linear hash table cannot support concurrent transactions, while threads utilizing the physical-undo implementation never hold locks on more than two buckets.[9]

---

[8]Strictly speaking, this statement is only true for the LLADD's core. However, there are variants of most popular data structures that allow us to preserve these invariants. LLADD can correctly support operations whether or not they have these properties.

---

[9]However, only one thread may expand the hashtable at once. In order to amortize the overhead of initiating an ex-

(2) Lock Bucket

(3)

(4)

New

(1) Allocate New Block

Insert Bucket Entry

(1) Lock Bucket

(2)

Old

(3) Free old Block

Delete Bucket Entry

(1) Lock Bucket

(4)

(3)

(5) Delete pointer
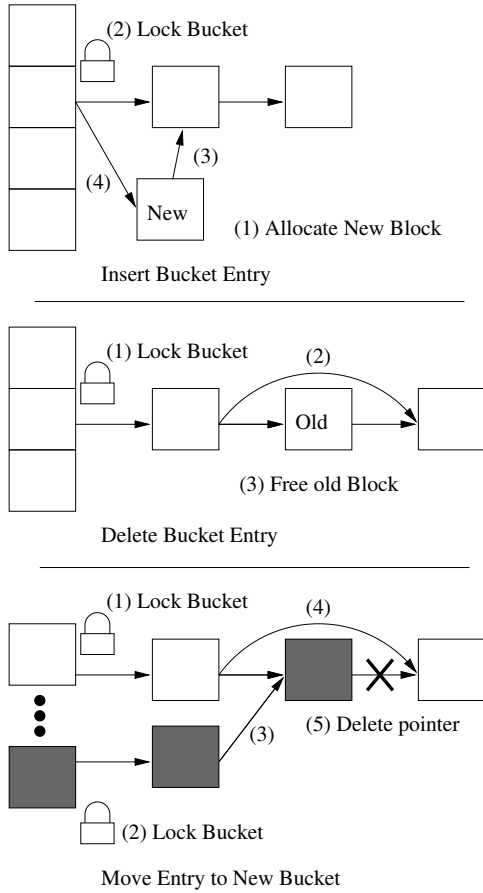
(2) Lock Bucket

Move Entry to New Bucket

Figure 4: Linear Hash Table Bucket operations.

Because another module provides the resizable arrays needed for the bucket list, the complexity of the linear hash algorithm is in two areas. The first, linked list management, is straightforward in the physical case, but must be performed in a specific order in the logical case. See Figure 4 for a sequence of steps that safely implement the necessary linked list operations. Note that in the first two cases, the portion of the linked list that is visible from LLADD's point of view is always consistent. This is important for crash recovery; it is possible that LLADD will crash before the entire sequence of operations has been completed. The logging protocol guarantees that some prefix of the log will be available. Therefore, as long as the run-time version of the hash table is always consistent, we do not have to consider the impact of skipped updates, but we must be certain that the logical consistency of the linked list is maintained at all steps. Here, the challenge comes from the fact that the buffer manager only provides atomic updates of single pages; in practice, a linked list may span pages.

The last case, where buckets are split as the bucket list is expanded, is a bit more complicated. We must maintain consistency between two linked lists, and a page at the begining of the hash table that contains the last bucket that we successfully split. Here, we misuse the undo entry to ensure proper crash recovery. Our bucket split algorithm is idempotent, so it may be applied an arbitrary number of times to a given bucket with no ill-effects. Also note that (for our purposes), there is never a good reason to undo a bucket split, so we can safely apply the split whether or not the current transaction commits.

First, an "undo" record that checks the hash table's meta data and redoes the split if necessary is written (this record has no effect unless we crash during this bucket split). Second, we write (and execute) a series of redo-only records to the log. These encode the bucket split, and follow the linked list protocols listed above. Finally, we write a redo-only entry that updates the hash table's metadata.[10]

We allow pointer aliasing at this step so that a given key can be present for a short period of time

---

pansion, and to allow concurrent insertions, the hash table is expanded in increments of a few thousand buckets.

