Lemon: A Flexible Transactional Storage System

Paper 198

Existing transactional systems are designed to handle specific workloads well. Unfortunately, these implementations are generally monolithic and hide the transaction support under a SQL interface, which forces many systems to "work around" the relational data model. Manifestations of this problem include the the poor fit of existing transactional storage systems to persistent objects and hierarchical or semi-structured data, such as XML or scientific data. This work proposes a novel flexible transaction framework intended for non-database transactional systems; for example, Lemon makes it is easy to develop high-performance transactional data structures. It generally outperforms Berkeley DB, and its extensibility enables optimizations that outperform Berkeley DB by 2x and MySQL by up to 5x. We present novel optimizations for object serialization and graph traversal that demonstrate this flexibility.

1 Introduction

Transactions are at the core of databases and thus form the basis of many important systems. However, the mechanisms that provide transactions are typically hidden within monolithic database implementations (DBMSs) that make it hard to benefit from transactions without inheriting the rest of the database machinery and design decisions, including the use of a query interface. Although this is clearly not a problem for databases, it impedes the use of transactions in a wider range of systems.

Other systems that could benefit from transactions include file systems, version-control systems, bioinformatics, workflow applications, search engines, recoverable virtual memory, and programming languages with persistent objects.

In essence, there is an *impedance mismatch* between the data model provided by a DBMS and that required by these applications. This is not an accident: the purpose of the relational model is exactly to move to a higher-level set-based data model that avoids the kind of "navigational" interactions required by these lower-level systems. Thus in some sense, we are arguing for the development of modern navigational transaction systems that can compliment relational systems and that naturally support current system designs and development methodologies.

The most obvious example of this mismatch is in the support for persistent objects in Java, called *Enterprise Java Beans* (EJB). In a typical usage, an array of objects is made persistent by mapping each object to a row in a table[1] and then issuing queries to keep the objects and rows consistent. A typical update must confirm it has the current version, modify the object, write out a serialized version using the SQL `update` command and commit. This is an awkward and slow mechanism; we show up to a 5x speedup over a MySQL implementation that is optimized for single-threaded, local access (Section 7).

[1]Normalized objects may actually span many tables [10].

The DBMS actually has a navigational transaction system within it, which would be of great use to EJB, but it is not accessible except via the query language. In general, this occurs because the internal transaction system is complex and highly optimized for high-performance update-in-place transactions.

In this paper we introduce Lemon, a flexible framework for ACID transactions that is intended to support a broader range of applications. Although we believe Lemon could also be the basis of a DBMS, there are already many excellent DBMS solutions, and we thus focus on the rest of the applications. The primary goal of Lemon is to provide flexible and complete transactions.

By *flexible* we mean that Lemon can implement a wide range of transactional data structures, that it can support a variety of policies for locking, commit, clusters and buffer management. Also, it is extensible for both new core operations and new data structures. It is this flexibility that allows the support of a wide range of systems.

By *complete* we mean full redo/undo logging that supports both *no force*, which provides durability with only log writes, and *steal*, which allows dirty pages to be written out prematurely to reduce memory pressure.[2] By complete, we also mean support for media recovery, which is the ability to roll forward from an archived copy, and support for error-handling, clusters, and multithreading. These requirements are difficult to meet and form the *raison d'être* for Lemon: the framework delivers these properties as reusable building blocks for systems to implement complete transactions.

With these trends in mind, we have implemented a modular, extensible transaction system based on on ARIES that makes as few assumptions as possible about application data and workloads. Where such assumptions are inevitable, we allow the developer to plug in alternative implementations or define custom operations whenever possible. Rather than hiding the underlying complexity of the library from developers, we have produced narrow, simple APIs and a set of invariants that must be maintained in order to ensure transactional consistency. This allows developers to produce high-performance extensions with only a little effort.

Specifically, application developers using Lemon can control: 1) on-disk representations, 2) data structure implementations (including adding new transactional access methods), 3) the granularity of concurrency, 4) the precise semantics

[2]A note on terminology: by "dirty" we mean pages that contain uncommitted updates; this is the DB use of the word. Similarly, "no force" does not mean "no flush", which is the practice of delaying the log write for better performance at the risk of losing committed data. We support both versions.

of atomicity, isolation and durability, 5) request scheduling policies, and 6) deadlock detection and avoidance schemes. Developers can also exploit application-specific or workload-specific assumptions to improve performance. These features are enabled by the several mechanisms:

Flexible page layouts provide low-level control over transactional data representations (Section 4.2).

Extensible log formats provide high-level control over transaction data structures (Section 4.4).

High- and low-level control over the log such as calls to "log this operation" or "write a compensation record" (Section 3.3).

In memory logical logging provides a data store independent record of application requests, allowing "in flight" log reordering, manipulation and durability primitives to be developed (Section 8).

Extensible locking API provides registration of custom lock managers and a generic lock manager implementation (Section 4.1).

Custom durability operations such as two-phase commit's prepare call, and savepoints (Section 7).

We have produced a high-concurrency, high-performance and reusable open-source implementation of our system. Portions of our implementation's API are still changing, but the interfaces to low-level primitives, and the most important portions of the implementation have stabilized.

To validate these claims, we walk through a sequence of optimizations for a transactional hash table in Section 6, an object serialization scheme in Section 7 and a graph traversal algorithm in Section 8. Benchmarking figures are provided for each application. Lemon also includes a cluster hash table built upon two-phase commit, which will not be described. Similarly we did not have space to discuss Lemon's blob implementation, which demonstrates how Lemon can add transactional primitives to data stored in a file system.

2 Prior work

A large amount of prior work exists in the field of transactional data processing. Instead of providing a comprehensive summary of this work, we discuss a representative sample of the systems that are presently in use, and explain how our work differs from existing systems.

Relational databases excel in areas where performance is important, but where the consistency and durability of the data are more important. Often, databases significantly outlive the software that uses them, and must be able to cope with changes in business practices, system architectures, etc., which leads to the relational model [4].

For simpler applications, such as normal web servers, full DBMS solutions are overkill and expensive. MySQL [17] has largely filled this gap by providing a simpler, less concurrent database that can work with a variety of storage options including Berkeley DB (covered below) and regular files. However, these alternatives affect the semantics of transactions and sometimes disable or interfere with high-level database features. MySQL includes multiple storage options for performance reasons. We argue that by reusing code, and providing for a greater amount of customization, a modular storage engine can provide better performance, transparency and flexibility than a set of monolithic storage engines.

The Postgres storage system [22] provides conventional database functionality, but also provides APIs that allow applications to add new index and object types [21]. Although some of the methods are similar to ours, Lemon also implements a lower-level interface that can coexist with these methods. Without Lemon's low-level APIs, Postgres suffers from many of the limitations inherent to the database systems mentioned above, as its extensions focus on improving query language and indexing support. Although we believe that many of the high-level Postgres interfaces could be built on top of Lemon, we have not yet tried to implement them, although we have some support for iteration.

Object-oriented and XML database systems provide models tied closely to programming language abstractions or hierarchical data formats. Like the relational model, these models are extremely general, and are often inappropriate for applications with stringent performance demands, or those that use these models in unusual ways. Furthermore, data stored in these databases often is formatted in a way that ties it to a specific application or class of algorithms [11]. We will show that Lemon can provide specialized support for both classes of applications, via a persistent object example (Section 7) and a graph traversal example (Section 8).

The impedance mismatch in the use of database systems to implement certain types of software has not gone unnoticed. In order to serve these applications, many software systems have been developed. 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 thumb nails [18, 6]. Others are simpler, such as Berkeley DB [19], which provides transactional storage of data in indexed form using a hashtable or tree, or as a queue.

