Transactions and Concurrency

G. Lawrence Krablin

Persistent Programming
Research Report 46
September 1987

Alan Dearle

Transactions and Concurrency

G. Lawrence Krablin

Glasgow University

University of Pennsylvania

Burroughs Corporation

Transactions and Concurrency

G. Lawrence Krablin
Glasgow University
University of Pennsylvania
Burroughs Corporation

14 Sept 1986

1 Introduction

Databases and other, more general, systems of persistent data storage and management have become a dominant feature of the computing landscape. The use of databases in concurrent and distributed environments has introduced a number of issues and problems. This paper reviews some of these issues and the mechanisms that have been developed to deal with them.

1.1 Databases

A database [5,24] is a collection of data related to some enterprise. In order for the data to be useful, some structure is imposed on it, either explicitly by a formal description of some kind, or implicitly through the conventions enforced for the use and update of the data. The structure of the database implies both constraints on the values of the data elements, and relationships among them. (The subject of 'data modelling' is discussed in [23]). As a whole, the structure and values of the data in the database provide an abstract model of the relevant aspects of the enterprise.

In most cases, the database will change over time (even a dictionary changes, reflecting corrections, new words, changes in usage, etc.) to continue to accurately represent the changing enterprise. The introduction of change to the database requires some definitions so that the process of change can be understood and managed.

1.2 Properties

For a database, there is some notion that it is correct, that it accurately represents the abstraction of the enterprise. Correctness implies that the structure is internally consistent, that the values and relationships maintain the consistency constraints implied by the abstraction, and that the values are an appropriate representation of the state of the enterprise. The first two of these three qualities can be lumped together and called consistency. In principle, it is possible to write a single statement in the predicate calculus, which completely expresses the requirements of consistency for the database. The consistency predicate is a static statement, defined only by the abstraction. The third part of the correctness criteria is a dynamic statement; that the collection of changes made to the database, including their order in time, is an adequate representation of the changes in the enterprise.

In any model, choices have to be made about what elements of the actual enterprise are to be included in the abstraction and how they will be represented in the model. As a natural result of the abstraction process a lot of detail is compressed into simple units, such as cases of nuts and bolts, rather than individual parts. On the other hand, there may be abstract objects in a model which are composed of smaller parts or which have a number of distinct qualities which are individually represented. When the model is a database on a computer, we also have to contend with the limited choice of units available on the computer. Computers work in terms of bytes or words internally, and much larger units (disk sectors, etc.) on their bulk, long term storage media. All computer operations are expressed in these units. The properties of consistency and correctness are explicitly defined in terms of the elements of the abstraction, not the underlying concepts (bytes, etc.) which are used to implement the elements.

These circumstances, individually and in combination, point out that even a simple change to the enterprise may require multiple changes in the database as it is represented on the computer. Within the database, a

change which is only partially carried out may result in the database being both inconsistent, in terms of the abstraction itself, and incorrect, in terms of its representation of the enterprise. Such inconsistency or incorrectness could even appear if the database were examined at an inpropituous time, in the middle of a change.

2 Transactions

In order to account for and alleviate the effects of this problem, the abstraction must implicitly define one more unit, the unit of change, or more generally, the unit of activity of any kind. A transaction [7,11,12] is a unit of work which corresponds directly to a single activity in the enterprise which is to be modelled in the database. A given model will have many kinds of transactions, some so simple as to not need the support of the past several paragraphs, others very complex indeed, involving thousands of individual operations in the database (perhaps millions of computer instructions). What they all (should!) have in common is that they preserve the consistency of the database and correctly model an activity of interest in the enterprise.

2.1 Transaction Properties

An examination of the database at a time between two transactions can be expected to find it consistent and, in terms of the abstraction, representative of the enterprise at a time between the two corresponding "real" activities. (Note that there is an implicit assumption here that the transactions occur in the same temporal order as the actual activities). If the database is examined while a transaction is in progress, it is likely to be both inconsistent and incorrect. Therefore, it would seem that the database should not be examined at such a time. Transactions are called atomic to emphasize the fact that they must appear to happen all at one instant, in spite of considerable potential internal complexity, to effectively eliminate the possibility of "seeing" the database in an inconsistent state.

If a transaction is going to successfully complete eventually, it is always possible to wait for that completion before attempting to observe the state of the database. However, transactions sometimes fail to complete successfully. The computer may crash, the program performing the transaction may fail, values and relationships of data in the database may preclude successful completion. Since this is a fairly common phenomenon, it won't do to invalidate the database when it happens. An atomic transaction must happen all at once, or not at all. The implementation of the database and its transactions must provide a means of restoring the database to its state at the beginning of a transaction (actually a state equivalent to one in which the transaction had never occurred), in case of failure. A transaction in such a system may be called reversible.

Since it is a rare database which experiences just one transaction in its lifetime, some consideration must be given to the relationships among transactions. In the discussion thus far, a serial sequence of activities has been assumed (concurrency appears below). In this context, there is a total order on the transactions based on their time of ("instantaneous") occurrence. A transaction operates on the database as it is left by the previous transaction, modelling the effects of sequential activities of the enterprise. It would be disastrous if an apparently successful transaction became "undone" sometime later. In the sequential environment, this might occur as a result of administrative measures taken to recover the database after a corrupting failure like a disk crash. The typical response is to restore the database from a backup medium, but as it is impractical to create the backup after every transaction, there is may be some possibility of lost transactions. There must be a means, within the database system, to protect it against such failures. The database must be resilient and the transactions recoverable.

There is a qualitative distinction to be made between atomicity and reversibility on the one hand, and resiliency and recoverability on the other. The first two properties can be expected to be absolute; that is, a system either provides them or it doesn't. At least in centralized systems, this is achievable. Recoverability implies the ability to survive failures which invalidate the results of already completed transactions. Commonly, this is a failure of long term (so-called "permanent") storage. The only protection against such failure is to store the relevant information in enough places with low enough probabilities of failure, that the overall probability

of unrecoverable failure is acceptably low. However, it can never be zero.

2.2 Concurrency

A sequential model is inadequate for computer databases. There are several reasons, including the disparity in processing speed between humans and computers, multi-programming and multi-processing in computer systems, and distribution of a database across several different computers, why concurrent processing of transactions must be accomodated. This means that the constituent sub-actions of a number of transactions will be interleaved. In effect, each transaction will be operating on a database which is inconsistent and incorrect because other partially complete transactions are also active. Clearly, unconstrained concurrency is untenable.