[10]Had we been using nested top actions, we would not need the special undo entry, but we would need to store *physical* undo information for each of the modifications made to the bucket, since any subset of the pages may have been stolen. This method does have the disadvantage of producing a few redo-only entries during recovery, but recovery is an uncommon case, and the number of such entries is bounded by the number of entries that would be produced during normal operation.

in both buckets. If we crash before the undo entry is written, no harm is done. If we crash after the entire update makes it to log, the redo stage will set the hash's metadata appropriately, and the 'undo' record becomes a no-op. If we crash in the middle of the bucket split, we know that the current transaction did not commit, and that recovery will execute the 'undo' record. It will see that the bucket split is still pending and finish splitting the bucket appropriately. Since the bucket split is idempotent, and we've arranged for it to behave correctly regardless of the point at which it was interrupted, the hastable is correctly restored.

Note that there is a point during the undo phase where the bucket is in an inconsistent physical state, although normally the redo phase is able to bring the database to a fully consistent physical state. We handle this by obtaining a runtime lock on the bucket during normal operation. This runtime lock blocks any attempt to write log entries that alter a bucket that is being split, so we know that no other logical operations will attempt to access an inconsistent bucket.

Since the second implementation of the linear hash table uses logical undo, we are able to allow concurrent updates to different portions of the table. This is not true in the case of the implementation that uses pure physical logging, as physical undo cannot generally tolerate concurrent structural modifications to data structures.

4.2 Two Phase Commit

The two phase commit protocol is used in clustering applications where multiple, well maintained, well connected computers must agree upon a set of successful transactions. Some of the systems could crash, or the network could fail during operation, but we assume that such failures are temporary. Two phase commit designates a single computer as the coordinator of a given transaction. This computer contacts the other systems participating in the transaction, and asks them to prepare to commit the transaction. If a subordinate system sees that an error has occurred, or the transaction should be aborted for some other reason, then it informs the coordinator. Otherwise, it enters the *prepared* state, and tells the coordinator that it is ready to commit. At some point in the future, the coordinator will reply telling the subordinate to commit or abort. From LLADD's point of view, the interesting portion of this algorithm is the *prepared* state, since it must be able to commit a prepared transaction if it crashes before the coordinator responds, but cannot commit

before hearing the response, since it may be asked to abort the transaction.

Implementing the prepare state on top of the ARIES algorithm constists of writing a special log entry that informs the undo portion of the recovery phase that it should stop rolling back the current transaction and instead add it to the list of active transactions.[11] Due to LLADD's extendible logging system, and the simplicity of its recovery code, it took an afternoon to add a prepare operation to LLADD.

5 Performance

We hope that the preceeding sections have given the reader an idea of the usefulness and extensibility of the LLADD library. In this section we focus on performance evaluation.

In order to evaluate the physical and logical hashtable implementations, we first ran a test that inserts some tuples into the database. For this test, we chose fixed length key, values pairs of integers. For simplicity, our hashtable implementations currently only support fixed-length keys and values, so this this test puts us at a significant advantage. It also provides an example of the type of workload that LLADD handles well, since LLADD is specifically designed to support application specific transactional data structures. For comparison, we ran "Record Number" trials, named after the BerkeleyDB access method. In this case, the two programs essentially stored the data in a large array on disk. This test provides a measurement of the speed of the lowest level primitive supported by BerkeleyDB.

The times included in Figure 5 include page file and log creation, insertion of the tuples as a single transaction, and a clean program shutdown. We used the 'transapp.cs' program from the Berkeley DB 4.2 tutorial to run the Berkeley DB tests, and hardcoded it to use integers instead of strings. We used the Berkeley DB "DB_HASH" index type for the hashtable implementation, and "DB_RECNO" in order to run the "Record Number" test.

Since LLADD addresses records as {Page, Slot, Size} triples, which is a lower level interface than Berkeley DB exports, we used the expandable array that supports the hashtable implementation to run the "LLADD Record Number" test.

