数理解析研究所講究録
第396巻 1980年 1-20

IMPLEMENTING CONSISTENT RECOVERY IN A DISTRIBUTED DATABASE SYSTEM

Akihiro Takagi

Yokosuka Electrical Communication Laboratory

ABSTRACT

Concurrent execution of transactions and various failures occurring during transaction processing in a distributed database system can lead to an inconsistent database state. Concurrency control and recovery scheme play a central role in the effort to preserve consistency. This paper proposes a new recovery scheme based on the concept of multiple uncommitted versions of database entities. This scheme is compatible with a highly concurrent yet consistent schedule of transactions. A rough sketch of an algorithm for database updates that adopts this scheme is also presented in order to show its utility.

1. INTRODUCTION

Components of a database are related to each other in certain ways. Such relations are usually called consistency constraints. Since these consistency constraints cannot necessarily be enforced at each primitive action on the components (usually called entities) such as read and write, sequences of actions are grouped to form transactions, which are the units of consistency. Each transaction must transform the database from one consistent state to a new consistent state [2] [3] [4] .

Although transactions preserve consistency when executed one at a time, concurrent execution of transactions and various failures occurring during transaction processing can cause anomalies such as lost updates, dirty read and unrepeatable read [4] . To prevent these anomalies from occurring, it is required that :

1) for a given concurrent schedule of transactions, there exists some serial schedule that is equivalent to it (schedules that satisfy such a property are called consistent schedules),
2) each transaction is either completed or backed out even if failures occur during transaction processing.

This paper proposes a new scheme for recovering from various kind of failures that is compatible with a highly concurrent yet consistent schedule. It also shows the utility of the new scheme by presenting a rough sketch of an algorithm for database updates that adopts this scheme.

2. CONSISTENT SCHEDULE OF TRANSACTIONS

2.1 Definition of Consistent Schedule

A schedule S for a set of transactions T1, T2, ... , Tn defines the binary relation $<$ as follows : $T1 < T2$ if transaction T1 performs action A1 on entity E at some step in S and transaction T2 performs action A2 on E at a later step in S and if A1 is not permutable with A2. Let $<*$ be the transitive closure of $<$. Then the definition of a consistent schedule can be restated as follows [2] [3] [4] :

S is a consistent schedule if and only if the relation $<*$ is a partial order.

2.2 Consistent Schedule based on Timestamps

Gray et al. [2] [3] [4] explored a consistent scheduling scheme that uses a lock mechanism. Although each transaction has to set an exclusive lock on any entity it dirties and a share lock on any entity it reads, it is possible to ensure that any legal schedule is consistent if each transaction observes a two-phase lock protocol [2] (i.e. transactions cannot request new locks after releasing a lock). However, the degree of concurrency achieved by this scheme is not very good since the two-phase constraint is only a sufficient condition for consistency [2] . In addition, if one wants to cope with possible failures during transaction processing, it is necessary to hold all locks to the end of the transaction. This seriously restricts concurrent execution of transactions since it almost serializes any pair of transactions if there is one or more entity that is needed in exclusive mode by both of them. In fact, the degree of concurrency in System R, which forces transactions to observe this lock protocol, is reported to be less than two [5] . Also, it is subject to deadlock that would be very costly in a distributed environment.

