Improving MMDB Distributed Transactional Concurrency

Weiwei Gong
Dept. of Computer Science
Univ. Massachusetts Boston
wwgong@cs.umb.edu

Patrick O'Neil
Dept. of Computer Science
Univ. Massachusetts Boston
1-617-354-6460
poneil@cs.umb.edu

Elizabeth O'Neil
Dept. of Computer Science
Univ. Massachusetts Boston
1-617-354-6460
eoneil@cs.umb.edu

ABSTRACT

Main Memory Database Systems (MMDBs) have been studied since the 80s [3,4], when memory was quite costly ($1500 per MByte in 1984). We can now buy memory for about $10 per GByte. An advantage of MMDBs is that serial execution of a non-distributed transaction on a uniprocessor from start to finish saves the work of disk I/O, locking, latching and deadlock handling [7]. The 2013 Bulletin on Data Engineering [11] had eight articles on recent MMDBs and only three mentioned distributed transactions. Implementing fast, serializable, distributed transactions on an MMDB is difficult, since communication delays typically leave some CPUs idle and reduce total throughput.

We began a project in Fall 2011 to improve distributed transactional performance of the open-source MMDB VoltDB system [14], which was based on an earlier academic prototype MMDB H-Store [1]. We developed a low-overhead concurrency method that executes consecutive Prepares with delayed Commits on each node (CPU) and takes Write locks but not Read locks to detect conflicts. We developed an Ordered Escrow Method, a variant of Escrow [14] to greatly speed up transactions with incremental updates. We named our VoltDB modification CVoltDB (C for Concurrency) and proved it supports replica consistency and serializability. Full TPC-B and TPC-C benchmarks demonstrate greatly improved performance due to new features in CVoltDB.

Categories and Subject Descriptors

C.2.4**: [Distributed Systems]** Distributed databases

General Terms

Algorithms, Measurement, Performance, Design.

Keywords

Concurrency Control, Distributed Database, Performance, Escrow, Shared Nothing

Topics: Concurrency Control and Recovery, Distributed and Parallel Databases, Database Performance, Database Services and Applications

IDEAS '14, July 07 - 09 2014, Porto, Portugal

http://dx.doi.org/10.1145/2628194.2628242.

1. INTRODUCTION

Classical RDBMS products, such as Oracle, DB2 and SQL Server, are Disk Resident DBs (DRDBs), with designs from the 1970s, whose data can only be read or updated after it is brought into memory. Enormous reductions in memory prices and ever-faster compute cycles have made disk accessed data much less desirable for all but the largest databases. Memory-resident transactions can run to completion on a uniprocessor with no stalls by avoiding I/O waits and transactional user interaction. This saves overheads for locking, latching and memory buffering, using about 64% of the CPU time in DRDBs [7].

A shared-nothing distributed database horizontally partitions data into shards of allied rows on distinct nodes. A bank database typically assigns accounts of bank branches to a distinct shard, with each node's transaction processor having exclusive access to its shard. If all the information needed by a transaction is held in a single shard owned by a certain node, we call the transaction a single-node transaction. A transaction needing data from different shards is a distributed transaction.

In a sharded MMDB running single-node transactions, latches and locks are not needed if transactions execute one at a time on each node and thus run without blocking. In this way, we eliminate buffering, latching, and locking. A lightweight logging method, for example to an SSD device, is still needed to provide recovery. Such a system provides very high performance for single-node transactions, as argued in [16] and shown in practice in H-Store and VoltDB. When we run distributed transactions on a sharded MMDB, each transaction needs to execute on several nodes and communicate between them. Under the serial execution rule at each node, many waits for messages will occur in mid-transaction, and performance nose-dives.

Distributed transactions are directed from a single Coordinator node and have two phases of execution: a Prepare P and a Commit C (or Abort A). Single-node transactions also have Prepare and Commit phases. With mixed distributed and single-node transactions, a sequence such as P1 C1 P2 A2 P3 C3… runs on each executing node. Each node of a distributed transaction must wait for a Coordinator Commit message after a Prepare phase completes and this is pure communication wait time. The challenge is to find a concurrency mechanism that allows useful work during communication wait, yet avoids re-introducing the latching and locking disk-based systems need. We note that communication wait times are about 100 times shorter than Disk waits (about 30us vs. 3ms), about the same length as the transactional work for a Prepare, so transactions need to overlap only in a minor way to fill in communication wait times.

Our CVoltDB scheme (like the one in [8] but simpler) has transactions perform Consecutive Prepares on a node, e.g.: P1 P2

C1 C2, or P1 P2 P3 C1 P4 C2 A3 C4, etc. Here the transactions can be single-node or distributed and new transactions can Prepare while distributed transactions wait for Commit messages. We run single-threaded on the node and assume transactions execute Prepares, and separately, Commits/Aborts in transaction ID (TxID) order. Write locking is used to handle conflicts while multiple Prepared transactions exist at once: Prepares that try to read or update a column of a row updated by a prior Prepare will block because of a conflicting Write. See Section 4 for further details. The Escrow Method [14] addresses this delay by supporting Escrow updates that don't block subsequent Escrow updates of the same column. For CVoltDB, we have developed the Ordered Escrow Method, described in Section 5**.**

In what follows, we describe a version of VoltDB [17] that we improved from the version we first encountered by guaranteeing distributed transactions on multiple nodes (CPUs) run on only the nodes required. See Section 3.2.1. The original VoltDB developers made little effort to optimize distributed transactions since their customers had no need of them. With this improvement, VoltDB is still a sharded MMDB with serial execution on nodes. We added our concurrency schemes of the prior paragraph, with runtime flags to allow selection of features. Implementation is further discussed in Section 6.

In a sharded MMDB with serial execution on nodes (without Consecutive Prepares or Escrow), even a few distributed transactions in the mix makes performance nose-dive due to communication delay between nodes. We ran the TPC-B benchmark with a varying proportion of distributed transactions to measure the effectiveness of Consecutive Prepares and Escrow updates to improve performance of transactions with incremental updates. Our changes have no effect at 0% distributed, but at 5% distributed (on a 4 host cluster) our scheme improves TPS by 90% over that of VoltDB and by 174% at 15% distributed, the standard TPC-B mix. We also tested the TPC-C benchmark, with its more complex transactions. As expected, the quantitative results are not as striking, but still significant. See Section 7 for more information.

