The Escrow Transactional Method

PATRICK E. O'NEIL
Computer Corporation of America

A method is presented for permitting record updates by long-lived transactions without forbidding simultaneous access by other users to records modified. Earlier methods presented separately by Gawlick and Reuter are comparable but concentrate on "hot-spot" situations, where even short transactions cannot lock frequently accessed fields without causing bottlenecks. The Escrow Method offered here is designed to support nonblocking record updates by transactions that are "long lived" and thus require long periods to complete. Recoverability of intermediate results prior to commit thus becomes a design goal, so that updates as of a given time can be guaranteed against memory or media failure while still retaining the prerogative to abort. This guarantee basically completes phase one of a two-phase commit, and several advantages result: (1) As with Gawlick's and Reuter's methods, high-concurrency items in the database will not act as a bottleneck; (2) transaction commit of different updates can be performed asynchronously, allowing natural distributed transactions; indeed, distributed transactions in the presence of delayed messages or occasional line disconnection become feasible in a way that we argue will tie up minimal resources for the purpose intended; and (3) it becomes natural to allow for human interaction in the middle of a transaction without loss of concurrent access or any special difficulty for the application programmer. The Escrow Method, like Gawlick's Fast Path and Reuter's Method, requires the database system to be an "expert" about the type of transactional updates performed, most commonly updates involving incremental changes to aggregate quantities. However, the Escrow Method is extendable to other types of updates.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; H.2.2 [**Database Management**]: Physical Design—*deadlock avoidance*; *recovery and restart*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Escrow transactions, hot spots, long-lived transactions, nested transactions, two-phase commit

1. INTRODUCTION

This section sketches the general background of systems for assuring transactional consistency in a multiuser environment, including salient details of two methods supporting high-speed transaction updates that do not forbid record access by concurrent transactions: Gawlick's Fast Path and Reuter's Method. It then introduces the basic idea of the Escrow Method, which has many similarities with the two earlier techniques. In Section 2 we present a detailed architecture to support the Escrow Method, applied to aggregate field quantities. In Section 3 we look at several areas where the benefits of the Escrow Method can best

Author's address: Computer Corporation of America, 4 Cambridge Center, Cambridge, MA 02142.

be realized, notably, distributed transactions and human interaction during a transaction. Conclusions are presented in Section 4.

1.1 General Transactional Methods

The concept of "transaction" has had important applications to the needs of business where a set of record updates must succeed all at once or not at all, as with a transfer of money between two account records. Transactions serve a dual purpose in that they represent logically indivisible sets of record access (reads or updates) both for purposes of concurrency and recovery. These purposes are theoretically separable in the following senses: (1) In the absence of concurrent users, we would still wish to provide for recovery of logical sets of record updates in case of memory or media loss; and (2) if we postulate a system with absolutely dependable hardware so that recovery becomes superfluous, we would still wish to exclude multiple users from simultaneous access to a record field quantity in order to avoid the well-known "lost update problem."

Although most real-life transactional systems internally dovetail the methods of recovery and concurrency so that they become almost inextricable, it is still good practice to separate the two concepts as much as possible in theory. Without such a modular approach, it becomes difficult to keep a sufficiently large set of concepts in mind at once; we need these categories to structure any attempts at rigorous analysis when new designs for transactions are proposed. As will be seen, the Escrow Method adds a new dimension of recoverability to a type of high-concurrency transactional method that has been around for some time, resulting in a technique with valuable new applications.

The most common solution to the problem of lost updates with concurrent transactions has been to lock records. The first transaction to update a record locks it so that another transaction cannot update it concurrently; the lock is held until the first transaction commits (concludes successfully) or aborts (concludes unsuccessfully), but in either case the database is left in a consistent audit-balanced state (where we assume that the transaction itself did not through some error introduce an inconsistency). For a thorough discussion of locking and other well-known methods of concurrency control such as time-stamping, see [1, 4, 5, 11].

1.2 The Fast Path Method of Concurrency Control

A method of concurrency control that is less well known than locking is the one used in Main Storage Databases in IMS/VS Fast Path, as presented in [1]–[3] and [7]. Although the theory for this feature was developed in 1974 and 1975, architectural details have not been available in published form until recently; furthermore, the approach itself requires extra sophistication in that it makes special distinctions in data types to speed up a certain kind of transaction involving "hot-spot" aggregate fields. It is for these reasons that Fast Path Concurrency has not received as much attention as some other methods that are perhaps less in actual use. For researchers interested in high-speed transactions, however, this method is quite familiar.

Fast Path Concurrency deals with transactional access to aggregate field quantities, such as "quantity on hand" or "total cash received," where the type of update envisioned nearly always has the purpose of performing an increment

or decrement to the quantity. It is further assumed that these quantities are accessed for update of this type with great frequency, so that these fields become "hot spots" and represent a major bottleneck to the transaction rate if a normal record-locking protocol is followed. It is therefore important to permit access to the field quantity without excluding other transactions from enjoying the same access, except for the shortest possible time; holding a lock on a quantity for the length of a transaction, even one that lasts just a second or so, is unacceptable.

Fast Path allows the programmer to make special requests to VERIFY that an attribute (field quantity) bears some relation ($<$, $\leq$, $=$, $\geq$, $>$) to a known value, and to MODIFY the field quantity by a constant amount (or to a different constant value). Thus, if the field QOH (quantity on hand) were constrained to remain nonnegative by the application, we would see a great many pieces of code performing the following logical steps:

$$\text{VERIFY QOH} \geq \text{CONST}$$
$$\text{MODIFY QOH} := \text{QOH} - \text{CONST}$$

Fast Path validates the VERIFY requests at the time the logic is executed and again during commit processing. The MODIFY requests, however, are not actually executed until commit time, after all VERIFY requests have once again been validated. The first test gives the application program a chance to take an alternative branch if the QOH field does not meet the VERIFY criterion, while the second test at commit time results in an abort if it fails. When the first VERIFY request is processed during in-line logic, no guarantee is given that the later commit processing will be successful, since the quantity is not actually updated until the transaction is ending. This is not possible without keeping some running idea of what values the quantity might take on if various outstanding MODIFY commands eventually succeed; presumably such a running value must be kept by making updates associated with the record when the MODIFY request is issued. Regarding this, Gawlick points out [2] that the idea of modifying a record when the MODIFY request is issued "creates problems for asynchronous image copies, and, more seriously, it does not allow retrieval of a consistent picture of the whole data base, unless one is willing to deadlock and/or delay the processing of many transactions." We will refer again to this comment later.

Note that the whole idea of Fast Path transactions of this sort is that the system becomes an "expert" on the intention of the program. At the time of commit, during a very short time window, the system is able to "replay" the series of VERIFY requests generated by the program logic, and be sure that the situation is acceptable so that the corresponding MODIFY commands can be applied. The real-time tests of the VERIFY requests during the program logic are simply to give confidence that this "optimistic" transaction will eventually succeed; the real-time tests could be omitted without loss of rigor.

1.3 Reuter's Transactional Method

In [9], Reuter presents a slightly different paradigm for transactions on aggregate quantity fields, meant as an expansion on the Fast Path approach. The basic idea is that the aggregate field quantity QOH is subject to TEST & MODIFY requests as in Fast Path. (The TEST is analogous to the Fast Path VERIFY and is immediately followed by the corresponding MODIFY request.) The

MODIFY request actually makes immediate changes to the field quantity in the following sense.