There are, however, a few observations which can be made about the behavior of transactions in a database system. The first is that some transactions are readonly; they do not alter the contents or structure of the database. The database is never, even temporarily, made inconsistent or incorrect by a readonly transaction. Any number of readonly transactions may be run concurrently without danger.

The second observation is that databases are not monolithic entities and most transactions involve only a portion of the database. Any two transactions which use totally disjoint sections of the database may also be active concurrently.

In neither case does it matter which transaction is "first". That is, the result of processing them concurrently is indistinguishable from processing them sequentially in either order.

The cases where the apparent order is significant are usually described in terms of read and write sets. The read set of a transaction is the collection of data elements within the database whose values are consulted by the transaction. The write set is the set of elements whose values are altered by the transaction. The read and write sets of a transaction frequently overlap. Two concurrent transactions will conflict precisely if the write set of one overlaps either the read or write set of the other (the definition is symmetric).

2.3 Serializability

The generally accepted criterion for correctness of concurrent database transactions is serializability [8]. In general, a collection of transactions run concurrently is serializable if there is some order in which the same transactions can be run, one at a time, that would give the same results as the concurrent execution did. The results include both the final state of the database and any output reported from the transactions.

The serialized order is, of course, a total order, with every transaction fixed in relation to every other transaction. The actual necessity of resolving conflicting subtransactions only demands a partial order, which does not fix the relationships among transactions which do not interfere with one another. A serialization may be any total order consistent with the partial order defined by the activities of the transactions.

As a practical matter, serializability is defined in terms of the sets of data objects consulted (read) and altered (written) by the transasction. Each object involved is assumed to be read at most once and written at most once (no loss of generality), with each action taking place at an identifiable time in the context of the whole system. "Simultaneous" actions can either be ruled out, based on the physical constraints of the system, or, for a distributed system, ignored as irrelevant, since there can still be no simultaneous actions on a single object. The overall record of the activities of a concurrent mix of transactions is known as the log or trace. The relative position of two transactions in a serialized order depends on three kinds of potential conflicts in the log: read-write, write-read, and write-write, Transaction T_i precedes T_j in the order if for any data item d in the database, one or more of the following hold:

1. T_i reads d before T_j writes d (T_i doesn't see the change made by T_j)
2. T_i writes d before T_j reads d (T_j does see the change made by T_i)
3. T_i writes d before T_j writes d (some later transaction will see the value left by T_j, not that left by T_i)

If none of the above hold (in either direction) between two transactions, for any data object, their relative order in the serialization is either irrelevant

(arbitrary) or possibly defined by the transitivity property of the order in relation to other transactions. If more than one of the above conditions hold, or the two transactions are related by common access to multiple items, there is the possibility that the transactions should each precede the other. Since this is impossible, these two transactions are not considered serializable, even though the results might be the same for either order.

It can be seen immediately that the only relationship of two transactions through a data item which does not force an order is read-read, where neither transaction modifies the item. This can be generalized to the notion of "commutative" operations , two primitive operations on an item, such as two reads, whose order of execution is of no consequence to the results. For data objects supporting primitive operations more complex than read and write, this is a weaker requirement than serializability stated in terms of read and write logs, potentially allowing a greater degree of concurrency.

3 Concurrency Control

In the various schemes which have been developed and put into use to guarantee serializability in a concurrent environment, one important property is the point, during the lifetime of a transaction, at which its position in the order is determined with respect to the other transactions running concurrently. In different systems allowing concurrency, the determination point may differ. In a strictly sequential environment, the order is obviously determined as each transaction begins (or is selected to begin by a scheduler). This transaction lies after the already finished transactions and before those not yet started. With some concurrency control mechanisms, there is a different determination of the order.

A number of disciplines have been developed in database management systems to prevent conflicting transactions from proceeding concurrently. Three of these, locking, optimism, and timestamping, will be discussed in the following sections. All (indeed any such mechanism) require one additional property of transactions. This is identity, that distinct transactions can be distinguished and identified so that operations on the database can be associated with the correct transaction.

An important consideration for all forms of concurrency control is how to define the "objects" which are the referents of the activities forming the log, and thus used to locate conflicts [10,12,13]. An object, in this sense, is a composite of one or more of the abstract elements of the database. As long as these objects do not overlap (in terms of primitive data elements) and cover the databse (that is, the collection of objects form a partition of the database), there are no correctness concerns involved in this choice. (Note that the use of the term partition is a little loose here, as it is meant to apply to the elements of the abstraction, not necessarily the bits used to represent them). However, with some mechanisms, there may be a significant impact on performance.

If the unit chosen as an object causes the whole database to be encompassed in a single object, there will be no concurrency at all, but the overhead of concurrency control will be small. On the other hand, the granularity of objects could be made as fine as individual fields in records (in principle, individual bits, but this has no meaning in the abstraction). This might permit maximum concurrency, but would impose a large time and space penalty. In a database organized around records, the natural choice for the object of interest seems to be the record. This is tied to the fact that there is generally already some overhead in storage and processing for record manipulation, and to the role of the record as a conceptual unit, representing a single aspect of the enterprise.

3.1 Locking

The most obvious dynamic mechanism for controlling concurrent execution of potentially conflicting transactions is to arrange to give one transaction exclusive access to some portion of the database for a period of time [8,13]. During this time, no other transaction can read or write the data covered by the lock, thus preventing any conflict. This kind of locking is derived directly from the principle of mutual exclusion that has been developed for operating systems in multi-processing, multi-programming computer systems.

A critical aspect of the design of a lock-based concurrency control system is the choice of interval over which the lock on any given data element is to be held. This interval could range from the entire duration of the

transaction down to exactly the time during which the data element is needed by the transaction.

The first of these is certainly effective, but it may constrain concurrency more than strictly necessary. There are also some problems with its implementation. The first is the difficulty of determining, before it begins, what elements in a database a transaction will use. This set is commonly determined dynamically by the nature of the transaction, conditions within the database at the time, and input data, which may be interactive and also depend on the current state of the database. If the lockable elements are classes of data, such as whole relations, then such an approach may be feasible.

The second problem here is that the locking process itself must be made atomic, so that all the locks are seen to be obtained simultaneously. This is also true for individual locking, but at the beginning of a large transaction, other transactions might have to wait a long time to get started, a potential problem in an interactive system.

The all-at-once approach to locking is free from deadlock. That is, the situation cannot arise where two transactions are each waiting on a lock held by the other. However, the locking algorithm must be prepared to abort, release all the locks obtained, and restart the process, if it is unable to obtain the lock for one or more of the desired elements of the database. Failure to do so could cause deadlock.