CVoltDB makes distributed transactions run faster, but still has problems with load balancing. Because of the tyranny of execution in TxID order, the system runs at the rate of the most loaded partition, among partitions interrelated by distributed transactions. VoltDB has recently changed the TxID generation algorithms to allow partitions assign compatible per-partition TxIDs to distributed transactions to help with this load balancing problem.

NOTE: **The CVoltDB sources are available at https://github.com/wwgong/CVoltDB.**

2. RELATED WORK

Mike Stonebraker, one of the authors of [3] who foresaw main memory databases with lower memory prices of the future, spearheaded a collaborative project to implement such a system in 2007. The academic prototype called H-Store [16], which showed the value of MMDB in OLTP applications, and in particular, the shared-nothing approach with single-threaded nodes in H-Store. H-Store was announced in [9] and the open-source commercial product based on this prototype is VoltDB.

H-Store runs transactions one at a time on each node and this approach was inherited by VoltDB. In [8], an approach called speculative execution was built on H-Store to execute a second transaction on the same node with the assumption that the first transaction would Commit, an attempt to utilize time between Prepare and Commit of a distributed transaction that would otherwise be idle. But if the first transaction Aborts then the second transaction would need to Abort (under the assumption the second transaction might have read something changed by the first transaction). Paper [8] also provided a second concurrency approach with both Read and Write locks. This approach performed better than speculation at higher abort rates or higher distributed transaction ratios. We consider CVoltDB to be closer to this approach, but without Read locks.

Many current commercial transactional MMDBs, including Oracle's TimesTen and Microsoft's Heketon, are centralized, so their threads access data in a single shared memory, limiting scale-up. Two notable commercial distributed MMDBs, VoltDB (open-source) and SAP's HANA (proprietary), both described in [11], use shared-nothing partitioning, i.e. sharding. HANA uses multi-threaded nodes and distributed snapshot isolation, allowing read-only transactions that read from a snapshot evolving from OLTP transactions. VoltDB uses single-threaded nodes and is discussed in Section 3.

The Calvin MMDB system developed at Yale [12,13] executes distributed transactions in a predetermined serial order without the delays of two-phase commit by pre-analyzing each transaction for its read and write sets, using "dry runs" if necessary, and then scheduling them to avoid conflicts. It uses horizontal partitioning and allows (limited) concurrency on each partition. The CVoltDB Consecutive Prepares execution qualifies as following Calvin's deterministic locking, without requiring pre-analysis and usually fills in the two-phase locking delay with useful work.

Escrow-like operations are so common in business applications that in 1976 IBM released Dieter Gawlick's IMS Fast Path [5], with Increment/Decrement updates of data kept in memory for high-speed changes that couldn't wait for regular database handling. This inspired the Escrow Method for non-blocking Aggregate-Increment update appearing in ACM TODS in 1986 [14]. It has been used to maintain indexed summary views [6], in the application tier of Oracle as Compensation-Aware Data Types [19], and in an academic project as "B data" layered on Amazon S3 cloud storage [10]. But we believe this is the first time Escrow has been offered as a native database capability to transaction programmers. We will see the value of Escrow performance when we run benchmarks in Section 7.

3. BACKGROUND: VoltDB & CVoltDB

VoltDB [17] is an open-source, sharded MMDB with single-threaded nodes that evolved from the academic prototype H-Store. Changes include robust error handling and many other production database features. VoltDB does not need buffering, latching, or locking, greatly simplified from any DRDB engine. After reviewing VoltDB code, we decided we could improve performance using consecutive prepares on each node and the Escrow Method, as discussed in the Introduction. We call our variant system "Concurrent VoltDB", or CVoltDB.

3.1 CVoltDB and VoltDB common features

VoltDB horizontally partitions data into replicated shards of allied rows on distinct hosts. If one host fails during a transaction, it continues to run on replica shards. If the number of shards on distinct hosts is K+1 ($K \geq 1$) then the system provides K-Safety,

meaning K hosts can crash leaving transactions still able to run, a feature called High Availability (HA). Transaction programs are stored procedures registered in the system and dynamically loaded when they are to be run. During transaction initiation, VoltDB writes a Command Log containing a TxID, Stored Procedure ID and Stored Procedure arguments. The Command Log has about 100 bytes and is quickly written (for example) to a Solid State Disk with a capacitor-backed memory buffer. VoltDB writes a Snapshot copy of the complete MMDB onto disk as of some committed transaction, using a copy-on-write technique. If all hosts crash, VoltDB and CVoltDB can recover from the last complete Snapshot on Disk, using later Command Logs to replay transactions.

Each transaction is deterministic in VoltDB and CVoltDB, in the sense that no external input (e.g., system clock time) is allowed during transaction execution. This is to guarantee replica consistency, i.e. so transactional data updates on replica shards are consistent, since clock time at different replicas might differ. When a transaction program is compiled as a stored procedure, it is registered in the system and later run by providing the Procedure ID and parameter values. Transactions get unique TxIDs based on the time they arrive (most significant bits) and the host where they arrive. The VoltDB on which we built our CVoltDB system executes transactions in TxID order, one at a time on each node, ensuring replica consistency and serializability

VoltDB and CVoltDB have three transaction classes: Single node transactions, such as a withdrawal or deposit at a bank branch, are most common. Distributed transactions can be either one-shot or multi-shot. A shot is a round-trip message exchange between participant nodes of a distributed transaction and the Coordinator. A one-shot Distributed Transaction might be a transfer of money from one bank branch to another on a different node. Multi-shot transactions can have multiple shots, each returning data to the coordinator before the next shot begins. This is needed when new data can lead to a new decision, since each shot runs only non-procedural SQL, but a decision can be made by the stored procedure code from data returned to the Coordinator. Some transactions in the TPC-C benchmark must use multi-shot transactions, as we explain and illustrate in Section 7.

VoltDB distributed transactions have performance problems. A single node transaction runs from start to finish without interruption and distributed transactions runs concurrently on many nodes, but in VoltDB a distributed transaction locks up all nodes on all hosts. This bottleneck was removed in CVoltDB, as explained in Section 3.2.

3.2 Changes from VoltDB to CVoltDB preparing for Concurrency

In order for Consecutive Prepares on a node (Section 4), and Ordered Escrow (Section 5) to work well, we needed to add features to speed up distributed transactions as described below.

