A RECOVERY SCHEME FOR DATABASE SYSTEMS WITH LARGE MAIN MEMORY

Sang Hyuk Son

Computer Science Report No. TR-86-26
October 17, 1986

Abstract

The availability of large, relatively inexpensive main memories would enable the advent of future database systems that have all the data in main memory. The challenge in the design of such systems is to provide fast transaction processing by different access methods and query optimization strategies, and to provide more efficient concurrency control and a rapid restart after a failure. In this paper a recovery scheme for main memory database systems is presented. It achieves non-interference and a fast restart through incremental checkpointing and log compression. Other issues in main memory database systems are addressed, and alternatives in restart logic are discussed.

Index Terms - database systems, main memory, recovery, consistency, checkpoint, transaction, non-interference, log

1. Introduction

The availability of large, relatively inexpensive main memories coupled with the demand for faster response time has brought a new perspective to database system designers: main memory databases. The price of main memory has been sharply decreased throughout the past decade, and is expected to continue to do so. By 1990, 1 megabit chips are expected to be commonplace, and it should further reduce the main memory price by another order of magnitude: a gigabyte of memory should cost less than $200,000, and if 4 megabit chips are available, the price might be as low as $50,000 [DeW84].

It is well-known that many database operations are I/O bound, i.e., limited by the speed at which data can be transferred from disks [Gar84]. Clearly, substantial performance gains can be achieved if a large portion of, or the entire database, could reside in main memory, eliminating the I/O bottleneck. High transaction processing rates can be obtained since I/O delays are significantly reduced. For example, if the entire database is resident in main memory, a transaction would never need to access data pages on disk. However, keeping a large portion of the database in main memory brings some critical problems to the recovery mechanism of the system. Since the least expensive form of main memory is volatile, any loss of electric power destroys all stored data. Constructing a recoverable main memory database system using volatile memory is a challenge to the system designer.

Although it is difficult to recover a main memory database system in a reasonable amount of time, there are applications for which the cost and the complexity of the recovery mechanism can be justified. For instance, meeting tight bounds on response time in real-time applications may not be able to tolerate delays caused by access to disk storage. Another example where main memory databases may be viable is an office workstation, in which memory is relatively cheap and the database system must respond very quickly [Amm85]. It has been argued that when the ratio of main memory to disk capacity is higher and disks are slower, the advantage of storing the database on disk diminishes [Bit86].

There has been not much literature written on recovery in main memory database systems. Garcia-Molina et al. [Gar84] discussed a massive memory machine, by presenting a novel architecture for such a machine and how it can lead to reduced memory times with higher reliability. However, they did not discuss on database recovery. DeWitt et at. [DeW84] addressed implementation techniques for main memory database systems, but the emphasis was on access methods not on recovery. Recovery methods for main memory database systems appeared in the literature are the "overlapped" checkpointing mechanism proposed in [DeW84] and the "fuzzy dump" checkpointing mechanism in [Hag86]. However, they either require the database quiesce [DeW84], or construct an inconsistent state of the database [Hag86]. Verhofstad [Ver78] surveyed recovery techniques for general database systems, and Haerder and Reuter [Hae83] has a survey of transaction-oriented recovery. Bitton [Bit86] discussed several issues of main memory database systems.

In this paper we address the issues of main memory for the primary copy of the database, and present a recovery scheme that uses the techniques of non-interfering checkpointing and logging. A checkpoint provides a starting point from which recovery can begin. One important requirement in constructing a checkpoint is that the interference of the checkpointing procedure with the transaction processing should be kept as small as possible. Consistency of the constructed checkpoint is also important since no undo operation would be necessary during recovery, reducing the recovery time. On-line log compression is needed to keep the log short to achieve a rapid restart. Compression can be used by any database system to improve restart time, but is essential for main memory database systems which may achieve very high transaction throughput.

The rest of the paper is organized as follows. Section 2 introduces a model of computation used in this paper. Section 3 addresses general issues in the design of main memory database systems. Section 4 presents several important notions used in the recovery of main memory database systems, and the overview of the recovery method. Section 5 describes the details of the recovery method, and Section 6 concludes the paper.