At the other end of the spectrum is the "exactly as long as necessary" approach to locking. The first restriction that must be applied is that a transaction must lock any given data element only once (that is, abstain from releasing it and later locking it again). The consequences of locking an element, unlocking it, then locking it again (assuming it is actually used during both locked periods) are that the transaction may be unserializable with respect to another transaction which uses the data element between the two locked periods.

Further analysis of this same kind of problem leads to the "two phase" locking discipline [8]. This is the practice of dividing a transaction into a growing phase and a shrinking phase. During the growing phase, locks are obtained. During the shrinking phase, locks are released. Once a transaction has initiated its shrinking phase by releasing any lock, no further locks may be obtained. There is a proof [8] that this discipline guarantees serializability. (The subject of serialization and locking disciplines is treated formally in [2]).

There are several important considerations involved with the use of two phase locking for concurrency control.

1. A transaction must have an identity with which to register its ownership of locks. This is necessary both to verify that it is this transaction which holds the lock, and to facilitate recovery from aborts and crashes. An alternative view of the same requirement is that each transaction must keep track of the locks that it holds, in a form which is usable for recovery should the transaction abort or the system crash.

2. A transaction must obtain ownership of the appropriate lock before using an object in the database. The abstraction itself defines which bits of state correspond to objects in this context. For example, the locks themselves, which have no meaning or existence in the abstraction, are not objects in this context. However, since they are also manipulated in a concurrent environment, they must also be protected in a somewhat similar way (primitive mutual exclusion, for instance). This hierarchy continues, until a level is reached where concurrency is not an issue.

3. All locks, once obtained, must be held by the transaction until it commits or aborts.

4. All locks held by a transaction are released when the transaction terminates, either by commit or abort.

5. Termination of a transaction means that no further activities will occur on its behalf and implies that its identity will be permanently retired.

The lock may be obtained in a separate operation before any other reference to the object, or it may be automatically requested on the first reference. In some situations, a programmer or compiler may be able to determine

statically some or all of the objects which must be locked and thus avoid the necessity of verifying the lock on each reference, but in general, each reference must be dynamically checked (since objects may be accessed by pointers rather than by explicit names appearing in the program).

3.2 Deadlock Handling

Although a two-phase discipline does guarantee serializability in any mix of transactions which all complete, it does not provide any protection against deadlock. It is possible for two transactions to request locks on two objects such that each has a lock on one object and cannot continue until it gets the lock on the other object. Neither transaction can continue until the other releases a lock, which it won't do until ...,. In a general object locking system, deadlock is essentially unavoidable. All the techniques used to avoid deadlock [1,6,13,20] involve some prior knowledge of the form and behavior of the transactions. This is possible in some limited situations, such as a small set of distinct transactions whose interactions and potential conflicts are understood in advance so that a restrictive scheduling regime may be employed.

If deadlock cannot be avoided, it must be handled, so that the whole system doesn't grind to a halt. Handling is done in two parts: detection and resolution. In a centralized system, deadlock may be detected by keeping track of which transactions hold locks on what objects, and which transactions are blocked trying to obtain a lock. A simple analysis (topological sort of a graph, for instance) will indicate any groups of two or more transactions which are mutually deadlocked. This method may be expensive if the granularity of lockable objects is small. The analysis may also be difficult and slow in a distributed system [9,16]. A parameter of this kind of deadlock detection is when to do the analysis: whenever a transaction blocks, periodically (continually), or only on some evidence of deadlock, such as failure of a transaction to progress for a long time.

A different approach uses a timeout to assume that any transaction which isn't progressing very quickly actually is involved in a deadlock. Then deadlock resolution is applied without actually verifying that the deadlock actually exists. This method is suitable for systems where transactions are expected to be of fairly uniform duration, so that a reasonable timeout value can be chosen to effectively eliminate spurious deadlock detections. Timeout is also an attractive method in distributed systems since it does not require any centralized analysis and it conveniently handles prolonged or permanent communications failures uniformly. The drawback of timeout based detection is the potential for false indications, especially where long running transactions compete with short ones.

The typical way of resolving a deadlock situation is to force one of the transactions involved to give up one of the locks involved, allowing some other transaction to proceed. The two phase discipline requires that giving up a lock initiate the shrinking phase of the transaction, so the 'victim' cannot obtain any more locks. Since the victim cannot progress any further, the solution is to force the victim to abort completely, release all locks, and restart from scratch.

The victim may be chosen as any of the transactions involved in the deadlock. If timeout is used for detection, the victim is necessarily the transaction which timed out. If a more sophisticated deadlock analysis provides a complete or partial list of transactions involved, a choice may be based on such factors as priority, age, number of times already victimized, etc.

When a lock is given up by one transaction, some scheduling policy will be applied to assign the lock to one of the transactions (if there are any) waiting for it. Such a policy might use an explicit priority, FIFO, age of transaction, or just a random choice. The policy has an effect on system throughput and the performance of individual transactions, but is transparent to correctness in terms of serializability.

3.3 Differentiated Locks

Although the two phase discipline described guarantees serializability, there are improvements to be made without sacrificing that guarantee [10,11,13,20]. Specifically, more concurrency can be obtained by specializing the locks to take advantage of the fact that there is no conflict involved in two or more transactions simultaneously reading the same item. Here the principle is that a read lock (permission to read) may be granted to

any transaction as long as no other transaction holds a write lock (permission to modify the item). A write lock may be granted only when no other transaction holds either a read or write lock. The conventions for use of an item are similar to those for an item with a simple lock except the differentiation of read and write. A transaction must hold the write lock to modify the item in any way. Any other access need only be protected, by a read lock, against a concurrent transaction modifying the item.

In terms of serializability, the results for two phase locking continue to hold, since interleaved reads by concurrent transactions do not represent conflicts. Concurrent transactions performing writes are protected in exactly the same way as with a simple undifferentiated lock.

A broader generalization of differentiated read/write locks can be obtained by considering which operations on an object are commutative (interchangeable in time without external effect). Clearly, in a model with simple objects where the only operations are read and write, two reads on the same object are always commutative and no operation is commutative with a write. To refine the locking scheme further requires a set of objects with higher level operations. Such an example, the semiqueue, is explored in [25,26]. Since locking is essentially object oriented, rather than transaction oriented, such generalizations are easy to build, using the same basic structure as simple locking.

