High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++ *

Arthur Whitney
KX Systems
1105 Harker Avenue
Palo Alto, CA 94031

Dennis Shasha
Courant Institute, NYU
251 Mercer Street
New York, NY 10012

Stevan Apter
Union Bank of Switzerland
299 Park Avenue
New York, NY 10171

Abstract

Imagine an application environment in which subsecond response to thousands of events gives the user a distinct competitive advantage, yet transactional guarantees are important. Imagine also that the data fits comfortably into a few gigabytes of Random Access Memory. These attributes characterize many financial trading applications.

Which engine should one use in such a case? IBM FastPath, Sybase, Oracle, or Object Store? We argue that an unconventional approach is called for: we use a list-based language called K having optimized support for bulk array operators, and that integrates networking, and a graphical user interface. Locking is unnecessary when single-threading such applications because the data fits into memory, obviating the need to go to disk except for logging purposes. Multithreading can be handled for OLTP applications by analyzing the arguments to transactions.

The result is a (private, size-reduced) TPC/B benchmark that achieves 25,000 transactions per second with full recoverability and TCP/IP overhead on an (167 Megahertz) UltraSparc I.

Further, hot disaster recovery can be done with far less overhead than required by two phase commit by using a sequential state machine approach. We show how to exploit multiple processors without complicating the language or our basic framework.

1 A Brief Description of the K Language

K is list-based language integrating bulk operators, interprocess communication, and graphical user interface facilities, designed and implemented by Arthur Whitney. It is extremely concise. For example, a simple spreadsheet has been implemented in about 100 characters, including the graphical user interface, as illustrated in Figure 1.

*Partly supported by NSF grants #IRI-9224601 and #IRI-9531554

Authors' e-mail addresses : nnyapt@ny.ubs.com (Steve Apter), shasha@cs.nyu.edu (Dennis Shasha), asap@netcom.com (Arthur Whitney)

```
F..t:".['D;(;);{. y};F[]]
      F[.;'f]:9$D[]"
/ The first line executes each formula of F[]
/ and assigns the results to the corresponding
/ positions of D. The second line assigns those
/ values of D to the corresponding display fields of F.

F:D:.+('a'b'c'd; 4 53#,"") / 4 across and 53 deep
'show$'F
```

Figure 1: This K program implements a 4 by 53 spreadsheet, including a user interface and full access to vector functionality. Comments begin with slashes. The program has two data structures: a set of formulas and constants F and a set of data D. Every time a formula is changed, the trigger F..t fires. This trigger updates the data D based on the data and formulas. The strings F and data D are initialized to be 4 by 53 arrays of blanks. The last line displays the spreadsheet. This program has slightly more functionality (because arbitrary formulas are allowed) than a simple spreadsheet written in Java presented at location http://www.javasoft.com/applets/applets/SpreadSheet/example1.html.

Unlike most other list-based languages, K is extremely fast. For example, sorting three million records in memory takes two seconds on an IBM 990. K is also small. The entire runtime system, written in C, fits in 300,000 bytes.

The basic data structures are arbitrary length multi-lists and dictionaries (associative arrays). Binary operations on equi-length multi-lists include element-by-element addition, multiplication, subtraction, and division. max, and min. Boolean comparison ($<, >, =$) operators are also element-by-element returning a list of 1s and 0s. These operators are overloaded with scalar variants as in APL. So one can, for example, compare 5 to every element in a list L and get a list of 1s and 0s of length |L| such that position i is 1 if 5 is equal to the ith position in the list and 0 otherwise.

Rearrangement operators include sort, duplicate removal, and partition. Partition on a list L returns a list of sublists of positions where a given sublist of positions consists of all those positions in L having equal values. For example applying partition to the list 105 20 30 20 30 30 105 returns a list consisting of sublists 0 6 (positions having value 105), 1 3 (positions having value 20), and 2 4 5 (positions having value 30).

Location functions include "member" (i.e. is a value in a list), "find" (viz., find first place in a list having a value), and "where" (find all places in a list having a value). Find can be optimized using hashing.

Any operator can be modified to form a cross-product as well as left and right "half-cross-products." Using the full cross-product, one can, for example, compare every member of list L1 with every member of another list L2. This returns a list of 1s and 0s whose length is the product of the cardinalities of L1 and of L2. Using the left half-cross-product, one can also compare (or add or divide or any other binary operations) each member of L1 with all of L2 returning a list of sublists where each sublist is of the length of L2 and the number of sublists is the length of L1.

Reduction operators include count and prefix scan. For example, one can modify the operator plus with prefix scan on the list 3 5 7 8 9 and get the result 3 8 15 23 32. Another operator (called "over") gives the last element of the plus prefix scan (32 for the example above).

Two-dimensional tables are built from multiple lists (a fully vertically partitioned representation in which each attribute is associated with a list). Each value in an attribute can itself be a list. Vertical partitioning works well for decision support, because only the attributes necessary for a query are brought in. (Also, the scanning rates in K are very fast. For example, 8 million records in RAM can be scanned in 1.9 seconds on an UltraSparc I to check that a field value is greater than 5 using the "where" feature mentioned above.)