At any time, we only expect to know a *range* of values within which the quantity falls, for example, $A \leq QOH \leq B$ for some constants A and B. Then a TEST criterion is TRUE if the quantity in question *must* obey the criterion (e.g., the criterion is $C \leq QOH$, and given the range $A \leq QOH \leq B$, we also have $C \leq A$), it is FALSE if the quantity in question *cannot* obey the criterion (e.g., as before, only now $B < C$), and otherwise the criterion is POSSIBLY TRUE (e.g., as before, only now $A < C \leq B$). A successful MODIFY request has the effect of *expanding* the range of uncertainty in which the quantity must fall; the range for the values of the quantity resulting from a MODIFY corresponds to the realization that the transaction that has requested this MODIFY may succeed or may fail: Accordingly, the incremental update may or may not occur, and this is the range of uncertainty. At a later time, when a commit or an abort eventually occurs, the range is reduced in the appropriate fashion, until the quantity has a single value with no uncertainty when there are no transactions outstanding that have modified the quantity. The detailed handling of the range for the quantity is presented later in Section 2.2 dealing with Escrow transactions for aggregate fields and is quite analogous to the scheme offered by Reuter.

The important new functionality offered by Reuter's Method over Fast Path is that Reuter attempts to provide a *guarantee* in real time that a TEST & MODIFY request that succeeds will never be denied during later commit processing. As will be seen, this is a crucial property for long-lived transactions that are supported by the Escrow Method, and an attempt will be made to journal this guarantee for later recovery. As a result, we will have completed phase 1 of a two-phase commit and will be able to UNDO or REDO the update at any future time, to use the notation of Gray [4].

In Reuter's paper, the question of recovery is not fully dealt with. This is undoubtedly due to a desire not to overburden the exposition, since Reuter clearly had extensive ideas in this area as a coauthor of a framework to classify recovery schemes [6]. As mentioned at the beginning, it is perfectly reasonable that this paper would concentrate on concurrency to the exclusion of recovery. The difficulty is that it is not totally clear without careful treatment how recovery is to be added to this scheme; it is this difficulty to which Gawlick was alluding in the earlier quoted statement. It turns out that the analysis that gives recovery of completed transactions has an important dividend, in that it suggests how recoverability of intermediate results for long-lived transactions can be handled.

To see some of the difficulty of recovery with this method, consider what would happen if a memory loss were suffered following a disk write of a field with an uncertain "range" of values. How would the true value for the field after a crash be determined, when the transactions that were in progress have lost their place in their trains of logic?

There seems to be a problem of synchronization here: the range of a field quantity that has been introduced by several transaction updates should not be written to disk without other information that the recovery manager can use to back out incomplete transaction updates on recovery. Note that several transactions may be involved in updating the current field, each with its own contri-

bution to the range found for the field. Without knowing exactly what these contributions are, it is impossible to recapture the true value of the field in a later recovery, when some subset of the transactions have committed and the remainder must be backed out. Journaling information after every field update to achieve this recovery information is clearly to be avoided for efficiency reasons. The Escrow Method proposed here handles the stated need in a natural manner. To do this, it creates an entirely new entity, which is called an Escrow journal.

1.4 The General Escrow Transactional Method

In its full generality, the Escrow Method is not restricted to transactions involving aggregate fields, although this is the major example that will be offered here. The basic ideas underlying Escrow transactions are the following:

(1) Some field quantities and a specified class of tests and updates for these quantities are designated to be of "Escrow type." These might be the aggregate field quantities encountered above, the tests that require range restrictions, and the class of updates involving all positive or negative incremental changes; basically, these are the building blocks for a type of transaction for which the system is willing to take responsibility for being an "expert" in the sense indicated earlier.

(2) When any attempt is made by a transaction to perform an "Escrow-type update" to some field, the system first uses its "expert" knowledge to guarantee that the update can be performed *at any time in the future*, and *in any order with any subset of updates for which this guarantee has already been made.* If a test is a necessary condition for the update, then it is considered a part of the update, and the guarantee must include assurance for the test as well, that it will be valid after any possible subset of already guaranteed updates have been applied, and of course that adding the current update to the set guaranteed will not make the test for some other update invalid in some possible schedule of commits.

What we are doing here is assuring that with these guarantees in place we can still UNDO any transaction while maintaining other guarantees, and that no conflicts in serializability can arise between different fields since all updates can be committed in any order. The algorithm used by the system to do this is immaterial, but the algorithm may certainly make use of *Escrow journals* for the affected field, explained in the next paragraph.

(3) After the guarantee above is made, an Escrow journal is created for this request by the system. The Escrow journal contains at least the following information:

(1.1) Escrow journal contents

Transaction ID making this request
Escrow pool ID (e.g., positive or negative increment)
Parameters of the request (e.g., test criteria and incremental change)
Field being updated (implicit by placement of the journal)

Additional system-accessible or application-accessible information may be included in the Escrow journals as necessary to the algorithm used.

These Escrow journals, as they are created, are placed in a data structure that is considered to be a logical *extension* of the field being updated. Thus, if an Escrow field being written to disk is referred to, it should be understood that all active Escrow journals are included. It is not assumed that an Escrow journal is automatically placed on disk when the update occurs, but simply that it is placed in memory in (logical) proximity to the field of which it is a logical extension, so it will go out to stable storage with the field in question. The information contained in the extended logical field should be exactly the information needed to perform recovery. It will turn out that it is necessary to include, with the extended logical field, a time stamp that will allow the system to put in sequence updates of the field to stable storage and transaction commit logs that will be found on stable storage during recovery. In addition, whatever other information is held by the system to aid in determining legal requests, such as the range of the field value, may also be considered part of the logical extension of the field.

(4) When a transaction that has made a request to update a given field eventually commits or aborts, the associated Escrow journal is applied as an update or discarded, as appropriate. Whichever action is performed, the logical field is updated appropriately, and the Escrow journal for that transaction is deleted from the logical field. (The Escrow journal, with the name of the field affected, should now be an appropriate object to log the success of the transaction in the event of a commit.) Note that, as the Escrow journals are applied, it is still the case that *any subset* of the Escrow journals remaining should be applicable in any order. In the case where several options are open for how an update is to be applied, the system must choose the one that makes possible all remaining subsets in any order.

In the following section, a detailed example of the Escrow Method will be considered, which associates with an aggregate field a data structure, range, and set of actions the system should take under various circumstances. Most of these details are matters of efficiency only. It is clear that, if a set of Escrow journals and a starting quantity for a certain field is given, it can be determined whether each new request can be accommodated with any subset of the existing journals in any order. If necessary, all possible orders of applying journals (or not applying them when an abort occurs) can be attempted. Note that in this case the concept of a range of values allows one to determine applicability of the journals in a more efficient manner because of commutativity of addition, but the abstract theory of Escrow journals does not require such an efficient algorithm! Any method will serve that can take update requests and make the guarantee of Paragraph (2) above. Indeed, we are willing to go through quite a bit of calculation to permit multiuser access to fields undergoing Escrow update, and we should not blind ourselves to solutions that at first appear "inefficient." Some examples are given in Section 4 to show how this Generalized Escrow Method can be useful in practice.

2. ESCROW TRANSACTIONS FOR AGGREGATE FIELDS

The term "Escrow journal" was originally chosen for its connotation in banking, where a portion of an account balance is taken to one side and held for some particular purpose. In its expanded meaning, the Escrow journal acts as a

representative of the guarantee that has been extended by the system that the update requested may be committed (or aborted) at any future time. Now a detailed example of the Escrow Method in the case of aggregate fields will be given. It is quite similar in some ways to Fast Path and Reuter's Method.

In Section 2.1 a notation for a programmer interface to use the Escrow Method on aggregate fields is introduced, and some of the issues that arise are investigated. In Section 2.2 details of some of the system design and an example of how changes occur within the extended logical field as Escrow requests are made and transactions commit and abort are given. In Section 2.3 it will be shown how recovery can be performed under the Escrow Method.

