Inform. Systems Vol. 13, No. 2, pp. 155–166, 1988
Printed in Great Britain. All rights reserved

0306-4379/88 $3.00 + 0.00
Copyright © 1988 Pergamon Press plc.

HANDLING HOT SPOT DATA IN DB-SHARING SYSTEMS

THEO HÄRDER†
IBM Almaden Research Laboratory, 650 Harry Road, San Jose, CA 95120-6099, U.S.A.

(*Received* 7 *May* 1987; *in revised form* 26 *November* 1987)

Abstract—On-line transaction systems with high performance demands need a variety of concurrency control methods used for synchronizing data access of even a single transaction. Simple protocols based on strict two-phase locking would not meet their performance goals. This is particularly true for usage patterns of special data elements known as "hot spots".

In this paper, we review various solutions for concurrency control on aggregate data, where the operations to be synchronized commute—at least for certain value ranges. In particular, the escrow mechanism introduced for centralized DBMS is discussed and extended. Our investigations focus on the escrow mechanism for a data sharing environment where transactions running on multiple, independent processors must be efficiently synchronized without sacrificing their serializability. First of all, we propose the use of global escrow services, which may be called asynchronously. Such an optimistic attitude seems to be appropriate, since rejections of escrow operations typically are rare events. The performance of the proposed scheme may be further improved be refining it to a hierarchical escrow scheme with a global escrow and distributed local escrows. Both approaches may be favorably integrated either in a centralized locking scheme or in a primary copy authority scheme for DB-sharing.

1. INTRODUCTION

The transaction concept provides a framework in which to execute a unit of work meaningful for the application environment in an "all or nothing" fashion despite the presence of failures. It allows for multi-user access to the shared database, thereby ensuring its semantic and physical integrity [1]. Nowadays, a database management system (DBMS) is expected to offer the full set of transaction services to the application programmer.

Business applications typically consist of a large number of transaction types, namely reading and updating record sets of varying size, frequency of reference, and usage. Because of such a diversity, transaction processing using a single method of concurrency control e.g. strict two-phase locking [2], would cause serious drawbacks, at least for special resource usage patterns or high traffic situations. This is especially true for large on-line transaction applications with demanding "high performance" requirements. To meet their typical needs, *the use of a variety of methods of concurrency control by a single transaction is reasonable* [3]. Fast Path, for example, provides three methods of concurrency control to optimize access to very active data items sometimes called "high traffic data elements" as well as application journal data (historical data) and to handle less frequently accessed data. This variety of methods was primarily designed to support the execution of short transactions with only a few data references [4]. Longer transactions may require even more methods tailored to their usage modes and reference patterns.

Banking applications as defined by the ET1 benchmark [4] currently represent one of the most time-critical workloads for high performance transaction systems (HPTS). Their dominant transaction type (called DEBIT_CREDIT transaction [5]) may serve as an example to study some of the concurrency control requirements, especially for DB-sharing systems [6] executing in the order of 1KTPS of that type [7]. As far as concurrency control is concerned, we need to synchronize one read/write access to a very large ACCOUNT file, two further read/write accesses to fairly small files called TELLER and BRANCH, and a final write access to a sequential HISTORY file for every transaction. Accessing an arbitrary record from ACCOUNT (out of 10^7) is not critical and may be performed by using standard R–X protocols. TELLER and BRANCH updates (concerning about 10^3 and 10^2 records respectively) are used to keep aggregate information such as "total cash paid", primarily for consistency reasons. These frequent modifications require special concurrency control methods, because strict two-phase locking protocols would cause extremely high lock contention and long processing delays due to the strict serialization needs. The particular reference behavior of the transactions and the relatively few data elements accessed provoke the problem of concurrency control for high traffic data elements. Another problem is created by "high speed" sequential insertion. Since every transaction desires to insert a record at the end of the HISTORY file (chained in LIFO manner to the other records of a given account number), a "hot spot" for free

†Permanent address: Department of Computer Science, University of Kaiserslautern, Postfach 3049, D-6750 Kaiserslautern, West Germany.

placement administration and record insertion must be resolved. (In the sequel, we will use the popular term "hot spot" for high traffic as well as hot spot data elements). Although record locking may relieve the problem to a certain degree, very high transaction rates are dependent on special mechanisms to avoid jam-like situations.

We are going to explore solutions for concurrency control to help alleviate contention of the type illustrated by the DEBIT_CREDIT transaction. The best way in which to attack this problem would be its avoidance by an appropriate DB-schema design [7]. Although highly recommendable, it may not be achievable at any rate, since such an approach is sometimes in conflict with other design goals. In this case, suitable concurrency control solutions are mandatory.

For our investigation, we assume a data sharing environment where transactions running on multiple, independent processors must be synchronized without sacrificing serializability, that is, level-3 consistency is guaranteed. Of course, recoverability is implied by these requirements. To approach the problem, we start to investigate the *escrow mechanism* [8] designed for centralized DBMS in detail. The main topic of our paper is an extension and optimization of the escrow mechanism for DB-sharing. We conclude with a summary of our results.

2. SOLUTIONS TO HOT SPOT SYNCHRONIZATION

The ET1 benchmark exemplified that aggregate information about other entities in the database may be anticipated in practical applications. Such "redundant" information typically appears in fields of record types having only a small number of occurrences. Since these fields often serve for some bookkeeping function, they usually have to be modified with extreme frequency. Typical examples of such aggregate field quantities are:

- "quantity on hand" for stock-room management,
- "total cash received" for teller applications,
- "current number of seats available" for flight reservation.

All information kept in those fields could be derived from the remaining data, but because of frequency and cost of such functions it is obviously impractical in large databases.

To observe consistency level-3 defined in [9], serializability has to be assured for all read and write accesses of a transaction to all elements of the database, that is, the result of the transaction is equivalent to its execution in some serial schedule. Hence, as soon as a transaction desires to see or modify the actual value of a data field, it has to be synchronized with all other concurrent transaction on the resp. field, because a read does not commute with any update operation. Therefore, such "strict" operations applied to high traffic data would drastically lower the degree of overall parallelism in a transaction system.

