Impact of Timing Constraints on Real-Time Database Recovery

Jing Huang
IBM Corporation
5 West Kirkwood, Room 3530
Roanoke, TX 76299
jingh@vnet.ibm.com

Le Gruenwald
School of Computer Science
The University of Oklahoma
Norman, OK 73019
gruenwal@cs.ou.edu

ABSTRACT In this paper, we discuss how timing constraints affect three major database recovery activities: logging, checkpointing and database reloading. We then present several schemes to handle these activities in a real-time main memory database system and their performance results obtained via simulation. Further improvement of these schemes is also provided.

1. INTRODUCTION

Time is an important aspect of real-time database systems (RTDBS). In such an environment, not only transactions are associated with timing constraints but also some *temporal data* may lose their validity after a certain time interval. In order to specify absolute temporal consistency requirements, two attributes, timestamp and absolute validity interval, are usually associated with a temporal data item. *Timestamp* indicates when the current value of the data item was obtained and *absolute validity interval* is the length of the time interval during which the current of the data item is considered to be valid. The value of a temporal data item x is said to be valid or meet the absolute temporal consistency requirement at the current time t_{now} if $tnow - tx \leq ax$ holds, where t_x is the timestamp of x, and a_x is the absolute validity interval of x [18].

There is a number of applications such as air traffic control, stock-trading, and telecommunications where RTDBS might be applicable. One aspect of these examples is that they all involve gathering data from the environment, processing of gathered information in the context of information acquired in the past, and providing timely response. Another aspect of these examples is that they may process both temporal data as well as persistent data which remain valid regardless of time [15]. The main goal of an RTDBS is to meet the timing constraints of transactions and data as much as possible [19]. In order to achieve a higher system performance, the use of a main memory database (MMDB) is a good choice. This is because in an MMDB environment, all or a major portion of the database can be memory-resident, thus transaction processing can be satisfied with few I/Os [1]. An MMDB system therefore has a potential for obtaining a substantial performance improvement over a disk-resident database system for real-time applications.

This research is funded in part by the National Science Foundation under grant NSF-9201596

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee
DART '96 Rockville MD, USA
Copyright 1997 ACM 0-89791-948-3/96/11 ..$3.50

However, due to the volatility of semiconductor memory, the contents of main memory may be lost at the time of a system failure which may occur due to an error in the database management system code, an operating system fault, a hardware failure or a power outage. As the database is not available when a system failure occurs, for a high performance system, such as those processing 1000 transactions or more per second [3], many transactions might be backlogged during system recovery. On average, a system failure occurs once every few weeks [4]. This may cause many transactions to miss their deadlines and a large amount of temporal data to lose their validity before they can be used. The degradation of the system performance may result in a catastrophe for safety-critical activities, such as those that respond to life or environment-threatening emergency situations [19]. Besides system failure, a transaction may fail to reach its successful end because of a deadlock, violations of consistency, time-out, user abort, and so on [4]. In order to guarantee the recoverability of a real-time MMDB system, logging and checkpointing activities must be performed during normal operations and reloading activities must be invoked at the time of a system crash.

A great number of recovery techniques have been studied for conventional database systems ([5], [6], [9], [11], [13]); however, not much work has been done for real-time database systems [2]. Sivasankaran et al [17] has recently studied some issues of real-time database recovery, but no detailed schemes were given in their work. In this paper, we discuss several techniques to handle real-time MMDB recovery activities which include logging, checkpointing and reloading. Key factors which affect the performance of these techniques are also studied via simulation. Finally, suggestions for further improvement of the proposed schemes are discussed.

2. LOGGING

2.1. Proposed Schemes

Logging notes on a stable storage called a log all updates made to the database [4]. When a system or transaction failure occurs, in order to construct the consistency of the database, all the effects made by successful transactions must be reflected in the database which is accomplished through REDO operations, and all the modifications done by incomplete transactions must be removed from the database by using UNDO operations. A REDO operation is performed by applying After Images (AFIMs), which are the values of data items after they are modified, of all committed transactions to the database, while an UNDO operation is accomplished by copying to the database BeFore Images (BFIMs), which are the values of data items before they are modified, of all unfinished and aborted transactions.

