High Contention in a Stock Trading Database: A Case Study

Peter Peinl, Andreas Reuter
University of Stuttgart

Harald Sammer
Tandem Computers, Frankfurt

Abstract

Though in general, current database systems adequately support application development and operation for online transaction processing (OLTP), increasing complexity of applications and throughput requirements reveal a number of weaknesses with respect to the data model and implementation techniques used. By presenting the experiences gained from a case study of a large, high volume stock trading system, representative for a broad class of OLTP applications, it is shown, that this particularly holds for dealing with high frequency access to a small number of data elements (hot spots). As a result, we propose extended data types and several novel mechanisms, which are easy to use and highly increase the expressional power of transaction oriented programming, that effectively cope with hot spots. Moreover, their usefulness and their ability to increased parallelism is exemplified by the stock trading application.

1. Functional and operational characteristics

Database systems, especially those of the relational type, are regarded as the basic utility to manage the operational data of an enterprise [1]. Data independence and the system-controlled data integrity facilitate the integration and extension of increasingly complex applications, especially in the online transaction processing (OLTP) field. Though in general, current database systems provide suitable means for OLTP application development and operation, increasing complexity of applications and throughput requirements reveal a number of serious weaknesses in both, the data model and the implementation techniques used [2]. They mainly arise from the inappropriate support for dealing with high frequency access to a small number of data elements present in the majority of applications. Though several proposals, like the introduction of field calls into IMS/FP [3], try to eliminate that problem, we claim that those primitives still do not provide an adequate solution for a wide range of real-life applications, and will describe several novel mechanisms to cope with hot spot data elements.

Our claims stem from observations made in a case study [4], which included a prototype implementation of a large high volume stock trading system, using the features of a standard relational database system. To make our point, the remainder of this section will outline the functional and operational characteristics of this application. Section 2 will provide some insight into the actual implementation, the problems caused by the use of standard data management primitives and the tricks required to circumvent them. Section 3 then will draw some general conclusions from these observations and come up with the definition of novel data management requirements. Thereafter, in section 4, we propose practical solutions for the management of hot spot data, and finally, in section 5, we put these functional extensions into the perspective of an enhanced DBMS.

Fig. 1 illustrates the general structure of the computerized stock trading system, which basically provides three types of service. The user, typically a stock broker, may enter bids to buy or sell a certain number of shares of a specified stock into the system, which then records the bid in a central database. After the insertion of a new bid, the system has to determine, whether that bid might result in a deal, i.e. whether it can find matching bids of the opposite type. If a deal is possible, the system automatically performs it, which entails several book-keeping functions and the appropriate modification of the database. In order to reflect a deal in the database, all the bids involved in the deal have to be deleted, the deal has to be recorded in a trade audit and the brokers concerned have to be informed about the completed deal. For the detailed description in the following we use the relational schema shown in Fig. 2; all the names should be fairly self-explanatory.

The third major function, called notification service in Fig. 1, broadcasts current stock prices to all con-

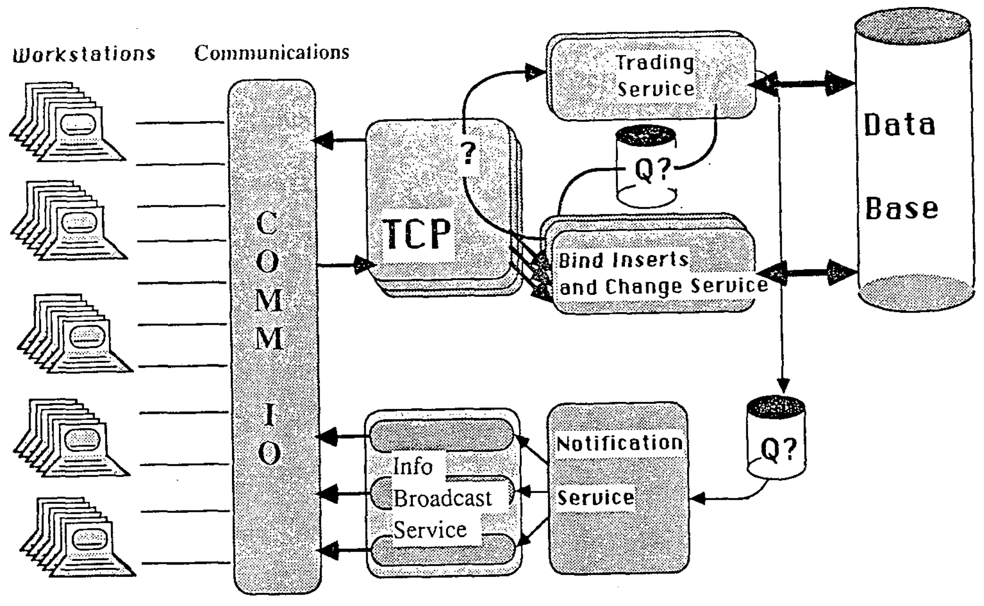

Figure 1: Overview of the computerized stock trading system

BROKER (**Broker-ID**, ...)