Fortunately, normal transactions are not interested in the actual and precise values of those fields. Usually, they only need to be confirmed that the actual field value A meets some condition, e.g. $L \leqslant A \leqslant U$. For example,

- QOH is used to prohibit negative stock quantities caused by normal transactions and may serve as a trigger to order new items of a particular kind as soon as a limit L is reached,
- some fields may represent repositories for the accumulated sums of money transfers by a specific teller or branch, essentially used for verification of cash requests (enough money available?) or for auditing and consistency checking by special programs as part of given business procedures,
- seat reservation does not require to modify the actual value of A immediately, but only the assurance that eventually x seats ($x \leqslant A$) will be reserved on behalf of the corresponding request.

These hot spot applications have the following properties in common:

- all fields have numerical data types (this property could be generalized to special types which allow for commuting operations);
- read operations on the actual field value can be replaced by a test or verify function on a typically very large key range;
- update operations are incremental ($-x$, $+y$) such that they commute; a preceding range test ensures their proper application.

A key observation derived from these properties is that tailored synchronization mechanisms allow for much more parallelism on such hot spots without sacrificing serializability of transactions. For centralized DBMS, a special implementation and some design proposals are well known.

2.1. *The fast path approach*

A first method was developed and implemented for Main Storage Databases in the Fast Path feature of IMS/VS introduced in 1976 [5]. Performance-critical transaction processing was supported by two new operations to be invoked by the transaction program:

VERIFY field **comp** value1;
CHANGE field **inc/dec** value2.

Using VERIFY, the programmer can test during transaction execution, whether or not a range condition is satisfied, for example, for QOH. The corresponding field is neither locked nor updated by this operation. Depending on the test result, the programmer may decide how to proceed. Updates of those fields are prepared by the transaction using a so-called intention list. During commit processing, all

CHANGE operations were executed thereby locking the fields, testing their values again, and applying the updates if the conditions still hold. Otherwise, the transaction is aborted. An example application could be sketched as follows:

VERIFY QOH $> L + x$
normal transaction processing;

VERIFY QOH $< U$
only, if range checking required;

CHANGE QOH:=QOH $- x$
commit processing.

Note, this approach for handling hot spot data is not transparent to the application program; it has to distinguish between "normal" and "special" updates. Another important property of this concept is that VERIFY does not make any guarantee as to whether or not the later modification will finally succeed.

2.2. *Reuter's method*

The Fast Path implementation was the starting-point of a design proposal by Reuter described in [10] which aimed at an expansion of the original approach. In addition to its regular functions, the centralized concurrency control component (C^4, e.g. lock manager) provides two new operations. Their basic functions are:

- TEST access: a test operation evaluates the current field value as to whether or not it is contained in a closed user-specified interval $[L, U]$. Since a number of concurrent updates on the corresponding field may not be committed while the test predicate is evaluated, C^4 keeps an *uncertainty interval* $[LV, UV]$ for the current field value V. It calculates the truth value of the predicate to be evaluated according to given rules regarding this fuzziness, that is, $[LV, UV]$ must be completely contained in $[L, U]$ for a TRUE result. Otherwise, if $[LV, UV]$ overlaps either L or U or even both, the predicate is not satisfied by V. TEST is more powerful than VERIFY, since closed intervals can be specified directly and since it may be combined with MODIFY.

- Modify access: a modify operation increments or decrements the current field value V (subject to a positive TEST) on behalf of a transaction T without prohibiting modify operations of other transactions while T is uncommitted. A later abort of T causes an inverse operation, which is unique and without side-effects to concurrent modifications of other transactions. A MODIFY later aborted may have had only some indirect influence on test operations of other transactions, since it temporarily increased the uncertainty interval thereby potentially provoking some unsuccessful tests or wait situations.

An uncertainty interval reflects the range of possible outcomes for a value at a given point in time when any, some or all uncommitted transaction would be aborted. The current value V contains all modifications of committed and uncommitted transactions up-to-date, whereas LV and UV are calculated to reflect the worst cases of the outcome of uncommitted transactions. Hence, if all transactions are committed, $LV = V = UV$ holds which is the actual value A (as assumed below with a starting $A = 20$). Since handling uncertainty intervals is important for the following, we try to explain the problem by using a short example:

Table 1. Handling an uncertainty interval

T_1	T_2	T_3	T_4	LV	V	UV
				20	20	20
−5				15	15	20
	−3			12	12	20
		+4		12	16	24
com				12	16	19
		com		16	16	19
			−6	10	10	19
	com			10	10	16
			abort	16	16	16

It is important to realize that the actual value A can only be determined when all transactions are committed ($A = V$).

Obviously, TEST and MODIFY should be uninterruptable by concurrent modify operations when the field is actually updated. Therefore, they are usually meant to be executed together in a TEST&MODIFY request, which could be described in its effect, for example, in the following form:

if **TEST** $(\text{QOH} + x, [L, U])$
then **MODIFY** QOH $- x$.

MODIFY applies its increment immediately to the aggregate field value V. Since a transaction's increment remains "provisional" until EOT, it expands the range of uncertainty of V. However, this uncertainty only concerns testing transactions in typically rare cases where the range boundaries are approached by the uncertainty interval. After successful MODIFY the update of V itself is guaranteed by the system, leaving the transaction the unilateral right to abort until EOT. As opposed to that procedure, VERIFY of Fast Path permits several transactions to check the current field value concurrently, which may turn out to be an old copy when the actual CHANGE is performed as part of commit processing. Hence, a second (implicit) test has to be applied which may result in a system-enforced abort.