Unlike conventional logging, real-time logging must not only ensure that data values are recovered correctly but also guarantee that timing constraints of data items are reflected

properly. As we mentioned before, to reflect the temporal consistency requirements, each temporal data item is associated with two attributes: time stamp and absolute valid interval. This means that for each temporal data item not only its data value needs to be maintained in a log record as in conventional logging for persistent data, but also its time stamp and absolute valid interval need to be kept in the log record. One simple way to achieve this goal is to adopt the conventional logging approach. For each persistent data item, the corresponding log record maintains its AFIM and/or BFIM depending on the update policy selected; for each temporal data item, besides AFIM/BFIM, ATIME/BTIME and INTERVAL are also kept in the log record. BTIME and ATIME are time stamps before and after a temporal data item is modified, respectively, and INTERVAL represents the absolute valid interval. Only one log buffer is used in the system, which means that both persistent data log records and temporal data log records are kept in the same log buffer. It is obvious that this approach is simple and easy to implement; however, as persistent data and temporal data have the different log record formats, extra overhead will be incurred during recovery time in order to check the log record format. To eliminate this overhead, multiple log buffers could be used which store persistent data and temporal data log records separately.

Another point needs to be noted is that some temporal data may have very short temporal valid intervals and thus may become invalid at the time when they are updated. It is not necessary to keep their old values in log records, which consequently leads to a smaller log record size and less amount of log information that needs to be processed at recovery time. Based on whether multiple log buffers are used and whether the values of invalid temporal data will be maintained in log records, we have four alternatives, as summarized in Table 1, to do real-time MMDB logging. The detailed description of the four logging alternatives can be found in ([7], [8]).

	Alternative 1	Alternative 2	Alternative 3	Alternative 4
Using Multiple Log Buffers	No	No	Yes	Yes
Logging Invalid Temporal Data	Yes	No	Yes	No

Table 1. Four Real-Time MMDB Logging Alternatives

In order to evaluate the performance of the above alternatives, we have built a simulation model using the simulation language SLAM II [14]. Based on the results obtained, we found that the *key factors* that affect the performance of real-time MMDB logging are multiple log buffers and validity of temporal data. Specifically, we observed two important points. First, the use of multiple buffers eliminates the need for a log format indicator and thus successfully reduces the amount of information that needs to be maintained in a log record. This in turn lessens the logging overhead imposed on normal transaction processing and reduces the amount of work needed at recovery time. Second, even though ignoring logging invalid temporal data can reduce the amount of log information that needs to be processed for recovery, as the overhead involved in checking the validity of temporal data during normal processing and database recovery outweighs the time saved from reducing memory access, the overall performance is not improved.

We also observed that the amount of temporal data does affect the performance of logging techniques. The more temporal data the system has, the higher overhead will be incurred during the logging process, and the worse the system performance becomes. Under this situation, in order to reduce the interference of logging activities with normal operations, an efficient logging scheme is desired.

2.2. Further Improvement

One drawback of the above logging approaches is that all temporal data is treated the same regardless of the length of their valid intervals. Usually a real-time database system contains some temporal data whose valid intervals may be very short. For these kind of temporal data, as discussed in [17], their values are likely to become invalid after they are recovered from a system crash. It is thus not necessary to perform UNDO or REDO operations for these data items at database recovery time. When immediate update is used, which allows modified data to be propagated to the primary database at any time, to ensure that the effects of an aborted transaction can be removed completely during normal operation, the system needs to maintain only BFIMs for short temporal data's updates. No AFIMs need to be recorded. When deferred update is applied, in which modified data by a transaction cannot be applied to the database until a successful completion of the transaction is assured, neither BFIMs nor AFIMs are required to be kept for short temporal data. In this way, both the overhead incurred in the logging process as well as the time required to perform post-crash log processing are reduced.

Another possible drawback of the existing logging techniques is that only one update policy (immediate or deferred) is employed at a time. Based on our studies, when transaction abort rate is low, the immediate update policy offers a better performance in terms of transactions and data meeting timing constraints than the deferred update policy does. This is because immediate update enables transactions to be processed faster than deferred update does. However, if transaction abort rate is high, due to a high abort overhead incurred in immediate update which may result in a poor system performance, the deferred update scheme is a good choice.