3.2.1 Distributed Transactions Limited to Needed Nodes

When VoltDB needs to run a distributed transaction, the Initiator sends the Stored Procedure Identifier and parameter values to a designated node acting as a Coordinator. The Coordinator sends out transaction code (compiled from a SQL batch) to ALL nodes. Various nodes execute the code when their turn to run arrives. The nodes not actually involved in the transaction will find no matching data and return a null set to the Coordinator. Thus all nodes are locked up during distributed transactions.

In CVoltDB we allow multiple parameters of a stored procedure to specify which subset of nodes are actively involved in a distributed transaction when this is fixed in advance, e.g.: in TPC-B and TPC-C. This allows us to run many distributed transactions concurrently on the system. We call this CVoltDB feature: Distributed Transactions Limited to Needed Nodes. This feature is the most important for performance of the three features mentioned in these subsections and was also supported by H-Store.

3.2.2 Early Code Distribution in CVoltDB

Another feature of distributed transactions in CVoltDB is named Early Code Distribution. A VoltDB coordinator readies a distributed transaction T_k after its work on the previous transaction has finished. This involves running the stored procedure, which assembles code and sends it out to the transaction participant nodes, which may be idle, waiting for the code. A CVoltDB Coordinator sends out code for T_k to participating nodes before it is time for T_k to be readied on the Coordinator. These nodes will queue this code and then run T_k as soon as all prior transactions on the node have Prepared, allowing T_k participants to run their Prepare phase earlier and return results to the Coordinator. The return of results may happen even before it is time for T_k to be readied, but the Coordinator will queue the results and when the time for T_k to be readied arrives the Coordinator can process the results for T_k immediately. The inevitable wait has become a probabilistic wait (depending on which prior transactions run on each node).

3.2.3 Optimal Coordinator Use in CVoltDB

In VoltDB systems the Coordinator node is arbitrarily chosen, but CVoltDB lets the programmer specify that the Coordinator sit on a node of a specified partition used by the transaction to reduce communication in distributed transactions.

4. CONSECUTIVE PREPARES ON A NODE

As discussed in the Introduction, CVoltDB transactions perform Consecutive Prepares on a node while delaying Commits e.g.: P1 P2 C1 C2, or P1 P2 C1 P3 C2 A3, etc. Note we run single-threaded on the node and all transactions run in TxID order on each node both in Prepares and Commits/Aborts. The Prepare phases of different transactions on a node must be executed one at a time. But when a multi-shot transaction Prepares on a node and returns data to its coordinator, it may have more Prepare work to do on this node in a later shot. Thus multi-shot communication can delay Prepares by later transactions.

With non-Escrow column updates we use Write locks on rows for serializability and if T_1 Updates a column C of a row and T_2 later Reads C, T_2 need not take a Read lock on C but it must TEST if the column has a prepared update, and if so it must block on T_1's Write lock. The lock will be released when T_1 Commits. For example: P1 P2 (blocks) C1 P2 (unblocks) C2. If T_1 Reads C, it need not take a Read lock since even if a Write lock on C is taken later by T_2, the Read by T_1 was valid at the time it was executed until the end of T_1's Prepare phase (by the rule of one-at-a-time Prepares), and the Write by T_2 occurs later in TxID order. Note that if T_1 Write locks a column C1 of R and T_2 later tries to Write

lock a different column C2 of R, T_2 will block. The rationale for this is that when T_1 Updates C1 it creates a Before Image (BI) of R, where C1 has its old value. If T_2 later updates C2 than a new BI would be needed, but the value C1 would have to be ambiguous in this BI (old value or new value). We provide for phantom detection of Prepared updates by creating a BI row R' when T_1 updates one of more columns in R, with both R and R' held in index entries as in KVL locking.

5. THE ORDERED ESCROW METHOD

CVoltDB tables can have one or more Escrow columns in each row, based on work in [14, 15].

5.1 Escrow Columns and Operations

An Escrow column such as "account_balance" can be incremented with a positive or negative increment, INC. A transaction Ti can make an Escrow Update Request of an Escrow column C in TBL1 below, using a SQL update statement with this special syntax:

Update TBL1 set C += INC where...
or
Update TBL1 set C -= INC where...
(Note: if INC is negative, C -= INC will add |INC|.)

After such a request succeeds, the result is an Escrow grant, and this grant will be part of an Escrow Journal on a node as defined below. The value of Escrow is that many increments by many different transactions can be granted on a single Escrow column without blocking and later committed or aborted. There can also be Escrow requests granted by a single transaction on many Escrow columns

Example 5.1 Consider an Escrow column with identifier ColIdx in the table TBL1. If the Column represents (say) money in a bank account, the largest value allowed (MAX) might be $100,000 (without a special account) and the smallest value (MIN) might be $10 (minimum balance). We explain below how MIN and MAX limit possible Escrow grants. VAL is the current value of the column if all outstanding Escrow grants are Committed. Two new values, INF and SUP, have the same type as VAL and after multiple requests, INF is the smallest possible value if all grants with positive increments Abort and all grants with negative increments Commit, while SUP is the largest value in the inverse case. Thus [INF, SUP] bracket possible committed values of active transactions. There are six values in a so-called Escrow column ColIdx. The data structures that hold this data are discussed in Section 6.1.

ColIdx; MIN = 10 MAX = 100000 INF = 1000, VAL = 1000 SUP = 1000

Any precise numeric type can be used in Escrow, but long integers are common (e.g.: pennies in bank balances). As increment and decrement grants occur, a chain-linked list of Escrow Journals for grants is created, with the header containing the updated Escrow Column. An Escrow Journal has this layout.
Escrow Journal

TxID = 12456 INC = -20

Hashing on a row pointer lets us access these Escrow journals (see Section 6.1). A sequence of updates to Escrow Column ColIdx is given below (no Chain-links are shown).

Start

ColIdx; MIN = 10 MAX = 100000 INF = 1000, VAL = 1000, SUP = 1000

After Escrow grant 1 for T_1

ColIdx; MIN = 10 MAX = 100000 INF = 950, VAL = 950, SUP = 1000
TxID = 1 INC = -50

After Escrow grant 2 for T_2

ColIdx; MIN = 10 MAX = 100000 INF = 950, VAL = 990, SUP = 1040
ColIdx; TxID = 1 INC = -50
ColIdx; TxID = 2 INC = 40

After T_1 Commits

ColIdx; MIN = 10 MAX = 100000 INF = 950, VAL = 990, SUP = 990
ColIdx; TxID = 2 INC = 40

After T_2 Aborts

ColIdx; MIN = 10 MAX = 100000 INF = 950, VAL = 950, SUP = 950

If T_1 and T_2 both aborted, the column would return to its initial state: INF = VAL = SUP = 1000.