In the earlier discussion, scheduling (choice of which of several competing transactions would get a lock next) was left to the underlying implementation. There is an assumption that this implementation will choose a strategy which is reasonably fair. That is, under normal loading, no transaction will be starved completely by having others continually placed ahead of it. This is fairly easy to manage with undifferentiated locks by having the scheduler maintain a FIFO list of requesters for each lock. Some sophisication may be applied to re-ordering these lists to minimize the probability of deadlock. The situation for differentiated read/write locks is more difficult, as late arriving read lock requests will be filled ahead of outstanding write lock requests as long as other reading transactions remain active. A modified set of rules, blocking new read locks while a write lock request is outstanding, is one approach to this problem.

Deadlock detection and resolution issues are (generally) unchanged by the differentiation of read and write locks, except that there are now two (or more) locking structures instead of one. This slightly complicates deadlock analysis, but has no effect at all on a timeout detection method.

3.4 Commit/Abort Considerations

Locking, as it has been described above, satisfies the requirements of serializability, and thus guarantees that successful transactions will be atomic. However, there can be expected to be unsuccessful transactions as well (deadlock resolution victims, for example), and these must also be atomic in the sense of having their effects reversed, never becoming visible to the outer world.

What is needed is to keep track of all the changes made to the database by a transaction, so that the original values may be re-installed on abort. The constraints of the two phase discipline prevent these uncommitted changes from being seen by, and thus affecting, any other transactions. The obvious time to note these original values is at the point where the lock is obtained which allows a modification to be made. It cannot be done earlier than that, as the object might be changed by some other transaction, and it must be done before any modification is actually made. The preserved value is, like the lock, associated explicitly with the data object, and implicitly, like lock ownership information, with the transaction whose abort would require the backup value.

The processing activity when a transaction commits is simply to release all the locks, and, in principle, discard the backup copies of altered objects. External requirements, such as legal auditing procedures, and reliability concerns may make it necessary to keep such information indefinitely. The only serializability concern for commit is that once the process is begun (the first object unlocked), it must be completed. Incomplete commit processing may leave the database in an indeterminate state, with some objects updated, and others still locked. The commit process itself may be considered a transaction in the context of a lower level of abstraction.

Abort processing is similar, with all locks being released. However, instead of being discarded, the backup information is used to restore the values of objects which were modified. The modified values are then discarded. (Again, auditing may necessitate the retention of the discarded values). The same serialization requirements apply here as for commit.

3.5 Locking Overhead

For each lockable object, the storage overhead for locking is just the lock itself, represented by the name of its holder (needed for checking and abort recovery) and an unlocked indication. Differentiated locks additionally require a list of transactions currently holding read permission, with an indication of an empty list. For commit processing, each transaction must have, implicitly or explicitly, a list of locked objects. Abort recovery further requires the original value to be available to be restored. In addition, deadlock analysis may require further data structures associated with both the objects and the transactions.

The processing overhead involved is to determine the necessity of locking, obtain the lock, and verify upon access that the lock is properly held. The backup copy must be made. On commit all the locks must be released, and, on abort, the backup values restored. It may be necessary to engage in deadlock analysis and victim selection, and the data structures associated with deadlock analysis must be dismantled upon transaction completion.

3.6 Locking Variations

Concurrency control by locking has been described in fairly simple terms: getting the lock, checking when necessary, releasing it, saving and restoring the value. There are a number of variations which may reduce the processing overhead. Where address mapping hardware is used and the object sizes involved fit well, the lock checking overhead can be reduced to the page fault checking in the hardware. It may be more convenient to leave the original object intact until commit, and do modifications to a private, shadow copy. Then commit processing will have to install the newly committed value. This approach is attractive when the permanent data are on a limited access medium, such as disk, and normal operations would use a copy in primary random access memory.

Where auditing is required anyway, "before images" may be copied to the audit trail. Then abort processing will back down the audit trail restoring objects touched by the aborting transaction. The audit trail may also contain the list of objects locked, eliminating the dependency on the internal record keeping of an aborted transaction.

4 Optimism

The optimistic method for concurrency control [14] is based on the observation that, in some database systems, concurrent transactions rarely conflict. That is, a transaction is almost never forced to wait for a lock to become available. To the extent that this is true, locking for concurrency control is just so much wasted time and space. The optimistic method of concurrency control allows a transaction to proceed unhindered by the activities of other transactions until it is ready to commit. At this time, a check is made that there was no unserializable interaction with an already committed transaction. If there was, the present transaction fails involuntarily (much like a deadlock victim) and must restart from the beginning. Otherwise it completes successfully.

The actual mechanism requires keeping track of the read and write sets of the transactions. In addition, all writes, and, implicitly, creates and deletes, are done to "shadow" versions (local copies) of the actual permanent data items. This protects against a transaction which eventually fails after writing, thus serving the same purpose as saving the old value in the locking routines described above. In the optimistic system, all transactions consist of a read phase (reads and shadow writes - the body of the transaction), validation, and a write phase when the shadow values are installed as the permanent values.

The validation process consists of assigning the transaction a place in the serial order and then verifying that it can be validly placed there in the order. The serialization numbers are assigned sequentially as transactions reach the validation stage. The validation check is to verify that at least one of the following criteria holds for every transaction T_i which precedes (in the serialization) the present transaction T_j (taken from [14]):

1. T_i completes its write phase before T_j starts its read phase (T_i completely precedes T_j)

2. the write set of T_i does not intersect the read set of T_j and T_i completes its write phase before T_j starts its write phase

3. The write set of T_i does not intersect the read set or the write set of T_j and T_i completes its read phase before T_j completes its read phase

These conditions ensure that T_j properly follows T_i: T_j has no affect on T_i and sees all of T_i's writes. Thus serializability is guaranteed.

Once the validation is complete, the write phase is executed, installing new values in the permanent data structure. The assignment of the location in the serialization must be done under mutual exclusion. The read and write sets of all transactions which completed after any currently running transaction began must be kept to allow validation of the later transactions. This is a potentially unbounded amount of information. If any of these records are lost because of limited storage, following transactions whose validation required that information must be forced to abort, as they cannot be validated. They will then be restarted and will eventually occupy a later position in the serialization. Another aspect of optimism comes into play here - it is hoped that this situation will not occur often and not repeatedly for any given transaction. It is suggested in [14] that, in such a case, a transaction may be restarted and run all by itself, and allowed to complete before concurrent processing is resumed.

The optimistic method has read/write differentiation built into it, as the read and write sets are separately maintained and play distinct roles in validation. Further specialization would require, for each special case, a restatement of the validation criteria and thus a new validation algorithm, possibly with different record keeping requirements.