Although Berkeley DB's feature set is similar to the features provided by Lemon's implementation, there is an important distinction. Berkeley DB provides general implementations of a handful of transactional structures and provides flags to enable or tweak certain pieces of functionality such as lock management, log forces, and so on. Although Lemon provides some of the high-level calls that Berkeley DB supports (and could probably be extended to provide most or all of these calls), Lemon provides lower-level access to transactional primitives and provides a rich set of mechanisms that make it easy to use these primitives. For instance, Berkeley DB does not provide access methods to access data by page offset, and does not provide applications with primitive access methods to facilitate the development of higher-level structures. It also seems to be difficult to specialize existing Berkeley DB functionality (for example page layouts) for new extensions. We will show that such functionality is useful.

LRVM is a version of malloc() that provides durable memory, and is similar to an object-oriented database but is much lighter weight, and lower level [20]. Unlike the solutions mentioned above, it does not impose limitations upon the layout of application data, although it does not provide full transactions. LRVM's approach of keeping a single in-memory copy of data in the application's address space is similar to the optimization presented in Section 7, but our approach includes full support for concurrent transactional data structures as well.

Finally, some applications require incredibly simple but extremely scalable storage mechanisms. Cluster hash tables [9] are a good example of the type of system that serves these applications well, due to their relative simplicity and good scalability. Depending on the fault model on which a cluster hash table is based, 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. Possibilities include in-memory replication across many nodes, or spooling logical logs to disk on dedicated servers. Similarly, atomicity semantics may be relaxed under certain circumstances. Lemon is unique in that it can support the full range of semantics, from in-memory replication for commit, to full transactions involving multiple entries, which is not supported by any of the current CHT implementations.

Boxwood provides a networked, fault-tolerant transactional B-Tree and "Chunk Manager". We believe that Lemon could be a valuable part of such a system. However, we believe that Lemon's concept of a page file and system-independent logical log suggest an alternative approach to fault-tolerant storage design, which we hope to explore in future work.

3 Write-ahead Logging Overview

This section describes how existing write-ahead logging protocols implement the four properties of transactional storage: Atomicity, Consistency, Isolation and Durability. Lemon provides these properties and also allows applications to opt-out of them as appropriate. This can be useful for performance reasons or to simplify the mapping between application semantics and the storage layer. Unlike prior work, Lemon also exposes the primitives described below to application developers, allowing unanticipated optimizations and allowing changes to be made to low-level behavior such as recovery semantics on a per-application basis.

The write-ahead logging algorithm we use is based upon ARIES, but has been modified for extensibility and flexibility. Because comprehensive discussions of write-ahead logging protocols and ARIES are available elsewhere [9, 14], we focus on those details that are most important for flexibility, and provide a concrete example in Section 4.

3.1 Operations

A transaction consists of an arbitrary combination of actions, that is protected according to the ACID properties mentioned above. Typically, the information necessary to REDO and UNDO each action is stored in the log. We refine this concept and explicitly discuss *operations*, which must be atomically applicable to the page file.

Lemon is essentially a framework for transactional pages: each page is independent and can be recovered independently. For now, we simply assume that operations do not span pages. Since single pages are written to disk atomically, we have a simple atomic primitive on which to build. In Section 4.3, we explain how to handle operations that span multiple pages.

One unique aspect of Lemon, which is not true for ARIES, is that *normal* operations are defined in terms of REDO and UNDO functions. There is no way to modify the page except via the REDO function.[3] This has the nice property that the REDO code is known to work, since the original operation was the exact same "redo". In general, the Lemon philosophy is that you define operations in terms of their REDO/UNDO behavior, and then build a user-friendly *wrapper* interface around them (Figure 1). The value of Lemon is that it provides a skeleton that invokes the REDO/UNDO functions at the *right* time, despite concurrency, crashes, media failures, and aborted transactions. Also unlike ARIES, Lemon refines the concept of the wrapper interface, making it possible to reschedule operations according to an application-level policy (Section 8).

3.2 Isolation

We allow transactions to be interleaved, allowing concurrent access to application data and exploiting opportunities for hardware parallelism. Therefore, each action must assume that the data upon which it relies may contain uncommitted information that might be undone due to a crash or an abort.

Therefore, in order to implement an operation we must also implement synchronization mechanisms that isolate the effects of transactions from each other. We use the term *latching* to refer to synchronization mechanisms that protect the physical consistency of Lemon's internal data structures and the data store. We say *locking* when we refer to mechanisms that provide some level of isolation among transactions. For locking, due to the variety of locking protocols available and degrees of

[3] Actually, even this can be overridden, but doing so complicates recovery semantics, and only should be done as a last resort. Currently, this is only done to implement the Juicer flush() and update() operations described in Section 7.

isolation available, we leave it to the application via the mock manager API (Section 4.1).

Lemon operations that allow concurrent requests must provide a latching implementation that is guaranteed not to deadlock. These implementations need not ensure consistency of application data. Instead, they must maintain the consistency of any underlying data structures. Generally, latches do not persist across calls performed by high-level code, as that could lead to deadlock.

3.3 Log Manager

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

An important concept in ARIES is the "log sequence number" or *LSN*. An LSN is essentially a virtual time stamp that goes on every page; it marks the last log entry that is reflected on the page and implies that *all previous log entries* are also reflected. Given the LSN, Lemon calculates where to start playing back the log to bring the page up to date. The LSN is stored in the page that it refers to so that it is always written to disk atomically with the data on the page.

ARIES (and thus Lemon) 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 increases the 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 can never be written back. To handle stolen pages, we log UNDO records that we can use to undo the uncommitted changes in case we crash. Lemon ensures that the UNDO record is durable in the log before the page is written to disk and that the page LSN reflects this log entry.

Similarly, we do not *force* pages out to disk when a transaction commits, as this limits performance. Instead, we log REDO records that we can use to redo the operation in case the committed version never makes it to disk. Lemon 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 order.

3.4 Recovery

We use the same basic recovery strategy as ARIES, which consists of three phases: *analysis*, *redo* and *undo*. The first, analysis, is implemented by Lemon, but will not be discussed in this paper. The second, redo, ensures that each REDO entry is applied to its corresponding page exactly once. The third phase, undo, rolls back any transactions that were active when the crash occurred, as though the application manually aborted them with the "abort" function call.

After the analysis phase, the on-disk version of the page file is in the same state it was in when Lemon 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.[4] Because we make no further assumptions regarding the order in which pages were propagated to disk, redo must assume that any data structures, lookup tables, etc. that span more than a single page are in an inconsistent state.

This implies that 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 REDO log entry must only rely upon the contents of that page.

Once redo completes, we have essentially repeated history: replaying all REDO entries to ensure that the page file is in a physically consistent state. However, we also replayed updates from transactions that should be aborted, as they were still in progress at the time of the crash. The final stage of recovery is the undo phase, which simply aborts all uncommitted transactions. Since the page file is physically consistent, the transactions may be aborted exactly as they would be during normal operation.

One of the nice properties of ARIES, which is supported by Lemon, 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. Because pages can be recovered independently from each other, there is no need to stop transactions to make a snapshot for archiving: any fuzzy snapshot is fine.

4 Flexible, Extensible Transactions

As long as operation implementations obey the atomicity constraints outlined above and correctly manipulate on-disk data structures, the write-ahead logging protocol will provide correct ACID transactional semantics, and high performance, concurrent and scalable access to application data. This suggests a natural partitioning of transactional storage mechanisms into two parts (Figure 1).

