A Simple and Efficient Implementation for Small Databases

Andrew D. Birrell
Michael B. Jones
Edward P. Wobber

11 August 1987

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

A Simple and Efficient Implementation for Small Databases

This paper describes a technique for implementing the sort of small databases that frequently occur in the design of operating systems and distributed systems. We take advantage of the existence of very large virtual memories, and quite large real memories, to make the technique feasible. We maintain the database as a strongly typed data structure in virtual memory, record updates incrementally on disk in a log and occasionally make a checkpoint of the entire database. We recover from crashes by restoring the database from an old checkpoint then replaying the log. We use existing packages to convert between strongly typed data objects and their disk representations, and to communicate strongly typed data across the network (using remote procedure calls). Our memory is managed entirely by a general purpose allocator and garbage collector.This scheme has been used to implement a name server for a distributed system. The resulting implementation has the desirable property of being simultaneously simple, efficient and reliable.

1. THE PROBLEM

There are many situations in the design of an operating system or of a distributed system where we desire to manage data that is structured and that must persist across system restarts. Although we often call these "databases", they are quite different from commercial databases such as those used by a bank or an insurance company. The databases we are considering have the following distinguishing characteristics. They are relatively small — up to about 10 megabytes. They have a moderate rate of updates — a burst rate of up to 10 transactions per second, and a long term rate of up to 10000 transactions per day. The update operations provided by these databases are single-shot transactions. In other words, there are no update transactions composed of multiple client actions, and the database implementation does not make any intermediate state visible to its clients. Examples of these operating system databases include records of user accounts, network name servers, network configuration information and file directories.

The purpose of this paper is to offer an implementation technique suitable for this class of databases. Our design is based on technology well known to the database community — a main memory database with checkpoints and a re-do log [10,11]. To this we add technology well known to the operating systems community — high level language data structures, remote procedure calls and strongly typed access to backing store. The outcome appears to be novel — an implementation of operating system databases that is simultaneously very simple, very efficient and very reliable. The other implementations we have seen for such databases suffer by lacking one (or more) of these properties.

Throughout the paper we illustrate the design with the example of a name server that we have built for our distributed computing environment. That name server design has other interesting aspects [6], but the present paper concentrates only on the data storage aspects of the name server.

Typically, we implement such databases by compiling into the implementation an understanding of the particular data types and record organizations required by the database. This contrasts with the flexibility of the data models offered by general purpose database systems; we will not consider this particular aspect any further in the present paper.

2. OTHER TECHNIQUES

Traditionally, one of a small number of implementation techniques has been used for these databases. The choice has seemed to depend partly on the size and characteristics of the data, partly on the access patterns, but largely on the predilections of the system builders. The techniques have characteristic differences in their performance and in the reliability of update operations, and in the complexity of their implementations.

In relatively simple operating systems such as Unix™, almost all databases are stored as ordinary text files (for example, */etc/passwd* for user accounts, */etc/hosts* for host names and addresses). Whenever a program (or the operating system) wishes to access the data it does so by reading and parsing the file. Changes to the data are made by invoking the normal text processing programs to modify the file. An update involves rewriting the entire file. All such updates must be serialized — an administrator obtains some form of exclusive lock prior to editing the file. The reliability of updates in the face of transient errors (halting the machine during an update operation) can be made quite good, by using an atomic file rename operation to install a new version of the file. Reliability in the face of hard errors (part of the file becoming unreadable) depends entirely on keeping backup copies of the complete database. The attraction of this technique lies entirely in its simplicity. The implementation consists of subroutines for parsing the file. Updates and browsing are performed with existing editors, text manipulation programs and user interfaces. Any computer science student could build such a database system and it would still be small and simple. It is generally not practicable to produce good performance with this technique; its applicability is restricted to databases with a fairly low rate of enquiries and a very low rate of updates.