2.3. *The escrow paradigm*

A further extension and generalization of the particular synchronization method on aggregate field values was proposed in [8] as the *General Escrow Transactional Method*. For single incremental operations on aggregate field values discussed so far, it represents an almost identical solution compared to Reuter's method, as will be seen below. To illustrate the underlying paradigm, we try to present a short summary of the abstract concept. Its basic ideas are the following:

(1) aggregate field quantities requiring range restrictions and involving commutable, incremental changes (+, −) are designated to be of *escrow type* (some further types may be added to this class as well);
(2) before an *escrow request* of a transaction is accepted by the escrow component, that is, before it returns a "done" message, it checks whether or not it will be eventually able to successfully perform this request. Accepting a request implies the guarantee that the update can be propagated to the field value *at any time in the future, in any order, and with any subset of updates for which this guarantee has already been made.* Note, this guarantee includes the assurance that an update of an accepted escrow request may or may not be eventually performed, whichever is appropriate;
(3) to make the acceptance of an escrow request crash-resistant, an appropriate *escrow log* is created by the escrow;
(4) when a transaction eventually commits or aborts, the associated escrow log is used to propagate the new field value or its corresponding entries are discarded.

This abstract specification tries to avoid any reference to a conceivable implementation. Compared to the proposals discussed earlier, this description illustrates the idea more clearly that the escrow equipped with global "expert" knowledge concerning state and accepted operations on the corresponding field, may act as a trustworthy mediator for the concurrent requests on the field value among transactions. No direct reference to the resp. field (neither read nor write) is allowed. Updates of committed transactions are guaranteed to be eventually propagated by the escrow. It should be pointed out that neither the method nor the time nor the propagation is specified. Hence, synchronous or asynchronous write operations could be used (not necessarily before or at EOT).

The principal difference to the approaches discussed so far, is the separation of the reservation of a requested quantity as the result of a positive test and the use of this quantity. At first, the programmer asks the escrow to put a quantity "in escrow" for later use. Then he can use this quantity "in escrow" whenever appropriate. Hence, the separation of the request and use actions offers the advantage of reserving the entire eventually needed quantity at a time and of requesting it in portions when they are used. Hence, the escrowed quantity may be used at once, processed by multiple use requests, or may even be only partially used. (A real world example is the reservation of travel expenses and the taking of one or more advances on these travel expenses). Of course, the quantity put "in escrow" cannot be exceeded by the sum of all use requests. Quantities or parts of them not used are released from the "in escrow" state (escrow pool) at the end of the requesting transaction. When a transaction aborts, all its requested quantities are released.

The syntactical form taken from [8] will help to illustrate how the escrow paradigm is reflected at the programmer interface. It clarifies the test and use operations on escrow fields of the aggregate type. An escrow request has the form:

```
if   ESCROW [field = F1,
            quantity = C1, test = (condition)]
then  "continue with normal processing"
else  "perform exception handling".
```

Note, a test parameter is part of the escrow interface. Hence, the user is requested to specify a range condition for the test. In order to satisfy the test condition, $C1$ is added/subtracted to the $F1$-value (depending on whether a quantity is added, e.g. deposit of money, or removed, e.g. withdrawal of money) before the test is performed. A positive test puts the requested quantity "in escrow" for safe and unrestricted use by the transaction. To use a reserved quantity, the following operation may be applied one or several times:

USE (field = F1, quantity = C2).

Both constructs describe the escrow interface according to [8] as seen by the transaction program.

3. PROPOSAL FOR AN ESCROW MECHANISM

An important observation is the fact that the given interface for escrow requests might interfere with responsibilities of the escrow. It implies that the programmer specifies a range test for the actual field value to ensure its consistency. (We feel that the range test is used to check an integrity constraint on the field value in most cases). An integrity constraint specified "manually", however, causes an unnecessary dependency of its correctness on the transaction program. What happens if incorrect or contradictory conditions w.r.t. globally defined ranges are used by individual transaction programs, e.g. QOH = −100?

A key idea in the context of DBMS is the system enforced control of integrity constraints. In order to avoid contradicting or inconsistent specifications of conditions on escrow fields, such globally valid integrity constraints should be exclusively controlled by the escrow itself, and should not be dictated by the individual programmer. Hence, the globally valid integrity constraint should be part of the field definition and publicly accessible in the database schema. As a consequence, the test option in the ESCROW command is not needed anymore.

3.1. A refined escrow mechanism

This observation leads to a simplification of the ESCROW command. In a typical escrow request we only need to specify the quantity to be put in escrow which may be expressed by the "grantable" option. This option should be sufficient for (almost) all uses. Therefore, we propose the following standard form of an escrow request:

```
if  ESCROW (field = F1, grantable = C1)
    then  "continue with normal processing"
    else  "perform exception handling".
```

The escrow accepts this request if it can safely guarantee $C1$ quantities given the current uncertainty interval. Otherwise, a message indicating the cause of rejection is returned upon which the transaction may retry later.

For exceptional cases, where the programmer desires to control the range of the current value more closely, we suggest the use of a special "restrict" option which is entirely programmer-controlled. The escrow will check and satisfy this private "integrity constraint" as long as it does not violate the global integrity constraint. This extra service will cause additional overhead; therefore, it should be only used in cases where it is really needed. The corresponding form of such an escrow request with "manual" integrity control could be:

```
if  ESCROWR [field = F1,
             grantable = C1,
             restrict = (condition)]
    then  "continue with normal processing"
    else  "perform exception handling".
```

The escrow request is accepted if $C1$ quantities can be granted without violating the restrict condition and if, in turn, the restrict condition does not violate the global integrity constraint of the escrow field. In case of a rejection, the transaction may relax the restrict condition or retry later. We assume that this kind of manual control is not very important in a database context. Moreover, it seems to be rather meaningless in highly concurrent situations (1KTPS). Therefore, we do not elaborate on the related problems and consequences in detail.