The lower layer implements the write-ahead logging component, including a buffer pool, logger, and (optionally) a lock manager. The complexity of the write-ahead logging component lies in determining exactly when the UNDO and REDO operations should be applied, when pages may be flushed to disk, log truncation, logging optimizations, and a large number of other data-independent extensions and optimizations. This layer is the core of Lemon.

The upper layer, which can be authored by the application developer, provides the actual data structure implementations,

[4] Although this discussion assumes that the entire log is present, it also works with a truncated log and an archive copy.

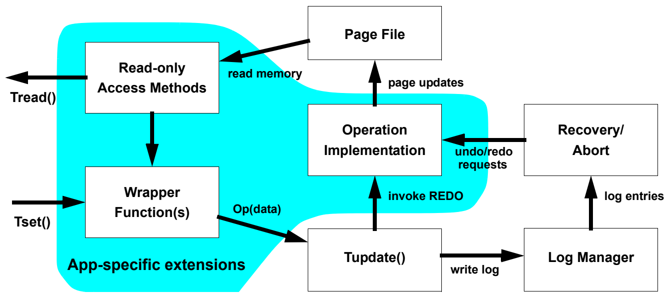


Figure 1: Lemon architecture. The shaded region covers extensions which we call *operations*. Arrows point in the direction of application data flow. Note that writes to the page file and log are protected by the Tupdate() call, and that wrapper functions may be built upon each other. Operation implementations are automatically invoked by the transactional library. Not shown are a set of convenience functions that make it easy to write high level operations and wrappers.

policies regarding page layout, and the implementation of any application-specific operations. As long as each layer provides well defined interfaces, the application, operation implementation, and write-ahead logging component can be independently extended and improved.

We have implemented a number of simple, high performance and general-purpose data structures. These are used by our sample applications and as building blocks for new data structures. Example data structures include two distinct linked-list implementations, and a growable array. Surprisingly, even these simple operations have important performance characteristics that are not available from existing systems. The remainder of this section is devoted to a description of the various primitives that Lemon provides to application developers.

4.1 Lock Manager

Lemon provides a default page-level lock manager that performs deadlock detection, although we expect many applications to make use of deadlock-avoidance schemes, which are already prevalent in multithreaded application development. The lock manager is flexible enough to also provide index locks for hashtable implementations and more complex locking protocols such as hierarchical two-phase locking [8, 16]. The lock manager API is divided into callback functions that are made during normal operation and recovery, and into generic lock manager implementations that may be used with Lemon and its index implementations.

However, applications that make use of a lock manager must handle deadlocked transactions that have been aborted by the lock manager. This is easy if all of the state is managed by Lemon, but other state such as thread stacks must be handled by the application, much like exception handling. Lemon currently uses a custom wrapper around the pthread cancellation mechanism to provide partial stack unwinding and pthread's thread cancellation mechanism. Applications may use this error handling technique, or write simple wrappers to handle errors with the error handling scheme of their choice.

Conversely, many applications do not require such a general scheme. If deadlock avoidance ("normal" thread synchronization) can be used, the application does not have to abort partial transactions, repeat work, or deal with the corner cases that aborted transactions create.

4.2 Flexible Logging and Page Layouts

Lemon supports three types of logging, and allows applications to create *custom log entries* of each type.

Physical logging is the practice of logging physical (byte-level) updates and the physical (page-number) addresses to which they are applied.

Physiological logging extends this idea, and is generally used for Lemon's REDO entries. The physical address (page number) is stored, along with the arguments of an arbitrary function that is associated with the log entry.

This is used to implement many primitives, including *slotted pages*, which use an on-page level of indirection to allow records to be rearranged within the page; instead of using the page offset, REDO operations use the index to locate the data within the page. This allows data within a single page to be re-arranged easily, producing contiguous regions of free space. Since the log entry is associated with an arbitrary function more sophisticated log entries can be implemented. In turn, this can improve performance by conserving log space, or be used to match recovery to application semantics.

Lemon also uses this mechanism to support four *page layouts*: *raw-page*, which is just an array of bytes, *fixed-page*, a record-oriented page with fixed-length records, *slotted-page*, which supports variable-sized records, and *versioned-page*, a slotted-page with a separate version number for each record (Section 7.1).

Logical logging uses a higher-level key to specify the UNDO/REDO. Since these higher-level keys may affect multiple pages, they are prohibited for REDO functions, since our REDO is specific to a single page. However, logical logging does make sense for UNDO, since we can assume that the pages are physically consistent when we apply an UNDO. We thus use logical logging to undo operations that span multiple pages, as shown in the next section.

4.3 Nested Top Actions

The operations presented so far work fine for a single page, since each update is atomic. For updates that span multiple pages there are two basic options: full isolation or nested top actions. By full isolation, we mean that no other transactions see the in-progress updates, which can be trivially achieved with a big lock around the whole structure. Usually the application must enforce such a locking policy or decide to use a lock manager and deal with deadlock. Given isolation, Lemon needs nothing else to make multi-page updates transactional: although many pages might be modified they will commit or abort as a group and be recovered accordingly.

However, this level of isolation disallows all concurrency among transactions that use the same data structure. ARIES introduced the notion of nested top actions to address this problem. For example, consider what would happen if one transaction, *A*, rearranged the layout of a data structure, a second transaction, *B*, added a value to the rearranged structure, and then the first transaction aborted. (Note that the structure is not isolated.) While applying physical undo information to the altered data structure, *A* would UNDO its writes without considering the modifications made by *B*, which is likely to cause corruption. Therefore, *B* would have to be aborted as well (*cascading aborts*).

With nested top actions, ARIES defines the structural changes as a mini-transaction. This means that the structural change "commits" even if the containing transaction (*A*) aborts, which ensures that *B*'s update remains valid.

Lemon supports nested atomic actions as the preferred way to build high-performance data structures. In particular, an operation that spans pages can be made atomic by simply wrapping it in a nested top action and obtaining appropriate latches at runtime. This approach reduces development of atomic page spanning operations to something very similar to conventional multithreaded development that uses mutexes for synchronization. In particular, we have found a simple recipe for converting a non-concurrent data structure into a concurrent one, which involves three steps:

1. Wrap a mutex around each operation. If this is done with care, it may be possible to use finer grained mutexes.

2. Define a logical UNDO for each operation (rather than just using a set of page-level UNDOs). For example, this is easy for a hashtable; e.g. the UNDO for an *insert* is *remove*.

3. For mutating operations (not read-only), add a "begin nested top action" right after the mutex acquisition, and a "commit nested top action" right before the mutex is released.

This recipe ensures that operations that might span multiple pages atomically apply and commit any structural changes and thus avoids cascading aborts. If the transaction that encloses the operations aborts, the logical undo will *compensate* for its effects, but leave its structural changes intact. Because this recipe does not ensure transactional consistency and is largely orthogonal to the use of a lock manager, we call this class of concurrency control *latching* throughout this paper.

We have found the recipe to be easy to follow and very effective, and we use it everywhere our concurrent data structures may make structural changes, such as growing a hash table or array.

4.4 Adding Log Operations

Given this background, we now cover adding new operations. Lemon is designed to allow application developers to easily add new data representations and data structures by defining new operations. There are a number of invariants that these operations must obey:

1. Pages should only be updated inside of a REDO or UNDO function.

2. An update to a page atomically updates the LSN by pinning the page.

3. 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.