2. A Model of Computation

This section introduces the model of computation used in this paper. We describe the notion of transactions and the assumptions about the effects of failures.

2.1. Data Objects and Transactions

A *database* consists of a set of data objects. Each data object has a *value* and represents the smallest unit of the database accessible to the user. Data objects are an abstraction; in a particular system, they may be files, pages, records, items, etc. All user requests for access to the database are handled by the *database system*. A database is said to be *consistent* if the values of data objects satisfy a set of assertions. The assertions that characterize the consistent states of the database are called the *consistency constraints* [Esw76].

The basic units of user activity in database systems are *transactions*. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction consists of different types of operations such as read, write, and local computations. The *read-set* of a transaction T is defined as the set of data objects that T reads. Similarly, the set of data objects that T writes is called the *write-set* of T.

A data object may have associated with it a lock. To read or write a data object, a proper lock must be set on the data object. A transaction can lock a data object in two modes: *exclusive* (write-lock) and *shared* (read-lock). When a data object is locked in exclusive mode by a transaction, then no other transaction can concurrently lock it. If a transaction has locked a data object in shared mode, other transactions can concurrently lock it in shared mode. This basic locking principle ensures that any data object already locked would be inaccessible to other transactions while in an inconsistent state.

A transaction is the unit of consistency and hence, it must be *atomic*. By atomic, we mean that intermediate states of the database must not be visible outside the transaction, and every updates of a transaction must be executed in an all-or-nothing fashion. A transaction is said to be *committed* when it is executed to completion, and it is said to be *aborted* when it is not executed at all. When a transaction is committed, the output values are finalized and made available to all subsequent transactions. We assume that the database system runs a correct transaction control mechanism, and hence assures the atomicity and the serializability of transactions.

Each transaction has a time-stamp associated with it [Lam78]. A time-stamp is a number that is assigned to a transaction when initiated and is kept by the transaction. Two important properties of time-stamps are (1) no two transactions have the same time-stamp, and (2) only a finite number of transactions can have a time-stamp less than that of a given transaction.

2.2. Failure Assumptions

A database system can fail in many different ways, and it is almost impossible to make an algorithm which can tolerate all possible failures. In general, failures in database systems can be classified as failures of *omission* or *commission* depending on whether some action required by the system specification was not taken or some action not specified was taken[Moh83]. The simplest failures of omission are *simple crashes* in which the system simply stops running when it fails. The hardest failures are *malicious runs* in which the system continues to run, but performs incorrect actions. Most real failures lie between these two extremes.

In this paper, we do not consider failures of commission such as the "malicious runs" type of failure. When the system fails, it simply stops running (fail-stop). When the failed system recovers, the fact that it has failed is recognized, and a recovery procedure is initiated.

A failure which causes a loss of some portion or all of the secondary storage medium is called the *media failure* [Dat83]. Although magnetic storage devices are usually very reliable, this type of failure may not be rare in large database systems with many secondary storage devices. There are several causes for such a failure, the most common of which are (1) head crash, (2) hardware errors in the disk controller, and (3) bugs in the routines for writing the disk. Media failure makes recovery in main memory database systems more complicated since recovery information such as log and checkpoints are saved in disk. Our recovery scheme can cope with media failures as long as database checkpoints are safe. We discuss it further in Section 5.

3. Issues in Main Memory Database Systems

In addition to the recovery problem, there are many important issues in the design of main memory database systems. They include access methods, memory management, and transaction management. In this section, we briefly discuss each of them.

Access Methods

The question raised here is whether or not the access methods and query processing algorithms that are known to be efficient for conventional disk-oriented database systems have the same advantages in main memory database systems. A number of studies investigate this question, using analytical models and simulations to evaluate access methods and join algorithms [DeW84, Leh85, Sha85]. It has been shown that a prevalent access methods used in disk-oriented database systems such as B-tree and B^+-tree may not be advantageous for main memory database systems, because space efficiency is critical in those systems. If most of the database fits in main memory, an AVL-tree may be a better choice [DeW84]. Also, fast query processing algorithms that create large intermediate results, such as a sort-merge join, are not appropriate for main memory database systems. Exact estimate of the size of a temporary relation are critical here, since errors in the estimation may result in substantial performance degradation or even failure of the system, due to limitations in the size of the address space [Bit86].