One should not look at Figure 5, and conclude "LLADD is almost five times faster than Berke-

---

[11] Also, any locks that the transaction obtained should be restored, which is outside of the scope of LLADD, although this functionality could be added relatively easily if a lock manager were implemented on top of LLADD.
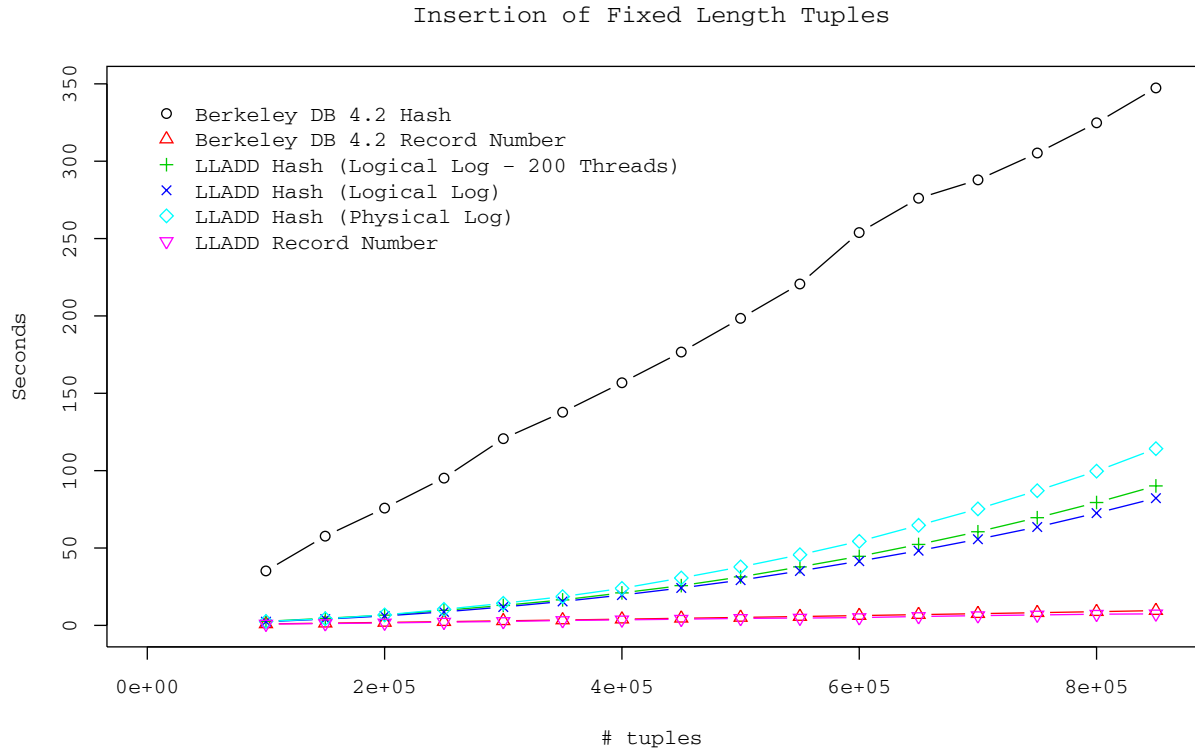
Insertion of Fixed Length Tuples

Figure 5: The final data points for LLADD's and Berkeley DB's record number based storage are 7.4 and 9.5 seconds, respectively. LLADD's hash table is significantly faster than Berkeley DB in this test, but provides less functionality than the Berkeley DB hash. Finally, the logical logging version of LLADD's hash table is faster than the physical version, and handles the multi-threaded test well. The threaded test spawned 200 threads and split its workload into 200 separate transactions.

ley DB," since we chose a hash table implementation that is tuned for fixed-length data. Instead, the conclusions we draw from this test are that, first, LLADD's primitive operations are on par, perforance wise, with Berkeley DB's, which we find very encouraging. Second, even a highly tuned implementation of a 'simple,' general purpose data structure is not without overhead, and for applications where performance is important a special purpose structure may be appropriate.

Also, the multithreaded test run shows that the library is capable of handling a large number of threads. The performance degradation associated with running 200 concurrent threads was negligible. The logical logging version of LLADD's hashtable outperformed the physical logging version for two reasons. First, since it writes fewer undo records, it generates a smaller log file. Second, in order to emphasize the performance benefits of our extension mechanism, we use lower level primitives for the logical logging version. The logical logging version implements locking at the bucket level, so many mutexes that are acquired by LLADD's default mechanisms are redundant. The physical logging version of the hashtable serves as a rough proxy for an implementation on top of a non-extendible system. Therefore, it uses LLADD's default mechanisms, which include the redundant acquisition of locks.