The USE operation may be taken in its proposed form. To make a generalized escrow mechanism available, it would be desirable to have a function for releasing (part of) quantities "in escrow", at least for its usage in longer transactions. A programmer might request a certain quantity to put "in escrow" before he actually knows his real demand, which may be dependent on other field values to be accessed. When he detects that this request was too optimistic, he should have a convenient way to return/remove quantities not needed before EOT. On the other hand, a request which turns out to be too pessimistic could be easily corrected by another escrow request. When a transaction requires multiple escrow request for its operation, it may happen that some of them are granted before one is rejected. For these situations, it is advantageous to endow a transaction with some mechanism to react more flexibly. Hence, the return of such quantities without aborting the transaction is another use of a RELEASE operation. Therefore, we propose a simple extension of the following form:

```
RELEASE (field = F1, quantity = C3).
```

A transaction can give back only granted quantities up to the granted amount. However, in some situations it would be useful to distinguish between "already used" and "not used" quantities. Since we don't want to burden the escrow interface by such a distinction, RELEASE is used to return quantities (including both kinds).

In short performance-critical transactions, typically one USE operation might be expected per escrow type referenced, since only a simple "unit of work" is performed. Therefore, a natural way of programming would be to put the resp. quantity "in escrow" and to immediately use it, e.g. modify QOH. Such a combined escrow request is conveniently expressed as follows:

```
if  ESCROW (field = QOH, grantable = C)
    then  USE (field = QOH, quantity = C)
    else  ABORT.
```

The escrow will honor this request when QOH $> = C$ holds before "accept".

The most important use of the escrow method is for synchronizing incremental operations on aggregate field values, by exploiting their commutivity property. Its benefit comes from the fact that in most cases further updates to such fields can be performed while the first one (and other accepted ones) still remains pending. Note, the refined escrow mechanism not only increases data integrity and flexible operation, but also facilitates the use of the underlying concept in a distributed environment (see Section 4).

The price to pay is a special treatment of escrow requests at the DB-programming interface. Sufficient (semantic) information has to be made available for the escrow whereas operations on other data just use (syntactic) R–X lock protocols. Therefore, removing a hot spot detected in an operational system by use of the escrow mechanism requires an adjustment of the application programs.

After this discussion, it becomes apparent that the refined escrow method is more flexible for general usage, but not *very different in flavor from Fast Path and Reuter's Method for a single USE request* on an aggregate field. O'Neil claims in [8] that one slight difference concerns the way the test is performed.

Escrow tests will report to the user a probable failure when the test result is only "possibly true" (depending on the behavior of uncommitted transactions). Reuter's method attempts to let the requesting transaction wait until the uncertainty has been resolved which may cause deadlock situations where escrow fields are involved. Again, this seems to be a minor issue.

However, there are some deeper differences in the various approaches. As mentioned earlier, Reuter's method allows for regular read and write accesses to an aggregate field, which must be synchronized to guarantee serializability. Such accesses to escrow type fields may be harder to achieve if the method is used in its generalized form. Longer transactions and multiple USE requests, do not seem to easily fit together with additional R–X lock protocols on escrow fields in practical applications. Since Reuter focuses exclusively on short transactions similar to the DEBIT_CREDIT type, it may be acceptable to lower parallelism on those fields (to one) in order to get an actual view of their committed values. On the other hand, the generalized escrow method tends to be general enough to hold more of some quantity "in escrow" than will be used immediately (advance request on travel expenses). This problem raises the question of long term escrow transactions which is not discussed further. In such applications, it does not seem to be reasonable to provide regular read and write access to escrow fields, since fluctuation of their values makes them "fuzzy" for most of the time. The exact value of those fields is usually needed once a session, once a day, or even once a month (auditing purposes). Therefore, special access functions could be provided.

3.2. *Further use of the escrow approach*

As pointed out in [8] the escrow method properly generalized is not restricted to the control of special operations on aggregate fields when its power to control and allocate resources is fully utilized. Some examples will help to explain its conceivable uses.

An escrow could handle fields containing *extremal* values as well as *aggregate* values. For example, a MIN or a MAX value of a collection of frequently changing values is easily maintained by the same principle; it could be checked and modified if necessary, even under the uncertainty created by the presence of uncommitted transactions.

D. Gawlick suggested that the key property of the escrow concept (the guarantee of eventual propagation of accepted requests) could further be exploited to solve resource allocation problems among sets of on-line transactions more effectively. His illustrative example is on seat reservation for a flight, where customers either book a number of window seats, aisle seats or "don't-care" seats. If we have the freedom to delegate the reservation of these seats to an escrow instead of booking them immediately, some flexibility might be gained to optimize resource allocation. In this case, the escrow method uses the fact that the order in which the requests are accepted may be different from the order in which allocation is performed.

4. A HIERARCHICAL ESCROW MECHANISM FOR DB-SHARING

Thus far, we have described the escrow mechanism and proposed a change of its interface, including an extension of its use. Originally designed for centralized DBMS, it may not lend itself smoothly to distributed applications—at least when frequent escrow services may be required everywhere in the distributed system. Especially, high performance transaction systems based upon DB-sharing architectures would benefit from the escrow idea represented so far only to a limited degree, since hot spot references to a single field may occur from all participating systems. Therefore, we are going to refine and adjust the escrow mechanism.

4.1. *Properties of DB-sharing*

In the following we will investigate the use of the escrow mechanism in high performance transaction systems where multiple DBMS share the database at the disk level—usually called DB-sharing. AMOEBA [6] represents a particular implementation of this type of coupled system architecture. Major reasons for system coupling for DB-applications are availability and performance. Since the number of processor nodes influences the performance of some kinds of cooperation, we assume that a DB-sharing complex typically consists of less than 10 and not of hundreds or even more processors.

Such architectures require the cooperation of "independent" DBMS running on multiple computers. Since every system has access to all data, there is no need for data partitioning among the DBMS. As a consequence, a DBMS is able to execute an entire transaction locally, that is, there is no need for function shipping or invocation of "remote" subtransactions. On the other hand, effective load partitioning is mandatory to help balance the usage of system resources. These properties of data sharing strongly influence the performance of concurrency control algorithms, since a share of remote data and lock requests has to be anticipated and since only local ones are desirable.