An RTDBS usually consists of two types of transactions [18]: periodic transaction and aperiodic transaction (or normal transaction). Periodic transactions are usually responsible for updating a sensor data item or reading several temporal data and deriving a temporal data item; they are likely to be short and have fewer chances to be aborted. Besides, as values of temporal data change frequently with time, it is desired that updates of temporal data can be reflected in the database as quickly as possible. Based on these considerations, the immediate update approach seems to be a good choice for periodic transactions.

Normal transactions is the transactions found in conventional database system which can read and write persistent data but can read only temporal data. For this type of transactions, they are usually longer than periodic transactions and thus have more chances to miss their deadlines. Depending on a transaction's characteristics, some transactions (for instance, firm deadline transactions), may be aborted if they cannot complete

within their deadlines. For this type of transactions, perhaps deferred update is a suitable selection. Therefore, a logging technique which applies both immediate update and deferred update at the same time may provide a better performance for a real-time MMDB system. This is an interesting topic for further research.

3. CHECKPOINTING

3.1. Proposed Schemes

Checkpointing is a process used to maintain on disks an up-to-date copy of the database. It is needed as it limits the time required to recover the database after a system crash [4]. When a system failure occurs, as checkpoints provide an almost up-to-date copy of the database, most data in the log are not needed at the time of recovery. The recovery process needs to process only the log information which is generated after the last complete checkpoint. It is obvious that an efficient checkpointing scheme enables the system to restart quickly from a failure.

We have studied two partition checkpoint techniques [8]. The basic idea of our approaches is that they no longer treat the entire database as a single object, instead, a database is segmented into smaller data partitions based on either pages' update frequencies or temporal valid intervals. Each partition contains only one type of data pages: temporal or persistent, and is checkpointed independently. With these checkpoint schemes, it is possible to checkpoint frequently updated data with short temporal valid intervals more often so that fewer log records need to be examined for these data at recovery time. Note that in order to facilitate the recovery process, each partition has its own log buffer. These techniques also enable transaction execution and database recovery to be processed in parallel after a crash so that the overall performance is enhanced. The features of the two checkpoint techniques are highlighted as follows.

In the first scheme, both persistent data and temporal data are partitioned based on their update frequencies. Partitions which contain most frequently updated pages (called hot partitions) will be checkpointed more often than those of few updated pages. Since the backup of hot partitions can be kept as recent as possible, at the time of database recovery, as there is less log information that needs to be examined, the recovery process is hastened.

In the second scheme, persistent data are partitioned based on update frequencies while temporal data are partitioned based on their valid intervals. Temporal data partitions with short valid intervals are given more opportunities to be flushed out by increasing their update frequencies than other partitions, this enables many temporal data to be recovered and used before losing their validity and in turn reduces the number of transactions aborted or delayed due to invalid data access. No logging and checkpointing will be invoked on temporal data whose valid intervals are shorter than a specified interval threshold; this not only reduces the interference of logging and checkpointing activities on normal operations but also hastens the recovery process.

The performance of the two schemes was evaluated via simulation. The results obtained showed that the key factors which affect the performance of real-time MMDB checkpointing are the number of partitions which the database has, whether or not the temporal data of short valid intervals is checkpointed and how the checkpoints among partitions are scheduled. Specifically, we have derived two important conclusions. First, the more partitions the database has, the more performance improvement the system tends to obtain. However, partition checkpoint does impose an overhead on normal system operations, and the more partitions the database has, the more overhead it incurs. When the number of partitions reaches a certain limit, the benefit obtained from further partitioning is not very significant. Second, not logging and checkpointing temporal data of short valid intervals does help to improve the overall performance. This is especially true when most of temporal data valid intervals are short and the system has a great amount of temporal data. As a temporal data item's log record size is larger than that of a persistent data item, it is desired that temporal data can be checkpointed more frequently than persistent data so that less log information needs to be processed for temporal data at recovery time.