On the other hand, a scheduling scheme that uses a timestamp mechanism is

based on the observation that a consistent schedule of transactions is merely a sequencing of actions performed on the underlying entities by these transactions such that the relation $<*$ is a partial order. Namely, sequencing is directly controlled using a timestamp mechanism [1] [11] rather than a lock mechanism. In this scheme, each transaction is assigned a globally unique timestamp(1), and thereby all transactions are totally ordered. The type manager of each entity schedules actions on the entity in the timestamp order of the transactions that requested these actions. This distributed (i.e. per-entity-based) scheduling algorithm guarantees that, for any pair of transactions T1 and T2 both of which access the same entities E1, E2 ... En, $T1 < T2$ if and only if the timestamp assigned to T1 is smaller than the timestamp assigned to T2. Therefore, the relation $<*$ defined by this scheduling algorithm is a partial order that can be extended to the timestamp order. This scheme ensures the maximum degree of concurrency since it imposes no more restraints than necessary (i.e., that the relation $<*$ be a partial order). In addition, it is deadlock free since $<*$ is an acyclic relation. However, it requires a non-trivial algorithm that ensures that actions are eventually executed in the timestamp order even if components of the system fail or sequence anomalies occur because of communication delays, processing delays etc.(2). Such an algorithm may induce a greater overhead than the previous scheme does.

This paper adopts the latter scheme since it

1) realizes highly concurrent execution of transactions,

---

(1) It is easy to guarantee that every (locally generated) timestamp is globally unique [13] . In addition, Lamport's method [7] guarantees that all local clocks are reasonably synchronized.

(2) For example, suppose that both T1 and T2 perform actions on two entities E1 and E2. Then it may happen that E1 gets a request from T1 before that from T2, but E2 gets requests in the reverse order.

2) is deadlock-free, and

3) fits in with multiple uncommitted versions as discussed later.

3. GUARANTEEING THE ALL OR NOTHING PROPERTY OF TRANSACTIONS

3.1 Two-phase Commit Protocol

In order to prevent inconsistencies from occurring when failures (1) are encountered, it must be always possible to decide whether or not to complete any outstanding transactions and perform the alternative thus selected. Unfortunately, there exists no finite length protocol which ensures that each transaction is either completed or backed out in a distributed system in which nodes or communication lines may fail at any time [4] . Therefore, the second best policy is to relax the requirement for finiteness of the protocol, but attempt to minimize the time window during which a failure causes unnecessary delay. This is the main aim of the two-phase commit protocol that was first mentioned publicly by Lampson et al [8] .
In the two-phase commit protocol, a commit point is established after the first phase of commitment (2) is successfully completed. If something goes wrong before the commit point, the transaction must be backed out. On the other hand, the transaction must be completed no matter what happens after the commit point (even though it may cause an infinite delay). The rest of this section discusses recovery schemes that back out transactions that have failed, that is, transactions that can-

---

(1) This paper excludes media failures such as head crashes, dust on magnetic media etc., and serious failures of operating systems. To cope with such failures, extra recovery schemes such as incremental dump, long-term checkpoint, differential files etc [4] [9] [14] are needed.

(2) Committing the change of an entity's state means making this change decisive to the users of the entity.

6

not be completed because some error has been experienced during their execution.

3.2 Recovery Schemes

Backout of a transaction consists of the following phases :

1) deciding which entities the transaction accessed,
2) restoring the states of these entities to their states before the transaction was invoked.

In addition to these, the information flow (if any) from the transaction must be undone and all other transactions affected by this information flow must also be backed out (this is called cascading of backouts [4] or the domino effect [10]). In this paper, recovery schemes used to implement the first phase are called transaction-oriented recovery schemes because the first phase associates entities that were processed together by a given transaction. Recovery schemes used to implement the second phase are called object-oriented recovery schemes because the second phase deals with the history of actions performed on a given entity (object) by different transactions.

A transaction-oriented recovery scheme basically remembers the identifiers of the entities accessed by the transaction and requests object-oriented schemes associated with these entities to restore the states of the entities when something goes wrong. Audit trails (or logs) [4] [14] and recovery caches [6] can be used as transaction-oriented schemes. However, a backout/commit cache that contains the actions to be performed in the backout process as well as the actions to be performed in (the second phase of) the commit process seems to be most appropriate [12] .

An object-oriented recovery scheme remembers the history of the changes of an entity's state and identifiers of the transactions that depend on each state.

Recovery schemes proposed so far, such as a careful replacement, multiple copies and differential files [9] [14] , can neither realize sufficiently concurrent execution of mutually dependent transactions nor control the cascading of backouts.