2.1 Programmer Interface for Escrow Method on Aggregate Fields

What is needed now is a new type of system request that an application programmer can make to put a quantity "in Escrow," as well as a way to use the quantity for requirements of the transaction as these become apparent.

Example (2.1) An ESCROW request:

IF ESCROW(field=F1, quantity=C1, test=(condition), recover=BOOL)
THEN continue with normal processing
ELSE perform exception handling (such as abort)

The ESCROW request above is a system function that asks that a quantity C1 (which for the moment is assumed to be nonnegative) of the field quantity F1 be set aside in Escrow for later use by the transaction. The caller can specify in the "test=(condition)" parameter some condition that must be satisfied by the quantity in the field F1 *after* the quantity C1 is removed. This condition must hold if the return from the ESCROW request is to be successful (TRUE). If the ESCROW request does not result in a TRUE return, the program will perform some sort of exception-handling logic, such as aborting the transaction or looking at another field for some "replacement" for this missing quantity. If the test=(condition) turns out to be true, however, and taking the quantity C1 from the field F1 will not invalidate the test of some earlier successful ESCROW request of a transaction that is still live, the system will have taken the quantity C1 of the field F1 into an "Escrow pool," and the programmer will be permitted to use a second system function of the form

USE(field=F1,quantity=K1)

where the quantity K1 now will be taken from the pool of Escrow items from the field F1 set aside by earlier ESCROW requests. The sum of quantities from a field that have been put in Escrow cannot be exceeded by the sum of the quantities used. This would be a logic error in the program and would be trapped by the system at the time of the offending USE call. When the transaction in process ultimately commits, all of each quantity that has been put in Escrow but not used will be returned to the appropriate field quantity. Of course, if the transaction aborts, the entire Escrowed quantity will be returned.

The "recover=BOOL" parameter in the ESCROW request allows the programmer, in the case of "recover=YES," to specify that the current Escrow pool state should be recovered in the event of later memory or media failure, with the

transaction restarted. This is not a *commit* action, since after recovery the program may still decide to abort the Escrow changes made so far. The purpose is to provide a way for the program to keep faith with outside agents (such as cohort processes in distributed transactions) that may be counting on the Escrow guarantee of eventual commit. Of course in the case of "recover=NO," the request will be rolled back in the event of a crash, the normal fate of an update of an incomplete transaction during recovery.

It is the responsibility of the program to recover its internal logical state and perform all appropriate USE requests when a recoverable ESCROW request is issued. Some mechanism to aid the programmer in this recovery is needed, but it is not proposed here. However, the value of an ESCROW request with a "recover=YES" parameter is that it will leave no concurrency window during recovery where a needed resource can be used up by another process.

The USE function is a convenience to the programmer to permit a certain sloppiness in ESCROW requests; it does not actually have any connection with the concurrency control of the system. Nevertheless, it is useful in clarifying the options of the programmer.

Thus, to transfer a nonnegative quantity C of some item from QOH while keeping the QOH nonnegative, we could give the following ESCROW request:

IF ESCROW(field=QOH,quantity=C,test=(QOH $\geq$ 0),recover=NO)
THEN USE(field=QOH,quantity=C) (continue processing . . .)
ELSE ABORT

Recall that the test=(QOH $\geq$ 0) takes place as if C were *already* subtracted. The effect of these combined ESCROW and USE requests is really quite simple. It should be clear that this rather complicated form is used only because of the considerations of multiuser concurrency and recovery that are present. The following is much preferred:

IF (QOH $- C \geq 0$)
THEN QOH := QOH $- C$
ELSE ABORT

The general form that the Escrow test=(condition) can take is subject to rather complex restrictions; in this paper it is restricted to the form (F1 $\geq$ C2). Note too that a *negative* incremental quantity can be put in Escrow for later USE of negative quantities. These negative quantities can be thought of, for example, as returns of stock to a warehouse where only some limited amount of stock can be housed, and empty places must be reserved; there is an upper bound on final stock, and thus the condition to be checked is (F1 $\leq$ C3). Of course, after a successful commit following a USE of a negative quantity, the field quantity will be *increased.* In the discussion that follows, the case of negative quantities is usually neglected in pursuing simplicity of explanation. The case of negative Escrow quantities is parallel to the case of positive quantities. It is also distinct, in that separate Escrow pools are required for positive and negative quantities of the same field. It is possible that other named Escrow pools may be useful to the

programmer under various circumstances; the defining characteristic of these pools is that each pool is conceptually distinct and treated separately during recovery.

Note that a user may wish to Escrow more of some quantity than will be used immediately, since later logic may dictate that more be used (think of taking an advance on travel expenses). It is also possible to make a second Escrow call in the same transaction (think of requesting a second advance).

2.2 Internal Design for the Escrow Method on Aggregate Fields

Analogously to the terminology of [9], every Escrow journaled (aggregate) field A has three quantities representing it at any time:

val(A): The value A will assume if all ESCROW requests to A of currently live transactions are applied without any aborts.
inf(A): The lowest value A might assume from any combination of commits and aborts of currently live transactions that have performed ESCROW requests on this field.
sup(A): The highest value A might assume from any combination of commits and aborts of currently live transactions that have performed ESCROW requests on this field.

In these definitions ESCROW requests are included for negative quantities as well as positive ones; indeed it is only by attempting to Escrow a negative quantity of the field A that inf(A) could be less than val(A). Note that only the ESCROW requests defined in Section 2.1 will make an actual change in the field quantities, inf(A), val(A), and sup(A). The USE request calls only on some quantity of an item that has already been placed in Escrow. For short, the ESCROW request asks for a "change to the field." No other action the user takes can change the field (except a final abort or commit).

If no transactions are currently live (exist uncommitted) that have requested a change to the field A, then inf(A) = val(A) = sup(A). From the definition, it is clearly the case that inf(A) ≤ val(A) ≤ sup(A). The need for the quantity val(A) as will be seen later is questionable, but it can be thought of as the most likely value of A, which may be of some value to the application programmer in certain edge cases. An Escrow journaled field can be thought of as existing physically with three values in a record. However, any attempt to access the three values inf(A), val(A), and sup(A) as normal fields in a user work area will fail; tests on these values can be performed only through an ESCROW request, possibly with a "quantity=0" parameter value. For example:

IF ESCROW(field=A,quantity=0,test=(sup(A) ≥ 100),recover=NO) . . .

Although an application program can specifically access any one of these three values within an ESCROW request, reference to inf(A), sup(A), and val(A) should be made only in tests and should not appear in ESCROW requests with a nonzero quantity. Standard usage should be to test the quantity A itself, and the system will follow a conservative policy when the quantity A is referenced.

Consider the following request:

(2.1)

IF ESCROW(field=A,quantity=C,test=($A \geq 0$),recover=NO)
THEN USE(field=A,quantity=C) ...
ELSE ...

The system considers the condition test=($A \geq 0$) to be TRUE only if in fact, after the quantity C has been taken from A and placed in Escrow, ($\inf(A) \geq 0$); that is, the smallest value A might attain still exceeds 0. This is the conservative assumption. If the condition were instead test=($A \leq D$), then the conservative policy would be to return TRUE only if ($\sup(A) \leq D$). In every case the conservative policy uses the value $\inf(A)$ or $\sup(A)$, which guarantees all eventual possible values will make the condition true. Note particularly that this policy makes the resultant THEN and ELSE clauses of the ESCROW request asymmetric: If test=($A \geq 0$) fails, this is not the same as having test=($A < 0$) succeed!

If the condition in the ESCROW request of (2.1) above is TRUE, then the assignment $A := A - C$ is performed by changing the values of $\inf(A)$, $\sup(A)$, and $\text{val}(A)$ in accordance with their definitions, *assuming* that this assignment will not invalidate another inequality condition for this field imposed by an earlier live transaction. The need to maintain the consistency of conditions for earlier live transactions was mentioned earlier, and details appear below.