Also built into this method is the backup mechanism for abort. Shadow update, much like the variation suggested for locking, avoids the need for explicit preservation of permanent values, at the cost of requiring explicit installation of newly committed values. Again, where permanent data is maintained on secondary media, this may not be a significant cost.

4.1 Overhead

The processing time overhead for the optimistic method is spread among four distinct activities. During the read phase of the transaction, the read and write sets must be maintained, and the shadow copies of objects to be modified must be made. Then, at commit, the validation process must be performed. The magnitude of this activity depends not only on the characteristics of the transaction to be validated, but also on the number of other, already committed, transactions which were (might have been) concurrent with it, and the sizes of all the read and write sets.

Finally, after validation, the new copies of updated objects must be installed. It is noted in [14] that some portions of the validation process must be performed under lower level concurrency control (such as mutual exclusion). Also, the process of installing new values must be completed promptly, as this write phase plays a role in the validation of later transactions.

The processing for abort is simple: just forget everything, read sets, write sets, and shadow copies, except what is needed to restart the transaction.

The space overhead is that required to maintain the shadow copies of updated objects and the read and write sets. The shadow objects (or the old permanent objects) are discarded during the write phase. However, the read and write sets (along with the times associated with the phases) of a sucessful transaction must be kept as long as there are still outstanding any other transactions which were concurrent with it. In a busy system, this can build up significantly.

5 Time Stamping

A third mechanism for managing the interactions of concurrent transactions is timestamping [3,18]. Each transaction is assigned a unique timestamp. This is a somewhat arbitrary identification of the place the transaction will have in a canonical serialization of the concurrent mix of transactions. In practice, each transaction will be assigned a timestamp higher, or "later", than that already assigned to other "older" transactions.

The timestamp is used to arbitrate conflicts arising among transactions in the following way. All read and write operations must be performed in timestamp order. This means that a read for a transaction with timestamp TS_1 cannot be performed on a data object which has already been written by a transaction with timestamp TS_2, where $TS2 > TS1$. There are similar restrictions for write-read and write-write conflicts. In a case where such an out-of-order request is made, the requesting transaction (with the older timestamp) is aborted and restarted with a new, higher timestamp.

The scheme is implemented by maintaining read and write timestamps with each object. These are the timestamps of the transactions which last performed the corresponding operations on the object. A read cannot be performed if there is already a higher write timestamp, and a write attempt would be aborted if the read timestamp is higher. Ostensibly, a write should also be denied by a higher write timestamp (a write-write conflict), but Thomas [21,22] observes that such a write, assuming it doesn't also conflict with a read, can simply be ignored, as "already obsolete". In [3], this is called the "Thomas write rule". If the same transaction later attempts to read back the value, the read will fail because the later write timestamp remains.

In a distributed environment, transactions may be assigned timestamps by autonomous authorities, as long as the resultant timestamps are unique. Using the real time for the most significant portion of the timestamp, and a unique "timestamp issuing authority identification" (hostname in a network, for example) keeps the timestamp unique while preserving some approximation of real time order among the various locations. It isn't even necessary to synchronize the clocks very closely, since the critical property of the timestamps is uniqueness. However, if the clocks differ by very much, or the assignment of timestamps isn't in real time order, transactions may suffer starvation because they cannot compete with other transactions whose timestamps are derived from a clock running "fast".

Reed [18,19] has suggested an elaboration of the timestamping mechanism, also studied in [4]. This is to maintain multiple versions of each data object, with a separate write timestamp for each version and a set of read timestamps for the object as a whole. Reads are performed by simply choosing the appropriate version (highest timestamp lower than that of the read) and are thus never rejected. A write may still fail if a read has occurred with a timestamp greater than that of the write, but less than the next greater timestamp associated with a write which has already occurred. This corresponds to the situation where a later read already depends on an earlier write. The current write would conflict with that relationship, and must be rejected. As with the read and write sets for optimistic concurrency control, it may not be reasonable (in terms of storage) to retain all the versions or even all the read timestamps necessary to process all transactions. In this case, reads with old timestamps may have to be aborted because the system cannot supply the necessary (old) version. Note that the multi-version form of timestamping automatically uses the Thomas write rule, as, although the older write is not discarded, the resulting version may never be used.

That timestamping guarantees serialization among successful transactions is obvious, as the order is effectively chosen as the transactions begin. Any action which would violate the order is aborted, causing the abort of the whole transaction. Multi-version timestamping provides a wider window for a slow or long transaction to succeed, but it is only a matter of degree. Deadlock is not a consideration, as any potential deadlock situation will already have been resolved by aborting an attempt at an out-of-order action on an object. However, inordinately slow transactions or communications failures in a distributed system can cause indefinite delays for an otherwise healthy transaction. As with locking, a timeout mechanism can be used to break up such a situation [18]. More formal treatment appears in [3].

Although the timestamping method may do well at maximizing concurrency, it must still be prepared to handle abort situations. These may be generated by serialization failures or by other conditions outside the timestamping mechanism. Reed [18] discusses the use of conditional "possibilities" to deal with this issue. The effect is to delay any transaction which depends on a possibility whose outcome hasn't yet been determined. Each version in a multi-version scheme is associated with a possibility, which is determined when the writing transaction (which created the possibility) commits or aborts. If the transaction aborts, the possibility also aborts and any other operations depending on the possibility must also abort or try another possibility (version) for the same object. This kind of interference can be lessened for read only transactions, which may be assigned arbitrarily old timestamps [3].

5.1 Timestamping Overhead

The processing overhead for single version timestamping without possibilities is very low, as it is only necessary to check and update the object timestamps. The only storage overhead is that needed for the object timestamps. Multi-version timestamping requires space for many versions of the object, each with a write timestamp, and a set of read timestamps for each object. Each read or write requires a search through the timestamps to determine validitiy. As with the optimistic method, it may prove impractical to retain all the relevant versions and read timestamps indefinitely.

The addition of possibilities, needed for abort, adds a delaying structure, similar to a lock, to each unconfirmed version, plus a list of objects affected to each transaction. Commit processing consists of confirming the possibilities, and thus freeing transactions dependent on them. Abort is done by denying the possibilities and either aborting or redirecting waiting transactions. When a write possibility is denied, the associated version is discarded.

6 Comparison of Concurrency Control Mechanisms

Three different concurrency control methods have been described. They have differing requirements in terms of auxilliary data structures and processing overhead. All must make some provision for keeping a permanent value and one or more temporary, tentative values for each object being modified by a currently running transaction. All must make some provision for forcing a transaction to abort, although the circumstances vary. Although all three methods call for overhead processing on every access to the object, clever organization of the addressing mechanism can limit this overhead to the first access by a given transaction.