The corresponding databases in larger scale operating systems are often implemented by ad hoc schemes, involving a custom designed data representation in a disk file, and specialized code for accessing and modifying the data. Typical read accesses involve perusing a small number of directly accessed pages from the disk (the number depending on the complexity of the data structure). Typical updates involve the same actions as reading, plus extra disk writes for recording the update. In such ad hoc schemes, updates are typically performed by overwriting existing data in place. This leaves the database quite vulnerable to transient errors, requiring restoration of the database from a backup copy. This is particularly true if the update modifies multiple pages, so such updates are best avoided if you use this technique. Again, recovery from hard errors depends on restoration of a backup copy. The performance of these databases is generally quite good for updates, requiring typically one disk write per update.

More sophisticated database systems (including most systems that actually dare to use the word "database") overcome these problems by implementing update transactions with an atomic commit mechanism [3,7,8]. There is a substantial literature on how to implement this, and it is not at all difficult to achieve a high degree of reliability; certainly all such systems will recover from any transient error by undoing or replaying an entire update transaction. At extra cost, they can recover from hard errors. Some of these general purpose systems provide options that correspond to the database techniques used in our design — main memory data storage with snapshots and logging [5]. The major skill in engineering these systems appears to be how to achieve acceptable performance and availability. A dominant factor in their performance is the number of disk writes for a typical update transaction. A naive implementation of atomic commit will require two disk writes: one for the commit record (and log entry) and one for updating the actual data. This is somewhat more complicated than a system without atomic commit, has much better reliability, and performs about a factor of two worse for updates. Sophisticated large-scale database systems can do much better than this [1,5]. Using a redo log they can delay updating the actual data, in the hope (and expectation) that the data update can be merged with that for a later transaction. Sometimes multiple transactions can be recorded in a single log entry. These top-end database systems can provide optimal combinations of performance and reliability, but at the expense of great complexity [1,4]. Engineering such a database system is a major task, requiring a large investment and producing a large system. Their complexity is completely out of line with the sort of databases we are considering.

Note, however, that general purpose databases often offer a feature that neither the simpler techniques, nor our design, include. They allow a client to perform multiple separate update operations in a single atomic transaction. This involves quite a lot of complexity, since they must hide the internal intermediate states of such transactions from other clients, must arrange to serialize them, and must be able to abort them if the client fails to complete a transaction. This complexity often extends to the locking schemes, since a client in the midst of a multi-step transaction generally wants to have a read lock on the related data. Other techniques, including ours, do not offer this. We believe that the databases we are considering can be manipulated satisfactorily without this feature.

With each of these techniques, the performance of read accesses can be made arbitrarily good by caching. In fact, in many general purpose database systems such a cache is often readily available as a side-effect of the buffer management schemes already required for their implementation. Unfortunately, the better the performance achieved by this caching, the more complex the implementation becomes. The best caching schemes are not much less complex than simple virtual memory systems. It is certainly possible to achieve better performance with a database-oriented caching scheme than would be achieved by just using a traditional virtual memory (by taking advantage of knowledge of the data structures and access patterns — particularly enumerations). But the virtual memory has already been implemented and paid for; with the databases we are considering the incremental performance gain from a custom designed cache would not be worth the costs.

So none of these techniques is entirely satisfactory. The designer must choose to sacrifice reliability, simplicity or efficiency. The purpose of this paper is to describe a design, which we have implemented and are using in our daily work, that we believe avoids this sacrifice.

3. THE DESIGN

The design is quite straightforward. At all times the database is represented as an ordinary data structure in virtual memory. Its counterpart on disk has two components: a checkpoint of some previous (consistent) state of the entire database, and a log recording each subsequent update to the database.

The virtual memory data structure is organized in whatever manner is convenient for your particular database. You can use all the data organizations provided by your favorite high level language.

A read access to the database consists purely of a lookup in the virtual memory data structure. The disk structures are not involved. There must be, of course, an appropriate locking strategy to mediate between read and write operations on the virtual memory structure.