2.2.1 *Definitions.* What follows is a definition of how inf, val, and sup quantities of a field are affected by ESCROW requests and transactional commits and aborts. In these definitions, unless otherwise stated, the value C is assumed to be positive. The definitions also give an account of how Escrow journals are created and deleted as the transaction proceeds. (See (1.1) above for the general layout of the Escrow journal. A detailed example of field structure changes during the life of several transactions is given directly after the definitions.)

(1) When ESCROW (field=A,quantity=C, ...) is requested successfully, the system sets $\inf(A) := \inf(A) - C$ and $\text{val}(A) := \text{val}(A) - C$. (If the quantity C were negative, then $\inf(A)$ would not be affected, and we would have $\sup(A) := \sup(A) - C$.) An Escrow journal is created and placed in an extended logical field data structure for the field A; this will inform the system what to do if a back out or commit is later performed. Note in passing that the value C must be a constant as seen by the Escrow system, but may well be a calculated variable value to the program logic.

(2) As later USE calls are made against the quantity placed in Escrow by the transaction, the Escrow journal is accessed to check that the quantity is available and to record the quantity remaining in Escrow that has been unused. Additional ESCROW calls (with a positive quantity) may also access this journal and add to the quantity remaining in Escrow. The logical field Escrow journal has an entry to keep track of the quantity USEd as well as the total quantity taken into Escrow by foregoing ESCROW requests. Separate Escrow journals are maintained for at least the two Escrow pools that must exist for every transaction and field where positive and negative quantities are placed in Escrow, and may exist for other named pools as well.

(3) If the transaction that has made this request eventually commits, the Escrow journal in the extended logical field is accessed, and the unused quantity K remaining in Escrow (a nonnegative quantity) is placed back in the field by setting $\inf(A) := \inf(A) + K$ and $\text{val}(A) := \text{val}(A) + K$; then the total quantity placed in Escrow and used, U, is reflected by setting $\sup(A) := \sup(A) - U$. In the case where a negative Escrow quantity was requested, reverse the roles above of $\inf(A)$ and $\sup(A)$. The result should set $\inf(A) = \text{val}(A) = \sup(A)$ again if no other simultaneous transactions were in operation on this field. The Escrow journal is deleted from the data structure of the extended logical field, and a logical copy of the Escrow journal is written with a commit log in case recovery is later required. In the commit log, the field affected in the Escrow journal is explicitly named, the transaction number is written once in the header, and the quantity named is the quantity U actually used. The commit log also contains a time stamp for sequencing with writes of extended logical fields to stable storage.

(4) If the transaction that has made this request eventually aborts, the changes made when the request was first applied are reversed. The Escrow journal in the extended logical field informs the system it should set $\inf(A) := \inf(A) + C$ and $\text{val}(A) := \text{val}(A) + C$ (again, this is the case where C is positive). Following this, the Escrow journal is deleted.

2.2.2 *Example.* In the example that follows is a time line of ESCROW Requests, commits, and aborts. The example is simplified in that the commit logs that are written are not detailed, so coverage is limited to changes in the extended field. Recall that the extended field must contain the inf, val, and sup values for the field, a time stamp, and all Escrow journals for updates to the field. A time stamp is assumed that is incremented for every successful ESCROW request, commit, or abort, although of course other time-stamp schemes are possible. Transaction ID numbers are assigned sequentially at transaction start-up.

We start with a layout for the Escrow journal:

LAYOUT OF ESCROW JOURNAL

TRANSACTION ID, POOL ID
TEST CRITERIA: LO, HI
ESCROWED = E | USED = U

The pool ID value is either "P" or "N" according to whether the request creating the current journal is for a positive or negative quantity. Of course this is determined from the sign of the Escrowed quantity, but since other named pools might be desired, the explicit notation is adopted. The LO and HI values specify the range the field is restricted to based on the Escrow test(s) used; note that quantities $-\infty$ and ∞ for LO and HI indicate no restriction.

A time line follows, detailing structure changes to the extended logical field QOH, starting with inf = val = sup when no transactions are outstanding.

LOGICAL FIELD QOH

INF	VAL	SUP
100	100	100
TIME STAMP = 0		

Note that quantities $-\infty$ and ∞ for LO and HI indicate no restriction.

TRANSACTION 1 MAKES REQUEST:
ESCROW(field=QOH,quantity=50,test=(QOH ≥ 0),recover=NO)
USE(field=QOH,quantity=50)

	INF	VAL	SUP
	50	50	100
	TIME STAMP = 1		
ESCROW JOURNAL 1	TRANS #1, P: P LO=0, HI=∞ E = 50, U = 50		

TRANSACTION 2 MAKES REQUEST:
ESCROW(field=QOH,quantity=50,test=(QOH ≥ 20),recover=NO)
* REQUEST FAILS, SINCE RESULT OF SUBTRACTING ANOTHER 50
* WOULD BE INF(QOH) = 0, SO test=(QOH ≥ 20) IS FALSE

TRANSACTION 2 MAKES REQUEST:
ESCROW(field=QOH,quantity=20,test=(QOH ≥ 30),recover=NO)
USE(field=QOH,quantity=20)

	INF	VAL	SUP
	30	30	100
	TIME STAMP = 2		
ESCROW JOURNAL 1	TRANS #1, P: P LO=0, HI=∞ E = 50, U = 50		
ESCROW JOURNAL 2	TRANS #2, P: P LO=30, HI=∞ E = 20, U = 20		

TRANSACTION 1 MAKES REQUEST:
ESCROW(field=QOH,quantity=20,test=(QOH ≥ 0),recover=NO)
* REQUEST FAILS, SINCE RESULT OF SUBTRACTING ANOTHER 20
* WOULD BE INF(QOH) = 10, SO (QOH ≥ 30) OF SECOND ESCROW
* JOURNAL WOULD FAIL

TRANSACTION 3 MAKES REQUEST:
ESCROW(field=QOH,quantity=−30,test=(QOH ≤ 200),recover=NO)
* NEGATIVE QUANTITY CHANGES VALUE OF SUP, NOT INF
USE(field=QOH,quantity=−30)

	INF	VAL	SUP
	30	60	130
	TIME STAMP = 3		
ESCROW JOURNAL 1	TRANS #1, P: P LO=0, HI=∞ E = 50, U = 50		
ESCROW JOURNAL 2	TRANS #2, P: P LO=30, HI=∞ E = 20, U = 20		
ESCROW JOURNAL 3	TRANS #3, P: N LO=−∞, HI=200 E = −30, U = −30		

TRANSACTION 1 COMMITS

	INF	VAL	SUP
	30	60	80
	TIME STAMP = 4		
ESCROW JOURNAL 1	TRANS #2, P: P LO=30, HI=∞ E = 20, U = 20		
ESCROW JOURNAL 2	TRANS #3, P: N LO=−∞, HI=200 E = −30, U = −30		

TRANSACTION 2 ABORTS

	INF	VAL	SUP
	50	80	80
	TIME STAMP = 5		
ESCROW JOURNAL 2	TRANS #3, P: N LO=−∞, HI=200 E = −30, U = −30		

TRANSACTION 3 COMMITS

INF	VAL	SUP
80	80	80
TIME STAMP = 6		

2.2.3 *Maintaining Earlier Test Conditions.* As noted in [9], a difficulty arises if the test condition of an earlier but still live transaction is made invalid by a modification (incremental change) of a later transaction. The problem, of course,

is that the earlier ESCROW request can no longer be commuted with the later one, and so a nonserializable schedule may be created (as soon as one transaction must precede another, it is easy to construct a deadlock involving two Escrow fields). The solution to this, as indicated above, is to disallow an ESCROW request that invalidates the test condition of an earlier transaction. (Note however that it is *not* meant to constrain the values of a field based on ESCROW requests with test conditions on inf(A), val(A), or sup(A). These tests are only for the information of the programmer in determining what to do if a normal ESCROW request on a quantity A fails. They are not guaranteed for any length of time thereafter.)