Every system is assumed to have its own local lock manager, which has to communicate with other local or global lock managers via messages across a high bandwidth network. The message system is typically much faster than the channels to the disks. Every system also has its own system buffer which implies the need to cope with fully replicated data; the shared disk pool is just the hardcopy of some recent state of the replicated database. A local log file is kept for transaction and node (crash) recovery whereas appropriate redundancy for media recovery is maintained globally.

We assume that the typical workload consists of short transactions similar to the DEBIT_CREDIT type. Hot spot synchronization does not allow for a straightforward implementation using R–X protocols as this would cause far more serious drawbacks than compared to a centralized system. Not only the strict serialization of transactions on a hot spot would have to be taken into account, but also the problems of buffer invalidation and exchange of actual pages, which would degrade response time. Hot spot modification would presumably enforce a frequent fluctuation of the page containing the hot spot data among the various system buffers. A FORCE-to-disk solution [1] is inappropriate for such cases, but a solution where the hot spot page is permanently traveling around via the communication network is equally impractical (bottleneck because of update frequency).

4.2. *A global escrow mechanism for DB-sharing*

Standard lock and update propagation protocols would provoke thrashing situations for hot spot pages, combined with a synchronization bottleneck. Therefore, some special handling for hot spot modification is mandatory in DB-sharing systems. The escrow idea seems to be a good starting-point for investigating an "urgently needed" solution for a DB-sharing complex.

A straightforward take-over of the escrow concept is possible, at least in principle. For every escrow type field, a global escrow is designated and allocated somewhere in the DB-sharing environment. Such a scheme could easily be combined either with a centralized lock manager (CLM) or with a primary copy authority (PCA) approach [11], since individual fields could be assigned to one escrow or another. If all escrow type fields are assigned to a single global escrow, the scheme resembles that of a CLM. Of course, the set of fields could also be partitioned appropriately to reflect load partitioning and allocated to multiple global lock managers at various processor nodes. Hence, this approach would support the idea of PCA. In any case, requests of transaction running on the node of the corresponding global escrow could be treated as in a centralized DBMS.

If the global escrow resides on a different node, a "long" request has to be taken into account for every hot spot access (TEST&USE) of a transaction *T*. In addition, COMMIT or ABORT of *T* must be reported to the escrow (soon) after the corresponding event. Usually this message is not time-critical which allows for buffering or piggybacking with other messages. Nevertheless, long requests are considered to be very expensive and sensitive to the overall performance of a HPTS:

- the duration of a long request is fully contained in the response time of *T*, because it synchronously waits for the answer (activation/deactivation of participating processes on sender/receiver sides, delay of message transfer by e.g. buffered transfers, message handling/queuing and transmission);
- the resulting time delay extends the allocation of *T*'s resources thereby obstructing or blocking other transactions, e.g. lock wait.

Therefore, long requests should be avoided as far as possible. For example, two long requests for hot spot modification of a DEBIT_CREDIT transaction would be intolerable in many applications. The CLM-approach combined with a global escrow seems to possess only a limited potential to optimize locality of requests using the concept of sole interest [11], since hot spot access and sole interest embody contradictory usage patterns. Global escrows integrated in a PCA-scheme exhibit a greater potential to reduce long requests to a hot spot element, since PCA-partitions are assumed to be designed in accordance to load partitioning in order to facilitate locality of requests. Hence, for a range of application, certain advantages may be anticipated by such a scheme as compared to a centralized approach.

Note, optimization considerations such as the idea of an optimistic attitude or asynchronous escrow calls could be taken into account equally well in both approaches. If we assume that escrow requests are mostly successful, the following optimization approach would be promising:

- a transaction asynchronously calls the global escrow (TEST&USE) whenever a hot spot element has to be accessed and continues its work. Since typical transactions are short they may already have finished before the escrow answer arrives. Commit processing, however, must be delayed until the outcome of all escrow requests is known. If positive, it can immediately commit; otherwise, it is forced to abort. When violation of an integrity constraint is the cause of rejecting the escrow request, an instantaneous repetition of the transaction would not be very helpful.

Since high performance requirements dictate group commit in many cases implying some delay for most transactions, buffering of escrow messages at one site could be regarded as well to avoid congestion of communication traffic. Such delays cause no trouble for concurrent transactions when the remaining data items of a transaction are low-usage elements.

4.3. *Why do we need a distributed escrow mechanism?*

The concept of global escrows (combined with CLM or PCA) introduces a number of important improvements in contrast to R–X locking discussed above. Page invalidation and frequent page traveling are totally eliminated. (The corresponding page could remain at the escrow's site). Lock contention on hot spots is avoided in most cases. Although important, these improvements are considered to be insufficient in cases where a substantial fraction of the hot spot requests cannot be performed locally, and when the optimistic variant is not feasible. Note, since many transactions have to access the same hot spot, it is not easy to achieve locality of requests for reasons of load balance, etc. On the other hand, if two hot spots of a transaction are allocated on two processor nodes, perfect locality would be impossible. Therefore, we investigate a hierarchically distributed escrow mechanism as an alternative solution for DB-sharing.

A distributed escrow mechanism which attempts to provide local requests in most cases, no matter how transactions and data are allocated, would exhance performance and simplify load balancing. Since it incorporates a kind of location independence for hot spots, load partitioning and load control will be greatly facilitated as far as hot spot processing is concerned. Without such a scheme, loading partitioning and balancing is strictly dictated by "keeping locality of such references". If these restrictions can simply be removed from the load distribution decisions, more flexible and more effective policies may be found.

For these reasons, we are going to develop a suitable distributed scheme. Here is a sketch of the ideas to fit the escrow mechanism to DB-sharing environments:

- since an escrow is a trustworthy mediator guaranteeing that accepted requests will be eventually propagated at some future point in time, we can avoid modifying the corresponding page for every single request;
- given some local escrow capability, quantities on aggregate fields could be put "in escrow" locally and a collection of local updates to the hot spot in question could be accepted without referring to the underlying page (owned by the global escrow);
- from time to time, bulk propagation of incremental updates to the resp. page could be issued;
- since distributed and concurrent modifications of the hot spot data are performed, some global coordination is necessary. Although, for example token schemes to represent current ownership are conceivable, we consider the idea of having a permanent global owner for every escrow type field as more natural and effective. Hence, we propose a *hierarchical escrow mechanism* where the global escrow distributes "ranges" for which the local escrows can safely handle requests without communication, and coordinates the bulk propagation of the local escrows.