An update to the database is made in three steps. First, the virtual memory data is read to verify that it is consistent with any preconditions of the update (for example, consistency invariants or access controls). Second, all the parameters of the update are gathered together and recorded as an entry in the disk log. Third, the update is applied to the virtual memory data. These three steps consist of two virtual memory operations and one disk write. The commit point is the disk write: if we crash before the write occurs on the disk, the update is not visible after a restart; if we crash after the write completes, the entire update will be completed after a restart. Again, the implementation must use an appropriate locking strategy to serialize updates and to prevent interference between updates and concurrent enquiries.

From time to time, the implementation records on disk a checkpoint containing the entire contents of the virtual memory data structure, then resets the log to be empty and removes any previous checkpoint from the disk. The implementation uses the primitives of its host file system to ensure that these actions are performed atomically. The implementation must use a locking strategy to ensure that the checkpoint is a consistent version of the database, not one with a partially performed update (and to serialize checkpoints).

Restarting the system from its disk files consists of three steps: determine which is the current checkpoint (and discard any partially written ones, old ones or old logs); read the current checkpoint to obtain an old version of the virtual memory data structure; replay the updates from the log and apply them to the virtual memory structure to obtain the most recent state of the database.

One major decision required is what strategy to use to decide when to make a new checkpoint. Making a new checkpoint is quite time consuming, but the alternative is to allow the log to grow longer. We discuss this question later in the section about "performance".

As an example, consider how this design was applied in building our simple name server. The name server offers its clients a general purpose name-to-value mapping, where the names are strings and the values are trees whose arcs are labelled by strings. It provides a variety of enquiry and browsing operations, and update operations for any set of sub-trees.

The virtual memory data structure for the name server's database consists primarily of a tree of hash tables. The tables are indexed by strings, and deliver values that are further hash tables. This data structure is implemented in a normal programming style: it is entirely strongly typed and it uses our general purpose string package, memory allocator and garbage collector.

Our name server uses three locking modes to mediate amongst concurrent access: *shared, update* and *exclusive*. The interactions amongst these locks are best described by the following matrix.

	shared	**update**	**exclusive**
shared	compatible	compatible	conflict
update	compatible	conflict	conflict
exclusive	conflict	conflict	conflict

An enquiry operation is performed with a *shared* lock. An update operation first acquires an *update* lock (thereby excluding other update operations but permitting enquiry operations). After the update operation has verified its pre-conditions it assembles its log record and commits it to disk. Finally the update operation converts its lock to an *exclusive* lock (thus excluding enquiry operations) and modifies the virtual memory structures. An *update* lock is held while writing a checkpoint. Note that these rules never exclude enquiry operations during disk transfers, only during virtual memory operations.

Our implementation uses the following sequence to write a new checkpoint (on top of a Unix-like file system). We make no claim that these particular steps are optimal, or even good, only that it is quite easy to use a simple file system to achieve the desired effect. We use a single directory for our disk structures. In the normal quiescent state the directory contains a version-numbered checkpoint, with a file title such as *checkpoint.35*, a matching log file named *logfile.35*, and a file named *version* containing the characters “35”. We switch to a new checkpoint by writing it to the file *checkpoint.36*, creating an empty file *logfile.36*, then writing the characters “36” to a new file called *newVersion*. This is the commit point (after an appropriate number of Unix “fsync” calls). Finally, we delete *checkpoint.35*, *logfile.35* and *version*, then rename *newVersion* to be *version*.

On a restart, we read the version number from *newVersion* if the file exists and has a valid version number in it, or from *version* otherwise, and delete any redundant files. We read the indicated checkpoint file to obtain an old virtual memory structure, then we read the succession of entries from the indicated log file. Each entry contains the parameters of an update, and we apply these in sequence to obtain the most recent virtual memory structure. Now the name server is available to its clients.

4. RELIABILITY

There are two aspects to the reliability of such a system (once the bugs have been removed!): recovery from transient failures, and recovery from hard failures. A “transient” failure occurs when the system just stops, possibly in the middle of an update or checkpoint operation, possibly during a disk write for one of these operations. A “hard” failure occurs when some data in the disk structures becomes unreadable. We do not attempt to handle the situation where some data becomes undetectably corrupted. In other words, we assume that our disks and virtual memory give either correct data or an error.