5.2 The Escrow Out-of-Bounds Rule

In edge cases, we need to block Escrow requests until Commits and Aborts can resolve ambiguity in an Escrow value. This preserves deterministic execution, needed for replica consistency and recovery.

Consider a request for an Escrow update by transaction T_k that would add INC to an Escrow column C starting with a certain [INF, SUP] interval. If INC is so large that INF+INC (and thus SUP+INC) is larger than MAX. This request cannot be granted, since if T_k were to Commit the ultimate committed value would be out of acceptable bounds. Similarly a request cannot be granted where INC is so negative that SUP+INC (and thus INF+INC) falls below MIN. In both cases T_k must Abort.

Now if INF+INC and SUP+INC both fall in the range [MIN, MAX], the Escrow request is perfectly acceptable. But what if only one of these conditions failed? That is, if [INF+INC, SUP+INC] intersects [MIN, MAX] but projects to one side or the other. Then some possible outcome of committing and aborting transactions might bring the committed Escrow value outside the range [MIN, MAX]. One thing we know for sure is that all Escrow requests prior to the one by T_k left us within the valid [MIN, MAX] range for any combination of Aborts and Commits. But we cannot simply Abort T_k, since it is possible that a replica node that has made further transactional progress than ours would

immediately see it can grant T_k's request. This means we must BLOCK this new Escrow Request until enough Aborts and Commits have occurred so we know if the new request should succeed or fail. Since transactions Prepare in TxID order, no transaction that comes after this uncertain case for T_k can Prepare until we decide whether to grant the T_k request or Abort it. We summarize this rule below

Escrow Out-of-Bounds Rule. As an Escrow request by Tk adds INC to column C, we have three cases

If [INF+INC, SUP+INC] falls within [MIN, MAX], the request is granted and the new [INF, SUP] interval is set to [INF+INC, SUP] if INC < 0 or [INF, SUP+INC] if INC > 0.

1. If [INF+INC, SUP+INC] doesn't intersect [MIN, MAX], the request fails and Tk Aborts.
2. If [INF+INC, SUP+INC] intersects but is not contained in [MIN, MAX], T_k blocks while all prior transactions Commit or Abort at which point [INF+INC, SUP+INC] is a 0-length interval in or out of the [MIN, MAX] range and Tk commits of Aborts accordingly. This decision might be clear earlier if [INF+INC, SUP+INC] lies inside or outside [MIN, MAX] before all prior transactions Commit or Abort.

Example. In Example 5.1, after T_1 and T_2 Prepare, we have INF = 950 and SUP = 1040. If now T_3 Prepares a Decrement of -1000, [INF+INC, SUP+INC] = [-50, 40], so we must block T_3 until we find if Committing will result in a number in the [MIN, MAX] range, [10, 10000]. If T_1 Aborts and T_2 Commits, then T_3 can Commit and leave a value of 40, but any other actions for T_1 and T_2 will cause T_3 to Abort.

An index cannot contain Escrow columns since their values are uncertain when updates are pending. BIs replace non-Escrow column updates in an Abort but Escrow columns handle Aborts differently.

5.3 Ordered Escrow vs. Original Escrow

Note that a transaction Prepare may read an Escrow column, but the read will block until Escrow updates by all prior transactions commit. In the original Escrow paper [11], VAL was defined to be the most recently committed value so uncommitted Escrow updates of this column might later Commit or Abort, changing VAL. Our definition of VAL is the assumed-commit value of outstanding updates on the column, so if a transaction to performs an Escrow update itself and later Reads the column; the Read will wait for all prior transactions to Commit or Abort and then see its own update as part of the value read.

In the original Escrow environment of [14], if we wanted to sum the balances of two frequently updated bank accounts, the Escrow reads of the two balances would not necessarily have been consistent. The fact that CVoltDB executes Prepares and Commits/Aborts in TxID order on all nodes, removes this limitation and serializable consistency of Escrow Reads in distributed transactions is guaranteed. Furthermore there is no chance of deadlock among Escrow updaters as was possible in [14], since a deadlock requires interleaved Prepares. Escrow-related deadlock was shown in [15] to be exponentially difficult to handle, with the NP-complete complexity of the generalized banker's algorithm.

Our new syntax for Escrow update in Section 5.1, using += and -=, was not in [14], which required specialized database calls.

6. CVoltDB IMPLEMENTATION

6.1 CVoltDB Data Structures

After T_1 updates a non-Escrow column in a row R, the row is said to be dirty and a BI is created; if T_2 then tries to update any non-Escrow column in the dirty row, it must wait until T_1 has committed: T_2 is blocked for non-Escrow updates. But the term Write lock is a bit of a misnomer, since after the non-Escrow update of R by T_1, Escrow columns of R are still updatable by T_2. Recall from Section 4 that Read locks are never required in CVoltDB. A BI created to reverse non-Escrow column updates in a row during Abort must not contain the Escrow columns in that row. One should think of Escrow columns as lying outside the row of non-Escrow columns in every normal transactional sense.

We record both non-Escrow and Escrow column updates in the row by creating a Row Journal, accessed by hashing on the row pointer. The Row Journal has the following layout.

Row Journal

Pointer to AI of this row: PA
Pointer to Row Schema: PS
Pointer to non-Escrow changes: PN
Pointer to Escrow Update data: PE

- PA is a pointer to the After-Image (AI) of the row.
- PS is a pointer to a Row Schema listing all columns, Escrow or non-Escrow, their type and ColIdx.
- PN points to an information struct for non-Escrow row changes (if any) including: TxID, type (Insert, Delete, Update) and a pointer to the BI, a temporary row in the table.
- PE points to a series of pointers to each column's Escrow update information shown in Section 5.1

If a row has no non-Escrow changes then PN will be null. If there are no Escrow updates, PE will point to a null sequence and if there are no Escrow columns in the row, PE itself will be null. But if a row has no uncommitted changes at all, it will have no Row Journal!

6.2 SQL Operations in CVoltDB

CVoltDB does not itself handle schema changes or table loads, but does support VoltDB transactions in the transaction sequence that can do such work, not allowing concurrency with preceding or following CVoltDB transactions. CVoltDB handles all the DML operations, as follows.