As a final note on our performance graph, we would like to address the fact that LLADD's hashtable curve is non-linear. LLADD currently uses a fixed-size in-memory hashtable implementation in many areas, and it is possible that we exceeded the fixed-size of this hashtable on the larger test sets. Also, LLADD's buffer manager is currently fixed size. Regardless of the cause of this non-linearity, we do not believe that it is fundamental to our implementation.

6   Future Work

LLADD is an extendible implementation of the ARIES algorithm. This allows application developers to incorporate transactional recovery into a wide range of systems. We have a few ideas along these lines, and also have some ideas for extensions to LLADD itself.

LLADD currently relies upon its buffer manager for page-oriented storage. Although we did not have space to discuss it in this paper, we have a blob implementation that stores large data outside of the page file. This concept could be extended to arbitrary primitives, such as transactional updates to file system directory trees, or integration of networking

or other operations directly into LLADD transactions. Doing this would allow LLADD to act as a sort of "glue code" among various systems, ensuring data integrity and adding database-style functionality, such as continuous backup to systems that currently do not provide such mechanisms. We believe that there is quite a bit of room for the development of new software systems in the space between the high-level, but sometimes inappropriate interfaces exported by database servers, and the low-level, general-purpose primitives supported by current file systems.

Currently, although we have implemented a two-phase commit algorithm, LLADD really is not very network aware. If we provided a clean abstraction that allowed LLADD extensions and operations to cross network boundaries, then we could provide a wider range of network consistency algorithms, and cleanly support the implementation of operations that perform well in networked and in local environments.

Although LLADD is re-entrant, its latching mechanisms only provide physical consistency. Traditionally, lock managers, which provide higher levels of consistency, have been tightly coupled with transactional page implementations. Generally, the semantics of undo and redo operations provided by the transactional page layer and its associated data structures determine the level of concurrency that is possible. Since prior systems provide a monolithic set of primitives to their users, these systems typically had complex interactions among the lock manager, on-disk formats and the transactional page layer. Finally, at recovery time it is often desirable to reacquire locks on behalf of a transaction. Without extensible logging and without modifying the recovery code, it is impossible to ensure that such locks are correctly restored. By providing extensible logging, data-structures, and undo/redo semantics, LLADD removes these reasons for coupling the lock manager and the rest of the storage mechanisms. The flexiblity offered by splitting the lock manager and the ARIES algorithm into independent sub-systems, and allowing users to independently extend either module seems to outweigh the extra complexity that will be added to LLADD's interface. In particular, most difficulties related to locking seem to be data-structure dependent, suggesting that, like page layout or the semantics of various types of log entires, they are largely orthogonal to the atomicity and durability algorithms implemented by LLADD.

By releasing LLADD to the community, we hope that we will be able to provide a toolkit that aids in the development of real-world applications, and

is flexible enough for use as a research platform.

7 Conclusion

8 Acknowledgements

9 Availability

LLADD is free software, available at:

`http://www.sourceforge.net/projects/lladd`

References

[Beazley] D. M. Beazley and P. S. Lomdahl, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5*, Parall. Comp. 20 (1994) p. 173-195.

[RealName] A. N. Author and A. N. Other, *Title of Riveting Article*, JournalName VolNum (Year) p. Start-End

[ET] Embedded Tk, `ftp://ftp.vnet.net/pub/users/drh/ET.html`

[Expect] Don Libes, *Exploring Expect*, O'Reilly & Associates, Inc. (1995).

[Heidrich] Wolfgang Heidrich and Philipp Slusallek, *Automatic Generation of Tcl Bindings for C and C++ Libraries.*, USENIX 3rd Annual Tcl/Tk Workshop (1995).

[Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).

[Perl5] Perl5 Programmers reference, `http://www.metronet.com/perlinfo/doc`, (1996).

[Wetherall] D. Wetherall, C. J. Lindblad, "Extending Tcl for Dynamic Object-Oriented Programming", Proceedings of the USENIX 3rd Annual Tcl/Tk Workshop (1995).