STOCK (**Stock-Code**, Highest-Buy-Price, Lowest-Sell-Price, ...)

BID (**Stock-Code, Broker-ID, Bid-No**, Bid-Type, Bid-Price, Bid-Time, Bid-Quantity, Bid-Tag)

DEAL (**Stock-Code, Buy-Broker, Sell-Broker, Deal-No**, ...)

Figure 2: Data structures of stock trading system

nected brokers on a regular basis. Though we will not discuss that component further, because it does not directly relate to our problem, it should at least be mentioned, that the implementation of that service also poses a challenging task. As a matter of fairness, all the brokers have to be notified at approximately the same time with strict bounds on the maximum allowable latency between the delivery to the first and the last broker. Moreover, each broker is permitted to define his own notification profile, specifying what kind of information he actually wants, and those profiles may be altered dynamically. The way in which brokers react to the news broadcast via the notification service, however, strongly affects the trading service. As the broker follows the development of the stock prices on his screens and relates them to the bids issued by himself, he may probably decide to modify some of his previous bids. Hence, bids are modified frequently. From the system's perspective, the change of a bid triggers the same kind of activities as does the issuance of a new bid.

From a strictly functional point of view, it seems quite obvious how to implement the application using standard database services: Each incoming bid triggers a transaction, which first inserts the bid into the database, and then tries to make a deal by examining the outstanding bids for the same stock. If a deal is possible, the corresponding bids have to be removed from the database, the stock price has to be updated, the deals have to be logged in the trade audit, and finally messages have to be sent to the participating brokers, at which point the transaction commits. Alternatively, the transaction is completed as soon as the system realizes the impossibility of a deal.

Unfortunately, this solution will turn out to be a disaster under heavy load. The observed distribution of bids among the available stocks (about 875) is highly skewed. For instance, the 25 most heavily traded stocks (2.86 %) receive roughly half the bids, another 100 stocks (11.4 %) receive an additional 40 % of the total load, and the remaining 750 stocks (85.7 %) share only 10 % of the load. And, of course, heavy trading means many transactions per second per stock. With standard two-phase locking our main approach would make all transactions referring to the same stock run in a strictly serial fashion. In that case, the maximal throughput with respect to a specific stock, irrespective of the number of available processors, obviously is the inverse of the transaction length.

In contrast to the well-known debit-credit transaction [5], a trading transaction is neither short nor simple. In fact, matching sell bids to corresponding buy bids turns out to be a rather complex, and therefore lengthy procedure. To appreciate that, one has to understand the fundamental algorithm of determining stock prices. It is based on three pieces of information associated with each bid. First, there is a price bracket coming with every bid, meaning in the case of a buy order, the highest price the customer is willing to pay, and for a sell order, the lowest acceptable price of a deal. Thus, regarding just that information, a deal should be possible, if the lowest acceptable sell price of all outstanding bids is below the highest acceptable buy price. If the situation were that simple, there would be no big problem. However, each bid also carries a quantitative information, specifying the exact amount of stock to be traded, and a tag indicating, whether or not a **partial** satisfaction of a bid is acceptable. Obviously, the large majority of the bids issued is all-or-nothing, and that is one source of complexity. The second source originates from the requirement of satisfying bids according to an FCFS discipline. Hence, every bid is timestamped, too. The example in Fig. 3 will provide some insight into the operation of the bid matching algorithm, the efficient implementation of which is the cornerstone of an acceptable performance of the whole stock trading application. Fig. 3 graphically sketches two different situations in the trading history of the same stock.

In both diagrams of Fig. 3 the outstanding buy bids are represented by the symbol x, sell bids by o. Prices of the bids are plotted on the x-axis, the volume of a bid on the y-axis, and a bid is registered with its respective maximal (x) or minimal (o) price. Distinct bids at the same price are separated by horizontal bars and are arranged bottom-up in timestamp order. Obviously, there is no overlapping of the price range in the lefthand diagram of Fig. 3, and thus no deal is possible. The righthand diagram shows the status of the trading system after the insertion of two more buy bids at prices $8 and $9. Now the lowest sell price is below the highest buy price and overlapping bids are symbolized by ⊗. Under the assumption that all registered bids are of the all-or-nothing type, a buy bid of 3 shares at maximally $9 may either be matched to sell bids of 6 or 9 shares at $8, and $9, resp. The second buy bid, 4 shares at maximally $8, can only be matched to the sell bid of 6 shares at minimally $8. Since there is no combination of bids that results in a complete matching, no deal is possible. However, the insertion of a buy bid of 2 shares at maximally $9, enables a deal involving 2 buy bids (4 and 2 shares) and one sell bid (6 shares) at a price of $8. To arrive at that solution, all the bids in the price range between $8 and $9 had to be examined.