This paper proposes a new object-oriented recovery scheme that permits highly concurrent execution of transactions. This scheme uses multiple uncommitted versions of entities ; it is an extension of careful replacement [14] . In this scheme, whenever a transaction tries to perform an action on a given entity for the first time, a new version of the entity is created and the actual action (and all subsequent actions) are performed on this new version. A transaction can create a new version before the immediately preceding transaction has committed the current version, although the commitment of the new version must be deferred until the current one is committed. Each version continues to exist until the immediately succeeding version is committed. Each version contains additional information such as the timestamp value assigned to the transaction that performed the action (i.e. created this version). Therefore, this new scheme records not only the complete history of the state changes of the entity caused by uncommitted actions, but also what transactions depend on each version. This makes it possible to control the cascading of backouts caused by the backout of an uncompleted transaction. Because of this capability, a transaction can safely access an entity before its predecessors have committed their accesses to that entity, and therefore transactions can be executed highly concurrently.

Note that although multiple uncommitted versions are somewhat similar to Reed's tokens [11] , there is one crucial difference : the former provides a powerful concurrency control whereas the latter does not.

3.3 Guaranteeing Complete backout/commitment

In order for a transaction to be completely backed out (/committed), actions performed during the backout process (/the second phase of the commit process) :

1) must not be lost even if a failure occurs,
2) must be repeatable (idempotent [4] [8]) (1) since if a failure occurs during a backout (/commit) process, this process may have to be repeated.

Stable storage [8] that holds objects safely across failures plays an important role in satisfying the first requirement. In particular, implementing versions of entities and backout/commit caches in such stable storage satisfies this requirement. To satisfy the second requirement, different methods can be chosen. One method is to reduce the actions performed during the backout (/commit) process to a sequence of write actions that are well known to be repeatable [8] . Another method is to prevent the actions from being performed more than once by using a mechanism that provides an unique identifier (such as a timestamp) for each invocation of an action.

4. AN ALGORITHM FOR DATABASE UPDATES

This section presents a rough sketch of an algorithm for database updates that uses multiple uncommitted versions coupled with a consistent schedule based on timestamps (for more details, see [12]). It will help the reader understand the implementation aspects and the utility of the proposed recovery scheme.

---

(1) Repeatability (or idempotency) of actions means that performing them several times produces the same result as performing them exactly once.

4.1 Assumptions

This paper considers a distributed database system that consists of a set of nodes interconnected via communication lines. Each node consists of a set of subsystems ; data management subsystems and transaction management subsystems. A transaction management subsystem consists of transaction management processes that execute transactions, one at a time, by communicating with data management subsystems. Each data management subsystem maintains a portion of the database (i.e. a set of entities) and controls accesses to them. It consists of type managers of entities and data management processes that access entities at the request of transaction management processes. A transaction management process retrieves (/updates) the content of an entity by sending a read (/write) message to the data management subsystem that maintains the entity. The message is received by one of the idle data management processes of the subsystem. Then this process accesses the entity (Figure 1).

For convenience, this paper classifies read messages into readr messages (i.e. read-only messages) and readu messages (i.e. read messages that are followed by write messages). An access to the database via a readr message is called a read-only access, and an access via two messages, a readu message and a write message, is called an update access. The set of entities to be updated by a transaction is called its update set, and the set of entities to be read is called its read set. For the sake of simplicity, this paper assumes that

(a) the update set of a given transaction T is a subset of its read set,

(b) T performs an (either read-only or update) access to each entity at most once.

4.2 Representation of Entities

It is assumed that database entities are represented in the stable storage in the following way. Each entry of an entity descriptor consists of the following fields : v#, acc, s, addr. The v# field contains the version number, which is equal to the timestamp of the transaction that created this version (by an access request). The acc field indicates whether the access was read-only or update. The s field indicates the current state of this version, which may take on one of the following values :

1) dirty : already read, but not yet written (meaningless in the case of read-only access)
2) dependent : already accessed, but not yet prepared for commitment
3) prepared : prepared for commitment
4) committed : already committed
5) discarded : already discarded because of failures, sequence anomalies etc..