6.2.1 Row Insert

We follow VoltDB practice by inserting rows immediately in the transaction Prepare phase, along with primary and secondary index entries, backing out the inserts if an Abort occurs. To avoid potential phantoms, we show a newly inserted row as dirty in the Row Journal until the transaction inserting it Commits or Aborts. This acts as a KVL-like phantom to stop table or index scans.

Row *Delete*

A Delete of a row R creates a Row Journal if one is not preexisting, to show the row is dirty; earlier Escrow column updates will complete in order before the Delete is committed; we leave the row and all index entries in place, to avoid phantoms, blocking later range reads until the delete of T_j Aborts or

Commits. After the deleting transaction T_j Commits, the row will be invisible until it is removed during later Snapshot processing (Section 3.1).

6.2.2 Non-Escrow Update Operations

Updates of a non-Escrow column will create a BI of the row in a new position in the table with entries in the same indexes, while the AI of the row will continue with the old secondary index entries, updated as needed. Both row pointers hash to the same Row Journal so other transactions block in reading either row (by index range or direct search) until the transaction performing the update Commits. We do not permit updates of columns involved in unique indexes at this time. On Commit we remove the BI and its index entries. On Abort, the updated row reverts to values in the BI for non-Escrow columns and drops Escrow changes.

6.2.3 Reading Escrow and Non-Escrow Columns

We saw in Section 5.3 that it is possible to read (by Select or in a WHERE clause) the value of an Escrow column by waiting for pending Escrow updates of that column to Commit or Abort. Thus reading Escrow columns temporarily nullifies the performance advantage of Escrow updates.

A Select statement or WHERE clause can retrieve one or more Escrow or non-Escrow column values from a row or a range of such rows. The query processor hashes on the row pointer of any row it encounters to read the Row Journal (Section 6.1). A range search to locate rows must block, as in KVL locking, if it finds a row with an updated column it retrieves or one that is involved in the WHERE clause. If the query blocks, it must delay its read until updates Commit or Abort. With an Escrow column, there can be multiple transactions updating it, which must all resolve before we can read that column.

Note we have more freedom performing reads of non-Escrow columns than in classical databases, where an update takes a full Write lock on the row. For single-variable search conditions, we can read columns of rows without blocking as long as the columns we read don't include columns being updated: the Row Journal PN (Section 6.1) points to a Boolean array that shows updated non-Escrow columns. The granularity of our locking is generally the same as in KVL locking.

6.3 Serializability and Replica Consistency

Given one-at-a-time Prepares on a node, all operations of a TxID are Prepared before operations of a later TxID can begin Preparing. Distributed transactions Prepare on several nodes, each in proper TxID order on that node. Thus all conflicts among transactions on each single node yield edges in the Precedence graph going from transactions with lower TxID to ones with higher TxID. A Precedence graph thus cannot have a cycle and the system provides serializable execution in TxID order.

For Replica Consistency we need to show two replicas cannot have divergent outcomes in CVoltDB. As explained in Section 3.1, transactions are deterministic in CVoltDB in the sense that no external input (e.g., system clock time) is accessed during transaction execution. Two Replicas will Prepare and Commit the same transactions in the same order, but they might Prepare and Commit at different times. If the transactions have only standard operations (non-Escrow), serializable execution on the two Replicas would have identical results.

Of course all Escrow updates will have identical increments on all replicas, but the replicas could have different numbers of recent uncommitted transactions at any point and thus different [INF, SUP] intervals. We will show that two replicas cannot diverge in outcomes because of this. Consider the first decision point in prepare P_i that has different decisions (succeed vs. fail) on different replicas, possibly at the end of a blocked period caused by the Escrow Out-of-Bounds Rule. All earlier Escrow grants were made consistently so they are the same for all replicas at this decision point in P_i. All previous prepares have completed, so the same sequence of commits/aborts are in process.

Although two replicas could have different [INF, SUP] intervals valid in P_i, they both contain one common point, the committed value implied by all Escrow grants on the column value, resulting from the already-determined (but not yet processed) sequence of commits and aborts of the earlier transactions. Because of the non-empty intersection between the two [INF, SUP] intervals, it is impossible that one [INF, SUP] interval will end up entirely inside [MIN, MAX] and the other one entirely outside.

7. BENCHMARKS

In this Section, we compare the performance of CVoltDB against VoltDB in two benchmarks, the TPC-B benchmark (no longer officially recognized) and the current TPC-C [18] benchmark. This is clearly an unfair comparison, since VoltDB was not optimized for distributed transactions, but it seems to be a proper comparison to characterize the new performance features of CVoltDB. NOTE: we ran TPC-B and TPC-C without replication of benchmark data on our rather limited four-host cluster. We did test replication and snapshot processing separately to ensure that CVoltDB fully supports them.

All performance tests for TPC-B and TPC-C are run on our 4-host cluster, with 4 Dell minitower systems, quad-core Core i5-3450 3.10GHz CPUs and 16GB memory. The systems are connected by a Gigabyte Ethernet switch, and run 64-bit Ubuntu Linux 12.04 LTS (Linux version 3.2).

We ran benchmarks four ways, each Successive Configuration adding new CVoltDB features.

Configurations

1. Original VoltDB. VoltDB 2.1.3, with one performance bug fix to avoid unnecessary shots.
2. Basic Features. CVoltDB with the additional features of Section 3.2, but still running only one transaction at a time on each node.
3. Consecutive Prepares. CVoltDB with its additional features of Section 3.2, and those of Section 4.
4. Ordered Escrow. Full CVoltDB processing, with Escrow Columns.

7.1 The TPC-B Benchmark

The TPC-B benchmark measures performance of banking transactions, with four tables, as follows.

Table Name	# of Rows	Row size	Primary key
Branch	N	100 bytes	B_ID
Teller	10N	100 bytes	T_ID
Account	100,000N	100 bytes	A_ID
History	Varies	50 bytes	

TPC-B explicitly allows horizontal partitioning, and appropriate partitioning is by B_ID. For each successive transaction, a Driver sends to the database four separate integers, Aid, Tid, Bid and Del

(values for A_ID, T_ID, B_ID and a Delta Increment). The transaction returns the Account Balance to the Driver.