The operators above allow K to implement relational algebra and aggregates including group by and having clauses. An efficient library for all of relational algebra fits comfortably on a page. Since K treats names as regular data, one can perform second-order queries. Suppose, for example, that you want to union certain fields of a group of tables, where the group is presented as a list of names. SQL doesn't allow this, but K's second order facility makes this a short one liner.

In addition, K has unified file input/output with TCP/IP support, and has a graphical user interface built on top of X-Windows in which the values of variables can determine their color. This is useful on Wall Street where painting losses red is a useful visual cue.

K does not have pointers, so does not need to worry about pointer swizzling. Instead data-structures are memory-mapped from files.

Having a language with bulk operators implies that there is no discontinuity between data intensive operators and other operators. In the Wall Street context, for example, K supports both trading and analytics in a seamless fashion. There is no need for cursors, 4th generation languages, embedding calls from C or C++, SQL, OSQL, PowerBuilder, Motif, or Open-Client/OpenServer packages.

Note: The Sigmod reviewers found it annoying that we had no references for K. The language is in fact proprietary, even its description. We therefore depend on the reader's cultural knowledge of APL as well as the descriptions above to get a sense of what the language is.

2 Transaction Processing Strategy

The rest of this paper concentrates on only the most essential aspects of the K language: A full and very fast programming language with bulk operators that can easily express SQL, embodies order (e.g., top ten queries), and that can both access disks and send and receive TCP/IP messages. How do we construct an online transaction processing from such a language?

It turns out to be embarrassingly simple. Assume that a transaction program is implemented as a K procedure, so it can be described by its name and its arguments. Typical Wall Street programs enter a trade and update the trader's position as well as many subsidiary risk tables and back office (bill-clearing) databases. A message may consist of many transaction instances (hereafter just transactions). Our basic strategy is to log all incoming messages onto disk and then to perform the transactions in those messages on the in-memory version of the database. Recovery consists of replaying the logged transactions on disk after reproducing the state from an appropriate dump. (In our applications so far, this is the dump of the state as of last night.) Message logging is sufficient because the database is updated sequentially.

2.1 Group Logging

The basic strategy is slow if every transaction is logged separately. So, we write many transactions (each transaction is represented by its transaction type and all its parameters) at a time to disk. Once they are written to disk, they are secure against main memory failure, because one can always recover the state of the database by replaying the log.

2.2 Disaster Recovery

Warm start recovery might be slow even with group logging, because one might have to recover from a transaction log that is very long. Avoiding this problem requires frequent (i.e. intra-day) dumping, but let us step back a minute.

In the Wall Street setting, we are worried not only about warm start recovery (processor failure) and media failure (local disk) but site failure as well (remember the World Trade Center bombing?). So, we are interested in remote backups. Further, we want to dump the database state periodically for fast recovery in case of a massive power failure.

For these reasons, the backup sites behave differently from the primary, as seen in figure 7.1. On the primary, a single process logs a batch of transactions and applies them to the in-memory database. In every backup, a **logger process** will receive transactions and log them both on disk and in memory, but is not responsible for applying the transaction to the in-memory database. Once a transaction t is logged on backup b, b will report that t *is stable on* b. A commit message for transaction t will be sent back to the client after it is stable at the primary and a sufficient number of backups (how many is a policy issue) and the primary completes the transaction in its main memory database.

After logging and replying to the client, the logger process sends large groups of transactions asynchronously to the **worker process** which performs the transaction against the main memory database. The logger also periodically instructs the worker process to dump its (the worker's) state to disk. While the worker dumps its state, new transactions will accumulate in its buffer.

One might wonder how the worker will ever catch up to the primary. Note that the worker will have a huge number of transactions in the buffer and won't have to log to disk. In our experiments, the in-memory performance of K on TPC/B yields transaction rates of slightly over 50,000 transactions per second on a (167 MHz) UltraSparc I in the absence of TCP/IP and logging vs. 25,000 with the overhead of TCP/IP and logging. So the worker processes of Figure 7.1 can catch up to the primary at the 50,000 transactions per second rate when it is finished dumping.

Like two phase commit, this strategy ensures that a sufficient number of backups can be brought up-to-date, but there is negligible overhead per transaction. The net effect is that we can continue processing seamlessly if the primary fails since at least one of the backups will be up-to-date. In the unlikely event, that there is a memory failure of all sites, we can recover by rolling forward from the most recent dump.

Ordering the Transaction Arrivals

We now must solve a distributed coordination problem: transactions must be applied in the same order to all data servers. A simple method is to have a single **time server** processor that assigns a sequence number to each transaction and then funnels them to all data servers. Failure of this processor however could cause the entire system to be unavailable.

Handling time server failures requires coordination that is as hard as the consensus problem[8] and is therefore prone to blocking, but can be made safe using ISIS process groups[2] and live with high probability. A possible alternative is to choose an application-specific approach which we describe in the appendix. Both schemes entail little overhead during failure-free execution (two messages per server per collection of transactions).

What Have We Gained?

To implement a replicated backup in a standard concurrent database setting, a commit protocol is necessary to ensure (among other things) that transactions commit in the same order at the several sites. This requires coordination among all sites (usually through a transaction manager or TP monitor)[11] for each transaction. Because our basic transactions execute sequentially, we can ensure that the primaries and backups are consistent merely by ensuring that the messages arrive in the same order at all sites. The moment of commit requires no coordination at all.