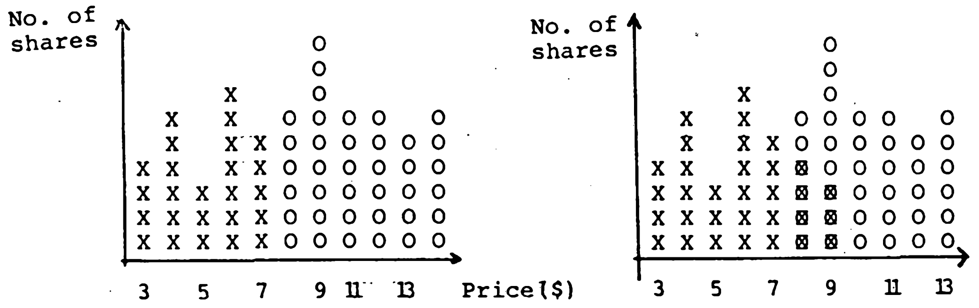

Figure 3: Two different trading scenarios

Since the specific regulations governing the implementation of the matching algorithm, which may even differ slightly from one stock exchange to another, are irrelevant with respect to the database problem, we will not further dwell on it. However, it should be noted that in order to determine the feasibility of a deal after the reception of a new bid, the following operations on the database have to be performed:

First, determine the lowest sell and the highest buy price pertaining to the respective stock. Second, if those intervals overlap, select all the bids in that price range, with their associated information from the database and deliver them to the deal testing algorithm. The latter will make the decision and eventually determine which bids participate in a deal. Certainly, a straightforward standard two-phase locking approach would result in disastrous performance, in particular on the heavily traded stocks. This is easily seen from Fig. 4 which sketches the naive solution. All the activities pertaining to the processing of a bid have to be performed in a single transaction protected

```
Begin-of-transaction
INSERT this bid into BID
SELECT price range from STOCK
IF this bid falls into the current price range or opens a
new one THEN
  {SELECT all bids in price range from STOCK
  try to find a deal
  IF deal is possible THEN
    {DELETE bids participating in deal from BID
    INSERT tuples into DEAL}
  UPDATE price range in STOCK}
COMMIT WORK
```

Figure 4: BIDDING-TRADING-transaction pseudo code in the straightforward solution

by long term locks, effectively serializing bid processing on a per stock basis. However, what can be done using current technology, to alleviate the problem, will be discussed in the following section.

2. The current solution

The only way to build a system fulfilling the performance requirements within the boundaries of current database technology is by the combination of a tricky application program design and an appropriate layout of the logical and the physical schema of the database.

We make the following assumption about the schema shown in Fig. 2: Primary keys are in bold-face, and there is a primary index in the physical schema. In additon to that, we have the following secondary indices on BID: Broker-ID, Bid-No and Stock-Code, Bid-Price, Bid-Time

The flow of control in the application program is summarized in Fig. 5, which in contrast to Fig. 1 focusses on the trading part.

Fig. 5 shows that in the real solution processing of a bid has been split into two distinct transactions. The rationale behind this was to keep transactions and thereby the duration of locks as short as possible to reduce data contention. Moreover, the second part of processing, called trading service in Fig. 5, need not be performed in all cases. As outlined before, the feasibility of a deal depends on the bid's price to fall into the proper range. Now, as preliminary studies pointed out, the vast majority of newly inserted or modified bids (84.4 %) will not affect the critical price range, and hence immediately rule out the possibility of a deal. In that case, the transaction terminates directly after the insertion of the bid tuple. If, on the other hand, the bid's price is in the proper range, the transaction terminates, too, and releases its locks, but at the same time triggers a second transaction to perform the bid matching algorithm on the respective stock. The most time-critical portion within that first transaction, the comparison of the bid's price to the current price range, is accelerated considerably by the maintenance of the highest buy and the lowest sell price as part of the stock tuples. Though that information, of course, is redundant and could be derived dynamically in each situation, its introduction obviates the examination of the BID relation altogether and immensely shortens the decision making process. That advantage, on the other hand, is partly offset by the creation of a crucial hot spot, which every transaction referring to the same stock must at least read, and, as has been pointed out before, the reference distribution is highly biased towards a small number of heavily traded stocks. However, contention can be minimized by performing the comparison as close as possible to the end of the transaction. But if the price range changes due to the new bid, there is no way to avoid an update of the stock record, which may become a bottleneck, unless more flexible synchronization mechanisms are provided by the database system. Fig. 6 summarizes the flow of control in both transactions in pseudo-code.