4. REDO operations use page numbers and possibly record numbers while UNDO operations use these or logical names/keys.

5. Use nested top actions (and logical UNDO) or "big locks" (which reduce concurrency) for multi-page updates.

An Example: Increment/Decrement

A common optimization for TPC benchmarks is to provide hand-built operations that support adding/subtracting from an account. Such operations improve concurrency since they can be reordered and can be easily made into nested top actions (since the logical UNDO is trivial). Here we show how increment/decrement map onto Lemon operations.

First, we define the operation-specific part of the log record:

```
  typedef struct { int amount } inc_dec_t;
```

Here is the increment operation; decrement is analogous:

```
// p is the bufferPool's current copy of the page.
int operateIncrement(int xid, Page* p, lsn_t lsn,
                     recordid rid, const void *d) {
  inc_dec_t * arg = (inc_dec_t)d;
  int i;

  latchRecord(p, rid);
  readRecord(xid, p, rid, &i);   // read current value
  i += arg->amount;

  // write new value and update the LSN
  writeRecord(xid, p, lsn, rid, &i);
  unlatchRecord(p, rid);
  return 0;                      // no error
}
```

Next, we register the operation:

```
// first set up the normal case
ops[OP_INCREMENT].implementation= &operateIncrement;
ops[OP_INCREMENT].argumentSize  = sizeof(inc_dec_t);
```

```
// set the REDO to be the same as normal operation
//   Sometimes useful to have them differ
ops[OP_INCREMENT].redoOperation = OP_INCREMENT;

// set UNDO to be the inverse
ops[OP_INCREMENT].undoOperation = OP_DECREMENT;
```

Finally, here is the wrapper that uses the operation, which is identified via `OP_INCREMENT`; applications use the wrapper rather than the operation, as it tends to be cleaner.

```
int Tincrement(int xid, recordid rid, int amount) {
   // rec will be serialized to the log.
   inc_dec_t rec;
   rec.amount = amount;

   // write a log entry, then execute it
   Tupdate(xid, rid, &rec, OP_INCREMENT);

   // return the incremented value
   int new_value;
   // wrappers can call other wrappers
   Tread(xid, rid, &new_value);
   return new_value;
}
```

With some examination it is possible to show that this example meets the invariants. In addition, because the REDO code is used for normal operation, most bugs are easy to find with conventional testing strategies. However, as we will see in Section 7, even these invariants can be stretched by sophisticated developers.

4.5 Summary

In this section we walked through some of the more important parts of the Lemon API, including the lock manager, nested top actions and log operations. The majority of the recovery algorithm's complexity is hidden from developers. We argue that Lemon's novel approach toward the encapsulation of transactional primitives makes it easy for developers to use these mechanisms to enhance application performance and simplify software design.

5 Experimental setup

The following sections describe the design and implementation of non-trivial functionality using Lemon, and use Berkeley DB for comparison. We chose Berkeley DB because, among commonly used systems, it provides transactional storage that is most similar to Lemon, and it was designed for high performance and high concurrency. For all tests, the two libraries provide the same transactional semantics.

All benchmarks were run on an Intel Xeon 2.8 GHz with 1GB of RAM and a 10K RPM SCSI drive, formatted with reiserfs.[5] All results correspond to the mean of multiple runs with a 95% confidence interval with a half-width of 5%.

We used Berkeley DB 4.2.52 as it existed in Debian Linux's testing branch during March of 2005, with the flags DB_TXN_SYNC, and DB_THREAD enabled. These flags were chosen to match Berkeley DB's configuration to Lemon's as closely as possible. In cases where Berkeley DB implements a feature that is not provided by Lemon, we enable the feature if it improves Berkeley DB's performance.

Optimizations to Berkeley DB that we performed included disabling the lock manager, though we still use "Free Threaded" handles for all tests. This yielded a significant increase in performance because it removed the possibility of transaction deadlock, abort, and repetition. However, after introducing this optimization, highly concurrent Berkeley DB benchmarks became unstable, suggesting either a bug or misuse of the feature. We believe that this problem would only improve Berkeley DB's performance in our benchmarks, so we disabled the lock manager for all tests. Without this optimization, Berkeley DB's performance for Figure 4 strictly decreases with increased concurrency due to contention and deadlock recovery. We increased Berkeley DB's buffer cache and log buffer sizes to match Lemon's default sizes.

Finally, we would like to point out that we expended a considerable effort tuning Berkeley DB, and that our efforts significantly improved Berkeley DB's performance on these tests. Although further tuning by Berkeley DB experts might improve Berkeley DB's numbers, we think that we have produced a reasonably fair comparison, and have reproduced the overall results on multiple machines and file systems.

6 Linear Hash Table

Lemon provides a clean abstraction of transactional pages, allowing for many different types of customization. In general, when a monolithic system is replaced with a layered approach there is always some concern that levels of indirection and abstraction will degrade performance. So, before moving on to describe some optimizations that Lemon allows, we evaluate the performance of a simple linear hash table that has been implemented as an extension to Lemon. We also take the opportunity to describe an optimized variant of the hash table and describe how Lemon's flexible page and log formats enable interesting optimizations. We also argue that Lemon makes it easy to produce concurrent data structure implementations.

We decided to implement a *linear* hash table [12]. Linear hash tables are able to increase the number of buckets incrementally at runtime. Imagine that we want to double the size of a hash table of size 2^n and that we use some hash function

[5]We found that the relative performance of Berkeley DB and Lemon is highly sensitive to filesystem choice, and we plan to investigate the reasons why the performance of Lemon under ext3 is degraded. However, the results relating to the Lemon optimizations are consistent across filesystem types.

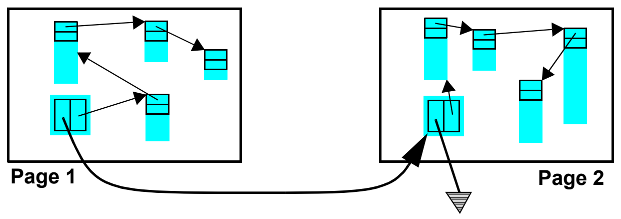


Figure 2: Structure of locality preserving (*page-oriented*) linked lists. By keeping sub-lists within one page, Lemon improves locality and simplifies most list operations to a single log entry.

$h_n(x) = h(x) \, mod \, 2^n$. Choose $h_{n+1}(x) = h(x) \, mod \, 2^{n+1}$ 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, we know that the contents of each bucket, m, will be split between bucket m and bucket $m + 2^n$. Therefore, if we keep track of the last bucket that was split then we can split a few buckets at a time, resizing the hash table without introducing long pauses [12].

In order to implement this scheme we need two building blocks. We need a map from bucket number to bucket contents (lists), and we need to handle bucket overflow.

6.1 The Bucket Map

The simplest bucket map would simply use a fixed-length transactional array. However, since we want the size of the table to grow, we should not assume that it fits in a contiguous range of pages. Instead, we build on top of Lemon's transactional ArrayList data structure (inspired by the Java class).

The ArrayList provides the appearance of large growable array by breaking the array into a tuple of contiguous page intervals that partition the array. Since we expect relatively few partitions (one per enlargement typically), this leads to an efficient map. We use a single "header" page to store the list of intervals and their sizes.

For space efficiency, the array elements themselves are stored using the fixed-length record page layout. Thus, we use the header page to find the right interval, and then index into it to get the $(page, slot)$ address. Once we have this address, the REDO/UNDO entries are trivial: they simply log the before or after image of that record.