If the Escrow field A has Escrow journals for outstanding transactions T1, T2, ..., TJ, ..., TN, denote by (low(J, A), high(J, A)) the interval to which A is restricted in order to keep the test for transaction J valid. For example, the condition "test=($A \geq 12$)" would result in the interval (low(J, A), high(J, A)) equal to (12, infinity); more tests by that same transaction might further restrict the interval. Now, while the transactions T1 through TN remain live, we want inf(A), the lowest value that A could take on after any set of updates, to be bounded below by MAX(low(J, A)) where J runs from 1 to N (we denote this by low(A) for short). That is, the validity of the most constraining lower bound, MAX(low(J, A)) is to be maintained. Similarly we want sup(A) to be bounded above by MIN(high(J, A)) where $J = 1 \ldots N$, or high(A). These "constraints" resulting from successful ESCROW requests will cause new ESCROW requests to fail, even when they have a successful test=(condition) of their own, if the Escrow change to the field in question would cause an earlier Escrow test to fail.

Note carefully that the condition of an ESCROW request is only later maintained as a constraint if the request condition was successful; the duration of the constraint is the duration of the transaction that contained that ESCROW request.

To implement these (low(J, A), high (J, A)) constraints, we can maintain a sort order using some sort of balanced tree scheme on the low values and separately for the high values of the Escrow journals associated with the field. When any new ESCROW request is made, the value low(A) can be read by direct access off the low tree, the largest value on the low tree; similarly, the value high(A) can be read off the high tree. When an ESCROW request is allowed, the test=(condition) of the request is translated to a finite lower or upper bound and is placed in the appropriate tree. When a transaction commits or aborts, its Escrow journal will be removed from the extended logical field, and associated values from the low value and high value trees can be simultaneously deleted. Naturally, a tree data structure of this kind is only appropriate if the design envisions numerous Escrow journals outstanding at the same time.

Note that, for many applications, the low(A) and high(A) constraints are better set by the Database Administrator (DBA) since the logical constraints are common to all applications and a single value to test will avoid possible program errors. For example, a QOH for some commodity may not fall below zero; at the same time, the aggregate QOH for some record should not (through vendor orders and order returns) go above the quantity MAX-QOH that can be stored in the relevant warehouse. There is a real question as to whether ad hoc tests should be allowed by individual program requests: the implementation becomes more

complex, and since individual constraints lose their effect when the transactions commit or abort, there will be no long-lived global constraint on any field. This seems like an unusual state of affairs, and the flexibility hardly seems worth it. Even if the user freedom to further test an Escrow field were allowed, some method of placing global high and low values should also be present so that a new program with (inappropriate) weak constraints cannot damage the database of a large inventory system.

2.3 Recovery of Escrow Updates of Aggregate Fields

When a transaction commits, all the Escrow journals created for the transaction are taken out of their extended logical fields and written to a transaction commit log on stable storage, together with a time stamp for the time when the transaction has completed. When the write is complete, the commit has been performed. In this section it is demonstrated how recovery from memory or media failure can be achieved for the Escrow Method in the case of aggregate fields. Note first that media recovery is easily handled: the archive copy of the database is put in place, and commit logs are applied up through the last one written. This is the same method used with classical transactional schemes. Special care is needed to recover intermediate results requested with the "recover=YES" option, and the prescription below must be followed for memory crash recovery in that case. Redundancy will be specified in writing extended logical fields to stable storage so that a copy can be found of this needed structure even in the case of media failure.

In what follows, a method is demonstrated for recovering from a memory crash to recapture completed transactions and roll back transactions that were incomplete at the time of failure; obviously, loss of memory could be treated in the same way that media failure is, but it can be hoped that a much less time-consuming method than applying all logs written since the last archive can be found. After treating completed transactions, the recovery of intermediate updates that have been made by an ESCROW request with the parameter "recover=YES" will be demonstrated. The many questions associated with the proper general interface to use so that applications can recover intermediate results would be material for another large paper. Here, it is simply indicated how the database recovery request might be honored, and other questions are left for later research.

The basic idea for memory crash recovery comes from the realization that the extended logical field structure, including the inf, val, and sup values and all extant Escrow journals, contains all the information of classical beforeimages and afterimages. Nondestructive writes to stable storage can be guaranteed by alternating updates of logical fields to two nonoverlapping areas; in this way, there is always a "good" stable copy. In addition, one good copy will be guaranteed for most cases of media failure. We wish to permit logical field updates to stable storage to be performed in a manner that is not synchronized with individual transactions, in a manner that is sensitive to channel capacity and does not act as a bottleneck to transactional throughput, but that still assures that updates do not collect in memory for excessive periods before being written out. This requirement is made more precise below.

Only two guarantees are needed to be able to perform crash recovery: (1) transactional logs are written to stable storage at the end of each transaction; and (2) checkpoints are taken from time to time that name a transaction number log from which to start recovery—they guarantee that all fields on stable storage are up-to-date as regards updates of transactions that had completed before that time. To clarify the second point, assume there is a guarantee at any time as transactions are being processed that all updates to extended logical fields more than two minutes old are up-to-date on stable storage—the fields updated are written out. What this means, in particular in the Escrow case, is that all Escrow journals that exist on the extended logical field on stable storage are associated either with transactions that have completed in that last two minutes or have not completed at all. (No earlier Escrow journals exist because they are deleted from the extended logical field when each transaction completes.) With this guarantee, a checkpoint can be written that stipulates that, if memory is lost, recovery should start with the commit log that was written two minutes ago. Then all later updates that went to commit logs will be considered during recovery; possibly some commit logs considered will be superfluous in that all fields updated were written to stable storage since the commit occurred, but this is something that can be dealt with as long as there is no missing information.

In performing recovery following a memory crash, take the most recent stable versions of extended logical fields, and start to process commit logs from the point indicated in the last checkpoint. If a commit log indicates a change to an extended logical field that matches an Escrow journal in the field itself, apply the change and discard the Escrow journal from the logical field. If the commit log has a greater quantity used than the Escrow journal indicates (the "match" was for the transaction number and pool ID, not the quantity), it is presumed that the commit log has the correct quantity. If a commit log indicates a change to an extended logical field that does *not* match an Escrow journal in the field itself, there are two possibilities: The first is that the transaction had already completed when the field was written out to stable storage; the second is that the transaction in question had not performed the update to the field at that time. To decide between these two alternatives, it is only necessary to compare the two time stamps for the field update and the transaction commit log. If the commit log has an earlier time stamp than the extended logical field update, then the field update in question is simply discarded since it must previously have been applied. If the commit log has a later time stamp, then it had not already completed at the time the field update was written, so the field update of the commit log is applied to the field in question, and no Escrow journal is created or deleted. When all commit logs have been processed, all remaining Escrow journals are deleted from the extended logical field and the inf, val, and sup are updated as though the associated requests had been aborted. Following this, normal transactional processing may resume.