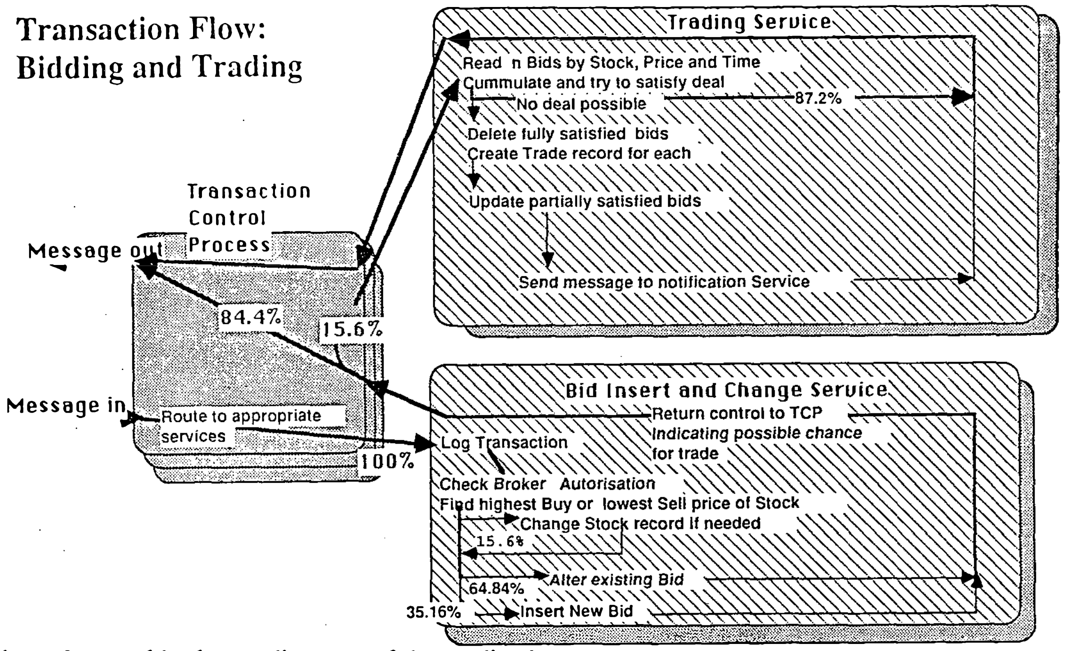
Transaction Flow:
Bidding and Trading
Transaction Control Process
Message out
Message in
Route to appropriate services
84.4%
15.6%
100%
Trading Service
Read n Bids by Stock, Price and Time
Cummulate and try to satisfy deal
No deal possible
87.2%
Delete fully satisfied bids
Create Trade record for each
Update partially satisfied bids
Send message to notification Service
Bid Insert and Change Service
Return control to TCP
Indicating possible chance for trade
Log Transaction
Check Broker Autorisation
Find highest Buy or lowest Sell price of Stock
Change Stock record if needed
15.6%
64.84%
Alter existing Bid
35.16%
Insert New Bid

Figure 5: Flow of control in the trading part of the application

Implementation of the trading part of bid processing is the real problem, because the second transaction touches more data and hence tends to be much longer. Due to the deficiencies of standard synchronization mechanisms, data contention has to be reduced primarily by cautious programming and the exploitation of knowledge about the real load. For instance, it is a well-established fact in that environment, that only a small fraction (12.2 %) of the bids reaching the trading service will finally result in a deal, in other words only 1.9 % of the total requests transform into a deal. This allows for the following strategy: The bid matching algorithm first obtains its input without requesting any locks (the optimistic approach [6]), and repeats its computation under lock protection, if the preliminary analysis indicated a possible deal. Thus, it keeps the number and duration of locks at a minimum (see Fig. 6).

To actually perform the bid matching, the algorithm has to analyze all bids pertaining to the price range specified in the co: ·sponding stock record. To facilitate processing, tl should be further sorted by

```
BIDDING-TA:
Begin-of-transaction
INSERT this bid into BID
SELECT price range from STOCK
IF this bid falls into the current price range or opens
a new one THEN
  {UPDATE price range in STOCK
  trigger trading-ta}
COMMIT WORK

TRADING-TA:
Begin-of-transaction
SELECT price range from STOCK
SELECT all bids in price range from BID
try to find a deal
IF no deal is possible THEN COMMIT WORK
ELSE
  {SELECT price range from STOCK
  SELECT all bids in price range from BID
  compare selected bids to first run
  IF change occurred THEN Restart-of-transaction
  ELSE
    {DELETE bids participating in deal from BID
    INSERT tuples into DEAL
    UPDATE price range in STOCK
    COMMIT WORK }}
```

Figure 6: Pseudo-code of the transaction in the real solution

price and timestamp, which in the current implementation is supported by a secondary index on the BID relation. Thus the data necessary for decision making can be quickly retrieved by first examining the stock tuple and subsequently reading along the index. With those provisions, though succeeding trading transactions on the same stock are effectively serialized in their final phase, the application as a whole meets its performance requirements. Finally, when a deal has taken place, some bookkeeping activity, for instance the deletion of bids, the creation of deal tuples, etc. is performed and the stock tuple is updated, if the price range changed due to the deletion of bids.

3. Novel data management requirements

The previous chapter has demonstrated some application specific request and processing patterns, which are not well-supported by current database systems and therefore have to be handled in the application program by 'handmade' solutions. But our claim is that the problems with this stock trading system indicate some general deficiencies in current relational data models and query languages that should lead to an extension of both the data types in the model and the primitives for accessing and synchronizing them.