If a transient failure occurs during an update, recovery is easy. If the update’s log entry was completed, then the update will be completed during the normal restart sequence, when the log is replayed. If there is no log entry whatever for the update, then the behavior is as if the update had not occurred. The implementation can detect a partially written log entry for the update, even if the log entry would span multiple disk pages; such a partial log entry is discarded. In our implementation, this detection comes from including the log entry’s length on the first page of the entry, combined with the known property of our disk hardware that a partially written page will report an error when it is read.

If a transient error occurs while writing a new checkpoint, the implementation restarts using the previous checkpoint and log, instead of the partially written new checkpoint.

For example (as described earlier), our implementation ensures that a new checkpoint and its empty log file exist on disk before writing a new version number file. On a restart we know that a complete checkpoint and log exist for whatever version number we find.

For the particular example of our name server, we chose not to build in local recovery from hard errors. This is because we already replicate the database on multiple name servers spread across the network. We respond to a hard error on a particular name server replica by restoring its data from another replica. This causes us to lose only those updates that had been applied to the damaged replica but not propagated to any other replica. In practice this is unlikely to amount to more than the most recent update, and such behavior seems acceptable for a name service. We have automatic mechanisms for ensuring the long-term consistency of the name server replicas.

Other applications might wish to build in similar or greater amounts of redundancy, depending on the perceived value of their data. This design gives some assistance here. For example, recovery from a hard error in the log could consist of ignoring just the damaged log entry (if the semantics of the application are such that updates are typically independent). Recovery from a hard error in the checkpoint could be achieved by keeping one previous checkpoint and log (preferably on a separate disk with a separate controller); the current state could be restored by reloading the previous checkpoint, replaying the previous log, then replaying the current log. Such redundancy measures cost disk space, but do not affect the performance for normal enquiries, updates, checkpoints or restarts.

There are other potential benefits from the existence of a complete update log, although we have not explored them. For example, the log files form a complete audit trail for the database, and could be retained if desired.

5. PERFORMANCE

It is clear that the performance of this design will be good. Enquiries take only the time necessary to access the virtual memory structure. Updates take the time for enquiries plus one disk write. A restart takes time proportional to the size of the checkpoint plus time proportional to the length of the log.

The actual cost of an enquiry will depend on the memory access patterns. If the working set can be retained in real memory, then the enquiry is extremely fast (this is the case with our name server and its 1 megabyte database). Otherwise the time will include the requisite number of page faults. The correct way to view this behavior is that the virtual memory system is being used to cache appropriate pages of the database in real memory. This should behave at least as well as any custom designed page-oriented caching scheme (actually better, since it has hardware support). Caching at the level of individual database records could produce better performance in a given amount of real memory, but at the expense of considerable added complexity.

An update behaves like an enquiry plus the cost of preparing the log entry and writing it to disk. The only schemes that will perform better than this involve arranging to record multiple commit records in a single log entry (in the presence of concurrent update requests). Such schemes are equally applicable to this design, with the same cost in added complexity.

The implementor (or the system manager) can trade-off between the time required for a restart and the availability for updates by deciding how often to make a checkpoint. With a simple locking scheme such as we used in our name server, updates are prevented while the checkpoint is being made, so frequent checkpoints are bad. Furthermore, making a checkpoint is quite an expensive operation. Finally, if checkpoints are too rare then the log file may consume excessive disk space. But if checkpoints are made too seldom then the restart time (which is mostly proportional to the log size) will be too long. However, with update rates of up to 10000 per day (our target long term rate) a simple scheme of making a checkpoint each night will suffice. For example, in our name server implementation, a log containing 10000 updates would cause the restart time to be about 5 minutes.

This design does require extra disk space. The total requirement consists of the virtual memory image for the virtual memory structure, two copies of the checkpoint and the log file. In addition, one extra checkpoint and log file can be retained for recovery from hard errors. This is more than would be required by the other techniques. However, for the moderate sized databases we are considering, the total amount of disk space involved is quite small.