4.4. A hierarchial escrow scheme

In the following, we are going to discuss the idea of a hierarchial escrow mechanism in detail. We propose a two-level application of the escrow idea permitting a high degree of local decisions on escrow requests in typical applications. At the first level, the global escrow guarantees that it is able to "handle" the quantities granted to its local escrows in the appropriate form when required. At the second level, the local escrows use this guarantee to satisfy requests of local customers directly.

Incremental operations on aggregate field values are commutable. Range tests are usually only necessary to prevent boundary violations. In most cases, however, these tests are not critical since ranges are typically broad (e.g. $0 \leqslant \text{QOH} \leqslant \text{MAXCAP}$) or "unlimited" for practical purposes (e.g. an event counter [3]). Given these prerequisites, the following key observation allows for a proper distribution of escrow tests:

- there exists a total ordering of all elements of the type. Then, we may partition a given range into disjoint subranges. A test that is satisfied by a subrange is obviously also satisfied by the full range;
- for simplicity, assume an integer-valued aggregation type with boundaries $[L, U]$. With

$$L < M_1 < M_2 < \ldots < M_n < U,$$

we can find some subrange partitioning:

$$[L, M_1], [M_1 + 1, M_2], \ldots, [M_n + 1, U].$$

If a request can be satisfied by referring to an isolated subrange, say $[M_2 + 1, M_3]$, we can check it without touching the full range $[L, U]$. This observation allows us to distribute quantities to local escrow which may independently grant them to their customers:

- fortunately, we need not apply the subrange distribution directly which would have only very limited use. Every test condition not completely contained in the subrange could not have been checked and had to be passed onto the global escrow. The usual escrow request does not contain the restrict option, asks only for a quantity and does not care where it is taken from $[L, U]$. Therefore, it is sufficient to hand out an amount of quantities C to local escrows, e.g. $(M_3 - M_2 - 1)$, but not range information. Since all quantities granted by the global escrow are safe, the local escrow produces safe accept operations as well, as long as it does not violate the rules, that is, observe the absolute amount.
- for the actual value A, always $L \leqslant A \leqslant U$ holds. $A - L$ quantities may be used "to be given away", that is, they are decremented, whereas $U - A$ quantities can be "received", that is, they are added. Decrementable quantities are denoted by C^- and incrementable ones by C^+. Both represent ranges in the discussed sense. Hence, we may apply our distribution idea to both separately. Then, increments and decrements may be treated and compensated locally within the given quantities;

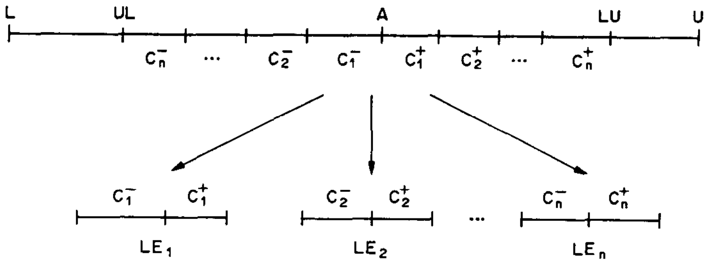

Fig. 1. Quantities drawn from a value range are distributed to local escrows.

- restrict conditions (expected in rare cases) are always passed to the global escrow which may have some spare quantities and safe (pessimistic) range boundaries, e.g. the information about all distributed subranges (see $[L, UL]$ and $[LU, U]$ in Fig. 1).

Figure 1 illustrates the abstract idea of distributing subranges of an aggregate field to local escrows. Note, for their test and use operations it is sufficient to deal with quantities only—we make the distinction between C^+ and C^- to reflect the need of increment and decrement operations. Therefore, these quantities (subranges) are interchangeable when assigned to a local escrow (LE). Each LE may use C^+ and C^- to build its own (absolute) value range which allows for keeping uncertainty intervals locally (see below). The global escrow (GE) keeps the subranges $[L, UL]$ and $[LU, U]$ (including $L = UL$ and/or $LU = U$) reserved as spare quantities. Restrict conditions falling into these subranges are directly satisfiable by the global escrow.

The idea of using subranges and of distributing quantities must be refined depending on the specific application. To clarify the issue, let us discuss conceivable distributions for our three application examples when a global and three local escrows are involved:

(1) *Teller applications*

The range of cash is assumed to be $[0, \infty]$ and the actual cash $A = 50\,\text{K}$. The upper limit means the bank (teller) is allowed to accept unlimited deposits. Note, this mechanism also offers some help to control the amount of deposits. An initial quantity distribution could be as follows:

$$C_g^- = 20\,\text{K} \quad \text{and} \quad C_g^+ = \infty \quad \text{for} \quad \text{GE}$$

$$C_i^- = 10\,\text{K} \quad \text{and} \quad C_i^+ = \infty \quad \text{for} \quad \text{LE}_i (i = 1, 2, 3).$$

(2) *Flight reservation*

Let us assume the range of available seats is $[0, 300]$. If no seats are booked at the beginning, then $A = 300$ and all C_i^+ become 0. Then, a conceivable initial distribution for C_i^- could be 60 for GE and 80 for each LE.