The only point of coordination is embodied in a time server process group that assigns sequence numbers to each transaction, sending them to each data server across a point-to-point network or using multicast. The data servers execute the transactions in sequence number order.

Recovering data servers can join the data server group by taking a feed from a dump being written by some other site, replaying the log from that site, and registering with the time server.

2.3 Results of this approach

While the TPC/B benchmark may be an inappropriate benchmark for retailing and perhaps insurance, it models trading applications quite well. Those applications are characterized by indexed lookups requiring subsecond response time. Here is a brief review of the benchmark:

- The database consists of four tables: Account, Teller, Branch, and History with a many-to-one relationship between Account.BranchID and Branch.BranchID and a further many-to-one relationship between Teller.BranchID and Branch.BranchID.
- Branch records must contain at least 100 bytes. The intended database size is 10 megabytes per TPS.
- The transaction consists of the following steps:
  1. update account.balance for a given accountid.
  2. update the teller.balance for a given tellerid.
  3. update the branch.balance for a given branchid.
  4. insert the accountid, tellerid, branchid, update amount, and timestamp into a history file
  5. commit the work

The approach we used admittedly doesn't fit the rules for this now outlawed benchmark. Our basic violation is that we don't let the data grow to 2×10^{11} bytes. Further, the history file was our log, so we wrote into history at the beginning of the transaction and never needed to commit work.

On an UltraSparc I having 250 megabyte of RAM, we ran a TPC/B benchmark having only 8 million records. (All the relevant fields are hashed, so changing the size from 1 million to 8 million records didn't change the timing. On an IBM 990 having 2 gigabytes of memory, the timings also didn't change with size but were 30% below these numbers. The point is that these numbers would change little if the UltraSparc had a bigger RAM.)

Two parameters are relevant: (i) whether messages are sent over TCP/IP or not and how big the messages are and (ii) whether we log the transactions to disk and in what size.

- No TCP/IP; No logging:
  50,505 transactions per second.

  This is relevant to our backup strategy because the backup catches up at this rate after it has dumped the database state.

- No TCP/IP; Logging:
  32,258 transactions per second.

- TCP/IP messages of 10,000 transactions each; No logging:
  26,316 transactions per second.

- TCP/IP messages of 1,000 transactions each; Logging in groups of of 2,500 transactions each:
  22,703 transactions per second.

  Note that the response time is still under a second

- TCP/IP messages of 10,000 transactions each; Logging in groups of of 10,000 transactions each:
  25,017 transactions per second.

- For data that spills to disk:
  5 transactions per second.

When data exceeds RAM size, the transaction rate is gated by the disk access that each transaction must make against the account table. We now show how to attack this problem.

2.4 Aggregate Style Queries

TPC/B is a low functionality benchmark. Let us consider a group by query which is typical in decision support applications. The data consists of 978,858 rows of telephone billing data (63 megabytes). Consider a group by operation that reduces the size to 5,400 rows.

```
set statistics time on;
select sum(ReportedCost) as charge, BillingID
```

```
into #temp from usage0197 group by BillingID;
set statistics time off;
drop table #temp;
```

Our hardware consists of two 200 megahertz Ultra Sparc IIs with 1.5 gigabytes of RAM and the latest version of a major commercial database. We run the query once without timing it. Then we run it several times and get times of around 10.8 seconds per query. Note that #temp is unlogged and will reside in memory.

The equivalent group by query in K takes 1.5 seconds once the initial data is in main memory. If K writes the data to a file on another machine, the time rises to 2.5 seconds.

3 Multithreading without Concurrency Control

The most straightforward way to implement the replicated state machine approach is to execute transactions in the order in which they arrive. This proved to be slow if some of the data is only on disk (causing TPC/B performance to decline to 5 TPS from over 20,000). This suggests the need for multithreading. Multithreading would also be useful for shared memory multiprocessors.

A good first impulse should be to partition the database and assign one process to each partition. That will make better use of the disks on a single server. If one can partition the database into several servers, then the entire problem goes away.

If partitioning is impossible or inconvenient, we can achieve parallelism across processors or to parallel disk arms by using multiple K processes while ensuring that transactions APPEAR to execute in the order in which they arrive. Provided all sites do this, we are assured of consistency across the sites. Note that we are asking for a tighter constraint than serializability. We are asking for a serializable execution that is equivalent to a particular order of transactions (the arrival order of transactions).

The algorithm OBEYORDER makes use of a programmer-provided predicate called CONFLICT that will determine whether two transactions conflict in the normal serializability sense [1]. For example, for TPC/B, CONFLICT(t1, t2) will hold if and only if t1 and t2 access the same account, branch or teller (and then only if we are worried about negative balances); all other updates are commutative. OBEYORDER works as follows:

1. Log the transactions in batches on disk as they arrive.

2. Construct a graph whose nodes are the transactions in a batch. Form directed edges from the undirected CONFLICT relationship as follows. If t conflicts with t' and t has an earlier sequence number than t', then form a directed edge (t, t'). This produces a graph G = (T, E) where T is the set of all the transactions in the batch (or batches) and E is the set of directed conflict edges.