The actual performance of our name server when running on a Microvax™ processor as follows. The working set of our 1 megabyte database remains constantly in real memory. A typical simple enquiry operation takes 5 msecs plus the network communication costs. This is entirely the computational cost of exploring the virtual memory structure. A typical update takes 54 msecs plus the network communication costs. This includes the costs of exploring (6 msecs) and modifying (6 msecs) the virtual memory structure, converting the parameters of the update from strongly typed values into bits suitable for preserving as a log entry (22 msecs), and using our file system for the disk write of the log entry (20 msecs). A checkpoint operation takes about one minute. This involves converting the entire virtual memory structure from a strongly typed value into bits suitable for preserving on disk (55 seconds), and the disk writes (5 seconds). Restart takes about 20 seconds to read the checkpoint, plus about 20 msecs per log entry. Notice that the times required for updates, checkpoints and restart are much greater than the raw disk times. This is partly because we use a general-purpose package (described later) for converting between strongly typed virtual memory data and raw bits suitable for long-term disk storage. Also, we have not yet tried to optimize our use of the file system when writing log entries. However, we believe these times are already very much better than other existing systems [2,8,9].

Our round-trip network communication costs are about 8 msecs for name server operations, so remote network clients can perform a name server enquiry in 13 msecs and an update in 62 msecs elapsed time. The name server can maintain a short term update rate of more than 15 transactions per second, unless it decides to make a new checkpoint.

6. SIMPLICITY

We do not understand how to provide a carefully reasoned analysis of the complexity of this design. What we have is a belief that it is indeed simple (to explain, build, modify and maintain), together with our experience in using this design to build a name server. In order to assist the reader in assessing the complexity of our name server, we present here some further details and measurements of it.

Our implementation makes use of a mechanism called "pickles", which will convert between any strongly typed data structure and a representation of that structure suitable for storing in permanent disk files. The operation *Pickle.Write* takes a pointer to a strongly typed data structure and delivers buffers of bits for writing to the disk. Conversely *Pickle.Read* reads buffers of bits from the disk and delivers a copy of the original data structure.* This conversion involves identifying the occurrences of addresses

* Pickling is quite similar to the concept of marshalling in remote procedure calls. But in fact our pickling implementation works only by interpreting at run-time the structure of dynamically typed values, while our RPC implementation works only by generating code for the marshalling of statically typed values. Each facility would benefit from adding the mechanisms of the other, but that has not yet been done.

in the structure, and arranging that when the structure is read back from disk the addresses are replaced with addresses valid in the current execution environment. The pickle mechanism is entirely automatic: it is driven by the run-time typing structures that are present for our garbage collection mechanism.

We use pickles for log entries and for checkpoints. To write the log entry for an update, we present the parameters of the update to *Pickle.Write*, then append the resulting bits (preceded by a count) to the log file. Note that this count allows us to identify the start and end of each entry in the log file. To replay a log entry we read the count from the log file, read the specified number of bits and present them to *Pickle.Read*. We then take the resulting value and perform an update operation with these parameters. To make a checkpoint we invoke *Pickle.Write* for the root of the entire virtual memory structure and stream the resulting bits to a file. To recover a checkpoint for restart we stream its bits into *Pickle.Read* to obtain the data structure. The performance of updates, making checkpoints and restarts is dominated by the performance of pickles. We are paying a little in our performance for using such a general package (about 40% of the cost of an update is in *Pickle.Write*), but we benefit greatly in the simplicity of our name server implementation.

Clients interact with our name server through a general purpose remote procedure call mechanism, well integrated into our programming language. In particular, our RPC mechanism automatically generates "marshalling" procedures to convert between strongly typed data structures and bit representations suitable for transport across the network.

The combined effect of these two facilities is that we can implement the name server entirely in a strongly typed language. It has no manually written code for casting values into low level disk or network bit patterns.