6.2 Bucket List

Given the map, which locates the bucket, we need a transactional linked list for the contents of the bucket. The trivial implementation would just link variable-size records together, where each record contains a $(key, value)$ pair and the *next* pointer, which is just a $(page, slot)$ address.

However, in order to achieve good locality, we instead implement a *page-oriented* transactional linked list, shown in Figure 2. The basic idea is to place adjacent elements of the list on the same page: thus we use a list of lists. The main list links pages together, while the smaller lists reside within one page. Lemon's slotted pages allow the smaller lists to support variable-size values, and allow list reordering and value resizing with a single log entry (since everything is on one page).

In addition, all of the entries within a page may be traversed without unpinning and repinning the page in memory, providing very fast traversal over lists that have good locality. This optimization would not be possible if it were not for the low-level interfaces provided by the buffer manager. In particular, we need to control space allocation, and be able to read and write multiple records with a single call to pin and unpin. Due to this data structure's nice locality properties and good performance for short lists, it can also be used on its own.

6.3 Concurrency

Given the structures described above, the implementation of a linear hash table is straightforward. A linear hash function is used to map keys to buckets, insertions and deletions are handled by the ArrayList implementation, and the table can be extended lazily by transactionally removing items from one bucket and adding them to another.

The underlying transactional data structures and a single lock around the hashtable are all that are needed to complete the linear hash table implementation. Unfortunately, as we mentioned in Section 4.3, things become a bit more complex if we allow interleaved transactions. The solution for the default hashtable is simply to follow the recipe for Nested Top Actions, and latch the entire table for each operation. We also explore a version with finer-grain latching below.

This completes our description of Lemon's default hashtable implementation. Implementing transactional support and concurrency for this data structure is straightforward; the only complications are a) defining a logical UNDO, and b) using the bucket list to handle variable-length records. Lemon hides the hard parts of transactions.

6.4 The Optimized Hashtable

Our optimized hashtable implementation is optimized for log bandwidth, only stores fixed-length entries, and exploits a more aggressive version of nested top actions.

Instead of using nested top actions, the optimized implementation applies updates in a carefully chosen order that minimizes the extent to which the on disk representation of the hash table can be corrupted. This is essentially "soft updates" applied to a multi-page update [6]. Before beginning the update, it writes an UNDO entry that will first check and restore the consistency of the hashtable during recovery, and then invoke the inverse of the operation that needs to be undone. This

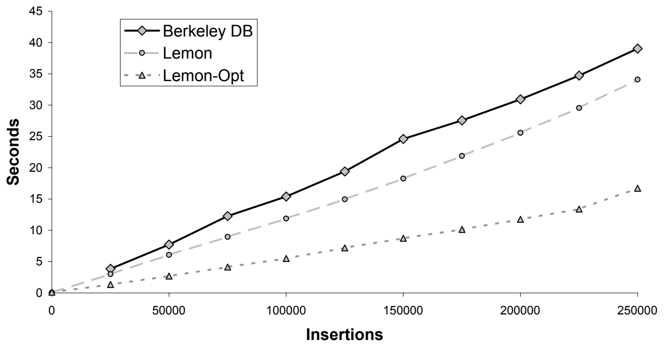


Figure 3: This test measures the raw performance of the data structures provided by Lemon and Berkeley DB. Since the test is run as a single transaction, overheads due to synchronous I/O and logging are minimized.

recovery scheme does not require record-level UNDO information, and thus avoids before-image log entries, which saves log bandwidth and improves performance.

Also, since this implementation does not need to support variable-size entries, it stores the first entry of each bucket in the ArrayList that represents the bucket list, reducing the number of buffer manager calls that must be made. Finally, this implementation caches the header information in memory, rather than getting it from the buffer manager on each request.

The most important component of Lemon for this optimization is Lemon's flexible recovery and logging scheme. For brevity we only mention that this hashtable implementation uses bucket-granularity latching; fine-grain latching is relatively easy in this case since all operations only affect a few buckets, and buckets have a natural ordering.

6.5 Performance

We ran a number of benchmarks on the two hashtable implementations mentioned above, and used Berkeley DB for comparison. The first test (Figure 3) measures the throughput of a single long-running transaction that loads a synthetic data set into the library.

Both of Lemon's hashtable implementations perform well, but the optimized implementation is clearly faster. This is not surprising as it issues fewer buffer manager requests and writes fewer log entries than the straightforward implementation.

The second test (Figure 4) measures the two libraries' ability to exploit concurrent transactions to reduce logging overhead. Both systems can service concurrent calls to commit with a single synchronous I/O.[6] Even when using the unoptimized hash table implementation, Lemon scales very well with higher concurrency, delivering over 6000 transactions per sec-

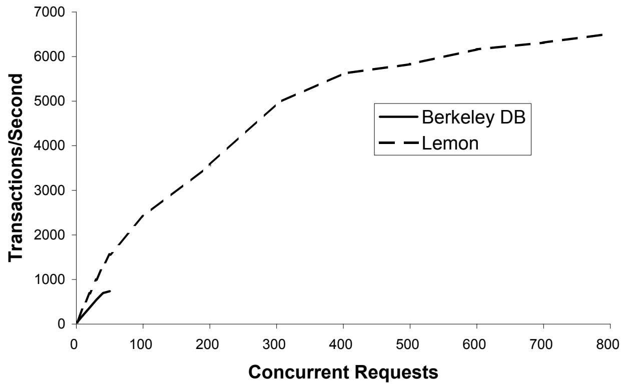


Figure 4: The logging mechanisms of Lemon and Berkeley DB are able to combine multiple calls to commit() into a single disk force, increasing throughput as the number of concurrent transactions grows. We were unable to get Berkeley DB to work correctly with more than 50 threads (see text).

ond.[7] Lemon had about double the throughput of Berkeley DB (up to 50 threads).

Finally, we developed a simple load generator which spawns a pool of threads that generate a fixed number of requests per second. We then measured response latency, and found that Berkeley DB and Lemon behave similarly.

In summary, there are a number of primitives that are necessary to implement custom, high-concurrency transactional data structures. In order to implement and optimize the hashtable we used a number of low-level APIs that are not supported by other systems. We needed to customize page layouts to implement ArrayList. The page-oriented list addresses and allocates data with respect to pages in order to preserve locality. The hashtable implementation is built upon these two data structures, and needs to generate custom log entries, define custom latching/locking semantics, and make use of, or even customize, nested top actions.

The fact that our straightforward hashtable is competitive with Berkeley DB shows that simple Lemon implementations of transactional data structures can compete with comparable, highly tuned, general-purpose implementations. Similarly, this example shows that Lemon's flexibility enables optimizations that can significantly outperform existing solutions.

This finding suggests that it is appropriate for application developers to build custom transactional storage mechanisms when application performance is important. Because we are advocating the use of application-provided transactional storage primitives, we only use the straightforward hashtable implementation during our other benchmarks.

We have shown that Lemon's implementation provides primitives that perform well enough to allow application-specific extensions to compete with highly tuned general purpose systems. The next two sections validate the practicality

[6]The multi-threading benchmarks presented here were performed using an ext3 file system, as high thread concurrency caused Berkeley DB and Lemon to behave unpredictably when reiserfs was used. However, Lemon's multi-threaded throughput was significantly better than Berkeley DB's with both filesystems.

[7]This test was run without lock managers, so the transactions obeyed the A,C, and D ACID properties. Since each transaction performed exactly one hashtable write they obeyed I (isolation) in a trivial sense.

of such mechanisms by applying them to applications that suffer from long-standing performance problems with traditional databases and transactional libraries.