The system environment also has impacts on the behavior of real-time checkpoint techniques. Our results indicated that the performance improvement offered by the proposed schemes is lessened when transaction arrival rate or system failure rate increases. This is because as transaction arrival rate or system failure rate gets higher, the number of transactions that are backlogged during a system crash increases tremendously. The savings obtained from reducing post-crash log processing time, therefore, becomes less significant to the overall performance compared to those obtained in a system of low system load and low failure rate.

3.2. Further Improvement

In the above checkpoint schemes, an MMDB is partitioned and checkpointed based on only the properties of real-time data, such as, temporal valid interval and data update frequency; transaction characteristics, such as criticalness and deadlines, are not considered. Criticalness indicates the level of importance attached to a real-time transaction relative to the other transactions. Depending on the functionality of a transaction, meeting the deadline of one transaction may be considered more critical than another [20]. For example, a transaction that reacts to an emergency situation, such as fire on the factory floor, probably will be more critical than the transaction that controls the movement of a robot under normal operating conditions. The goal of a real-time database system is to enable as many highly critical transactions as possible to meet their timing constraints. To achieve this goal, it is desired that data needed by highly critical transactions can be checkpointed as often as possible. This will reduce the amount of log information that needs to be processed at recovery time and allow the transactions to have more opportunities to meet their deadlines.

On the other hand, some critical transactions may not have short deadlines. Therefore, if high checkpoint priority is only given to data accessed by critical transactions during the checkpoint process, the overall performance may be degraded because too many updates are accumulated for data needed by transactions of short deadlines. To improve the overall performance, a data item should be assigned with a priority, called *data priority*, based on both criticalness and deadline of transactions that access the data object [20]. The objective of a checkpoint technique should be to keep the backup copy of data objects with high data priorities as recent as possible so that less log information needs to be examined for these data at recovery time and the overall system performance can be enhanced.

To fulfill the above objective, the idea of our proposed partition checkpoint schemes could be used. However, as the more partitions the database has, the more overhead these

schemes will incur, the number of partitions in which the database is divided should not be too high. One possible way to solve this problem is to combine partition checkpoint and segment checkpoint; the latter is studied in [12]. The main concept of the segment checkpoint scheme is that the database is divided into segments. Checkpoints among segments are performed in a round-robin fashion. Only one log buffer is used in the system. The advantage of this approach is that less overhead is incurred in normal operation than that in partition checkpoint. One possible drawback of the segment checkpoint is that as all the log records are kept in one log buffer, it is difficult to recover some urgent data first before other data at recovery time. In partition checkpoint, extra overhead will be incurred in order to schedule checkpoints among partitions and to ensure that log records distributed among multiple log buffers be used correctly.

To take advantage of both segment checkpoint and partition checkpoint, we may improve our proposed checkpoint schemes as follows. An MMDB is divided into five partitions. The first partition contains temporal data which have higher data priorities and need to be brought into MM before the system is brought up. Note that data referenced by transactions which have very short deadlines, for example, shorter than the time needed to reload two or three cylinders from AM to MM, can be considered to have higher data priorities. This is because of the fact that data priority is defined based on a transaction's deadline and criticalness, and during the reload process, user data cannot be brought into MM until system data is memory-resident. The second partition contains persistent data of higher data priorities which will also be reloaded into MM before the system is up. The third partition consists of temporal data which are not in partition 1 and whose valid intervals are longer than the interval threshold. The fourth partition consists of the rest of persistent data. As the third and fourth partitions are usually much larger than the first two partitions, in order to reduce the amount of log information that needs to be processed for these two partitions, they are further divided into segments. The last or the fifth partition contains temporal data whose valid intervals are shorter than the interval threshold, and no logging and checkpointing activities will be invoked on this partition. The goal of this checkpoint scheme should be to keep the backup copy of higher priority data as recent as possible so that these data can be recovered quickly from a system failure, and at the same time, reduce the amount of log information that needs to be processed for other data. How to schedule checkpoints among five partitions will have a crucial impact on the overall recovery performance, and thus needs to be investigated.