The implementation of the checkpoint and log facilities (excluding the pickle mechanism) occupies 638 source lines. The code to implement the name server's database semantics occupies 1404 source lines.

The package for checkpoints and logs (which is independent of the specialized semantics of the name server) was implemented by one programmer in about three weeks. The name server's database semantics took about three weeks of one programmer then two weeks of another. The RPC stub modules were generated automatically. Other parts of the name server occupied about two programmer-months; these included update propagation to other replicas, long-term replica consistency, management interfaces to the servers, user interfaces for browsing and modifying the database, and integrating the name server into our distributed computing environment.

The automatically generated RPC stub modules for client access to the name server occupy 663 source lines in the server and 622 source lines in the client. The (pre-existing) pickle package occupies 1648 source lines.

7. CONCLUSIONS

We have described our design for small to medium sized databases, such as the many organizational databases that occur in operating systems and distributed systems. We have shown how the design was used to implement a name server. The design has proved easy to implement. Its performance is at least as good as any other implementation we have seen, and better than most. It provides well understood reliability that is at least sufficient for our purposes and can be made as good as any other mechanism (including full-scale database systems).

The major limitations on the applicability of this design concern availability. There are two aspects that limit our availability: the time required for making a checkpoint (when updates are excluded), and the restart time (which involves replaying the entire log of updates made since the last checkpoint). As we have said, the system manager can make less frequent checkpoints, but at the expense of longer restarts. Ultimately, the limiting factor here is the update rate. At a high update rate, we would need either frequent checkpoints or very slow restarts. As we have described earlier, a single overnight checkpoint is probably sufficient for the databases we are considering.

A secondary limiting effect comes from the total size of the database. As it becomes larger, checkpoints take longer (thereby restricting the acceptable frequency of updates) and restarts take longer. However, it seems likely that many larger databases (for example the directories of a large file system) could be handled by considering them as multiple separate databases for the purpose of writing checkpoints. In that case, we could either use multiple log files or a single log file with more complicated rules for flushing the log.

The size of virtual memory, and the total amount of disk space required are not restrictive since the limitations on checkpoint size and frequency are more restrictive with present hardware. The virtual memory paging traffic provoked by this design appears to be no worse than the disk access traffic produced by other designs, and may well be better.

We believe that this is the preferred design for the class of databases we have described: moderate size (up to 10 megabytes), moderate update traffic (bursts of up to 10 update transactions per second, but only up to 10000 transactions per day), and no requirement for client-visible multi-step atomic transactions.

REFERENCES

1. Gawlick, D. Processing hot spots in high performance systems. *Proceedings Compcon-85.* IEEE, 1985.
2. Gealy, M. Experience with Clearinghouse in large internetworks. *Proceedings of Second SIGOPS Workshop on Distributed Systems.* ACM, 1986.
3. Gray, J.N. Notes on database operating systems. *Lecture notes in computer science, 60.* Springer-Verlag, Berlin, 1978.
4. Gray, J. et al. One thousand transactions per second. *Proceedings Compcon-85.* IEEE, 1985.
5. *IMS/VS version 1 general information manual* (form GH20-1260-12), IBM, 1984.
6. Lampson, B.W. Designing a global name service. *Proceedings of Fifth Symposium on the Principles of Distributed Computing.* ACM, 1986.
7. Lampson, B.W. Atomic transactions. *Lecture notes in computer science, 105.* Springer-Verlag, Berlin, 1981, 246-265.
8. Oki, B.M. et al. Reliable object storage to support atomic actions. *Proceedings of Tenth Symposium on Operating System Principles.* ACM, 1985.
9. Schroeder, M.D. et al. Experience with Grapevine. *ACM Trans. Comput. Syst. 2,* 1 (Feb, 1984).
10. Lehman, T. et al. Query processing in main memory database systems. *Proceedings of 1986 SIGMOD conference.*
11. Lehman, T. et al. A recovery algorithm for a high performance memory-resident database system. *Proceedings of 1987 SIGMOD conference.*