Memory Management

In the design of conventional database systems, memory management is designed to reduce the number of disk accesses. The idea is that data records may be prefetched, or commonly accessed data such as indices are stored permanently, in a large buffer pool or cache, and hence they are available with smaller delays. This task is mostly performed by the operating system of the host. The question regarding the memory management in main memory database systems is whether to rely on the host operating system completely or not. The database system may need an interface to the operating system that allows it to give hints on the usage of data to the operating system, and allows the operating system to optimize the overall performance [Gaw86].

Transaction Management

Concurrency control mechanisms in database systems have been studied in depth during the last decade [Ber81], and our understanding of serializability and other criteria for the correctness of transaction processing remains applicable in the context of main memory database systems. However, the fact that the primary copy of the database resides in main memory may require new algorithms for concurrency control and transaction commitment. For example, the concurrency control in the conventional meaning may no longer be needed in main memory database systems [Gar84]. One of the main reasons for executing several transactions concurrently is that transactions encounter long delays as they wait for disk pages to be brought into main memory. Since the transactions are not independent, their actions cannot be interleaved in arbitrary ways. The concurrency control mechanism ensures that only interleavings that preserve database consistency are run. In achieving this, concurrency control introduces substantial overhead and complexity into the database system. If disk delays do not occur in main memory database systems (i.e., the data required by each transaction are already in memory), and if transactions are short, they can simply be scheduled sequentially, eliminating the overhead and complexity of concurrency control altogether.

4. Basic Idea of Recovery

The recovery management in database systems essentially consists of two parts: the preparation for the recovery by saving necessary information during normal operation of the system, and actual recovery in order to reconstruct a consistent state. The preparation for recovery is performed through checkpointing and logging.

A checkpoint provides a starting point for recovery using the transaction log. A simple recovery scheme would proceed by first reloading the checkpoint from the disk, and then applying the transaction log to bring it up-to-date as follows (for details, see [Gra79]). First, the time t in the log older than the oldest transaction active at the time of the crash is determined. Second, process the log forward starting at time t until the end of the log and build a table of the status of all transactions started after t. Third, take all transactions that were active at the end of the log and make them aborted. Then reprocess the log to the time t while undoing all updates of aborted transactions. Finally, process the log forward, starting from t, while performing any necessary redo operations for committed transactions.

This kind of normal recovery technique does not perform well enough in main memory database systems. If updates of transactions are recorded both in the database and the log as in a normal database system, the time to recover from a crash would be unacceptably high. An optimistic estimate of recovery time in a "small" database system that performs 100 transactions per second, with each transaction writing two log pages, would be one hour [Hag86]. Therefore, it is clear that some optimization and new techniques are necessary to achieve a fast recovery in main memory database systems.

Our recovery scheme is based on the techniques of non-interfering incremental checkpointing and log compression using non-volatile areas of memory. They are combined to increase the transaction throughput by reducing the number of log write operation and the interference of checkpointing with transaction. The scheme also provides a rapid restart of the database system from failures. The idea of log compression is based on previous work in [DeW84, Hag86].

An approach for reducing recovery time is to periodically checkpoint the database to stable storage [Gra79]. In case of a failure, the saved checkpoint can be used to restore the database, and hence limiting recovery activities to those transactions that have begun since the last checkpoint. System R [Gra81] takes an action-consistent checkpoint, during which no storage system operation (which corresponds to read or write operation of transactions) is allowed in progress. Dirty pages are forced to disk. If the database is large, a large number of dirty pages will need to be written to disk, making the database unavailable for an intolerably long amount of time [DeW84].

The checkpointing of our scheme is performed concurrently with transaction activity while constructing a consistent database state on disk. This is particularly important in an environment where many update transactions generate a large number of updated pages between checkpoints, or in a real-time application where high availability of the database is desirable. Our checkpointing mechanism does not require the database quiesce to save a consistent state as in [DeW84], and constructs a transaction-consistent checkpoint.