The addr field contains the address of the storage cell that contains this version (1). Entries of a descriptor are sorted in the timestamp order (Figure 2).

4.3 Processing of Read Actions

A transaction can access an entity before preceding transactions that accessed the same entity commit their accesses. The only constraint on concurrency is that accesses to the same entity must be performed in the timestamp order.

When a readu action is invoked, the type manager processes the request in one

---

(1) If the acc field is "read-only", this version shares the storage cell with the previous version.

of the following ways, depending on the timestamp value ts assigned to the requestor.

1) If ts is greater than the version number of the current version, the type manager creates a new version (that is, creates a new descriptor entry) whose version number (v#), state (s) and access mode (acc) are "ts", "dirty" and "update", and returns the content of the (former) current version, except when the current version is "dirty", in which case creation of a new version is deferred (that is, the respective data management process is suspended) until the current version becomes "dependent".
2) If ts is smaller than the version number of the current version but greater than those of the committed and prepared (if any) versions, the type manager
   a) discards the versions whose version numbers are greater than ts
   b) creates a new version and returns the content of its immediate predecessor (that is, the closest older version).
3) Otherwise, the request is rejected (1).

Processing of a readr is similar to that of a readu except that

1) even if the action is not the latest one, it is not necessary to discard the versions that have greater version numbers. Instead, it is sufficient to insert a newly created, but outdated version immediately before these versions,
2) the state and the access mode are set to "dependent" and "read-only"

---

(1) Requests older than some "prepared" version are rejected since it is highly probable that such a version will be committed. Moreover, according to the two-phase commit protocol, once a version is in the "prepared" state, only the transaction management process that originated the transaction may abort it (and thus discard the "prepared" version).

12

respectively,

3) if the queue of waiting processes is not empty, then the type manager wakes up the process that has the smallest timestamp.

4.4 Processing of Write Actions

When a write action is invoked and the version created by the corresponding (preceding) readu action is not "discarded", the type manager acquires a free storage cell for the new version, writes the content of the buffer into it, and changes the state to "dependent". Otherwise, it returns as "discarded". If the queue of waiting processes is not empty, the type manager wakes up the process that has the smallest timestamp. Each version, unless discarded, is deleted when it and a newer version are committed.

4.5 Commitment of Transactions

The central principle of the commit protocol proposed in this paper is that no transaction can commit the versions it created until the states of all previous versions of these entities become "committed". The commit protocol is basically a two-phase commit protocol but it is considerably different from others [4] [8] [11] because it must co-operate with the concept of multiple uncommitted versions.

In the first phase, a transaction management process sends prepare messages to all involved data management subsystems to confirm that the versions it accessed are eligible to be committed. When the type manager receives the request, it performs one of the following operations, depending on the state of the designated version and the state of the immediately preceding version :

1) changes the state of the designated version to "prepared" if the designated version is "dependent" and the immediately preceding version is already committed,

2) suspends the data management process until the state of the immediately preceding version is changed to "committed" or "discarded" if the designated version is "dependent" and the immediately preceding version is not yet committed,

3) returns as "discarded" if that version is already "discarded".

If all return messages are "prepared", a commit point is established and then the second phase begins. Otherwise the transaction management process sends undo messages to all involved data management subsystems to backout the transaction. This is done by executing the set of actions (i.e. "send undo message" in this case) saved in the backout cache associated with this transaction.

Once begun, the second phase must be completed no matter what happens. The transaction management process sends commit messages to all involved data management subsystems to commit the versions the transaction created. This is done by executing the set of actions (i.e., "send commit message" in this case) saved in the commit cache associated with this transaction. When the type manager receives the request, it

1) changes the state of the designated version to "committed",

2) deletes the older committed version,

3) if a data management process is waiting for this version being committed, wakes up the process.