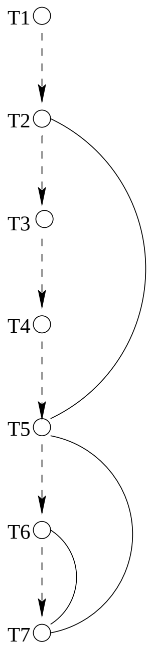
3. Execute T in parallel, respecting the order implied by E:

```
while G is not empty do
  R := roots of G.
  execute R in parallel
```

```
   remove R from G as well as all edges in G that touch R
 end while
```

That is, execute the roots of G in parallel. Then remove those nodes from G, forming a new graph G', and execute the roots of G' in parallel. Continue until there are no more nodes left.

Figure 3 illustrates this algorithm.



Dashed arrows represent the intended order of transactions according to sequence numbers.
Solid lines represent conflicts.

The algorithm would first turn the conflict edges into ordered ones in agreement with sequence numbers.
This would yield edges (T2, T5), (T5, T7), and (T6, T7).

Then the algorithm would find the roots of this graph:
T1, T2, T3, T4, T6. These can be processed in parallel.

After these are processed, they are removed from the graph and only T5 and T7 remain.
Since there is an edge (T5, T7), T5 is the only root and it is processed. Then T7 is processed.

This processing order agrees with a serial execution according to sequence number (viz. T1, T2, ... , T7).

Fig: Illustration of ObeyOrder Algorithm

Figure OBEYORDER shows the intended order of transactions (according to arrival order) and the conflict edges.

Theorem 3.1 *Algorithm OBEYORDER yields an execution that is equivalent to a serial execution of transactions in ascending order of sequence number.*

PROOF.
To be equivalent, the two executions must order conflicting transactions in the same way. Since OBEYORDER executes conflicting transactions in sequence order (i.e. arrival order), the result follows. □

Two possible practical objections to this algorithm can be raised. First, OBEYORDER depends on knowledge of conflicting transactions. This may not always be possible (though we often find it easy in our transaction processing applications, where conflicts are determined by the keys). Second, there may be many conflicts, thus excessively limiting concurrency (and more than a normal database system would).

This second objection leads to a PREFETCHING optimization that is independent of OBEYORDER and exploits the following observation: One can always prefetch data provided one holds it in a shared buffer. So, if a transaction t must access account x, it fetches x. If another transaction s updates x and s precedes t in the update order, everything is fine provided they access the same copy of x. The PREFETCHING operation reads data into RAM for as many transactions as possible, according to the same priority arrangement that OBEYORDER gives. That is, form a topological sort according to the graph defined in the OBEYORDER algorithm and read as much data as possible for those transactions in that order.

As a side note, one might remark that we may eject pages once they have been used by some transaction, since it is unlikely that they will be used again. Thus, improvements to the currently implemented LRU buffer replacement policy are possible.

4 Attacks and Parries

Here we list all the objections we have heard against this approach and our responses to them. In some cases, we must agree with the attackers.

4.1 The Worried Programmer Argument

Attack: "Your programmers have to avoid concurrency whereas our programmers (i.e. ones using a system with system-directed deadlock detection and rollback) can use it without worrying about it."

Parry: First, an application benefits from concurrency only if it exceeds the size of RAM or runs on a multiprocessor and can't be partitioned. (We discuss ways K can handle this further in the next subsection.) Second, it is not true that application programmers on relational database management systems can remain unconcerned about concurrency. If a Sybase or Oracle application runs into a deadlock, it must retry it, and often must recover its own program state, because rollback doesn't recover local variables. In our experience on Wall Street, this constitutes some of the most error-prone code in a database application even for experienced applications developers.

4.2 The Single-threading is History Argument

Attack: "As shared memory multiprocessors get more and more popular (especially with large memories and 64 bit addressing), the single-threaded argument will be history."

Parry: This argument appears to attack the core of our design. Our speed advantage at each site depends partly on the lack of concurrency control. Ditto for our doing without two phase commit. But even accepting the limitation of single-threaded equivalence, K can do several things with shared memory multiprocessors:

1. Partitioning: many applications can be partitioned on geographical or other grounds, in which case each partition can proceed in a single-threaded manner.

2. Once K has multithreading, intra-transaction parallelism can be used (e.g. to update the trade and position tables in a trading system in parallel). Intra-transaction parallelism also preserves the illusion of single-threaded transactions. The programmer would have to recognize the potential for such parallelism, but would be able to apply it easily since K

has an operator that applies to **each** member of a list conceptually in parallel. In this case, the list would consist of operators on different tables and one would apply all these operators in parallel.

3. Many of the operators in K such as scan, sort, sum, and partition can be parallelized both in-memory and on-disk. Such work is in the medium-term horizon for K. Parallelizing such operators requires no changes to the programming model and preserves the semantics of single-threading from a concurrency point of view.

4. Inter-transaction parallelism can be supported using the OBEYORDER algorithm. One might object that this fails in the face of ad hoc queries, but there are few ad hoc queries during a busy online window.

This said, we admit that our lack of concurrency control makes K unsuitable for IRS (U.S. tax authority) style databases. So be it.

4.3 The Buggy Whip/Bandwagon Argument

Attack: “SQL is the only (real) game in town. You guys are crazy to refight a battle that is long over.”