In order to construct transaction-consistent checkpoints, the updates of a transaction must be either included in the checkpoint completely, or they must not be included at all. To achieve this, transactions are divided into two groups according to their relationships to the current checkpoint: *included-transactions* (INT) and *excluded-transactions* (EXT). The updates belonging to INT are included in the current checkpoint while those belonging to EXT are not included. Time-stamps are used to determine membership in INT and EXT. A transaction-consistent state of the database is always maintained on disk. At a checkpoint time, only the portion of the saved state which has been changed since the last checkpoint is replaced. When a checkpoint completes, next one begins with only negligible interruption of service to delete old versions that are checkpointed.

When the log is written, some compression may occur. We assume that a portion of memory can be made non-volatile by using batteries as a backup power supply for memory chips requiring low power. The non-volatile memory is used to hold an in-memory log so that the log writes to disk can be delayed. In-memory log can be considered as a reliable disk output queue for log data. When the log data must be written to disk since in-memory log buffer is full, many transactions that have entries in the log may now be completed. For aborted transactions, the whole entries in the log may be eliminated; for committed transactions, the undo part of log records may be eliminated.

In addition, the log entries can be stored in disk in a more compact form. In the conventional approach, log entries for all transactions are intermixed in the log. A more efficient alternative is to maintain the log on a per transaction basis, and it is possible by using the non-volatile memory. Non-volatile memory also assists in reducing the recovery time by maintaining the information such as the log entry of the oldest update that refers to an uncheckpointed page, to help to determine the point in the log from which recovery should begin.

5. The Recovery Scheme

Our recovery scheme consists of two algorithms: a checkpointing mechanism and a restart logic. We present each of them separately in this section. To present our non-interfering incremental checkpointing mechanism, we need to first describe an event closely related to recovery in database systems: commitment of transactions. Time and ordering of transaction commitment is important in database recovery, because the goal of recovery in database systems is to reconstruct a state that is transaction-consistent.

5.1. Commitment

In conventional logging schemes, a transaction cannot commit until its log commit record has been written to stable storage [Gra79]. In main memory database systems, transactions no longer need data objects to read from or write to disk. However, they still need to perform at least one disk I/O for its commit record, making the time to write the log the major bottleneck of the transaction throughput.

Precommitment and group commitment of transactions can be used to reduce the number of disk I/O for writing commit records, resulting in the increased performance [DeW84]. When a transaction is ready to commit, the transaction manager places its commit record in the log buffer. The transaction releases all locks without waiting for the commit record to be written to disk. However, the transaction is delayed from actually committing until its commit record actually appears on disk, and the user is not notified about the commitment of the transaction until this event has occurred. The transactions with commit records on the same log page are committed as a group. They are called the *commit group*. A single log I/O is all that is required to commit all transactions within the group. The size of a commit group depends on how many transactions can fit their logs within a log buffer page.

A transaction releases its locks at the precommitment, allowing other transactions to access data objects in its read-set and write-set. In such a situation, it is possible that a cycle among dependency relations of committed transactions is developed, resulting in an inconsistent state of the database. To prevent the inconsistency, the commit ordering between dependent transactions must be preserved. It can be achieved as the following. The concurrency control mechanism of the system maintains the dependency list in the transaction's descriptor of each active transactions. When a transaction commits, it is removed from the dependency lists of all the transactions that depend on it. After the precommit, a transaction joins a commit group. The dependency list of the newly added transaction of a commit group is added to those on which the commit group depends. A commit group cannot be written to disk until all the groups it depends on have been committed.

5.2. Checkpointing

A checkpointing begins by setting the checkpoint number (CN) as the current clock value. Transactions with the time-stamps smaller than the CN are the members of included-transactions (INT); others are the members of excluded-transactions (EXT). The transaction activity continues with the checkpointing process. When an EXT updates data objects, new in-memory versions are created, leaving the old versions to be written to disk.