Sending of a commit message is repeated until it is successfully processed. When the transaction management process confirms that all commit messages were successfully processed, the whole commitment process is completed.

4.6 Backout of Transactions

Backout of a transaction occurs either when the transaction is aborted be-

14

cause of a failure [1] of any participants in the transaction processing or when it is involved in a sequence anomaly. A transaction is also backed out when transactions on which it depends are backed out. Suppose that T is the transaction that must be backed out. The backout of T causes all transactions that depend on T (in terms of the relation $<*$) to be backed out. This cascading of backouts is done in the following way.

1) The transaction management process executing T sends undo messages to all involved data management subsystems. This is done by executing the set of actions saved in the transaction backout cache.
2) When a data management process receives the undo message, it requests the appropriate type manager to delete the version created by T by invoking an undo action. If the access mode of the version is "update", the type manager not only deletes the version, but also changes the states of all newer versions (if any) to "discarded".
3) When a data management process invokes a write or prepare action on one of these discarded versions at the request of the transaction management process that created it, the type manager returns as "discarded". Then the data management process returns a "discarded" reply to the transaction management process that sent the write or prepare message to it.
4) Each transaction management process that has received a "discarded" reply must also be backed out by following the above procedures 1) 2) and 3).

---

(1) This does not include failures that occur during the second commit phase.

5. CONCLUSION

The main goal of this paper was to develop a new recovery scheme that realizes not only reliable but also highly concurrent execution of transactions in a distributed database system. This paper proposed multiple uncommitted versions (coupled with a backout/commit cache) as such a recovery scheme, and showed that this new scheme satisfied the above requirements. The new scheme is in striking contrast to other recovery schemes proposed so far that suffer (to a considerable extent) from the serious restraint that "no transaction can access any entity until all previous transactions that accessed the entity are completed". The memory overhead induced by this new scheme may be sufficiently low since, at any given time, each of the vast majority of entities is expected to have only one version. The processing overhead may also be acceptable since, in a normal situation where sequence anomalies do not occur frequently, the overhead is almost comparable to that induced by careful replacement.

Several extensions of the new scheme will be possible. First, if a version is read-only, it is not necessary for later versions to wait until it becomes committed. That is, a later version could be committed once the immediately preceding update version was committed. Second, if the older committed versions were not deleted, then an old read request could be processed (without creating a new version) long after subsequent versions had been committed ; this is what the multiple versions of Reed's scheme [11] provide. Third, the assumption in Section 4.1 that any transaction performs an access to each entity at most once could be removed by slightly extending the timestamp mechanism. Namely, if each timestamp were composed of a transaction id part and an access id part, then it would be possible to decide whether two different accesses were performed by the same

transaction.

ACKNOWLEDGMENT

This work was done while the author was visiting at the Laboratory for Computer Science, Massachusetts Institute of Technology from 1977 to 1978. The author wishes to thank Professor Liba Svobodova for her many helpful comments and suggestions.

REFERENCES

[1] P.A.Bernstein, D.W.Shipman, J.R.Rothnie and N.Goodman, "The concurrency control mechanism of SDD-1 : A system for distributed database (The general case)," Computer Corp. America, Cambridge, MA, Tech. Rep. CCA-77-09, Dec. 1977.

[2] K.P.Eswaran, J.N.Gray, R.A.Lorie and I.L.Traiger, "The notions of consistency and predicate locks in a database system," Commun. Ass. Comput. Mach., vol.19, pp. 624-633, Nov. 1976.

[3] J.N.Gray, R.A.Lorie, G.R.Putzolu and I.L.Traiger, "Granularity of locks and degree of consistency in a shared database," IBM Research Rep. RJ 1654, Sept. 1975.

[4] J.N.Gray, "Notes on database operating systems," in Lecture Notes in Computer Science, vol.60, Springer-Verlag, Now York, 1978.