Transaction Logic Profile: Given Aid, Tid, Bid and Del and a Timestamp TS from the Driver. We use the Escrow += extension to SQL syntax explained in Section 5.1.

```
BEGIN TRANSACTION
Update Account set Balance += Del where A_ID = Aid;  -- Escrow
Select Balance from Account where A_ID = Aid;  -- Read Escrow
Update Teller set Balance += Del where T_ID = Tid;
Update Branch set Balance += Del where B_ID = Bid;
Insert into History Values (Aid, Tid, Bid, Del, TS);
COMMIT TRANSACTION
Return Account Balance Aid to Driver (i.e., Client);
```

Note that this program uses normal SQL updates until the Escrow capability is added in Configuration 4.

A transaction is single-node if the Account's Branch, Aid/100,000 equals Bid; if they're different, it is a distributed transaction with two Branches, the Branches are in two partitions and the transaction is distributed. By design, 85% of the transactions use a single branch and 15% a second foreign Branch at random. TPC-B has no test that balances remain in a given range, a surprising lack. The Select from the Account Balance will force all prior transactions updating this row to Commit (unlikely, since there are 100,000 Accounts to a Branch) and return this value to the Driver. Note that the Account Balance, Teller cash and Branch balance are all updated by the amount of the Account change.

Note too that there is a scaling rule in TPC-B requiring a Branch row for each nominal TPS recorded, which would require, for a measurement of 10,000 TPS, at least 100 billion bytes for Account tables. We ignore this requirement. Some TPC rules were clearly intended to guarantee that no little upstart company would be able to challenge IBM, Oracle or Teradata. For example neither Vertica nor Sybase IQ were allowed to run the query benchmark TPC-D because of a rule that no vertical partitioning was allowed. Why not? Unfair competition!

7.2 TPC-B Benchmark Performance

Although the TPC-B benchmark is officially obsolete, it provides a well-known distributed workload that has hot-spot behavior. The benchmark specifies that 15% of transactions should involve two branches, thus two partitions, and others just one. We scale this distributed percent from 0 to 100% to see the effect of the mix.

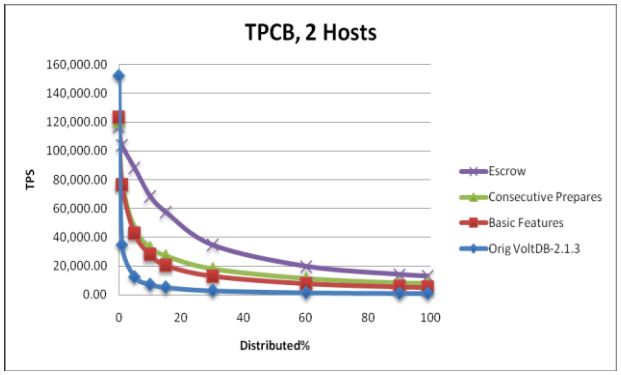

Figure 7.1 TPC-B by distributed%, on 2 hosts

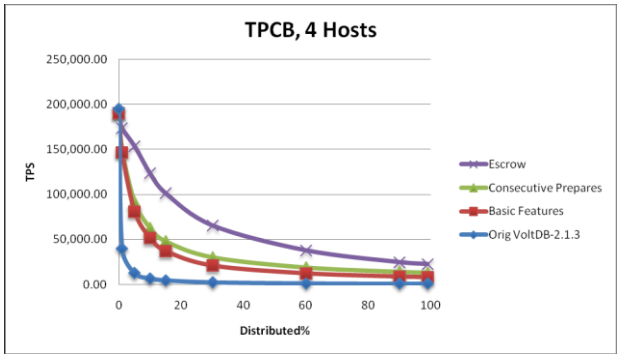

Figure 7.2 TPC-B TPS by distributed%, on 4 hosts

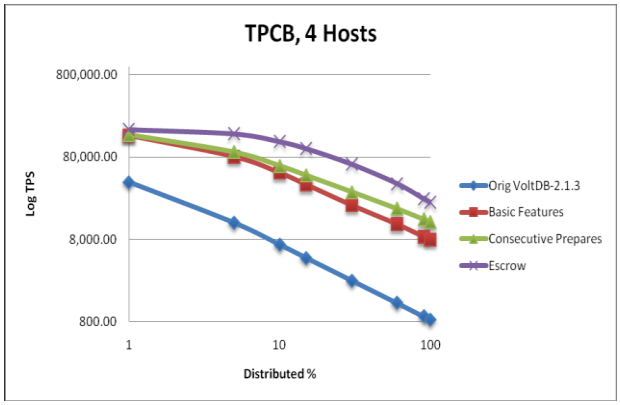

Figure 7.3: Figure 7.2 with log-log scale

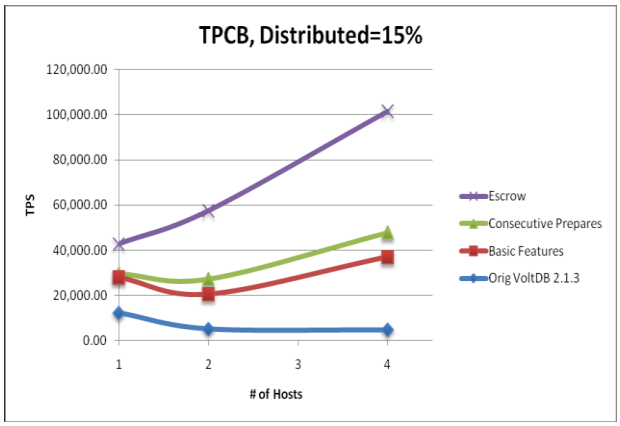

Figure 7.4 TPC-B TPS by # of hosts, 15% distributed

These graphs show VoltDB runs pure single-node transactions (Distributed=0%) faster than CVoltDB, but as soon as 1% of transactions are distributed, CVoltDB is much faster, about ten times faster at 15%, the standard TPC-B mix. Figure 7.4 shows how CVoltDB scales up in all its versions. The original VoltDB was very slow for distributed transactions because they locked up all nodes.

Basic H-Store [8] pre-loaded stored procedures so Early Code Distribution wasn't needed. It ran transactions one at a time on each node and distributed transactions were limited to needed nodes. In that form H-Store had comparable performance to VoltDB with Basic Features (Red curves of Figures above). H-Store can also do dynamic loading in its current form.

Consecutive Prepares but no Escrow (green curve) improves distributed transactions, but not by much. All transactions update the branch balance by a non-Escrow operation and thus each one blocks until the previous transaction Commits. The Escrow Method makes a big performance difference.