Parry: In our setting, some applications are written using K database servers and others use Sybase or Oracle. Applications running on top of relational database management systems that require very high performance do virtually all their processing outside of the RDBMS anyway. The interface to the persistent storage manager is then extremely painful and inefficient.

Further, K’s single-threaded approach allows excellent disaster recovery without the overhead of two phase commit or the window of vulnerability offered by replication servers (where the user can be told that a transaction has committed before it has been applied at the backup site). If you admit that you want a sequential machine approach, then you must be concerned about the following fact: SQL isn’t deterministic even if executed sequentially as observed by Richard Gostanian of ISIS[10].

The basic reason is that set-oriented operations are non-deterministic, e.g. book a 9 am flight from jfk to lax on feb. 4 Which flight is booked and which seat is chosen may depend on record layouts. SQL enforces no order. If one gets exactly the same disks and lays out tables in exactly the same way, one may hope that the results everywhere are the same, but (i) this is hard to do (ii) the guarantee you get depends on implementation details way below the level of the relational data model.

One might counter-argue that certain operations are inherently non-deterministic, e.g. calls to a timer. K programs that make such calls must make those calls at the client and submit any necessary times as parameters of the transaction.

Moreover, set-oriented (orderless) languages are inferior to list-preserving ones for many practical applications. Queries on the ten most recent events, year-by-year comparisons, cost of goods sold calculations as well as extensions proposed for decision support by Redbrick (http://www.redbrick.com) and in recent issues of DBMS by Ralph Kimball in his “Data Warehouse Architect” column [14] require notions of order. Scientific databases [21] also require order. Writing such queries in SQL requires a Joe Celko (editor of DBMS magazine and SQL puzzlist extraordinaire) to do at all and even he may have trouble doing them efficiently.

As a first step away from legacy systems, developers can use K only in a call-out mode. One of our applications, for example, uses K to perform bond valuation calculations. It is called from Sybase through Open Server.

Finally, the truly faint at heart can use K's SQL 89 library if they insist on using that language. It runs fast, but is painful for those used to the power of the K language.

By the way, didn't relational systems encounter this same bandwagon-style objection in the early 80s?

4.4 The Physical Data Independence Argument

Attack: "Anyone can design a great algorithm and data structure for a particular problem, but you will never be able to reuse the structures you have designed."

Parry: Tables are easily represented as multi-lists. Physical data independence happens as a result of the clever implementation of the bulk operators. We have never found it to be a problem to write new queries against our global data structures.

Further, multivalued dependencies do not need to be decomposed. If a user is associated with a set of permissions, one can make the association in the natural way: a user list and a set-of-permissions list.

```
steve {trade, sales-order-match, position}
arthur {risk-management, sales-order-match}
dennis {risk-management, position}
```

There is no need for further decomposition. The relational partisan might respond that putting sets in the authorizations column makes queries of the form *Which users are associated with a certain permission?* hard to answer. But these are easy in K. For example, the following query finds those user ids having authorization for trade.

```
user.id[ & ‘trade _in ‘user.auths]
```

4.5 The Query Optimizer Argument

Attack: "Your programmers must figure out how to implement queries, whereas SQL programmers may state their queries declaratively and the optimizer will figure out what to do."

Parry: None, really. A cynic might say that RDBMS query optimizers leave a lot to be desired — witness the temporaries people create to avoid bad processing of DISTINCT, ORs or NOT IN, or witness the facilities offered by the DBMS vendors to force indexes and table orderings. Witness also the errors introduced by "improvements" to the optimizers[1] But we are not cynics.

In K, a little thinking (e.g. avoiding cross-products) yields extremely fast running times for complex queries because scan, sort, and duplicate elimination are very well implemented.

[1] One major vendor improved its "not in" and "in" processing in such a way that it gave wrong answers. In one example, we tested two queries A and B which differed only in that query A had "... and trade.pt_id in (5,2)" and query B had "... and trade.pt_id in (2,5)". Query A returned 13 rows and query B returned 0.

4.6 The Interoperability Argument

Attack: "Ok, if someone comes to you with a brand new application, you might have a chance, but you have to interoperate with other database systems. How will you do that?"

Parry: This objection does in fact cause some pain when we design K applications. The problem boils down to a multi-database concurrency control and recovery problem: if a K database fails, its recovery consists partly of simply replaying its transactions from the log, including return values from other database systems. This allows the K database to recover its state correctly.

The K system should not, however, replay the writes it has made to foreign (e.g., relational) databases. Somehow, it must determine which writes it has done and which it must do upon recovery. Our solution to this is to keep "persistent breadcrumbs" (in the form of a table) of all transactions issued by the K system on each foreign database and to keep the breadcrumb table on the foreign database. Every write transaction from K to the foreign database includes a write to the breadcrumb table that uniquely describes the transaction. The code also includes a check to see whether that transaction has already occurred.