Locking requires extra storage per data object to keep track of the lock status and owner. Depending on the sophistication of the locking scheme, this varies from a single transaction identifier to a set of such ids. Deadlock analysis may also require a list of locked objects to be kept with each transaction. The appropriate lock must be obtained before the data is accessed, which implies that the lock must be checked on every access. When a transaction completes, it disappears from all these data structures.

Locking requires some mechanism to detect the presence, or probable presence, of deadlock, and a corresponding way of dealing with it. Deadlock is generally resolved by choosing as a victim a transaction which is party to the deadlock and forcing the victim to abort. A transaction can only become a victim when it is blocked trying to obtain a lock.

For transactions operating in a locking environment, the determination of their relative order in any equivalent serialization occurs dynamically, whenever an access is made by one transaction to an object already accessed by the other and the accesses conflict. The first transaction accessing the object will precede the second in the order. The second will wait to complete its access until the first finishes. Deadlock arises when an attempted access conflicts with an already established (through another object) order relationship. Deadlock analysis and detection may be a major factor in the complexity, size, and performance of the system.

Once a transaction has obtained access to all the objects it will use, it will not suffer further delay or interference, and, in particular, can no longer become involved in a deadlock.

The data structures required for locking are distributed between objects and transactions. Obviously, the lock itself is associated with the object. A list of objects affected must be kept with the transaction to facilitate commit and abort processing. The backup values needed for abort may be associated with either the transaction or the object.

Optimistic concurrency control requires the maintenance of read and write sets for each transaction. In addition, a private copy must be made of each object to be modified. In principle, all of these things must be done or checked on every access. The read and write sets must be kept for an indeterminate time after successful completion, along with the beginning and ending times for the read and write phases.

With the optimistic method, transactions suffer no delays or apparent interference until they are ready to commit. During the validation process, a transaction may be found to be in conflict with an already committed transaction. If so, the not yet validated transaction must be aborted. This is the only possibility for involuntary abort. The serialization order is determined as each transaction enters the validation process. The circumstances requiring abort during validation are that the actual access patterns (conflicting usage) contradict the order established by the race to commit.

The data structures for optimistic concurrency control are transaction centered. The read and write sets, and the private copies are associated with the transaction, with object names being used only to access the originals and as tags in the sets. However, in a distributed system, the objects could be partitioned to allow local management.

In its simple form, timestamping just requires a read timestamp and a write timestamp for each object. The timestamp of the transaction must be checked against these. If the operations would not be performed in timestamp order, the late transaction must be aborted.

Timestamps are assigned to transactions as they begin, thus immediately fixing their places in the serialization order. A forced abort is the result of this predetermined order being violated by actual events. In order to allow for abort, a form of conditional update, which may delay following transactions, is necessary.

Multi-version timestamping requires the maintenance, for each object, of a set of read timestamps and a set of versions, each also timestamped. The characteristics are essentially the same as for single version timestamping, except there is greater provision made for allowing slow transactions to keep their place in line, at the cost of maintaining records related to already committed transactions. Again, conditional update is necessary at the level of individual versions.

Under timestamping, as under locking, the data structures are distibuted between objects (versions, possibilities, and read timestamps) and transactions (the transaction timestamp and the list of possibilities to be confirmed or denied).

The serialization order determined by locking is a partial order. That is, there may be many consistent serializations. The order is determined dynamically. Both timestamping and optimism generate total orders somewhat arbitrarily, either as the transactions begin or as they finish. Only locking encounters deadlock, but the others must remember transactions after they have finished. Only optimism can force the abort of an apparently successfully complete transaction. Under timestamping, a transaction may be aborted involuntarily if it is unable to find or create a version with an appropriate timestamp.

In terms of maximizing concurrency, multi-version timestamping appears to allow otherwise interfering transactions to potentially operate concurrently without delay or forced abort. In the other forms of concurrency control, the argument is a tradeoff of decreased overhead versus fewer aborts. None of locking, optimism, or single version timestamping allow more concurrency among successful transactions.

7 Nested Transactions

In a database system, transactions are an attractive concept because the properties of atomicity and reversibility allow the programmer to abstract away some of the immense complexity of concurrent computation, or even just simplify the process of undoing an aborted activity. The ideas of top-down design suggest that it would be equally attractive to break transactions down into sub-units with the same properties [15,17]. A number of advantages accrue from such an approach.

Procedures are used to compose a program, implementing abstract operations on abstract data objects. The use of transactions adds to these abstract operations the same properties, in the narrower context, that make them so useful at the user interface. They will behave predictably even when concurrent processing is used to implement the higher level transaction. They may also be used to contain the effects of local failures, to permit recovery and retry at as low a level as possible. These low level activities can then be straightforwardly composed in the higher level framework. This is the ultimate goal of abstraction: to be able to ignore the details of lower level implementation, even when it involves concurrency or allows for failures.

In a distributed environment, transactions must act, in a coordinated

way, on objects scattered among various hosts. A nested subtransaction may be effectively used to represent the activities of the parent transaction in each location. This approach may also be used within a single host to deal with resources (objects) under the control of disjoint managers.

A nested, or subordinate, transaction must have a number of properties in order to be useful. In the first place, it will be invoked by a parent transaction for the purpose of accomplishing something. The subtransaction must operate in the environment of the parent, in terms of work already done by or for the parent. Thus the subtransaction must see tentative changes made by the parent, even though the parent has not committed these changes.

On the other hand, when a subtransaction finishes successfully, it must commit its effects only within the parent. That is, the effects will become visible to the parent and thus to any other subtransaction of the parent. If a subtransaction aborts, it must restore any objects it changed to the state perceived by the parent when the subtransaction began.

From this discussion, it should be evident that a subtransaction functions as a component of its parent. This means that, to transactions outside of the parent (competing with the parent), the activities of the subtransaction appear as indivisible activities of the parent.

If the implementation of a parent transaction supports concurrency, that is, allows subtransactions to compete concurrently, then the usual properties of transactions, applied to the subtransactions, require concurrency control to be employed to effectively serialize the sibling subtransactions. The effects of this on the different concurrency control mechanisms is described below. In all respects, competition among sibling subtransactions is identical to competition among disjoint parents, and, indeed, the disjoint parents may be considered siblings of each other.

For nested subtransactions, the concurrency control mechanisms encounter several new aspects of the problem. The first is that a child and its parent are obviously not serializable with respect to each other. There will in general be intermixed reads and writes of common objects.