7.3 The TPC-C Benchmark

VoltDB programmers wrote TPC-C Benchmark code and we made changes to optimize shots and add Escrow updates in our version. The TPC-C benchmark has nine different tables and five different transactional profiles to order, pay for, and deliver goods from warehouses. Figure 7.5 lists the TPC-C transactions and their properties.

Transaction RO = read only	% in mix	% distributed	#Partitions involved
New-Order	45%	10%	1-15
Payment	45%	15%	1-2
Delivery	4%	0%	1
Order-Status (RO)	4%	0%	1
Stock-Level (RO)	4%	0%	1

Figure 7.5. Transactions in TPC-C

The TPC-C benchmark scales by number of warehouses W, with a total of 500,000W rows, plus 100,000 rows for Item. We used one warehouse for each node, and at least 4-12 nodes/host, so W = 16-48 for 4 hosts, with 8.1M to 24.1M rows.

7.3.1 The Home Warehouse Partition

TPC-C data can be partitioned by warehouse by TPC-C designer intention. Each transaction relates to a specific home warehouse (data in the home warehouse partition) and accesses only this home partition's data for Delivery, Order-Status, Stock-Level and most cases of New-Order and Payment, making them single-node transactions. As listed in Figure 7.5, distributed transactions are used for 10% of executions for New-Order and 15% for Payment. For the distributed Payment transaction, exactly two partitions are accessed, the home warehouse and one other, called the remote warehouse partition. For distributed New-Order, up to 15 remote partitions might be accessed, but most cases access only one or two remote partitions for up to 15 line items, each remotely stocked only 1% of the time.

7.3.2 TPC-C Transactions

New-Order. A new order is placed for a Customer (a row in the Customer table). The home warehouse is the customer's, and its node is the optimal Coordinator in CVoltDB (See Section 3.2.3). The order inserts a row in the Order table, a row in New-Order, and 5 to 15 rows in Order-Line. Some Line Items may be stored in remote warehouses, and require remote stock-level adjustments. Here are the major steps of New-Order:

1. An average of 10 stock items are deducted from quantity on hand in Stock table (with 1% chance of each item being remote). If remote, Stock-related data returns to the Coordinator.
2. The transaction Aborts if an Item is not found (1% non-existence required by the spec). On the home partition, inserts occur for Order, New-Order and Order-Line rows using data gathered in Step 1.

On VoltDB, in the fastest case where the coordinator runs on the home partition, Figure 7.6 shows shots performed for steps 1 and 2 and then Commit in the distributed case. The Java stored procedure runs on Coordinator (J) and sends work to the remote node; then both nodes execute step-1 work "111". Results are returned to the coordinator (second J) where the program runs again and sends out step-2 work to the remote node, and both nodes executes the step-2 work "222". Results are returned to the program (third J), the commit decision is made, and Commit sent to the remote node, then acknowledged back.

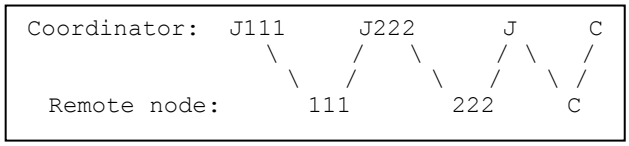

Figure 7.6. Time-Line for NewOrder on VoltDB

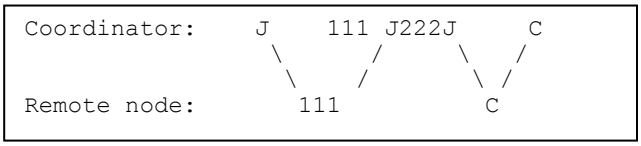

Figure 7.7. Time-Line for NewOrder on CVoltDB

The exclusive period for VoltDB, the period on a node when no other transaction can execute, runs from Prepare start to Commit time, since no other transaction can start on the node until after Commit. Figure 7.6 shows the exclusive period is 3RTT on the Coordinator and 2RTT on non-Coordinator nodes.

Figure 7.7 shows CVoltDB execution of the same New-Order transaction. Early Code Distribution (Section 3.2.2) is shown as J at the Coordinator, well ahead of the step-1 execution ("111") on the home warehouse partition, the optimal location (Section 3.2.3). The second work step occasionally needs remote access for a required update to add to stock levels if they've fallen too low. If not, CVoltDB runs on the Coordinator only for one step.

By Figure 7.7, the exclusive period for CVoltDB is possibly less than one RTT, but if the remote node needs to finish earlier transactions before executing this one, it may be longer. On the one remote node, the exclusive period extends to the Commit, roughly one RTT.

Payment Program The Payment program accepts a payment for a customer, possibly belonging to a remote warehouse, increments the customer balance and also increments the home warehouse and district year-to-date values, all increments using Escrow. Then it inserts to the History table.

Delivery (Single node Transaction) delivers the oldest undelivered order for the district. Its row is deleted from New-Order, and its Order-Line rows updated. The Customer balance is decremented, by Escrow update.

Order-Status (Single node Transaction, Read-Only) queries the status of a customer's last order, including the customer balance, an Escrow quantity.

Stock-Level (Single node Transaction, Read-Only) determines which of the items ordered by the last twenty orders in a given warehouse and district, have fallen below a specified threshold value.

7.4 TPC-C Benchmark Performance

Figure 7.8 (below) shows the performance dependence on the number of nodes per host in use. With a quad-core system, we expect to use at least 4 nodes per host, and a typical value is 6 nodes per host even for pure single-node executions. We ran multiple nodes per host up to 12 nodes per host. The fact that additional nodes (more than 1.5 per CPU) provide better performance means the system is encountering significant delays on each node, although not as great as VoltDB. This situation is also reflected in observed CPU percentages well below 100%. VoltDB cannot take much advantage of additional nodes because each distributed transaction takes over the entire cluster.