Before we start investigating these extensions, let us first identify the unsupported features of our case study in a way which helps us recognize the type of the problem, independent of the given application.

3.1 Types of processing requests

Looking at the problem itself as well as the current implementation, we can identify six properties which are significantly different from those supported by 'state-of-the-art' database systems and their implementation techniques. Here is the list:

a) The two values required to make the guess as to whether or not a deal might be possible are aggregate fields which can be handled by so-called hot-spot synchronization techniques [7,8].

b) While the trading transaction is active, the group of bid tuples it works on becomes a (potential) hot spot. There is no single hot spot element, but a set of objects (identified by a predicate) exhibits this property.

c) For coping with generalized hot spots (which should better be called 'hot sets') the implementation chose to read without locks, or with short locks at best, while doing the analysis. If a deal is possible, it acquires exclusive locks, performs the deal, provided the bid tuples have not changed meanwhile, and commits within a comparatively short time. This is an application of a well-approved Fast-Path-technique [3], which should be available as a general feature.

d) Bids at the same price must be traded in timestamp ordering. This means, in general we need the concept of a queue in our data model.

e) In the implementation, access paths are used for maintaining the processing order of bid tuples. This gives rise to new and more efficient synchronization techniques, because the attributes used for this access path cannot be subject to 'normal' update operations. Rather, they reflect insertions in a queue, and sequences of pop-operations from the top of a stack, which can interfere only in very special situations.

f) Once a bid tuple has been inserted - and the corresponding transaction has committed - a very strict obligation is put on the system: It has to guarantee that this bid will be processed according to its timestamp position as soon as a trade at the bid price is possible. So there are semantic dependencies between transactions (which have to be maintained by the application program, because the system has no means for expressing them), and moreover: The trading transaction must not fail on any bid tuple, because a rollback would imply the tuple not being processed in time stamp order.

Some of these special features have been discussed in the literature occasionally. There are suggestions for handling hot spot data, there are attempts to introduce new datatypes, and there are proposals for binding transactions in order to maintain complex integrity constraints. However, the scope of most of these solutions is too narrow to be readily applicable to our stock trading application. Before we sketch possible solutions, we will go through the item list again and explain, where and why existing proposals are insufficient.

3.2 Where to extend existing proposals

- Requirement a: The methods for allowing a high degree of parallelism on hot spot data [7,8] are designed for **incremental updates** on counters and sum fields. The DO-UNDO-REDO semantics are very simple, and consequently efficient implementations can easily be incorporated into existing systems. An extension to other kinds of aggregations like the max/min operators in our application is not quite as trivial for the UNDO case. Since, however, we will definitely not perform UNDO on the bid tuples (see requirement f)), the obvious extension will work efficiently.
- Requirement b: Hot spot synchronization on dynamically qualified sets of tuples is not possible with the techniques proposed so far.
- Requirement c: The Fast Path field call facility [3] is an optimization for incremental updates on fields with high **update** contention. It does, however, not easily carry over to tuple sets with **inserts** and **deletes** being the only high contention update operations. The general principle is applicable by recognizing that increase in parallelism can be achieved by executing **relative** updates depending on a condition which is very unlikely to be invalidated by the updates. And in fact, incrementing and decrementing a numeric field allows the same degree of parallelism like insertion and deletion of tuples in a set - we only need a simple and general means for deriving and checking the conditions on sets.
- Requirements d/e: Existing data models do not contain queue-type structures, and there are no access structures for implementing them efficiently. From the problem description it is quite obvious that due to the specific operations on queues, there is potential for high parallelism.
- Requirement f: There are papers on transaction scripts [9] for maintaining semantic integrity constraints, and on so-called transaction scenarios [10] for scheduling general transaction execution patterns. None of these schemes, though, can deal with the problem of executing transactions on existing tuples according to their queueing order with guaranteed delivery at the earliest possible moment.

4. Proposed solutions

Since in this paper we cannot discuss functional extensions to solve all the problems mentioned above, we will therefore focus on the generalized hot spot handling.

The generalization of the existing solutions to increment/decrement parallelism with interval tests to max/min parallelism with interval tests is straightforward and left as an exercise to the reader - provided you do not require efficient UNDO. But, as was explained, there is no UNDO in our application for these transactions. Except for this simple extension, we propose three functional enhancements that might help in a variety of situations where there are no fixed hot spots, but dynamically evolving areas of high contention in the database due to lengthy computations like those for finding the trading price.