However, there will be a similar requirement (to serializability) that there exist an equivalent order (schedule or log) of reads and writes such that none of the parent's activities are interspersed among the child's. This is required to guarantee the atomicity of the child.

The second new feature is the limited scope of commit and abort. Upon commit, the effects of a subtransaction are made visible only to its siblings, if any, and its immediate containing parent. This is required to guarantee the atomicity of the parent. Similarly, recovery from the abort of a subtransaction must only undo the effects of the subtransaction, not anything accomplished by or on behalf of the parent "before" the subtransaction was invoked, or concurrently by sibling subtransactions.

This is necessary to allow the parent to continue correctly (in terms of the programming model).

7.1 Locking and Subtransactions

Handling dependent subtransactions with lock based concurrency control complicates the locking algorithms and also multiplies the opportunities for deadlock. The criterion for granting a read lock is still fairly simple. A subtransaction may become a reader as long as the writer, if there is one, is an ancestor of the prospective reader (or the prospective reader itself is already the writer - an extension of the notion of ancestor).

A prospective writer will be granted a write lock so long as all readers and any existing writer are ancestors. This criterion follows naturally from the notion of the subtransaction acting for the parent, in the environment provided by the parent.

For write locks, the preservation of the "old" value, for restoration in case of abort, is more involved. An abort of a subtransaction affects only that subtransaction, but allows siblings and parent potentially to carry on. The value to be restored is the one the subtransaction found when it obtained the write lock. For nested subtransactions, there is, in effect, a stack of old values. Obtaining a write lock pushes another value onto the stack. In addition, the identity of each previous writer must be kept in the same stack, and also restored on abort.

On commit, the parent becomes the holder of the write lock. If the parent was the previous writer, in the stack of old values, then the top entry on the stack is discarded, as the value installed by the committing subtransaction becomes the current value for the parent. If the parent was

not the previous writer, the stack is left as is, so that the value and identity of the previous writer may be restored if the parent aborts.

Read locks are also passed on to the parent when the child commits. In the case of either commit or abort of a subtransaction, its name is removed from the set of readers.

Undifferentiated (as to read or write) locks are treated like write locks. To obtain a lock, the existing lock holder must be an ancestor.

Old values and lock holder identities must be stacked. On commit, the lock is passed onto the parent, with appropriate manipulation of the stack of previous values. On abort, the previous value and lock holder are restored.

One complication with locking and subtransactions arises from the fact that a subtransaction pre-empts the write lock from its parent, in order to ensure that the activities of the child appear to be properly nested as an atomic action within the parent. If the parent continues to run, it will have to be removed from the list of readers (and restored to it upon commit or abort of the subtransaction) as a part of granting a write lock to any descendent. The parent must also be prepared to be blocked from objects to which it formerly held read or write permission. This blocking is not of course evident to the transaction itself, being a part of the underlying implementation, but the possibility does cast shadows on some obvious strategies for efficient implementation, such as manipulating address maps to effect read or write access rights. These problems may be avoided by suspending the parent for the duration of any dependent subtransactions. This sort of problem does not immediately arise if subtransactions are restricted to different hosts in a network. However, in a fully general such environment, these subtransactions could spawn further subtransactions, including in the host of the original parent.

7.2 Optimism and Subtransactions

There are significant changes necessary to adapt the optimistic strategy for dependent subtransactions. The optimistic approach does not track the timing of conflicting usages by competing transactions, only that two transactions overlapped in time and read or wrote one or more objects in common. This implies that the parent must absolutely be suspended during the lifetime of any subtransactions. In order to handle subtransactions "in the environment of the parent", the read and write sets must be maintained and checked for all immediate offspring of a parent. This checking will not involve the read and write sets of the parent's siblings or their ancestors, nor, directly at least, those of children of siblings of a subtransaction. Thus subtransactions compete directly only with immediate siblings.

Similarly to the stacking of values for locking, private copies of objects to be modified must be maintained for each subtransaction. When validation is successful and a subtransaction commits, its modified objects become the private copies of the parent, replacing any copies the parent had of the same objects. The read and write sets of the subtransaction are merged (set union) with those of the parent. They must also be kept separately, within the environment of the parent, to be used for validation of siblings. They may, however, be discarded completely when all concurrent siblings terminate, and certainly when the parent terminates. The process of validation is unchanged except for being restricted to recognizing conflicts among immediate siblings.

7.3 Timestamping and Subtransactions

Timestamp based concurrency control must also undergo significant changes to support dependent subtransactions. As described earlier, a transaction is given a "timestamp" which places it in the serialization order. With subtransactions, a simple order, partial or complete, will not suffice. The subtransactions must appear embedded among the activities of the parent, although they are not distinguishable from the parent from a viewpoint outside the parent.

Reed [18] suggests a solution to this problem which, in effect, makes each timestamp represent an interval on the timeline. Subtransactions may be assigned timestamps within the interval. (A problem with this is that the size of a timestamp is unbounded unless the nesting depth of subtransactions is limited). This places a family of sibling subtransactions effectively within the parent.

The construction of the composite timestamp confirms the desirability

of delaying the parent during the lifetime of the subtransaction, as a range is not ordered with respect to a subrange of itself. This could be avoided by altering the parent's timestamp to just beyond the subrange assigned to each new subtransaction as it is spawned, but this could be difficult to manage in a distributed system.

The management of possibilities for nested subtransactions is similar to the management of locks. Subtransactions are not delayed by uncommitted possibilities held by an ancestor. On the other hand, when a subtransaction commits, its possibilities are confirmed only to its siblings and parent. Older ancestors and outsiders must continue to wait. If a subtransaction aborts, any siblings, ancestors (including the parent), or outsiders waiting for one of its possibilities must either be aborted or redirected to look for a different possibility.

8 Summary

In a single threaded, non-concurrent environment, a database may be treated purely as an abstraction on data structure, with appropriate operations. Transactions may be added to provide both a convenient higher level procedural abstraction for activities taking place in the database, and a mechanism to allow for and manage failure (abort). The elements which are specifically represented in the data abstraction are also important as the objects of reversible operations invoked by transactions.

In a concurrent environment, the relationships among objects, operations on those objects, and transactions are the basis for the management of concurrency. The transactions are defined at the same level of abstraction as the objects, and, with the operations, must provide the desired properties for the system, particularly in the face of concurrent transactions. In this context, the objects, operations, and transactions must be carefully specified, so their properties mesh to support the overall abstraction.