The checkpointing process waits until there is no active INT in the system. It then begins to write the pages updated by INT to disk, constructing a portion of the state to be replaced in the checkpoint. Let δ_1 be the set of pages that have been updated by INT since the last checkpoint. First, the members of δ_1 must be identified. This might be done with hardware assist by setting a dirty bit when the page is updated by INT. Then δ_1 must be written on disk without the loss of previous checkpoint in the case of a system failure. To achieve this, a variation of stable storage technique [Lam79] can be used in two steps. During the first step, δ_1 is written to a new disk file δ_2. When it is successfully completed, then δ_1 is written into the desired location in disk. If the system crashes while the checkpoint state is being replaced by δ_1, a consistent checkpointed state can be reconstructed from δ_2.

After δ_2 has been successfully created, a "begin checkpoint" record is written to the log, indicating the completion of the first step of checkpointing procedure. When δ_1 is written to their original locations on disk, making the checkpoint state on disk identical to the database state in memory as if all transaction activities are completed and no more transaction has been started after CN, an "end checkpoint" record is written to the log. It indicates that the checkpoint in disk is a complete transaction-consistent state of the database, and it includes all the updates of transactions active at time CN. The address of the checkpoint record and the value of CN are saved in non-volatile memory so that they can be used in locating the most recent checkpoint record in the log during recovery. When a checkpointing procedure completes, pages in δ_1 are overwritten by the new versions, and their dirty bits are set. They are the members of δ_1 for the next checkpoint. Then the next checkpoint begins with the new CN.

The checkpointing mechanism presented here is different form that of [DeW84] in two important ways. First, it constructs a transaction-consistent state of the database instead of action-consistent state. Second, it does not require the database quiesce. It is also different from that of [Hag86], which constructs an inconsistent state of the database.

5.3. Restart Logic

Now we present an outline of the restart logic used in our recovery scheme. With a consistent checkpoint, the database system has two alternatives in restart: a *fast restart* and a *complete restart*. A fast restart is a simple restoration of the latest checkpoint. Since each checkpoint is consistent, the restored state of the database is assured to be consistent. However, all the transactions committed during the time interval from the latest checkpoint until the time of crash would be lost. A complete restart is performed to restore as many transactions that can be redone as possible. The trade-offs between the two restart methods are the recovery time and the number of transactions saved by the recovery.

A rapid restart from failures is critical for some applications of database systems which require high availability (e.g., ballistic missile defense or air traffic control). For those applications, the fate of the mission, or even the lives of human beings, may depend on the correct values of the data and the accessibility to it. Availability of a consistent state is of primary concern for them, not the most up-to-date consistent state. If a simple restoration of the latest checkpoint could bring the database to a consistent state, it may not be worthwhile to spend time in restart by executing a complete restart to save some of the transactions. Furthermore, a fast restart is the only choice if the log is damaged during the failure.

For the applications in which each committed transaction is so important that the most up-to-date consistent state of the database is highly desirable, or if the checkpoint intervals are large such that a lot of transactions may be lost by the fast restart, a complete restart is appropriate to use. The cost of a complete restart is the increased restart time which reduces the availability of the database. Searching through the transaction log is necessary for a complete restart. Complete restart mechanisms based on the special time-stamp of checkpoints (e.g., CN) in distributed environments have been studied in [Kus82, Son86].

A complete restart in our recovery scheme works as the following. At restart from a failure, parameters for locating the compressed log and the latest checkpoint are read from non-volatile memory. The checkpoint saved on disk is read into main memory, hence reconstructs a transaction-consistent state of the database. The next step is to determine the transactions that have started after the checkpoint and have committed before the failure. Other transactions do not need to be processed since they must be aborted and their updates are not included in the checkpoint. It may be possible to perform this task "on the fly", as the log data are read. Finally, perform the updates of the committed transactions to the main memory database. Since we already know which transaction must be redone, this task can be performed very fast. The only thing left now is to establish an internal state of the system, and restart the system.

Hagmann [Hag86] claimed that the restart procedure using "fuzzy dump" for a 1-Gbyte main memory database can be completed within 5 minutes. Since the restart logic presented above performs similar steps as in [Hag86], it would also take 5 minutes to recover for a main memory database system of the same size. However, the recovery in [Hag86] would take much longer than 5 minutes if a failure occurs before the purification of the log by eliminating the updates of "thoroughly" committed and "thoroughly" aborted transactions, while in our recovery scheme, the time required remains the same. It is because our checkpointing mechanism constructs a consistent state on disk from the beginning, whereas the scheme in [Hag86] constructs an inconsistent state first, then use the log purification to reduce the recovery time.