THEOREM. *The recovery process outlined is valid if there are no intermediate recoverable results.*

Consider the following cases involving the times when an extended logical field in the database was written out to stable storage (FW), the time when a specific transaction first updated the field in question (TU), and the time when the

transaction committed (TC). It is clear that the event (TU) must have preceded (TC), that is, (TU) < (TC), but it is also possible that a commit did not occur prior to the crash (either the transaction was still in process or else it aborted.) Therefore we see the following possible cases:

$$(TU) < (TC) < (FW) \tag{1}$$

By the definition in Section 2.2.1, the logical extended field has been updated and the Escrow journal discarded at time (TC), prior to the time that the field was written out. Therefore an Escrow field will not be found during recovery to match the commit log, the time stamps will be checked to find that the transaction in question had completed at the time the field was written, and the commit log will be correctly discarded as already applied.

$$(TU) < (FW) < (TC) \tag{2}$$

By the definitions, there has been an Escrow journal created associated with an update performed by the transaction in question. During recovery a commit log will be found for the transaction detailing an update to the field—since the Escrow journal might be out-of-date, the more recent commit log for the quantity to apply is taken, and the Escrow journal from the extended logical field is deleted. (The possibility of two different Escrow journals in a single transaction, one involving a positive and one a negative Escrow quantity, indicates a need for separate commit logs that is differentiated by the Escrow pool identifier in the Escrow journal. Then finding only one of the Escrow journals present during recovery indicates that there are two different (TU) times to be examined, one for each type of Escrow, and that there is a mixed case with (3) below.)

$$(FW) < (TU) < (TC) \tag{3}$$

During recovery it is found that a commit log exists for which there is no corresponding Escrow journal, and that the transaction in question had not completed at the time the extended logical field had been written. Therefore, the commit update is applied, and Escrow journals from the logical field are neither added nor deleted.

$$(TU) < (FW), \quad (TC) \text{ did not occur} \tag{4}$$

During recovery no commit log is found to correspond with the Escrow journal created at time (TU). The transaction was in train at the time of the crash and must be rolled back, or else it was aborted after the field write, with the same result. The Escrow journal is deleted and treated as an aborted transaction in setting the field values after all the commit logs have been processed.

$$(FW) < (TU), \quad (TC) \text{ did not occur} \tag{5}$$

No Escrow journal is found with the extended logical field since the field write occurred too early. No commit log is found since no commit occurred. Recovery never learns of this update event, with the effect that the incomplete transaction is properly rolled back.

Note that the checkpoint manager must actually be aware of how long (time stamps) various extended logical fields have been in memory without having been written to disk after they have been updated (if they have only been read,

they need not be written). It is assumed that recovery takes some reasonable fraction of normal processing time, and the longer this is allowed to go on, the further back each checkpoint will indicate recovery must start relative to the current time.

Normally one would picture some sort of least recently used algorithm governing the logical extended fields to be written out (freeing up memory space if that is an issue). This has the advantage of a large number of updates for each write to stable storage. However, in situations where the stable storage copy of a field is getting too much out-of-date, the system must have some way of "pushing" the field out, possibly with several priorities, ending with a forced write before any further transactional access is permitted.

To permit recovery of intermediate results, updates requested using an ESCROW request with the parameter setting "recover=YES," the design is varied as follows: Escrow journals that correspond to requests of this form must be specially flagged, and the logical extended field must be written to stable storage immediately, before returning from the request. The recovery manager will recognize Escrow journals with this "recover" flag and will not delete these Escrow journals from the field when it has finished processing commit logs. Thus, the Escrow journals in question will still be in existence, and the field will have the appropriate range.

3. UTILITY AND BENEFITS OF ESCROW TRANSACTIONS

3.1 The Escrow Method Does Not Cause Deadlocks

Since ESCROW requests always return to the caller immediately, without waiting for any event in the case the request cannot at once be satisfied, no deadlock can occur in the pure Escrow case. To be sure, one can decide to try the request again in some edge cases, but a deadlock will not occur without a chance for the application programmer to avoid it by ceasing repeated requests.

An Escrow access can be viewed as locking an incremental quantity of an aggregate field (placing the quantity in Escrow) so that the rest of the field quantity becomes available again to other users. In this sense, the Escrow method is obviously more sophisticated than the standard record locking method, and this point will be referred to in considering distributed transactions.

For now, consider the edge case where an attempt to access an incremental quantity fails under conservative assumptions, but might later succeed if currently live transactions simply do not conclude in the worst possible way. For example, we have the following ESCROW request:

IF ESCROW(field=QOH,quantity=C,test=(QOH $\geq$ 0))
THEN USE(field=QOH,quantity=C) (continue processing . . .)
ELSE ABORT

Of course, if inf(QOH) is currently less than C, the test condition will fail. However, it may be possible that we are being too stringent in our test. If the current set of outstanding transactions is such that val(QOH) $\geq C$, or even if sup(QOH) $\geq C$, we might not want to accept this test failure as final and abort the entire transaction so far. In fact, Reuter [9] had the system wait under

conditions of uncertainty rather than reporting the test failure to the calling program. However, this approach entails the possibility of deadlock. Furthermore, the presence of deadlock can be particularly difficult to detect because, rather than finding a single other transaction blocking access to a desired field as could have been done in the classical record-locking case, there may be several transactions with Escrowed quantities from this field. Quite complex situations can arise, since perhaps no single transaction with these Escrows would release enough of the desired quantity by aborting for the current transaction request to succeed! It has been demonstrated in a separate paper that an algorithm to detect this sort of deadlock is NP-complete. For this reason and the considerations that follow, it seems that appropriate techniques for deadlock avoidance in the Escrow case may be more heuristic in character, such as timeout. In any event, a true deadlock detection scheme must conclude a "possible deadlock" is present in cases where perfect correctness may result in a nonpolynomial algorithm.

This problem of Escrow deadlocks is not as important in real-world applications as it might appear, because of the following consideration: in the normal course of business, it must be rare that an order can be filled only through some combination of aborts and commits of other outstanding transactions. Rather, a long period of success would be expected that is followed by a very short period when edge cases are important, and then another (possibly long period) when the requests fail completely. This is not to say that consideration of such an event can be dispensed with, but rather that it is proper to treat it as an exception. Naturally there are a few applications that seem to consist nearly uniformly of exceptions of this kind: airline reservations, for example. Even so, the event of finding a situation where a WAIT for a field is appropriate is usually rather rare; a deadlock arising from such events is rarer still. Additionally, the situation where a deadlock occurs is entirely different than the one that obtains under classical record locking. In that case, a deadlock can occur whenever records are locked in a cycle, even though there may be plenty of each of the quantities desired by the various transactions. When a transaction is blocked, we hesitate to abort the requesting transaction, since a very good chance of eventual success remains. In the Escrow case, by contrast, we would only find ourself in such an edge condition if it was quite likely that the quantity required by some requesting transaction was exhausted.

When the Escrow quantity has inf and sup values such that a current ESCROW request fails but a later retry of the transaction may prove successful, it is normally sufficient to give warning to the order-entry clerk (or equivalent) that this condition has arisen. The business will then have some policy regarding how hard the clerk should try to get the last quantity from the Escrow field in question. Creating a back order is probably the ultimate effort in this regard: the test can be performed again after most transactions have completed. If the system has some way to notify transactions that have enqueued on an item when it becomes free, so much the better, but we want to enter some form of exception processing in this case; at the very least the waiting transaction should be separated from the waiting terminal. For all these reasons, it seems that the more friendly system behavior is to report failure back to the calling application, but allow for exception handling.

```
IF ESCROW(field=QOH,quantity=C,test=(QOH ≥ 0),recover=NO)
THEN USE(field=QOH,quantity=C) (continue processing . . .)
ELSE
  IF ESCROW(field=QOH,quantity=0,test(sup(QOH) ≥ C),recover=NO)
  THEN perform exception handling.
  ELSE ABORT
```

Recall that the ESCROW request could fail if it brought the interval (inf(A), sup(A)) into a region that made some former test of a live transaction untrue. This condition is entirely comparable in its effects to the first Escrow test above failing, so that the initial ELSE clause will be activated. However, special handling is appropriate in this case, where the MODIFY action would make a former test invalid, so some method of testing for failure due to preexisting constraints should be available to the program.