4. RELOADING

4.1. Proposed Schemes

Reloading activities are performed in case of a system crash. For a large database, simply transferring it from archive disks to main memory can be very time-consuming. As estimated in [11], 28.43 minutes are needed to recover one gigabyte database. In order to resume transaction processing quickly without degrading system performance, an efficient reload scheme is needed. The objective of real-time reloading should be not only to reduce system restart time, but also to minimize the number of timing constrains which are violated

We have proposed two reload algorithms for real-time database reloading [8]. The common features of our approaches include (1) the system is brought on-line before the entire database is reloaded into MM in order to reduce down time; (2) transaction execution priorities are taken into account during the reload process to give immediate attention to high priority transactions so that they have more opportunities to meet their deadlines and (3) data accessed frequently are reloaded before other data.

In the first algorithm, a partition is selected as the recovery unit. The system resumes its execution when a specified number of partitions is memory-resident and their corresponding log information is processed. This algorithm gives partitions that are most frequently accessed a higher reload priority than those which are seldom accessed so that the number of page faults can be reduced and transaction execution can be processed with fewer interruptions. This algorithm also allows pages needed by high execution priority transactions to be brought into MM as quickly as possible so that these transactions have more opportunities to meet their deadlines. Besides, this algorithm takes transaction deadlines, reload prioritization, reload preemption and transaction priority inheritance concept [16] into account during the reload process. One drawback of this approach is that as a partition is the recovery unit, transaction processing will be suspended for a long time when a page fault occurs, which may result in many missing deadline transactions.

In the second approach, a page is used as the recovery unit. Transaction execution can proceed when its requested page is brought into MM and the corresponding log information is processed. This algorithm also takes transaction execution priority, priority inheritance concept, reload priority and preemption into consideration during the reload process. However, unlike the first approach, this algorithm reloads most frequently accessed pages, instead of partitions, into MM before less frequently accessed pages. Besides, temporal data are given higher reload priority than persistent data so that many temporal data can be used before losing their validity. This may in turn reduce the number of transactions which are aborted or delayed due to invalid data access. Another advantage of this approach is that as transaction processing is not suspended as long as that in the first approach when page faults occur, this approach has a potential to further improve the system performance in terms of transactions and data meeting timing constraints.

Our performance analysis indicated the following results. (1) In order to achieve a good recovery performance, reducing system unavailability is more important than reducing database reload time. The conventional reload approach, which brings the system up after the entire database is memory-resident, can finish database reloading in the shortest amount of time. However, as it delays transaction processing longer than the proposed schemes, it finally results in the worst performance. (2) To reduce the amount of invalid data and the number of transactions aborted due to invalid data access, temporal data of short valid intervals should be reloaded into MM as quickly as possible. (3) Pages needed by high execution priority transactions should be given higher reload priority so that execution of these transactions will not be suspended too long. (4) Taking transaction's execution priority into account during database reloading can help to reduce the number of transactions missing deadlines. (5) The reload threshold which specifies when the system should be brought up must be selected carefully and should not be a small number. The loss in system performance when choosing a too large reload threshold is much less than the loss when choosing a too small reload threshold. (6) Recovery

unit also plays an important role in improving the overall system performance. In most cases, the smaller the recovery unit is, the less time a transaction's execution is suspended when a page fault occurs, and the better performance the system obtains.

4.2. Further Improvement

One drawback of the above schemes is that the overall performance is degraded seriously when the system has a great amount of temporal data. The reason for this is that when the amount of temporal data continues increasing, most data that can be brought into MM before the system is brought up is temporal data. As few persistent data, which are usually accessed frequently by normal transactions, can be memory-resident before transaction processing is resumed, many page faults will occur during normal system operation. As the overhead incurred in handling page faults outweighs the gain obtained from reducing system unavailability, the overall performance will decrease. This indicates that reloading some temporal data, especially those of short valid intervals, into MM before bringing the system up helps to reduce the amount of invalid data and the number of transactions aborted due to invalid data access. However, enabling some persistent data, especially those of high access frequencies, to be memory-resident before resuming transaction execution is also desired. Therefore, to further improve the efficiency of database reloading, a reload scheme, which decides reload priority based on not only data access frequency and data type (temporal or persistent) but also the amount of data of each type that can be memory-resident before bringing the system up, should be developed.