6. Concluding Remarks

In this paper, we have presented a recovery scheme for main memory database systems. The scheme is based on techniques of non-interfering incremental checkpointing and log compression. This recovery scheme is very stable under load because the time for recovery is dominated by the database size, not the update transaction rate.

Since checkpointing is executed during normal operation of the database system, the interference with transaction processing must be kept to a minimum. To reduce the interference, inconsistent checkpoints have been considered [Hag86, Gra79]. However, to achieve a rapid restart which is highly desirable in main memory database systems, we need a transaction-consistent checkpoint.

Other benefit of having a consistent checkpoint is the fault-tolerance of the recovery scheme. Even if the log is partially destroyed during a failure (e.g., media failures), a consistent database state can be reconstructed by a fast restart. Furthermore, it does not require long recovery time to achieve consistency. Using duplicated or triplicated log may reduce the probability of of the loss of it. However, it cannot achieve zero probability of the log loss. The fault-tolerance feature achieved by consistent checkpoint is desirable in developing a highly reliable database system because of the following two reasons: first, Murphy's law ensures that all the replicated logs will sometimes fail together regardless of the number of replication [Gra81], and second, replication would increase processing and storage overhead during normal operation of the database system.

REFERENCES

[Amm85]Ammann, A., Hanrahan, M., and Krishnamurthy, R., Design of a Memory Resident DBMS, *Proc. IEEE COMPCON*, 1985.

[Ber81] Bernstein, P., Goodman N., Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, June 1981, pp 185-222.

[Bit86] Bitton, D., The Effect of Large Main Memory on Database Systems, *Proc. ACM SIGMOD 1986*, pp 337-339.

[Dat83] Date, C. J., *An Introduction to Database Systems*, vol.2, Addison Wesley Publishing Co., 1983

[DeW84] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D., Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD 1984*, pp 1-8.

[Esw76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., The Notion of Consistency and Predicate Locks in a Database System, *Commun. ACM* Nov. 1976, pp 624-633.

[Hag86] Hagmann, R., A Crash Recovery Scheme for a Memeory-Resident Database System, *IEEE Trans. on Computer Systems*, Sept. 1986, pp 839-843.

[Gar84] Garcia-Molina, H., Lipton, R., and Valdes, J., A Massive Memory Machine, *IEEE Trans. on Computer Systems*, May 1984, pp 391-399.

[Gaw86] Gawlic, D., Panel Discussion on Memory Management in Main Memory Database Systems, *ACM SIGMOD 1986*.

[Gra79] Gray, J. N., Notes on Database Operating Systems, *Operating Systems: An Advance Course*, Springer-Verlag, N.Y., 1979, pp 393-481.

[Gra81] Gray, J. et al., The Recovery Manager of the System R database Manager, *ACM Computing Surveys*, June 1981, pp 223-242.

[Hae83] Haerder, T. and Reuter, A., Principles of Transaction-Oriented Database Recovery, *ACM Computing Surveys*, Dec. 1983, pp 287-318.

[Kus82] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. *ACM SIGMOD 1982*, pp 293-302.

[Lam78] Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, *Commun. ACM*, July 1978, pp 558-565.

[Lam79] Lampson, B. and Sturgis, H., Crash Recovery in a Distributed Data Storage System, Xerox Palo Alto Research Center, November 1979.

[Leh85] Lehman, T. and Carey, M., A Study of Index Structures for Main Memory Database Systems, *Tech. Rep. TR-605, University of Wisconsin*, July 1985.

[Moh83] Mohan, C., Strong, R., and Finkelstein, S., Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors, *Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, August 1983.

[Sha85] Shapiro, L., Join Processing in Database Systems with Large Memories, *Tech. Rep. North Dakota State University*, Dec. 1985.

[Son86] Son, S. H. and Agrawala, A. K., An Algorithm for Database Reconstruction in Distributed Environments, *Proc. 6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, pp 532-539.

[Ver78] Verhofstad, J., Recovery Techniques for Database Systems, *ACM Computing Surveys*, June 1978, pp 167-195.