7 Object Serialization

Object serialization performance is extremely important in modern web application systems such as Enterprise Java Beans. Object serialization is also a convenient way of adding persistent storage to an existing application without managing an explicit file format or low-level I/O interfaces.

A simple object serialization scheme would bulk-write and bulk-read sets of application objects to an OS file. These simple schemes suffer from high read and write latency, and do not handle small updates well. More sophisticated schemes store each object in a separate, randomly accessible record, such as a database or Berkeley DB record. These schemes allow for fast single-object reads and writes, and are typically the solutions used by application servers.

However, one drawback of many such schemes is that any update requires a full serialization of the entire object. In some application scenarios this can be extremely inefficient as it may be the case that only a single field from a large complex object has been modified.

Furthermore, most of these schemes "double cache" object data. Typically, the application maintains a set of in-memory objects in their unserialized form, so they can be accessed with low latency. The backing store also maintains a separate in-memory buffer pool with the serialized versions of some objects, as a cache of the on-disk data representation. Accesses to objects that are only present in the serialized buffer pool incur significant latency, as they must be unmarshalled (deserialized) before the application may access them. There may even be a third copy of this data resident in the filesystem buffer cache, accesses to which incur latency of both system call overhead and the unmarshalling cost.

To maximize performance, we want to maximize the size of the in-memory object cache. However, naively constraining the size of the data store's buffer pool causes performance degradation. Most transactional layers (including ARIES) must read a page into memory to service a write request to the page; if the buffer pool is too small, these operations trigger potentially random disk I/O. This removes the primary advantage of write-ahead logging, which is to ensure application data durability with mostly sequential disk I/O.

In summary, this system architecture (though commonly deployed [10, 22]) is fundamentally flawed. In order to access objects quickly, the application must keep its working set in cache. Yet in order to efficiently service write requests, the transactional layer must store a copy of serialized objects in memory or resort to random I/O. Thus, any given working set size requires roughly double the system memory to achieve good performance.

7.1 Lemon Optimizations

Lemon's architecture allows us to apply two interesting optimizations to object serialization. First, since Lemon supports custom log entries, it is trivial to have it store deltas to the log instead of writing the entire object during an update.

The second optimization is a bit more sophisticated, but still easy to implement in Lemon. This optimization allows us to drastically limit the size of the Lemon buffer cache, and still achieve good performance. We do not believe that existing relational database systems or Berkeley DB could support this optimization.

The basic idea of this optimization is to postpone expensive operations that update the page file for frequently modified objects, relying on some support from the application's object cache to maintain transactional semantics.

To implement this, we added two custom Lemon operations. The "`update()`" operation is called when an object is modified and still exists in the object cache. This causes a log entry to be written, but does not update the page file. The fact that the modified object still resides in the object cache guarantees that the (now stale) records will not be read from the page file. The "`flush()`" operation is called whenever a modified object is evicted from the cache. This operation updates the object in the buffer pool (and therefore the page file), likely incurring the cost of both a disk *read* to pull in the page, and a *write* to evict another page from the relatively small buffer pool. However, since popular objects tend to remain in the object cache, multiple update modifications will incur relatively inexpensive log additions, and are only coalesced into a single modification to the page file when the object is flushed.

Lemon provides several options to handle UNDO records in the context of object serialization. The first is to use a single transaction for each object modification, avoiding the cost of generating or logging any UNDO records. The second option is to assume that the application will provide a custom UNDO for the delta, which increases the size of the log entry generated by each update, but still avoids the need to read or update the page file.

The third option is to relax the atomicity requirements for a set of object updates and again avoid generating any UNDO records. This assumes that the application cannot abort individual updates, and is willing to accept that some prefix of logged but uncommitted updates may be applied to the page file after recovery. These "transactions" would still be durable after commit(), as it would force the log to disk. For the benchmarks below, we use this approach, as it is the most aggressive and is not supported by any other general-purpose transactional storage system (that we know of).

7.2 Recovery and Log Truncation

An observant reader may have noticed a subtle problem with this scheme. More than one object may reside on a page, and we do not constrain the order in which the cache calls flush() to evict objects. Recall that the version of the LSN on the page implies that all updates *up to* and including the page LSN have been applied. Nothing stops our current scheme from breaking this invariant.

This is where we use the versioned-record page layout. This layout adds a "record sequence number" (RSN) for each record, which subsumes the page LSN. Instead of the invariant that the page LSN implies that all earlier *page* updates have been applied, we enforce that all previous *record* updates have been applied. One way to think about this optimization is that it removes the head-of-line blocking implied by the page LSN so that unrelated updates remain independent.

Recovery works essentially the same as before, except that we need to use RSNs to calculate the earliest allowed point for log truncation (so as to not lose an older record update). In practice, we also periodically flush the object cache to move the truncation point forward, but this is not required.

7.3 Evaluation

We implemented a Lemon plugin for Juicer, a C++ object serialization library that can use various object serialization backends. We set up an experiment in which objects are randomly retrieved from the cache according to a hot-set distribution[8] and then have certain fields modified and updated into the data store. For all experiments, the number of objects is fixed at 5,000, the hot set is set to 10% of the objects, the object cache is set to double the size of the hot set, we update 100 objects per transaction, and all experiments were run with identical random seeds for all configurations.

The first graph in Figure 5 shows the update rate as we vary the fraction of the object that is modified by each update for Berkeley DB, unmodified Lemon, Lemon with the update/flush optimization, and Lemon with both the update/flush optimization and delta- based log records. The graph confirms that the savings in log bandwidth and buffer pool overhead by both Lemon optimizations outweighs the overhead of the operations, especially when only a small fraction of the object is modified. In the most extreme case, when only one integer field from a 1KB object is modified, the fully optimized Lemon corresponds to a 2x speedup over the simple version.

In all cases, the update rate for MySQL[9] is slower than Berkeley DB, which is slower than any of the Lemon variants. This performance difference is in line with those observed in Section 6. We also see the increased overhead due to the SQL processing for the MySQL implementation, although we note that a SQL variant of the delta-based optimization also provides performance benefits.

In the second graph, we constrained the Lemon buffer pool size to be a small fraction of the size of the object cache, and bypass the filesystem buffer cache via the O_DIRECT option. The goal of this experiment is to focus on the benefits of the update/flush optimization in a simulated scenario of memory pressure. From this graph, we see that as the percentage of requests that are serviced by the cache increases, the performance of the optimized Lemon dramatically increases. This result supports the hypothesis of the optimization, and shows that by leveraging the object cache, we can reduce the load on the page file and therefore the size of the buffer pool.

The operations required for these two optimizations required a mere 150 lines of C code, including whitespace, comments and boilerplate function registrations. Although the reasoning required to ensure the correctness of this code is complex, the simplicity of the implementation is encouraging.

In addition to the hashtable, which is required by Juicer's API, this section made use of custom log formats and semantics to reduce log bandwidth and page file usage. Berkeley DB supports a similar partial update mechanism, but it only supports range updates and does not map naturally to Juicer's data model. In contrast, our Lemon extension simply makes upcalls into the object serialization layer during recovery to ensure that the compact, object-specific deltas that Juicer produces are correctly applied. The custom log format, when combined with direct access to the page file and buffer pool, drastically reduces disk and memory usage for write intensive loads. Versioned records provide more control over durability for records on a page, which allows Lemon to decouple object updates from page updates.