4.1 The CHECK/REVALIDATE-access

This is a generalization of the Fast-Path-technique. Without caring for any syntactic beauty, one might imagine features like the following added to standard SQL:

- SELECT ..... CHECK <check-expression> AS <check-name>
  The <check-expression> is either a list of values or a predicate on the attributes in the select list, which is evaluated for each result tuple of the query. The system is asked to remember either the attribute values or the predicate value under the <check-name>. This name can be used multiple times in a transaction, thus adding things to be remembered to the set. The key point is that all reads with a CHECK-option will grab no locks. Later on, one can refer to a <check-name> by:
- REVALIDATE <check-name> as an option in an update DML-statement. The update will only be performed if all items checked evaluate to the same result as before. If such a transaction fails, one can issue FORGET <check-name>. As a general feature, it would be desirable to be free to use REVALIDATE in a different transaction than the corresponding CHECK. This is no problem, because the context can be kept by the DBMS. The only question is how the <check-name> gets passed from one transaction to another one.

By the way: The use of the CHECK option identifies a set of tuples as a potential hot-spot item to the system, which can then optimize its buffer and access strategy such that these things are accessible fast, at least until REVALIDATE/FORGET has been issued. The implementation of this general scheme poses no serious problems, and allows to define the current solution in terms of a system-supported interface. Fig. 7 illustrates the basic idea.

4.2 The tuple passing primitive

In many situations, a set of tuples becomes a hot spot because some non-trivial execution is going on on these tuples, which may eventually result in an update on some or all of them, and at the same time other transactions try to ins t tuples into the set or delete

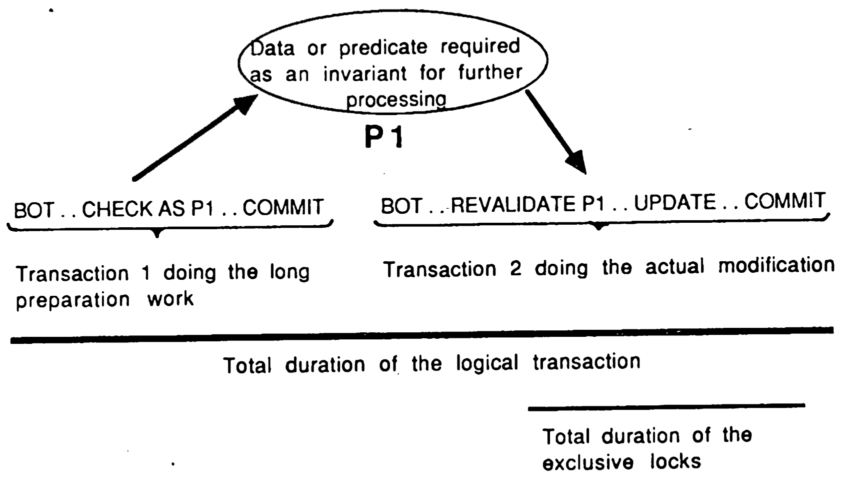
Data or predicate required as an invariant for further processing
P1
BOT . . CHECK AS P1 . . COMMIT
BOT . . REVALIDATE P1 . . UPDATE . . COMMIT
Transaction 1 doing the long preparation work
Transaction 2 doing the actual modification
Total duration of the logical transaction
Total duration of the exclusive locks

Figure 7: Use of the CHECK/REVALIDATE mechanism to bind two transactions into one logical unit

or modify existing ones. If the computation is of moderate length, the revalidate method can be used to support that. If, however, the computation is more or less permanently going on (as can be the case with heavily traded stocks) and the selection predicate of the 'hot set' is being changed due to the updates performed, we would have to do CHECKs and REVALIDATEs at unnecessarily short intervals, which decreases performance due to the overhead caused by the revalidation procedure. In this situation we could make use of a primitive supporting communication among ongoing transactions, as is shown in Fig. 8. The key idea here is to talk to the owner of a hot set rather than trying to access it.

Let us assume the following scenario:

Transaction A is the trading transaction, trying to compute the trading price for a 'hot' stock. Transaction B tries to insert a bid tuple which is in the price range currently being investigated by A. Hence, B is either blocked, or A's eventual revalidation will fail. Now since we know that for each stock, there will be at most one trading transaction running at any instant, we can directly associate that transaction (A in our example) with the hot spot it creates. So rather than storing the bid tuple in the database, running into the synchronization problems mentioned, it could be passed to the transaction working on the hot spot, which will then decide what to do with it. Look at the scenario:

B does an insert in the bid relation. If the bid is outside the current price range, it can be inserted into the database directly. If it is a potential member of the 'hot set', the insertion is automatically translated into a send to A. In either case, B is committed without delay.

On A's side, we need a system-maintained buffer with a special retrieval operation to be used by A, such that it can access those tuples belonging to the current hot set that have arrived during its lifetime. If A wants to process such a tuple, fine; if not, it does the insert for the tuple into the database, which now is no problem, because it holds all locks required to do that.