5. CONCLUSIONS

In this paper, we discussed issues related to real-time MMDB recovery. Several alternatives of logging, checkpointing and reloading were proposed and evaluated. Based on our simulation results, the key factors that affect the performance of real-time logging techniques are multiple log buffers and the validity of temporal data. Besides, transaction abort rate and percent of temporal data are also the control factors in the selection of logging schemes. The main factors that affect the behavior of checkpointing schemes are number of partitions the database has, whether temporal data of short valid intervals is checkpointed and how checkpoints among partitions are scheduled. In order to obtain a higher performance during the database reloading process, system unavailability, transaction priority, validity of temporal data, recovery unit as well as reload threshold must be taken into account. Finally, we discussed how to further improve the performance of the proposed recovery schemes.

REFERENCE

[1] H. Garcia-Molina, K. Salem, "Main Memory Database Systems: An Overview", IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6 Dec. 1992.

[2] M. H. Graham, "Issues in Real-Time Data Management", In The Journal of Real-Time Systems, 4, 185-202 (1992), 1992 Kluwer Academic Publishers, pp. 185-202.

[3] J. Gray, etc., "The 5 Minute Rule for Trading Memory for Disk Accesses And the 10 Byte Rule for Trading Memory for CPU Time", Proceedings of 1987 ACM SIGMOD Conference, San Francisco, CA, May 1987, pp. 395-398.

[4] J. Gray, A. Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publishers, Inc. 1993.

[5] L. Gruenwald, M. H. Eich, "MMDB Reload Algorithm", Proceedings of ACM SIGMOD International Conference on Management of Data, May 1991, pp. 397-405.

[6] R. B. Hagmann, "A Crash Recovery Scheme for a Memory-Resident Database System", IEEE Transactions on Computers, Vol. C-35, No. 9, Sept. 1986.

[7] J. Huang, L. Gruenwald, "Logging Real-Time Main Memory Databases", Proceedings of International Computer Symposium, December 1994, pp. 1291-1296.

[8] J. Huang, "Recovery Techniques in Real-Time Main Memory Databases", Ph.D. Dissertation, School of Computer Science, University of Oklahoma, December 1995.

[9] H. V. Jagadish, A. Silberschatz, S. Sudarshan, "Recovering From Main Memory Lapse", In The Proceedings of the 19th Very Large Database Conference, 1993, pp. 391-404.

[10] E. Levy, A. Silberschatz, "Incremental Recovery in Main Memory Database Systems", In IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6 Dec. 1992, pp. 529-540.

[11] X. Li, M. H. Eich, "Partition Checkpointing in Main Memory Database", Technical Report 93-CSE-23, Department of Computer Science and Engineering, South Methodist University, 1993.

[12] J. L. Lin, M. H. Eich, "Speedup Recovery From Fuzzy Checkpoints", Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas, Technical Report, July 1995.

[13] C. Mohan, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Roll Backs Using Write-Ahead Logging", In ACM Transactions on Database System, Vol. 17, No. 1, March 1992, pp. 94-162.

[14] A. Alen B. Pritsker, "Introduction of Simulation and SLAM II", John Wiley & Sons, Inc., New York, 1986.

[15] K. Ramamritham, "Real-Time Database", Invited Paper in International Journal of Distributed and Parallel Database, Vol. 1, No. 2, 1993, pp. 199-226.

[16] L. Sha, etc. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol. 39, No. 9, September 1990, pp. 37-47.

[17] R. M. Sivasankaran, K. Ramamritham, J. A. Stankovic, D. Towsley, "Data Placement, Logging and Recovery in Real-Time Active Databases", International Workshop on Active and Real-Time Active Databases, June 1995.

[18] S. H. Son, "Predictability and Consistency in Real-Time Database Systems", The Proceeding of Information Science, 1993.

[19] J. Stankovic, K. Shin, H. Kopetz, K. Ramamritham, L. Sha, D. Locke, J. Liu, A. Mok, S. Davidson, I. Lee, J. Strosnider, "A Real-Time Computing: A Critical Enabling Technology", Technical Report, University of Massachusetts, July 1994.

[20] O. Ulusoy, G. Belford, "Real-Time Transaction Scheduling in Database Systems", Information Systems, Vol. 18, No. 8, 1993, pp. 559-580.