For example, suppose that a K transaction updates tables R and S in a foreign database OTHER. There is no two phase commit across the OTHER and K databases, so the interface includes a third (breadcrumb) table in OTHER called idInK. The pseudo-code to update R and S now looks like this (seqnum is the sequence number of the K transaction and is a parameter of the transaction that is stored in the K log):

```
if (recovering on K side) {
 begin transaction on OTHER side
  if (seqno not in idInK) then update R and S and put seqno in idInK
  else do nothing
 end transaction
} else { /* normal operation on K side,
OTHER side is working normally.
so there is no way seqno could be in idInK */
 begin transaction on OTHER side
  update R and S and put seqno in idInK
 end transaction
}
```

There are two other matters to take care of:

1. If there is a deadlock in an OTHER side transaction, then the K side must do deadlock retry. This is as arduous for us as it is for anyone else.

2. If the OTHER side crashes, the K side must wait for it to recover and must be able to handle the problem that transactions with the same sequence numbers may arrive. The K side program must avoid doing a transaction twice.

4.7 The "This is All Obvious" Non-argument

Attack: "You are telling us nothing new. Others have already told us that main memory databases are great. You are just rediscovering that fact."

Parry: It is indeed true that there has been excellent research in main-memory databases. Telecommunications databases are often main-memory[6, 5]. Toby Lehman and Mike Carey have looked at many main memory issues having to do with query processing[15], indexes (small fanout trees known as T-trees work well)[16], concurrency control[18], and recovery[17]. Some of these ideas found their way into the Starburst main-memory storage manager[20] where the authors find that the overhead of locking can dominate the cost of database accesses.

The Smallbase project led by Marie-Anne Neimat also uses T-trees as well as hash indexes[12]. The project supports a subset of SQL. As in our approach, Smallbase serializes transactions. Smallbase plans to implement undo/redo recovery through value logging. On their size-reduced TPC/B benchmarks, they achieve 20,000 transactions per second through their storage manager interface without recoverability, serializability, or TCP/IP. In a nice use of the P2 configurable database manager, Thomas and Batory achieved 100,000 transactions per second on an HP 755 (99 MHz)[22] assuming collision-free hashing.

Li and Naughton[19] proposed a multiprocessor main memory databases. One thread groups input transactions into an input queue, though they don't use that queue for recovery. They write modified records to their log at commit. Their checkpoint thread sniffs the log and applies the changes to a shadow database. Periodically, it writes the shadow database to disk. Unlike them, our input queue is also the log since we use operation-based recovery. Further, our checkpoint process is completely asynchronous to the execution of the primary. They have one important safety feature that we lack: they allow rollbacks due to transaction logic or to exceptional conditions. The reason is that they keep the before-images of all data items.

In the Dali project, researchers Jagadish, Dan Lieuwen, Phil Bohannon, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan have implemented multi-level recovery algorithms (including fuzzy checkpointing) and recoverable T-tree concurrency control algorithms[13, 4, 3]. Margaret Eich and her students have also worked on concurrency control and recovery[7]. Hector Garcia-Molina and Ken Salem have been among the first to observe that concurrency control might be unnecessary in a main memory environment[9].

ISIS[10] takes a complementary approach that ends almost at the same point. The ISIS mechanism delivers messages in order to the primary and backup site(s). Each site is a standard concurrent database management system, however, so some form of coordination is needed to ensure that conflicting transactions commit in the same order at all sites. The claim is that the overhead for this coordination is lower than that of two phase commit.

There are only a few new points in our work:

1. We have exploited the no-concurrency-control idea to achieve recoverability without doing image-based recovery. This allows us to achieve much higher performance levels than conventional database management systems without sacrificing fault tolerance (indeed we enhance it).

2. In addition to high performance, we have shown how to avoid other distributed system

problems: we have hot backups while avoiding two phase commit (replication servers offer only warm backups).

3. Finally, we show how to obtain good disk bandwidth when the database size exceeds main memory, by using application-specific concurrency semantics.

5 Conclusions

Relational database systems grew up in a time of RAM scarcity. In such an environment, concurrency control/two phase commit/replication server style solutions made sense. Today, many applications fit comfortably in RAM. In such an environment, different rules apply (as predicted by Garcia-Molina, Eich, and Lehman), perhaps enabling us to do without concurrency control for many applications. This would avoid a host of problems having to do with blocking, deadlock, two phase commit, and implementation complexity.

A language having well-implemented list-based bulk operations yields a system that is simple and powerful. APL, not relational algebra, may be a better starting point for such a language.

The performance gains from this approach follow from

1. The possibility (realized in K) of a very efficient implementation.
2. No overhead to enter and exit a database management system, because there is no separate database management system.
3. No overhead for concurrency control.
4. Simpler logging and recovery code.

Further, applications can be made more reliable in K than using relational systems because

1. There is no deadlock recovery code (because there are no deadlocks).
2. One can realize true hot backups through replicated state machines with much less overhead than required by commit protocols, because operations are processed (or appear to be processed) sequentially.

Update applications that don't fit into memory are a potential problem, since a naive single-threaded application can have its performance reduced by a factor of several thousand if it becomes single-threaded on random disk accesses. Fortunately, applying a little serializability theory along with pre-fetching heuristics can mitigate this problem. Further, this solution also provides a mechanism for exploiting shared memory multiprocessors without sacrificing the single-threaded philosophy. The solution does, however, impose a burden on the programmer who must implement the CONFLICT predicate on transactions.