8 Graph Traversal

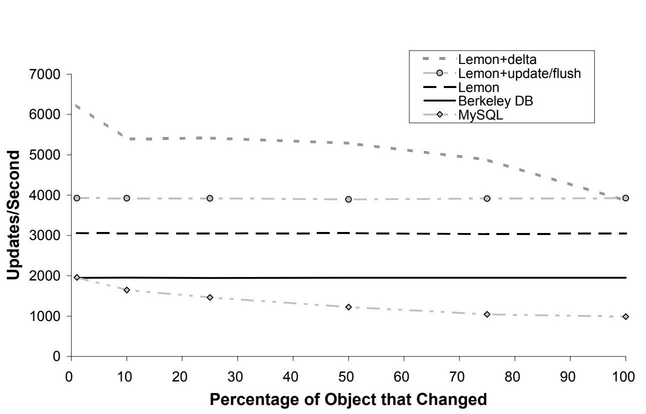
Database servers (and most transactional storage systems) are not designed to handle large graph structures well. Typically, each edge traversal will involve an index lookup, and worse, since most systems do not provide information about the physical layout of the data that they store, it is not straightforward to implement graph algorithms in a way that exploits on disk locality. In this section, we describe an efficient representation of graph data using Lemon's primitives, and present an optimization that introduces locality into random disk requests by reordering invocations of wrapper functions.

8.1 Data Representation

For simplicity, we represent graph nodes as fixed-length records. The ArrayList from our linear hash table implementation (Section 6) provides access to an array of such records with performance that is competitive with native recordid accesses, so we use an ArrayList to store the records. We could

[8] In an example hot-set distribution, 10% of the objects (the hot set size) are selected 90% of the time (the hot set probability).

[9] We ran MySQL using InnoDB for the table engine, as it is the fastest engine that provides similar durability to Lemon. For this test, we also linked directly with the libmysqld daemon library, bypassing the RPC layer. In experiments that used the RPC layer, test completion times were orders of magnitude slower.



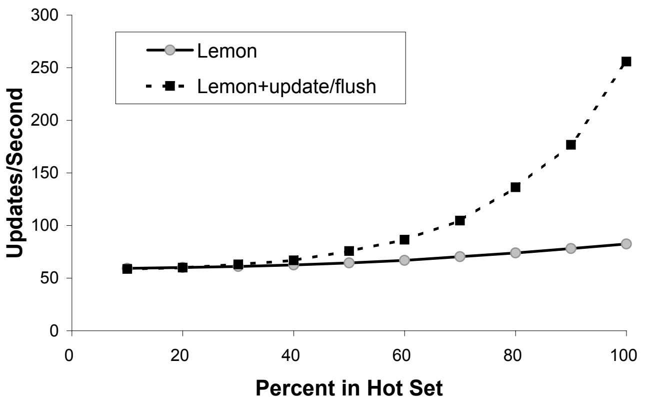


Figure 5: Lemon optimizations for object serialization. The first graph shows the effect of the two Lemon optimizations as a function of the portion of the object that is being modified. The second graph focuses on the benefits of the update/flush optimization in cases of system memory pressure.

have opted for a slightly more efficient representation by implementing a fixed-length array structure, but doing so seems to be overkill for our purposes. The nodes themselves are stored as an array of integers of length one greater than their out-degree. The extra `int` is used to hold information about the node; in our case, it is set to a constant during traversal.

We implement a "naive" graph traversal algorithm that uses depth-first search to find all nodes that are reachable from node zero. This algorithm (predictably) consumes a large amount of memory, as it places almost the entire graph on its stack.

For the purposes of this section, which focuses on page access locality, we ignore the amount of memory utilization used to store stacks and work lists, as they can vary greatly from application to application, but we note that the memory utilization of the simple depth-first search algorithm is certainly no better than the algorithm presented in the next section.

For simplicity, we do not apply any of the optimizations in Section 7. This allows our performance comparison to measure only the optimization presented here.

8.2 Request Reordering for Locality

General graph structures may have no intrinsic locality. If such a graph is too large to fit into memory, basic graph operations such as edge traversal become very expensive, which makes many algorithms over these structures intractable in practice. In this section, we describe how Lemon's primitives provide a natural way to introduce physical locality into a sequence of such requests. These primitives are general and support a wide class of optimizations.

Lemon's wrapper functions translate high-level (logical) application requests into lower level (physiological) log entries. These physiological log entries generally include a logical UNDO (Section 4.3) that invokes the logical inverse of the application request. Since the logical inverse of most application requests is another application request, we can *reuse* our operations and wrappers to implement a purely logical log.

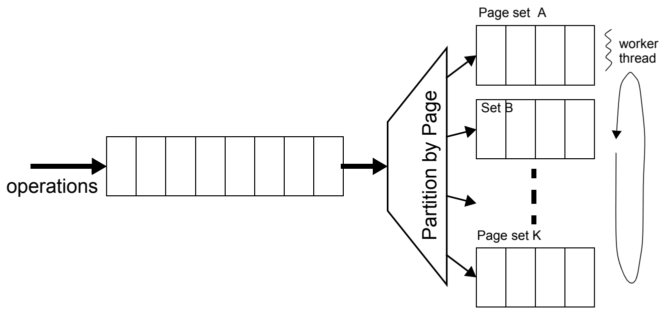


Figure 6: Because pages are independent, we can reorder requests among different pages. Using a log demultiplexer, we partition requests into independent queues, which can be handled in any order, improving locality and merging opportunities.

For our graph traversal algorithm we use a *log demultiplexer*, shown in Figure 6, to route entries from a single log into many sub-logs according to page number. This is easy to do with the ArrayList representation that we chose for our graph, since it provides a function that maps from array index to a $(page, slot, size)$ triple.

The logical log allows us to insert log entries that are independent of the physical location of their data. However, we are interested in exploiting the commutativity of the graph traversal operation, and saving the logical offset would not provide us with any obvious benefit. Therefore, we use page numbers for partitioning.

We considered a number of demultiplexing policies and present two particularly interesting ones here. The first divides the page file up into equally sized contiguous regions, which enables locality. The second takes the hash of the page's offset in the file, which enables load balancing.

Requests are continuously consumed by a process that empties each of the demultiplexer's output queues one at a time. Instead of following graph edges immediately, the targets of edges leaving each node are simply pushed into the demulti-

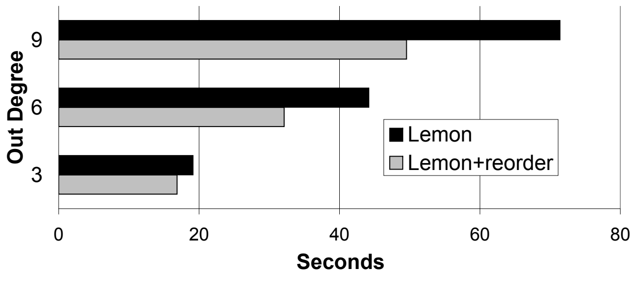


Figure 7: oo7 benchmark style graph traversal. The optimization performs well due to the presence of non-local nodes.

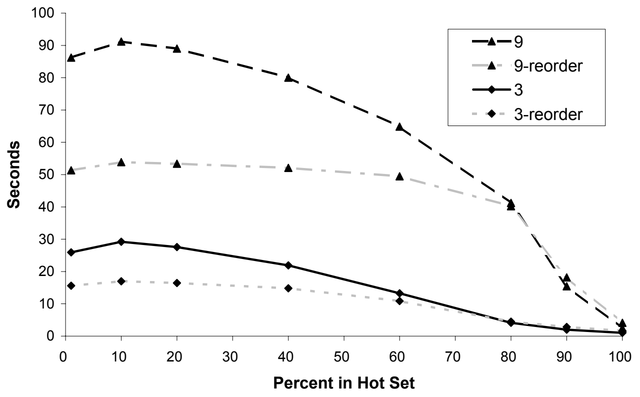


Figure 8: Hot set based graph traversal for random graphs with out-degrees of 3 and 9. Here we see that the multiplexer helps when the graph has poor locality. However, in the cases where depth first search performs well, the reordering is inexpensive.

plexer's input queue. The number of output queues is chosen so that each queue addresses a subset of the page file that can fit into cache, ensuring locality. When the demultiplexer's queues contain no more entries, the traversal is complete.