3.2 High-Concurrency Items Will Not Act as a Bottleneck

This is the major use planned for the Fast Path Method and by Reuter. New functionality may be offered in this regard if it can be demonstrated that the generalized method applies to new field types in common use. More investigation is needed to demonstrate the utility of such field types. See Section 4.

3.3 Human Interaction in the Midst of a Transaction

The most natural desire in the world on the part of a customer is to know "how much it would cost to buy 10 gross of those widgets." But a calculation of this kind can be complex if it involves sliding volume discounts, delivery charges, and sales tax. In addition, the order-entry clerk would like to be able to advise the customer as regards availability—would a back order be necessary? After saying that the quantity is available, it then becomes most embarrassing to try to order the quantity and find that it is not available after all. It does no good to explain that this transaction cannot be run to the desired point and then permit human interaction when the necessary information has been generated; as is well known, the resources involved under classical locking cannot remain locked for such an indeterminate period. But neither the customer nor the clerk understands this point, and the disappearance of the quantity because of a concurrent update transaction seems to them like a bug! Why couldn't we hold this quantity aside for a few moments? In a real-world legal transaction, where the customer might have committed with another person based on the availability of this item, this behavior is unacceptable, and the quantity must be guaranteed by physically placing it in Escrow (buying it and then returning it in the case of an abort).

What the clerk would like to do is enter a tentative order of this sort as if it were actual and read off the resulting information to the customer (price, delivery date, etc.). We do not want to duplicate all the logic of an order-entry transaction for such an inquiry, so the ideal method for doing this would be to run the transaction this far, tell the result to the customer, and wait for a decision. After the customer decides whether the result is acceptable, the clerk either clears (aborts) or accepts (commits) the transaction. Of course it might also be possible to make some small change in the order (e.g., quantity) and try again.

Simple as all this seems, the condition under classical locking that transactions cannot last across terminal interactions makes it extremely difficult to perform in practice. With the Escrow Method, since concurrent access to the records is possible while a transaction is live, the logic for terminal interaction becomes just as simple as it seems. It is simply allowed, as long as only Escrow updates have been performed by the transaction in question so far and they have been performed with the "recover=YES" parameter.

3.4 Distributed Transactions

The most significant problems with distributed transactions involve worst-case scenarios. What happens if the communication line goes down? What if a node goes down after it commits, but the other nodes do not know it has committed? It might seem that transactions are tailor-made to handle situations of this sort: every change made is reversible by an abort, so simply hold all transactions in that state until some agency decides a commit or abort is appropriate; it is clear that sooner or later all nodes will be informed of this decision. Where is the difficulty?

The difficulty of course lies in the unstated assumption that we are in a *hurry*. While a distributed transaction is taking place and cohorts of that transaction remain uncommitted at various nodes, no other transactions can access the records that have been locked. Under normal circumstances, because of the time needed for communication, this might mean seconds elapse without concurrent access. If a line or node goes down, this condition has to be noticed quickly, and something has to be done about transactions that were involved so that other work can get done with the records that were involved. From being a natural form for distributed work, transactions become almost impossible to support in highly concurrent situations when record locking is involved.

For applications where the Escrow method can be used, most of the time pressure is removed. If some incremental quantity is locked in Escrow on a node while a transaction is in limbo, that is an unfortunate happening, but presumably normal work can progress even so. In addition, there is some argument that the Escrow method is the best method that can be applied to this problem.

Example 3.1 Distributed Transactions—A Natural Use of the Escrow Concept. Consider an international corporation about to perform a business transaction on several continents at once under conditions of uncertainty about the communication links involved. What would be thought of a design for the transactions that made the corporate bank account in the United States unavailable while the transaction was in progress? But of course, no one would do it that way. Instead, an Escrow officer at the beginning of the transaction would establish a special Escrow account, withdrawing from the company's general account the funds needed to perform this transaction. Now, if at any time a communication link was lost, or a revolution or hurricane put one of the sites of the transaction into an indeterminate state, the company would be able to continue its usual business, putting off the remote transaction until more normal conditions were restored. Furthermore, it seems clear that the resources of the company that are placed in limbo by the uncertain condition of this transaction

are exactly the minimum ones that were needed; the uncertainty was a part of the decision to do business. The Escrow officer, protecting all parties, will return the sum involved to the company only when it is clear that another party acting in good faith might not have concluded the transaction to the first company's benefit. Assuming a reasonable communications protocol, it is arguable that this design is the best possible, and of course it is mirrored by the Escrow method of transactional consistency.

A distributed transaction using the Escrow method would obey a rather standard two-phase commit protocol. Assume that there is one master transaction at some node N1, and a group of subordinate transactions are created at other nodes. A reason might be that a local supply station is out of a needed commodity so that a remote transaction must be started to transfer some quantity of the item from the remote supply station. The only special precaution needed during the subordinate transaction is to log the Escrow quantity to disk, a recoverable phase one commit using an ESCROW request with the parameter "recover=YES," before notifying the master transaction that we have been successful; thus, if memory is lost at the remote node, this transactional state can be recovered. After the master transaction decides it is ready to commit, it may do so as far as its own record updates are concerned, although it must continue in existence until all subordinate transaction nodes have been notified. Notification consists simply of the message "Transaction ID commits," and the usual method of message acknowledgment will notify the master transaction when all messages have been received and the transaction is complete. Note that there is no hurry at all about sending the commit message. All resources are already appropriately allocated, and finalization can take advantage of slack times on the message links.

If the master transaction decides at any time that an abort is needed, it first aborts the local update increments and then sends a message to all nodes of subordinate transactions, "Transaction ID aborts." When all nodes have acknowledged, the transaction is aborted.

If a remote node crashes after it has acknowledged success of a subordinate transaction, later recovery will recover the abort/commit state, and the master transaction need not even be aware of this.

If the node of the master transaction crashes at any time, recovery must have some record of remote sites where subordinate transactions took place so that they can be sent abort messages until the decision to commit is taken, after which commit messages should be sent; thus, information needs to be logged when the master transaction requests acknowledgment of such remote transaction precommits and when the master transaction makes a commit decision. The various problems of lost messages are quite easily handled with this model, since at every point the various states are robust and reversible.

4. CONCLUSIONS

It is not hard to see why early computer practitioners would think of a record field as something to be copied to a local buffer and then tested and altered in an arbitrary fashion. To limit the programmer to Escrow tests and changes of some field is clearly not a course to be embarked on without well-thought-out

and visible benefits. The major effort involved is in teaching the transactional system how to be an "expert" in transactions of the given type. The benefits that would accrue from the Escrow Method seem very important in applications where they are an issue, and the underlying concept is clearly one that has existed in the normal business world for centuries. In addition, it seems quite possible to adopt this method as a *part* of the more traditional methods of transactional consistency, leaving it to the application programmers to decide what applications might benefit from this new approach.

4.1 Compatibility with Record Locking

It may happen that we wish to perform actions on fields of a record of a different kind where more standard lock types of read and write access are required. For example, we may want to change some character field, an address, or other informational field. It was shown in [9] that a locking policy that disallows mixing of read, write, and TEST/MODIFY access will result in serializable updates. The proof is immediately applicable to the Escrow case. This is hardly surprising since the locking policy amounts to the requirement that no Escrow journals be active, and therefore $\inf(A) = \text{val}(A) = \sup(A)$. It then becomes simple to permit normal locking on the (single-valued) $\text{val}(A)$. Since the Escrow Method permits extremely long-lived transactions, this result may be less useful than it appears. However, it seems possible to extend this result in certain ways, restricting activity in the Escrow journals rather than forbidding their existence while read and write locks are present. This is left as a topic for future research.