The overall criteria for integrity in the database is that the transactions be serializable. This property is defined in terms of the objects in the database and records of accesses to them (through operations) by transactions. Concurrency control is applied to guarantee serializability among successful transactions.

Several different concurrency control methods have been described, each with its own demands on the objects, operations, and transactions. They have differing characteristics with respect to potential concurrency, resolution of conflicts among transactions, and overhead costs. They also differ in their determination of the relative order of transactions in the serialization.

Also discussed were dependent subtransactions, which provide both a further degree of abstraction and a straightforward way of partitioning transaction management according to geographic or other criteria. Adaptations of the concurrency control mechanisms to handle subtransactions have also been examined.

Bibliography

[1] Bernstein, P.A., Rothnie, J.B., Goodman, N., Papadimitriou, C.A., The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case), *IEEE Trans. on Software Engineering* SE-4(5) May 1978 pp154-167

[2] Bernstein, P.A., Shipman, D.S., Wong, W.S., Formal Aspects of Serializability in Database Concurrency Control, *IEEE Trans. Software Engineering* SE-5(3) May 1979 pp203-216

[3] Bernstein, P.A. & Goodman, N., Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems, *Proc. 6th VLDB* 1980 pp285-300

[4] Bernstein, P.A. & Goodman, N., Multiversion Concurrency Control - Theory and Algorithms, *ACM TODS* 8(4) Dec 1983 pp465-483

[5] Date, C,J.,*An Introduction to Database Systems*, Second Edition, Addison Wesley 1977.

[6] Date, C.J., *An Introduction to Database Systems, Vol II*, Addison-Wesley 1983

[7] Davies, C.T., Data processing spheres of control *IBM System Journal*, 17(2) 1978 pp179-198

[8] Eswaren, K.P., Gray, J.N., Lorie, R.A., & Traiger, I.L., The Notions of Consistency and Predicate Locks in a Database System, *CACM*, 19(11) Nov 1976 pp624-633

[9] Gligor, V., Shattuck, S., On Deadlock Detection in Distributed Systems *IEEE Trans. Software Engineering*, SE-6(5) Sept 1980 pp435-440

[10] Gray, Lorie, Putzolu, Traiger, Granularity of locks and degrees of consistency in a shared database, in *Modeling in Database Systems, Proc. IFIP TC-2 Working Conference on Modelling in Data Base Management Systems*, ed. G.M. Nijssen, North Holland Jan. 1976

[11] Gray, J.N., Notes on Database Operating Systems, in *Operating Systems: An Advanced Course*, ed. Bayer, Graham, Seegmuller, LNCS 60 Springer-Verlag 1978

[12] Gray, J., A Transaction Model, in *Automata, Languages and Programming, Seventh Colloquium*, ed. de Bakker, J.W. & van Leeuwen, J., Noordwijkerhout July 1980 LNCS 85 Springer-Verlag pp282-298

[13] Kohler, W.H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, *ACM Computing Surveys* 13(2) June 1981 pp149-183

[14] Kung, H.T. & Robinson, J.T., On Optimistic Methods for Concurrency Control, *ACM TODS* 6(2) June 1981 pp213-226

[15] Liskov, B. & Scheifler, R., Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *ACM Trans. on Programming Languages and Systems*, 5(3) July 1983 pp381-404

[16] Menasce, D.A. & Muntz, R.R., Locking and Deadlock Detection in Distributed Data Bases,*IEEE Trans. on Software Engineering*, SE-5(3) May 1979 pp195-202

[17] Moss, J.E.B., *Nested Transactions: An Approach to Reliable Distributed Computing*, Ph.D. dissertation & MIT/LCS/TR260 MIT Cambridge, Mass. 1981

[18] Reed, D.P., *Naming and Synchronization in a Decentralized Computer System*, Ph.D. dissertation & MIT/LCS/TR205 MIT Cambridge, Mass 1978

[19] Reed, D.P., Implementing Atomic Actions on Decentralized Data, *ACM TOCS* 1(1) Feb 1983 pp3-23

[20] Rypka, D.J. & Lucido, A.P., Deadlock Detection and Avoidance for Shared Logical Resources, *IEEE Trans. on Software Engineering* SE-5(5) Sept. 1979 pp465-471

[21] Thomas, R.H., A Solution to the Concurrency Control Problem for Multiple Copy Databases, *Proc. 1978 COMPCON Conference*, IEEE, 1978.

[22] Thomas, R.H., A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases, *ACM Trans. on Database Systems*, 4(2) June 1979 pp180-209.

[23] Tsichritzis, D.C. & Lochovsky, F.H.,*Data Models*, Prentice-Hall 1982

[24] Ullman, J.D.,*Principles of Database Systems*, Computer Science Press 1982

[25] Weihl, W.E., *Specification and Implementation of Atomic Data Types*, Ph.D dissertation MIT & MIT/LCS/TR-314 , MIT, Cambridge MA. 1984

[26] Weihl, W.E., Linguistic Support for Atomic Data Types, in *Proceedings Appin Workshop on Persistence and Data Types*, University of Glasgow, Aug 1985 pp161-189

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Gray, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, NO. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 (March 1987)

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Atkinson, M.P, Morrison, R. & Dearle, A.
"A strongly typed persistent object store", 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, California (September 1986).

Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA : A persistent information space architecture", ICL Technical Journal 5, 3 (May 1987),477-491.

Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Cooper, R. & Atkinson, M.P.
"Requirements Modelling in a Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987

Wai, F.
"Distribution and Persistence". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987

Guy, M.R.
"Persistent Store - Successor to Virtual Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Dearle, A.
"Constructing Compilers in a Persistent Environment". Presented at the 2nd Internaional Workshop on Persistent Object Stores, Appin, August 1987.

Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Brown, A.L.
"A Distributed Stable Store". Presented at the 2nd International Workshop on Persistent object Stores, Appin, August 1987

Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment". Proceedings of the Thirteenth Internaional Conference on Very Large Databases, Brighton, Septermber 1987.

Atkinson, M.P. & Morrison, M.
"Polymorphic Names and Iterations", presented at the Workshop on Database Programming Languages, Roscoff, September 1987

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 9th July 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-87	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - 2nd edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00

PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00
PPRR-44-87	Persistent Object Systems: their design, implementation and use. (proceedings of the Appin workshop August 1987) ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£20.00
PPRR-45-87	Delayed Binding and Type Checking in Database Programming Languages - Atkinson, M.P., Buneman, O.P. & Morrison, R.	£1.00
PPRR-46-87	Transactions and Concurrency - Krablin, G.L.	£1.00