[5] J.N.Gray, Talk at Laboratory for Computer Science, M.I.T., Mar. 1978.

[6] J.J.Horning, H.C.Lauer, P.M.Melliar-Smith and B.Randell, "A program structure for error detection and recovery," in Lecture Notes in Computer Science, vol.16, Springer-Verlag, Now York, 1974.

[7] L.Lamport, "Time, clocks and ordering of events in a distributed system," Commun. Ass. Comput. Mach., vol.21, pp. 558-565, July 1978.

[8] B.Lampson and H.Sturgis, "Crash recovery in a distributed data storage system," Computer Science Laboratory, Xerox Palo Alto Research Center, CA, 1976.

[9] R.A.Lorie, "Physical integrity in a large segmented database," ACM Trans. Database Syst., vol.2, pp. 91-104, Mar. 1977.

[10] B.Randell, P.A.Lee and P.C.Treleaven, "Reliability issues in computing

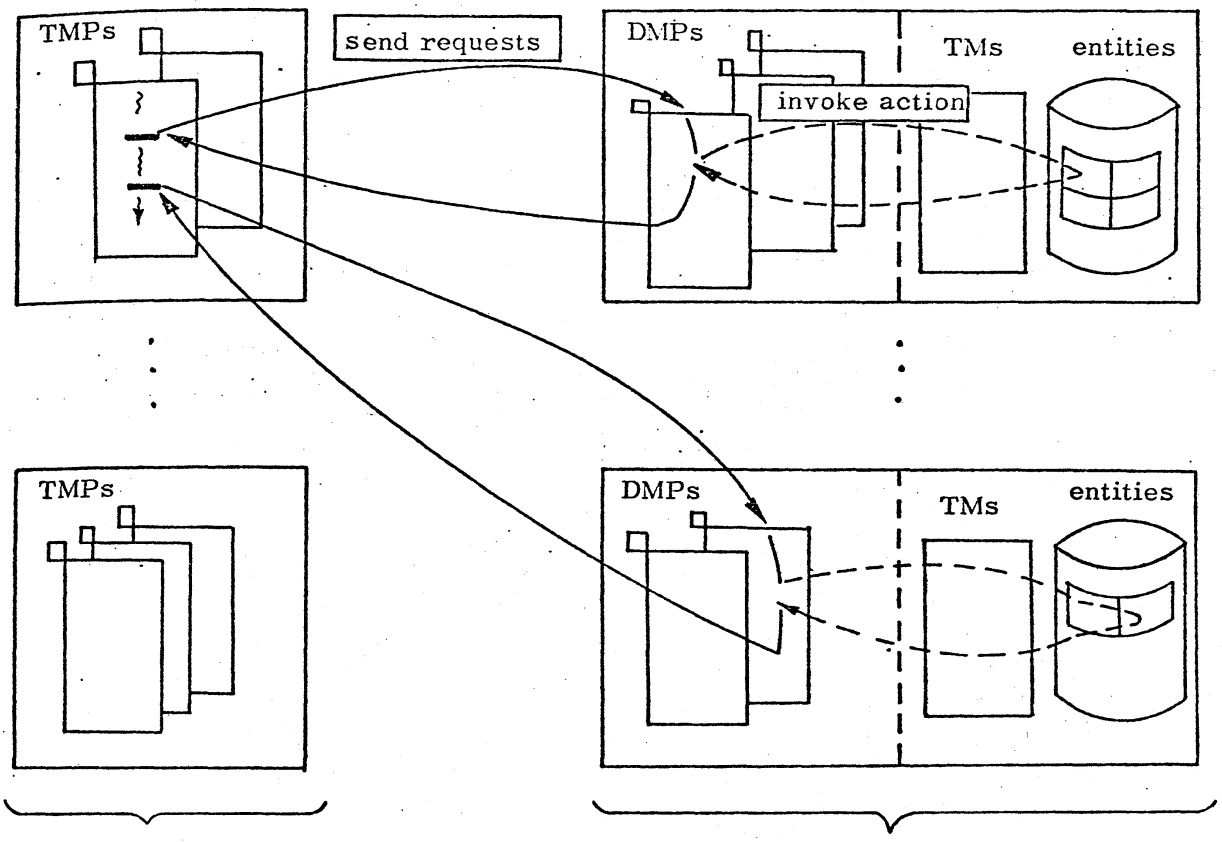
system design," ACM Comput. Surveys, vol.10, pp. 123-165, Jun 1978.