It is not necessary to elaborate on the language extensions for retrieving from the tuple buffer, it suffices to

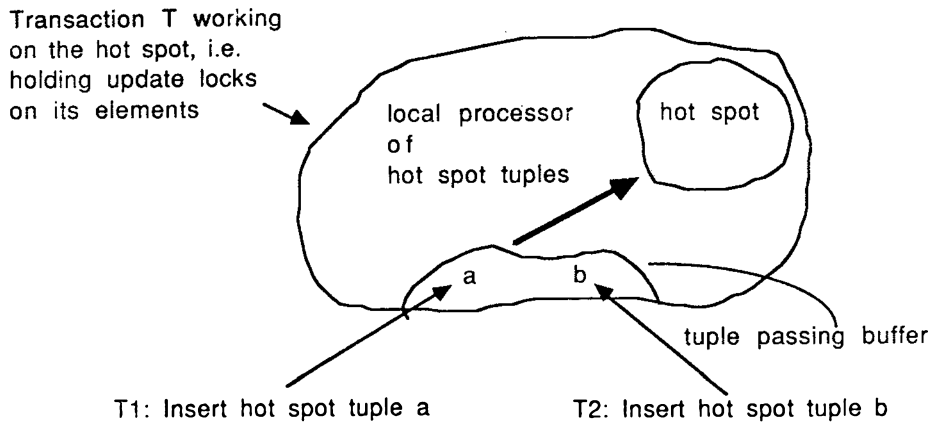
Transaction T working on the hot spot, i.e. holding update locks on its elements
local processor of hot spot tuples
hot spot
a
b
tuple passing buffer
T1: Insert hot spot tuple a
T2: Insert hot spot tuple b

Figure 8: The tuple passing mechanism: Rather than working on hot spot data, transactions communicate with the transaction 'holding' the hot spot

understand that in case of A's commit or abort the contents of the queue is inserted into the database by the system. Note that this extension maintains serializability between A and B-type transactions. The argument for that exceeds the scope of that paper.

4.3 Interval locks on access paths for maintaining queues

Tuples which have to be processed according to some queueing protocol (price and timestamp in our example) exhibit a number of specific properties in terms of request patterns, insert/delete points etc. that call for adequate synchronization support.

First, we have to take into account that queues are dynamic rather than static objects, i.e. they contain subsets of tuples of some relation. There is an application-dependent algorithm determining which tuples are 'in the queue' at any given point in time; the queueing order is more general than FIFO - in our example it is a price range and FIFO within the same price. The operations are simple generalizations of the standard push/pop-operations (see Fig 9.):

Tuples can be removed from the top of the queue; tuples can be inserted into the queue (the position depends on the queueing order); and the queue can be shrunk at the end, i.e. the last n tuples are removed from the queue - which does not mean deletion of these tuples.

Now in our example the queue is used for holding those bid tuples actually under consideration, which means the queue holds the hot set. Only tuples in the queue can be updated for trading, in which case they will be popped. Other tuples outside the queue can be inserted/deleted/updated without any restriction.

As Fig. 9 indicates, this request pattern can be supported by mapping the queueing criteria onto a search key structure. For proper synchronization , we need a key-range lock mechanism which allows the holder of the queue to have update locks on the queue-tuples and supports a number of non-2-phase lock conversions:

- shrink the predicate from left to right (pop)
- shrink the predicate from right to left (remove)
- extend the predicate from right to left (insert at the top of the queue)
- extend the predicate from left to right (insert at the bottom of the queue)

It is fairly easy to demonstrate that for the specific request patterns assumed, these lock conversions will not violate serializability of transactions.

5. Discussion of the solutions proposed

The application of the CHECK/REVALIDATE-mechanism is obvious. It is a simple extension of a whole range of ad-hoc solutions which are currently in use for different purposes:

- The field call facility in IMS/FP is restricted to one transaction.
- In pseudo-conversational transactions, which make heavy use of data invariants, the gathering, passing and testing of the data is left to the application programmer.
- Some TP-monitors have the notion of a general transaction context, which is kept either in special storage areas or in the associated database. But these context areas, apart from some system-defined structures, are simple containers with no system supported relation to any predicate on the database. Furthermore, there is no way to inform the system about forseeable high access frequencies on some of those data.