(3) *Stock management*

Let the given situation be 0, 1000 with $A = 700$ implying that the increments and decrements are limited. Since we expect frequent little requests, but few larger deliveries we may assign the GE to handle these deliveries when the current situation of an LE prohibits local acceptance (not enough C^+). Hence, the following initial distribution may be appropriate:

$$C_g^- = 100 \quad \text{and} \quad C_g^+ = 270 \quad \text{for} \quad \text{GE}$$

$$C_i^- = 200 \quad \text{and} \quad C_i^+ = 10 \quad \text{for} \quad \text{LE}_i (i = 1, 2, 3),$$

The last example may help to make clear that local increments and decrements are used to handle uncertainty intervals locally within the bounds of the given C^+ and C^- quantities. For example, LE_1 handles its value range $[0, C_1^- + C_1^+]$ and the corresponding uncertainty interval as follows:

Table 2. Uncertainty interval handled by a local escrow

T_1	T_2	T_3	T_4	T_5	LV	V	UV
					200	200	200
−50					150	150	200
	−30				120	120	200
	com				120	120	170
		+10			120	130	180
			−40		80	90	180
abort					130	140	180
		com			140	140	180
				+20	140	160	200
			com		140	160	160
				abort	140	140	140

LE_1 has $C^- = 140$ and $C^+ = 70$ available after the following sequence of committed requests: −30, +10, −40.

Based on these general observations, the design of a hierarchial escrow scheme may be pursued. Its algorithmic description would require a considerable refinement of the environment expected, e.g. a refined system model for DB-sharing which would exceed our framework of discussion. Instead, we are going to present the guidelines and goals of a protocol for the cooperation of local and global escrows. Since requests to escrows are application-dependent, that is, they vary in frequency, amount of quantities and site of request, it is impossible to determine limits assigned to local escrows in a general way. A responsible person, e.g. the database administrator, has to distribute the initial values; a suitably designed algorithm should quickly react to changes of request patterns, etc.:

(a) Distribution of appropriate ranges only

It does not make sense to distribute subranges of every escrow type field. Appropriate ranges are large enough that local escrows can work for a while without the need of communication to the global escrow. There are no general rules as to what "appropriate" means, since the average amount of quantities requested is application-dependent. For example, when the full range contains 5 quantities you should not try to distribute them among 3 local escrows.

(b) Local escrow requests

A transaction sends a request for a quantity C to its local escrow. If available, the request is accepted and the result is immediately returned to the transaction.

(c) Global escrow requests

If the available quantities are not sufficient, the global escrow is asked for sufficient quantities [ASK(min +)]. Both parties understand that more than the minimum quantities should be delivered—the actual quantities are subject to availability and other distribution considerations, e.g. at the very first request a large start quantity is transferred to a local escrow.

For global escrow requests, the optimistic idea is applicable in principle as discussed in sub-section 4.2. However, the programming interface gets much more complicated. Therefore, we don't want to elaborate this idea.

(d) Distribution of quantities

The global escrow grants some quantities to the requesting local escrow as long as it has some to distribute. Otherwise, it rejects the request which implies that the local escrow also has to reject the transaction's request given its local situation has not changed.

Quantities of committed transactions are added locally, where increments and decrements may compensate each other. As soon as the sum becomes too large, it is transferred to the global escrow which may propagate a "bulk" update of the stored field value. Since such a transfer has reduced the locally available quantities, it should invoke a supply of new quantities.

(e) Redistribution of quantities

The global escrow may collect some information concerning the activity of its local escrows, in order to adapt its distribution policy to their needs. (A very simple scheme would be the distribution of uniform subranges with or without some capacity for later use). In any case, it may happen that some local escrow has exhausted its share while others still have unused quantities. Therefore, it seems necessary to give the global escrow a revoke option for (part of) granted quantities. It sends a REVOKE(max −) to a local escrow which tries to satisfy this request by giving up to 'max' quantities back depending on its individual situation.

(f) Threshold to protect against thrashing and starvation

When the actual aggregate value approaches the L or U limit, it seems reasonable that all requests are directly accepted by the global escrow, in order to use marginal quantities otherwise distributed more efficiently. The global escrow may enforce such a processing mode (as a special option) by withholding requested quantities as soon as a (fuzzy) threshold is approached. Then, the local escrows are advised to pass all local requests to the global escrow (and to give their residual quantities back). New quantities will allow for changing back to the local escrow mode (Step d). Hence, the hierarchical scheme behaves like a purely global escrow scheme for only a critical margin of quantities.

4.5. Special actions of the global escrows

The following aspects are discussed for the distributed scheme. Similar solutions may be applied to the "centralized" scheme.

As explained above, only the global escrow is able to check (absolute) restrict conditions, because an amount of quantities is a relative unit taken from somewhere within a range. Since the global escrow knows the sum of the granted and returned quantities, it is able to calculate an uncertainty interval for the current situation. Of course, the uncertainty interval will be usually larger than in a centralized situation. On the other hand, a high performance transaction system based on a centralized system could also produce large uncertainty intervals when parallelism is high.

If the restrict predicate can be satisfied, the corresponding request is accepted. Otherwise, an appropriate reject message is delivered. If this simple solution is not satisfactory, a special (costly) action could be taken by the global escrow to shrink its current uncertainty interval, e.g. by using the REVOKE function. Once again, such manual tests of a single transaction (which do not correspond to global integrity constraints) are not a prime issue of a DBMS (they are not assumed to be treated as first class citizens). Nevertheless, their result would be very sensitive to other multi-user operations, even with a centralized escrow mechanism (e.g. in the wake of 1000 transactions per sec).

As mentioned earlier, the actual value A of an escrow type field is needed in rare cases, e.g. for inventory control, usually at specific and preplanned points in time. Such read accesses imply serializability of the "special" transaction, that is, all transactions contributed to the actual value A must preceded it in the serialization order. Since all these transactions have modified A, they must have been committed before. The global escrow can enforce this kind of (partial) system quiescence by revoking all quantities in a broadcast action. In such situations, escrow requests should be rejected with a special notification. Another solution would be the blocking of the requesting transactions which, however, may provoke deadlocks and requires resolution measures for the escrow, too.