[11] D.P.Reed, "Naming and synchronization in a decentralized computer system," Laboratory for Computer Science, M.I.T., Cambridge, MA, TR-205, Sept. 1978.

[12] A.Takagi, "Concurrent and reliable updates of distributed databases," Laboratory for Computer Science, M.I.T., Cambridge, MA, TM-144, Nov. 1979.

[13] R.H.Thomas, "A solution to the update problem for multiple copy databases which uses distributed control," Bolt Beranek and Newman Inc., Cambridge, MA, BBN Rep. 3340, July 1975.

[14] J.S.M.Verhofstad, "Recovery techniques for database systems," ACM Comput. Surveys, vol.10, pp. 167-195, June 1978.



TMP : Transaction Management Process
DMP : Data Management Process
TM : Type Manager

Figure 1 The assumed system structure

20

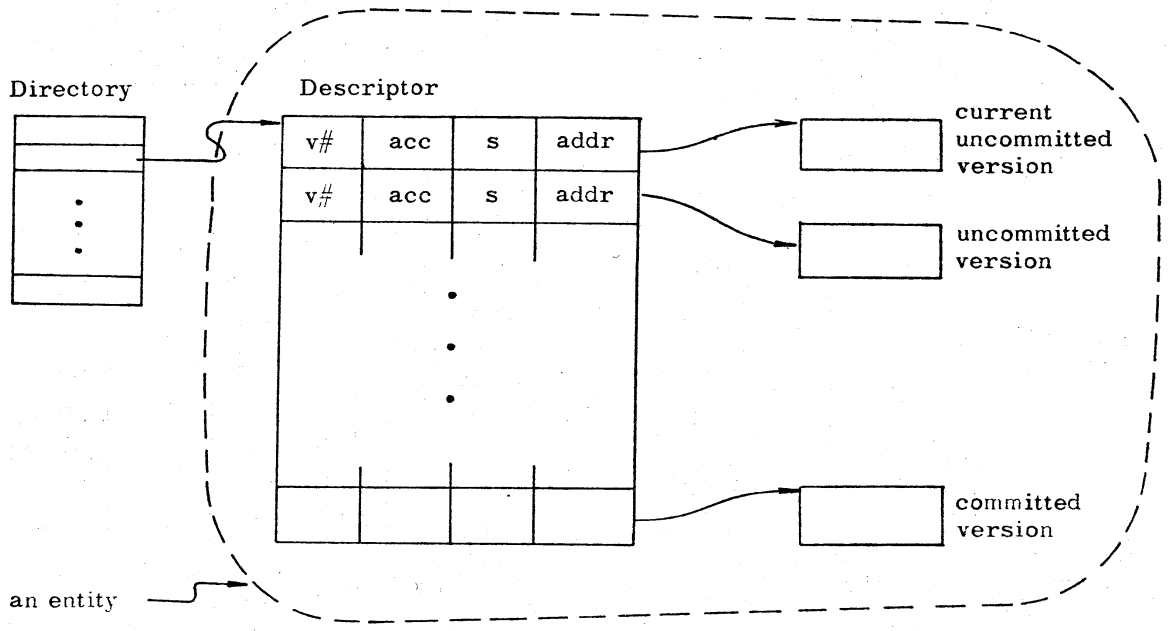


Figure 2 . Storage representation of entities