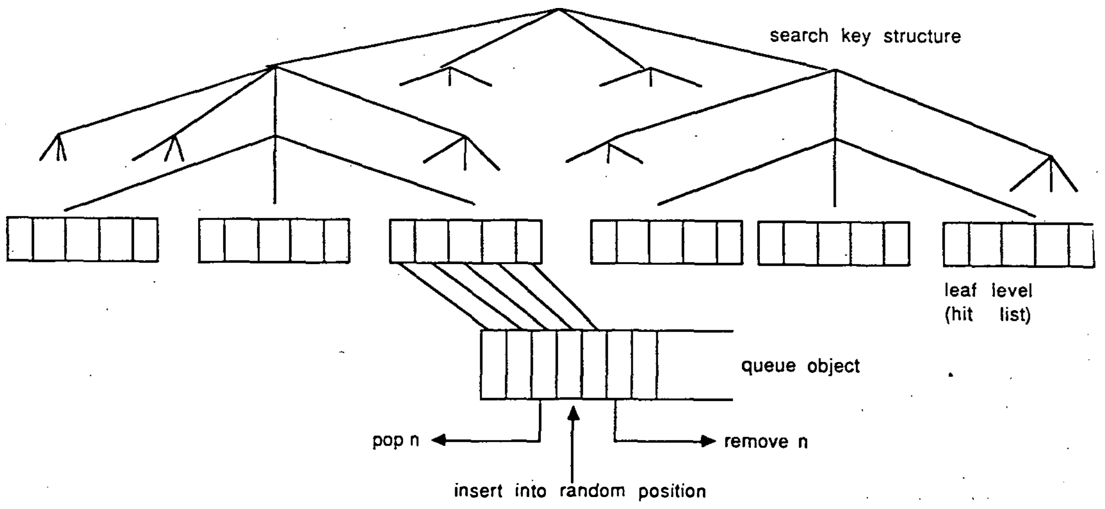

Figure 9: Mapping of queue-type objects into key intervals

The mechanism we have proposed is a fairly simple, yet general extension of each of these specialized techniques. It fits nicely into the overall language structure - one can perceive the data involved in CHECK/REVALIDATE as a temporarily materialized view - and conforms with all synchronization techniques used for ordinary transactions. It should be easy for the reader to re-phrase the current solution in terms of that mechanism.

However, as was pointed out before, under conditions of heavy trading for one stock, the overhead for revalidation may become a bottleneck. Estimates published in [4] say that this point will be near 100 TPS on the same stock.

To allow for higher throughput, we will demonstrate the use of the other techniques proposed. Our basic assumption is that for heavily traded stocks the trading transaction is running permanently (more or less). The bid tuples within the current price range are maintained as a queue-type object according to the protocol described in 4.3.

Now the transactions cooperate as follows:

```
BIDDING-TA:
Begin-of-transaction
IF this bid falls into the current price range or opens
a new one
THEN SEND the tuple to the TRADING-TA
ELSE INSERT INTO BID < ...... >
COMMIT WORK

TRADING-TA
Begin-of-transaction
DO FOREVER
  {ACCEPT tuple from the tuple-passing buffer
  MODIFY lowest-sell and highest-buy
  try to find a deal
  IF deal is possible THEN
    {SEND tuples to NOFITICATION-TA
    POP tuples from the top
    REMOVE tuples from the end
    MODIFY lowest-sell ad highest-buy}
  decide upon commit}
COMMIT
```

The MODIFY-operations are specialized incremental updates of the type described in [7], the NOTIFICATION-TA is not shown explicitly.

It becomes clear from the example that using a 'never-ending' transaction requires the use of savepoint-techniques [11], but we will not discuss this any further.

6. Conclusion

We have demonstrated the needs for extended data objects and synchronization techniques by analyzing an existing stock trading application. For solving the problems identified in this domain, we have suggested three extensions of current DBMS features and briefly described them in an informal way. Their usage was sketched in chapter 5.

The surprising result is that with three simple observations, namely:

- Keeping invariants about data in the database is fully consistent with the transaction paradigm.
- A hot spot can be associated with a transaction owning the hot spot.
- A queue locking protocol allowing for high parallelism with operations typically applied to queues can simply be associated with interval locking on an access path structure.

One can define additional objects and functions for current relational databases, which are easy-to-use and highly increase the expressional power of transaction oriented programming.

7. References

[1] Date, C.: An Introduction to Database Systems, Addison-Wesley.

[2] Gray, J.: The Transaction Concept: Virtues and Limitations, in: Proceedings 7th VLDB Conference, Cannes, 1981, pp. 144-154.

[3] Gawlick, D.: Processing 'Hot Spots' in High Performance Systems, in: Proceedings Spring COMPCON 85, San Francisco, pp. 249-251.

[4] Sammer, H.: Online Stock Trading Systems: Study of an application, in: Proceedings Spring COMPCON 87, San Francisco, pp. 161-163.

[5] Anon et al.: A Measure of Transaction Processing Power, in: Datamation, April 1985.

[6] Kung, H., Robinson, J.: On optimistic Methods for Concurrency Control, in: ACM TODS, Vol. 6, No. 2, June 1981, pp. 213-226.

[7] Reuter, A.: Concurrency on High-Traffic Data Elements, in: Proceedings PODS Conference, Los Angeles, 1982.

[8] ONeil, P.: The Escrow Transaction Method, in: TODS, Vol. 11, No. 4, December 1986, pp. 405-430.

[9] Chung, L., Rios-Zertuche, D., Nixon, B., Mylopoulos, J.: Process Management and Assertion Enforcement for a Semantic Data Model, Department of Computer Science, University of Toronto (submitted for publication).

[10]Duppel, N., Reuter, A., Schiele, G., Zeller, H.: Progress Report No. 2 of PROSPECT, Reserach Report, Department of Computer Science, University of Stuttgart, 1987.

[11] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I.: The Recovery Manager of the System R Database manager, Computing Surveys, Vol. 13, No. 2, June 1981, pp. 223-242.