Decision support is a good next application. Preliminary experiments indicate that K with its vertically partitioned tables works very well for this application (more than a factor of ten improvement over one previous record holder's TPC/D benchmark number on equivalent hardware). Further, the bulk operators such as sort, scan, and duplicate removal all admit parallel implementations. This library should include a query optimizer and perhaps give hints to the user about which data structures to build.

6 Acknowledgments

We would like to thank Tom Brown, Marc Donner, Toby Lehman, Mike Rosenberg, and John Turek for their comments on this paper. Wayne Miraglia and Don Orth helped greatly with their ideas and implementations.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman *Concurrency Control and Recovery in Database Systems* Addison-Wesley, 1987.

[2] Ken Birman "The Process Group Approach to Reliable Distributed Computing" Communications of the ACM 1993, pp. 37-48

[3] Philip Bohannon, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz, S. Sudarshan The Architecture of the Dali Main-memory Storage Manager personal communication, 1996

[4] Philip Bohannon, Rajeev Rastogi, Avi Silberschatz, S. Sudarshan Multi-level Recovery in the Dali Main-memory Storage Manager personal communication, 1996

[5] S. K. Cha et al. "Object-Oriented Design of a Main-Memory DBMS for Real-Time Applications" Proc of the Int Workshop on Real-Time Computing Systems and App (RTCSA 1995)

[6] M. Driouche, Y. Gicquel, Brigitte Kerherv, G. Le Gac, Yann Lepetit, G. Nicaud "Sabrina-RT, A Distributed DBMS for Telecommunications." EDBT 1988: 594-599

[7] Margaret H. Eich Main Memory Database Research Directions. IWDM 1989: 251-268

[8] M. J. Fischer, N. A. Lynch, and M. S. Patterson "Impossibility of Distributed Consensus with One Faulty Process" JACM April 1985

[9] Hector Garcia-Molina, Kenneth Salem Main Memory Database Systems: An Overview. TKDE 4(6): 509-516 (1992)

[10] Richard Gostanian, rg@isis.com personal communication

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques* San Mateo, Calif.: Morgan-Kaufmann, 1992.

[12] Michael Heytens, Sheralyn Listgarten, Marie-Anne Neimat, Kevin Wilkinson Smallbase: A Main-Memory DBMS for High-Performance Applications (release 4.2) Personal communication, September 7, 1995

[13] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, S. Sudarshan Dali: A High Performance Main Memory Storage Manager. VLDB 1994: 48-59

[14] Ralph Kimball "Data Warehouse Architect" DBMS, August 1996, vol. 9, no. 9, pp. 14-16

[15] Tobin J. Lehman, Michael J. Carey Query Processing in Main Memory Database Management Systems. SIGMOD Conference 1986: 239-250

[16] Tobin J. Lehman, Michael J. Carey: A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986: 294-303

[17] Tobin J. Lehman, Michael J. Carey: A Recovery Algorithm for A High-Performance Memory-Resident Database System. SIGMOD Conference 1987: 104-117

[18] Tobin J. Lehman, Michael J. Carey: A Concurrency Control Algorithm for Memory-Resident Database Systems. FODO 1989: 490-504

[19] Kai Li, Jeffrey F. Naughton Multiprocessor Main Memory Transaction Processing. DPDS 1988: 177-187

[20] Tobin J. Lehman, Eugene J. Shekita, Luis-Felipe Cabrera An Evaluation of Starburst's Memory Resident Storage Component. TKDE 4(6): 555-566 (1992)

[21] Joel Richardson "Supporting Lists in a Data Model (A Timely Approach)" VLDB-92, pp. 127-138

[22] Jeff Thomas and Don Batory "TPC-B on Smallbase and P2" manuscript obtainable from batory@cs.utexas.edu, August 1995

7 Appendix: application-specific maintenance of time servers

Recall the basic scenario of our backup strategy: clients send requests to a time server process group which in turn attaches sequence numbers to transactions and sends them to all available data servers. We call the sequence number associated with a transaction its *transaction number*.

Transaction numbers are the concatenation of the *epoch* and a simple number. Each time a new time server takes over the management of transaction number generation, the epoch number is incremented by one.

In addition, there are *client tickets* that uniquely identify each transaction with a concatenation of client id and client-generated sequence number. Client tickets are used to identify duplicate requests from a client for the same transaction.

The data servers are sequential state machines that process transactions in transaction number order (or appear to if they use algorithm OBEYORDER). A transaction t is *committed*, if the client receives the message that t has committed. The *last committed transaction* is the committed transaction having the highest transaction number. A transaction t is *stable at data server d*, if d has stored the transaction parameters for t on disk or the dump from which d was loaded includes the effects of t. A data server d is *up-to-date* if all transactions up to the last committed transaction are stable on d.

Tolerated failures include lost messages and fail-stop processors and disks. Availability should not suffer provided at least one time server is up and a majority of data servers are up-to-date. No committed transaction should be lost provided at least one data server is up-to-date. Here is the protocol.

Normal Operation Rules

1. The active time server assigns consecutive transaction numbers to transactions. If t and t' are from the same client, and t has a smaller client ticket than t', then the time server will assign a smaller transaction number to t than to t'. (There is no ordering between transactions coming from different clients.)

2. A data server d reports that t is stable after all transactions having lower transaction numbers are stable on d and t is logged.