Although this algorithm may seem complex, it is essentially just a queue-based breadth-first search implementation, except that the queue reorders requests in a way that attempts to establish and maintain disk locality. This kind of log manipulation is very powerful, and could also be used for parallelism with load balancing (using a hash of the page number) and log-merging optimizations such as those in LRVM [20].

8.3 Performance Evaluation

We loosely base the graphs for this test on the graphs used by the oo7 benchmark [4]. For the test, we hard code the out-degree of graph nodes to 3, 6 and 9 and use a directed graph. The oo7 benchmark constructs graphs by first connecting nodes together into a ring. It then randomly adds edges between the nodes until the desired out-degree is obtained. This structure ensures graph connectivity. If the nodes are laid out in ring order on disk, it also ensures that one edge from each node has good locality while the others generally have poor locality. Figure 7 presents these results; we can see that the request reordering algorithm helps performance. We re-ran the test without the ring edges, and (in line with our next set of results) found that reordering helped there as well.

In order to get a better feel for the effect of graph locality on the two traversal algorithms we extend the idea of a hot set to graph generation. Each node has a distinct hot set which includes the 10% of the nodes that are closest to it in ring order. The remaining nodes are in the cold set. We use random edges instead of ring edges for this test. Figure 8 suggests that request reordering only helps when the graph has poor locality. This makes sense, as a depth-first search of a graph with good locality will also have good locality. Therefore, processing a request via the queue-based demultiplexer is more expensive then making a recursive function call.

We considered applying some of the optimizations discussed earlier in the paper to our graph traversal algorithm, but opted to dedicate this section to request reordering.

9 Future work

We have described a new approach toward developing applications using generic transactional storage primitives. This approach raises a number of important questions which fall outside the scope of its initial design and implementation.

We believe that development tools could be used to improve the quality and performance of our implementation and extensions written by other developers. Well-known static analysis techniques could be used to verify that operations hold locks (and initiate nested top actions) where appropriate, and to ensure compliance with Lemon's API. We also hope to re-use the infrastructure that implements such checks to detect opportunities for optimization. Our benchmarking section shows that our simple default hashtable implementation is 3 to 4 times slower than our optimized implementation. Using static checking and high-level automated code optimization techniques may allow us to narrow or close this gap, and enhance the performance and reliability of application-specific extensions.

We would like to extend our work into distributed system development and believe that Lemon's implementation anticipates many of the issues that we will face in distributed domains. By adding networking support to our logical log interface, we should be able to demultiplex and replicate log entries to sets of nodes easily. Single node optimizations such as the demand-based log reordering primitive should be directly applicable to multi-node systems.[10]. Also, we believe that logical, host independent logs may be a good fit for applications that make use of streaming data or that need to perform transformations on application requests before they are materialized in a transactional data store.

[10]For example, our (local, and non-redundant) log demultiplexer provides semantics similar to the Map-Reduce [5] distributed programming primitive, but exploits hard disk and buffer pool locality instead of the parallelism inherent in large networks of computer systems.

We also hope to provide a library of transactional data structures with functionality that is comparable to standard programming language libraries such as Java's Collection API or portions of C++'s STL. Our linked list implementations, ArrayList and hashtable represent an initial attempt to implement this functionality. We are unaware of any transactional system that provides such a broad range of data structure implementations. We may also be able to provide the same APIs for both in-memory and transactional data structures.

Finally, due to the large amount of prior work in this area, we have found that there are a large number of optimizations and features that could be applied to Lemon. It is our intention to produce a usable system from our research prototype. To this end, we have already released Lemon as an open-source library, and intend to produce a stable release once we are confident that the implementation is correct and reliable.

10 Conclusion

We believe that transactions have much to offer system developers, but that there is a need to enable transactions for wider range of systems than just databases. We built Lemon and showed how its framework simplifies the creation of transactional data structures that have excellent performance and flexibility, including arrays, hash tables, persistent objects, and graphs. Lemon provides a wide range of transactional semantics, all the way up to complete ACID transactions with high concurrency, archiving and media recovery. We also demonstrated that the low-level APIs enable many optimizations, including optimizations for deltas, locality, reordering, and durability. We have released Lemon as open source and believe it makes it easy to benefit from the power of transactions.

References

[1] Agrawal, et al. *Concurrency Control Performance Modeling: Alternatives and Implications*. TODS 12(4): (1987) 609-654

[2] Carey, Michael J., DeWitt, David J., Naughton, Jeffrey F. *The OO7 Benchmark.* SIGMOD (1993)

[3] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks.* CACM 13(6) p. 377-387 (1970)

[4] Jeffrey Dean and Sanjay Ghemawat. *Simplified Data Processing on Large Clusters.* OSDI (2004)

[5] Greg Ganger. *Soft Updates: A Solution to the Metadata Update Problem in File Systems* ACM Transactions (2000)

[6] David K. Gifford, P. Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. *Semantic file systems*. Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, (1991) p. 16-25.

[7] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann (1993) San Mateo, CA

[8] Jim Gray, Raymond A. Lorie, and Gianfranco R. Putzulo. *Granularity of locks and degrees of consistency in a shared database*. In 1st International Conference on VLDB, September 1975. Reprinted in Readings in Database Systems, 3rd ed.

[9] Gribble, Steven D., Brewer, Eric A., Hellerstein, Joseph M., Culler, David. *Scalable, Distributed Data Structures for Internet Service Construction.* OSDI (2000)

[9] Haerder & Reuter *"Principles of Transaction-Oriented Database Recovery."* Computing Surveys 15(4) (1983)

[10] Hibernate, `http://www.hibernate.org/`

[11] Lamb, et al., *The ObjectStore System.* CACM 34(10) (1991)

[12] Litwin, W., *Linear Hashing: A New Tool for File and Table Addressing*. Proc. 6th VLDB, Montreal, Canada, (Oct. 1980)

[13] Mohan, et al., *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.* TODS 17(1) (1992) p. 94-162

[14] Mohan, Lindsay & Obermarck, *Transaction Management in the R* Distributed Database Management System* TODS 11(4) (1986) p. 378-396

[15] Mohan, Levine. *ARIES/IM: an efficient and high concurrency index management method using write-ahead logging* International Converence on Management of Data, SIGMOD (1992) p. 371-380

[16] *MySQL*, `http://www.mysql.com/`

[17] Reiser, Hans T. *ReiserFS 4* `http://www.namesys.com/`

[18] M. Seltzer, M. Olsen. *LIBTP: Portable, Modular Transactions for UNIX*. Proceedings of the 1992 Winter Usenix (1992)

[19] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., AND Kistler, J. J. *Lightweight Recoverable Virtual Memory*. ACM Transactions on Computer Systems 12, 1 (Februrary 1994) p. 33-57. Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.

[20] Stonebraker. *Inclusion of New Types in Relational Data Base.* ICDE (1986)

[21] Stonebraker and Kemnitz. *The POSTGRES Next-Generation Database Management System.* CACM (1991)