4.2 Audit-Balanced Read-Only Transactions

Another example of a different type of transaction on aggregate fields is where we wish to read all the aggregate fields of a database to produce an audit-balanced report. It seems possible to vary the design of the Escrow Method to accommodate such a requirement. Basically, the idea would be to preserve *versions* of the database extending into the past as is done on some Codacyl databases now. The variation would simply require that time-stamped Escrow journals be kept around after the transactions that created them complete, simply flagging the journals as complete; the journals would be kept so long as some audit-balanced transactions display an interest in a slice of the database corresponding to a time-stamp number that precedes the one that created the current journal. Since very long-lived transactions are envisioned, more sophisticated methods may be required to allow a time slice of the database while some transactions are in progress, but this can probably be achieved by inventing new Escrow pools for use while old Escrow pools are frozen.

4.3 Generalized Escrow Field Types

A primary area for future research should be to determine what other kinds of fields and updates might be amenable to the General Escrow Method of concurrency control. A simple but useful example follows:

Consider a quantity that, instead of being "aggregate," or the result of summation, is instead "maxmin," or the result of taking the max or min of a set of quantities. The method of creating Escrow journals for updates arising out of

new values to compare to the current one is immediately obvious. Indeed, such a structure had to be created in the aggregate journaling facility because of the need to keep track of extant tests on bounds for the aggregate quantity. This is also a serious problem for real-life applications as a survey of field types in standard record layouts in reference [8] shows. There were numerous fields with descriptions such as oldest order number unfilled, date of last invoice, date item last received in inventory, etc. The difficulty in a normal transaction of updating such comparative values in a valid way is obvious. It makes one wonder how it was done at all without the Escrow Method.

Another type of field update that would seem to be amenable to the Generalized Escrow Method was suggested by Gawlick. It illustrates the statement made in Section 1.4 that any method for making the guarantee of eventual acceptance is usable. Assume that there are clients who wish to reserve seats on an airline, either window seats, aisle seats or "don't-care" seats. If the seats are actually chosen as the callers ask for reservations, too many window seats might be handed out to "don't-care" reservations and window-seat requests of late callers might not be satisfied; this is the case even though all requests could be satisfied if the "don't-care" seats were apportioned at the end. With the Escrow Method, the only guarantee that needs to be made is that the reservations can be filled (an obvious requirement) and Escrow journals are held for all "don't-care" transactions, apportioning the aisle seats and window seats as they come in. At any time, the reservations can be guaranteed if the total of aisle seats and window seats remaining exceeds the number of "don't-care" seats requested. But with this method, the order in which the requests are received need not be the order in which allocation is performed.

The advantage seen in this resource-allocation problem can obviously be extended to much more complex situations. A characterization of the most general type of allocation problem that can be attacked with the Escrow Method has not yet been made. However, there seems to be a strong similarity between the Escrow Method and the Generalized Banker's Algorithm for allocation of operating-system resources. The Escrow Method must solve a somewhat more general problem in certain ways since (1) resources may be permanently depleted by certain requests, and (2) recovery from nonvolatile storage must be supported. However, the similarity to the Banker's Algorithm certainly hints at a wider field of allocation problems to be treated.

Note that the recovery method presented for the Escrow Method in the aggregate field case in Section 2.3 seems to be reasonably easy to generalize to more general Escrow systems. The key idea lies in the Escrow journals themselves.

4.4 Recoverability of Intermediate Results

The sketch of recoverability of intermediate transactional results at the end of Section 2.3 is meant only to show that a method is envisioned whereby the problem can be solved. Many details remain to be worked out, in particular, as regards a usable application interface for recovering through some set of updates that have been made "safe." This is left for future research.

4.5 Distributed Transactions with Replicated Data

Another possible area of investigation is how to generalize the Escrow Method to distributed transactions in the presence of replicated data. The Escrow Method seems to be extremely useful for one-copy distributed transactions because much of the commit becomes asynchronous. Is it possible to extend this property to the multicopy case, or is greater synchronism required while a consensus is reached?

5. PREVIOUS WORK

Important previous work in this area was performed by Reuter. Reuter took the rather theoretical work of Schlageter [10] and demonstrated several new concepts that would be needed to make such a transactional system actually useful. He developed the idea of interval tests, basing these tests as he said on the IBM Fast Path feature. The current form of the ESCROW request is new, an evolution of Reuter's TEST & MODIFY, which I believe solves some minor problems that had existed. For example, it is unclear from Reuter's paper how the test performed and then put a constraint on modifications of other transactions (the example on his page 89 seems to be a problem arising from this). Further, the TEST & MODIFY action seems to run into difficulty because it tests quantities *before* the MODIFY is applied. The constraint that maintains the truth of this test cannot apply to the transaction that made the test, since then the transaction would not be able to withdraw a quantity C from a field, even though it contained more than C (but less than $2C$). This problem cannot be patched. One way of viewing this difficulty is to say that the test constraints of Reuter seem to be MIN/MAX rather than aggregate, as in the case of our ESCROW requests.

Gray in [5] also refers to the value of commuting UNDO and REDO methods of one transaction with DO operations of others. He points out that this can be achieved where objects are updated only by additions and subtractions where the journal records the delta rather than the old and new values. He also attributes use of this fact to the IMS Fast Path method of reducing lock contention.

After I had written the first version of this paper, the two papers of Gawlick [2, 3] became available to me. It is clear that this field owes a great deal to the theory underlying Fast Path, which was developed by Gawlick and others as early as 1974 and 1975!

ACKNOWLEDGMENTS

I would like to acknowledge the help of Sunil Sarin of CCA who discussed some of these concepts with me and pointed out the important paper of Reuter. My wife, Elizabeth J. O'Neil, identified a problem with the constraints and offered the suggestion of applying the TEST condition as if the MODIFY had already been performed. Several discussions with Gawlick based on an early version of this paper were of great help in focusing the concepts in Escrow transactions to the areas where new power was provided; Gawlick also pointed out a great deal of the generality of the Escrow Method, and I am most grateful for his insights. Reuter made several suggestions, in particular, pointing out the probable difficulty of deadlock detection with transactions of this form.

REFERENCES

1. DATE, C. J. *An Introduction to Database Systems.* Vol. 2. Addison-Wesley, Reading, Mass., 1983.
2. GAWLICK, D. Processing "hot spots" in high performance systems. In *Proceedings of Spring COMPCON 85, 30th IEEE Computer Society International Conference* (San Francisco, Calif.), IEEE, New York, 1985, 249–251.
3. GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS Fast Path. *Bull. IEEE Database Eng. 8,* 2 (June 1985), 3–10.
4. GRAY, J. *Notes on Data Base Operating Systems: Operating Systems—An Advanced Course.* Springer-Verlag, New York, 1979.
5. GRAY, J. The transaction concept: Virtues and limitations. In *Proceedings of the 7th VLDB Conference* (Cannes, France). 1981, pp. 144–154.
6. HAERDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15,* 4 (Dec. 1983), 287–317.
7. IBM. *IBM Program Product General Information Manual GH20-9069-2, IMS/VS Version 1 Fast Path Feature.*
8. KEYDATA CORP. *Keydata Order Entry Service Summary.* Keydata Corp., Watertown, Mass., 1977.
9. REUTER, A. Concurrency on high-traffic data elements. In *ACM Symposium on Principles of Database Systems* (Mar. 1982). ACM, New York, 1982, 83–92.
10. SCHLAGETER, G. *Enhancement of Concurrency in Database Systems by the Use of Special Rollback Methods, Data Base Architecture.* North-Holland, Amsterdam, 1979.
11. ULLMAN, J. D. *Principles of Database Systems.* 2nd ed. Computer Science Press, Rockville, Md., 1982.

Received September 1985; revised September 1986; accepted September 1986