4.6. *Some aspects of recovery*

We don't want to discuss aspects of fault-tolerance in detail. However, the prime goal of using escrow, namely preventing hot spot data to act as a concurrency control bottleneck, would be compromised if a logging bottleneck were introduced, that is, to apply (extra) synchronous I/O at commit for transaction-related escrow information to a log. A solution to this problem can be sketched as follows:

- escrow use in a centralized DBMS requires escrow information to be added to the transaction's commit record which must be written anyway. At restart, the actual values (A) of escrow type fields are computed using the corresponding log entries. Of course, escrow type fields should be saved periodically, including all contributions of all committed transactions so far. Such checkpoints will limit the restart overhead;
- a global escrow scheme (sub-section 4.2) must take into account crashes of non-escrow nodes as well as the escrow node. Of course, transaction commit should not be made dependent on atomic propagation of updates or log information on the transaction site and the escrow site. Such a solution would require a distributed two-phase commit protocol for every transaction. Therefore, a better approach can be designed as follows: a transaction proceeds as in the case of a centralized DBMS, collecting and adding escrow information (delta-values on escrow type fields) to the transaction's commit record. The global escrow is forced to periodically checkpoint the values of escrow type fields. Restart of a non-escrow node delivers a list of winners (committed transactions) to the global escrow which updates its uncertainty intervals, thereby removing the losers' requests of that particular node. Restart of an escrow node requires the most recent checkpoint information (local), and from each participating node a list of committed transactions not reflected on the checkpoint (aggregated values are sufficient) as well as a list of in-doubt transactions. Hence, the global escrow is able to reconstruct the current states of the uncertainty intervals;
- a hierarchial scheme basically adheres to the same principles as the global scheme with checkpointing and local delta logging in the commit records. Furthermore, if local escrows perform some kind of checkpointing, it should be possible to handle node crashes locally, that is, to reconstruct the uncertainty intervals of a local escrow during restart.

In any case, adding of escrow–related log activity (extra I/O) to the critical transaction path is avoided. Even more efficient recovery schemes appear to be possible for our purpose when safe RAM becomes available [12].

5. CONCLUSIONS

We have presented a discussion of hot spot synchronization in DB-sharing systems for incremental operations on aggregate field values. The focus of this paper has primarily been the escrow concept for which we proposed a modification of its interface and an extension to optimize its use in data sharing environments.

Hot spot values would imply strict serialization in time for all accessing transactions when "standard concurrency control" is applied, e.g. two-phase locking. Fortunately, special operations on these fields don't require such an "encapsulation" to ensure serializability of transactions. In most cases, the escrow mechanism allows for removing typical operations on aggregate field values from synchronization protocols without violating the serializability requirement.

The concept of global escrow may be directly integrated in a synchronization concept for DB-sharing. When locking is used, it is applicable either as part of the CLM or the PCA approach. Compared to R–X protocols, it obtains the following properties/benefits:

- synchronization bottlenecks are removed;
- page thrashing of hot spot pages is prevented;
- one external escrow request per hot spot access is necessary, if the transaction does not run at the escrow site;
- asynchronous escrow requests seem to be a powerful concept to reduce response time, since their acceptance is very likely in most cases;
- a COMMIT/ABORT message is required for the escrow; it is, however, not time-critical.

Locality of escrow requests may be greatly improved by using a hierarchial escrow concept:

- although more complex, it permits local synchronization in typical escrow applications;
- it facilitates load balancing, since hot spots become "location independent";
- it may be combined with centralized escrow requests handled by the global escrow;
- remote escrow requests allow for asynchronous calls as well;

- smooth transitions seem to be achievable from distributed operation to centralized operation when the "escrow situation" makes it advisable.

These benefits of the escrow concept may be obtained for DB-sharing. Other (future) application areas like distributed transactions, nested transactions or multi-step conversational transactions may take advantage of the escrow paradigm as well (even for operations on some "more general" data types). In any case, reduced lock contention and improved overall parallelism may be anticipated.

Acknowledgements—D. Gawlick provided helpful information on the use of concurrency control methods for hot spots. Many of the discussed concepts were derived from problems and solutions described in his papers, as well as papers of P. O'Neil and A. Reuter. I. Narang, J. Palmer and K. Rothermel have read an earlier version of this paper and have contributed to clarify and improve the representation of important issues. The helpful comments of the referees are gratefully acknowledged.

REFERENCES

[1] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* **15(4),** 287–318 (1983).

[2] J. N. Gray. Notes on database operating systems. *Operating Systems—An Advanced Course, Lecture Notes in Computer Science 60* (Edited by Bayer R., Graham R. M. and Seegmueller G.), pp. 393–481. Springer, Berlin (1978).

[3] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS-VS. Tandem Research Rept TR85.6 (1985).

[4] Anon *et al.* A measure of transaction processing power. *Datamation* (April, 1985).

[5] D. Gawlick. Processing "hot spots" in high performance systems. *Proc. IEEE Spring Comput. Conf.*, pp. 249–251. San Francisco (February, 1985).

[6] K. Shoens. The AMOEBA project. *Proc. IEEE Spring Comput. Conf.*, pp. 102–105. San Francisco (February, 1985).

[7] J. Gray, B. Good, D. Gawlick, P. Holman and H. Sammer. One thousand transactions per second. *Proc. IEEE Spring Comput. Conf.*, pp. 96–101. San Francisco (February, 1985).

[8] P. E. O'Neil. The escrow transactional method. *Proc. Int. Workshop on High Performance Transaction Systems*, Asilomar, CA (September, 1985).

[9] J. N. Gray, R. A. Lorie, F. Putzolu and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. *Proc. IFIP Working Conf. on Modelling of Database Management Systems*, 365–394. Freudenstadt, Germany (1976).

[10] A. Reuter. Concurrency on high-traffic data elements. *Proc. Conf. on Principles of Database Systems.* 83–93. Los Angeles, CA (March, 1982).

[11] A. Reuter and K. Shoens. Synchronization in a data sharing environment. IBM Technical Rept, San Jose, CA (1984).

[12] G. Copeland, R. Krishnamurthy and M. Smith. *Recovery using safe RAM.* Technical Rept, MCC, Austin, TX (1986).