3. The active time server tells a client c to commit a transaction t after a majority of the up-to-date data servers report t to be stable and the primary data server gives a return value and it has sent commit messages for all transactions having lower client tickets from c. The client c commits t only after it has committed all transactions having lower client tickets.

Failover:

1. If the backup time server detects a problem, it sends a robot to close down the primary time server. (In practice, this might be a system administrator who is beeped for this purpose.)

2. After the primary has been shut down, the backup time server then checks with the data servers with whom it can communicate to determine the highest transaction number stable on at least one of those servers. The backup time server stops if it cannot talk to a majority of the data servers. (In that case, the backup time server calls for human assistance, because a network partition may have occurred. The person in question must determine the highest transaction number and cause all servers to be up-to-date, possibly after repairing the network.)

3. Suppose the backup time server can talk to a majority of data servers. Identify this majority set as S. Suppose n is the highest transaction number stable on any data server within S. The backup time server causes all data servers in S to be up-to-date up to n. It also resends to the client any necessary commit messages.

4. The backup time server then requests clients to send uncommitted transaction requests to it. It checks these against the client tickets of all previously committed transactions to ensure that it is not doing the same transaction twice, but executes those that have not yet been committed. It then changes its name to the new active time server.

5. The new active time server starts a new epoch. Sequence numbers are the concatenation of epochs with counters within epochs.

7.1 Guarantees

- After the new time server takes over, but before it accepts client requests, every committed transaction is stable on all data servers with which the new time server can communicate. All those data servers are up-to-date with respect to committed transactions.

Proof: Let the last committed transaction be t. Transaction t must be stable on a majority M of the data servers by the normal operation rule. That majority and the majority S with which the new time server communicates must intersect. So, t is stable on one of the data servers in S. Since the protocol specifies that all data servers in S are brought up to date with respect to the highest transaction number in S (step 3 of failover), t will be stable on all members of S.

- No transaction will be given different transaction numbers.

  Proof: No single time server would do this. Two time servers can never be active at the same time since the primary is turned off before the backup comes on-line. If a transaction t is committed by time server ts1 and then time server ts2 receives the same transaction as uncommitted, time server ts2 will detect the repetition (step 4 of failover) by using the client tickets.

- No two transactions will be assigned the same transaction number.

  Proof: Every time server starts a new epoch. As long as the order in which time servers take over from one another is well-defined, only one will start a given epoch number. If there is just one timeserver backup, then only one time server can take over. If there are many timeserver backups, then this can be ensured by using timeserver identifiers (but we don't deal with that here).

- A given transaction is written at most once to any data server.

  Proof: By the last two propositions, there is a one-to-one correspondence between transaction numbers and transactions. Each data server writes transactions in transaction number order to the local log.

- Data servers write and process transactions in the same order.

  Proof: By the normal operation rule, data servers process operations in transaction number order (or appear to).

- If a client c commits t, then t is stable on a majority of data servers, and any t' submitted by c such that t' has a lower client ticket than t is also stable on a majority of the data servers.

  Proof: Follows from the 1-1 correspondence between transaction numbers and transactions and the Normal Operation Rules.

- The system will continue to run even after the failure of one time server and a minority of data servers, provided there is communication connectivity among all surviving nodes. It will stop otherwise.

  Proof: Nothing in the protocol depends on more than a majority of data servers or a single time server.

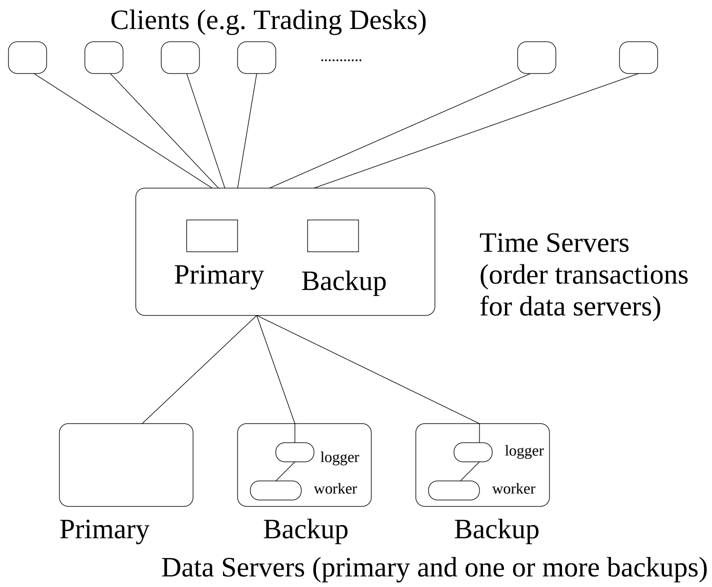


Figure: Hot Backup Strategy
Primary logs a group of transactions to disk and then executes them on an in-memory copy of the database.

Each backup has a logger process that logs transaction parameters to disk (like the primary). The logger also sends two kinds of commands to the worker: process this transaction and dump state. The backup tells the time servers that a transaction is safe as soon as the logger has written the transaction's parameters to disk, so response time doesn't suffer during dumps.

This figure illustrates a scenario in which a primary and one or more backups remain in synchrony by logging transactions and applying them in the same order.