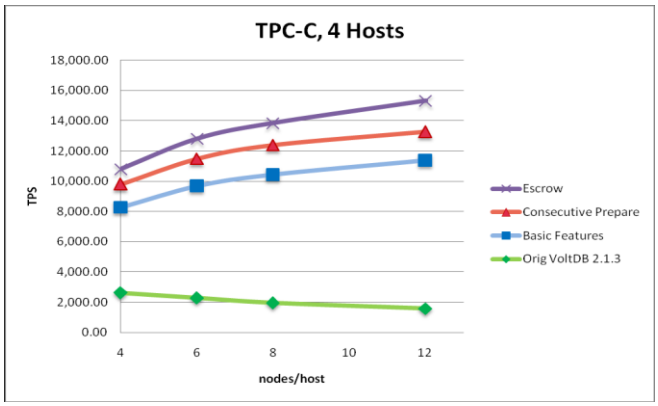

Figure 7.8 TPC-C performance on 4 hosts

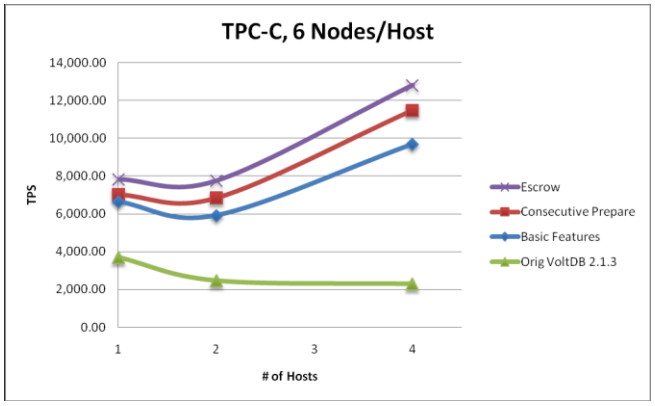

Figure 7.9 TPC-C performance by count of hosts

Figure 7.9 shows the scale-up of TPC-C on CVoltDB once multiple hosts are in use. It is not surprising that the change from one to two hosts degrades performance, because some of the distributed transactions are now communicating between hosts rather than just between nodes on the same host.

In summary, TPC-C is not as perfect a fit as TPC-B for CVoltDB, because the incremental manipulations of the customer balance are only part of the transactional work, and some of the distributed transactions are multi-shot. Still, we are seeing significant improvement in performance, in a fairly realistic setting.

8. FUTURE WORK

Several aspects of the CVoltDB system are incomplete. Schema changes and table loads may not be executed in a CVoltDB transaction, but instead must be done in a separate VoltDB transaction. Escrow columns depend on a naming convention for designation.

CVoltDB guarantees that transaction Prepares and Commits take place in TxID order. However Commit ordering is not required for serialization and replica consistency as long as Prepares are in TxID order, so some additional performance may be possible to allow out-of-order commits of transactions (but final command logging of transactions must occur in TxID order, so commit results must be queued on the coordinator node in this order).

Although CVoltDB has been able to utilize the wait time of the second phase of two-phase commit, the nodes still become idle between shots of a multi-shot transaction. To utilize this time, we would need to add Read locks and deadlock detection, as was done in [8], or wound-wait locks as in Google's Spanner [2] for deadlock avoidance. The latter option is particularly promising because it avoids possible distributed deadlocks at the cost of possible extra aborts, requiring retry. Note that these proposals still maintain single-threaded nodes to avoid latching.

The current version of VoltDB (starting from version 3) has a new transaction initiation protocol that reduces latency and may improve load-balancing behavior. However, it coordinates all the distributed transactions from a single node, creating a bottleneck if a significant fraction of transactions are distributed (common in our benchmarks but not in current VoltDB deployments). Possibly the new approach could be reworked to remove this bottleneck and still help with load balancing and latency.

9. CONCLUSIONS

In this paper we have introduced features into CVoltDB that provide much faster distributed transaction performance. We simplified the lightweight locking concurrency method of [7] by dropping Read locks, while keeping Write locks. Read locks are not needed since Prepares are executed one at a time on a node, and replica consistency follows from execution in TxID order. We also devised an Ordered Escrow method for OLTP MMDB systems that speeds up incremental operations while maintaining serializability and replica consistency. Escrow quantities can be read at the cost of synchronizing the stream of Escrow updates. The Escrow capability is provided to programmers with a simple and expressive SQL extension.

10. ACKNOWLEDGMENTS

Our thanks to Vertica for supporting Weiwei Gong.

11. REFERENCES

[1] Abadi, D., Hugg, J., Jones, E., Kallman, Kimura, R. H., Madden, S., Natkins, J., Pavlo, A., Rasin, A.M., Stonebraker, M., Zdonik, S., Zhang, Y. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB, 1,2* (2008) 1496–1499.

[2] Corbett, J, Dean, J, Epstein, M, Fikes, A, Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh,W., Kanthak,S., Kogan, E., Li, H., Lloyd, A., Melnik,S., Mwaura,D., Nagle,D., Quinlan, S., Rao, R Rolig, L., Saito,Y., Szymaniak,M., Taylor,C., Wang, R., Woodford, D. 2012 Spanner: Google's globally-distributed database. *Proceedings of the 10th USENIX conference on Operating SystemsDesign and Implementation*, OSDI'12, 251–264.

[3] DeWitt, D. J., Katz, R. H., Olken, F. , Shapiro, L. D., Stonebraker, M., Wood, D. A. 1984. Implementation Techniques for Main Memory Database Systems. *ACM SIGMOD 2014, 2* (June 1984) 1 – 8.

[4] Garcia-Molina, H., and Salem, K. 1992. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng. 4(6)(1992) 509-516.*

[5] Gawlick, D. and Kinkade, D. 1985.Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bulletin* 8,2 (June 1985) 3-10.

[6] Graefe, G., Zwilling, M. J., 2004. Transaction support for indexed views. *ACM SIGMOD 2004*: 323-334.

[7] Harizopoulos, S., Abadi, D. J., Madden, S. and Stonebraker, M. 2008. OLTP through the looking glass, and what we found there. In ACM *SIGMOD* (2008), 981–992.

[8] Jones, E. P. C., Abadi, D. J., and Madden, S. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. *ACM SIGMOD* 2010, 603-614.

[9] Kallman, R., H. Kimura, Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C. , Madden, S. , Stonebraker, M., Zhang, Y. , Hugg, J., Abadi, D.J. 2008. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB*, 1, 2 (2008) 1496-1499.

[10] Kraska, T., Hentschel, M., Alonso, G., Kossmann, D. 2009. Consistency Rationing in the Cloud: Pay only when it matters. *Proc.VLDB*, 2, 1 (August 2009) 253-264.

[11] Lomet, D., ed., 2013. *IEEE Data Engineering Bulletin 36, 2* (June 2013).

[12] Thomson, A, Abadi, D.2010: The Case for Determinism in Database Systems. *Proc.VLDB*, 3,1: 70-80

[13] Thomson, A, Abadi, D.2013 :Modularity and Scalability in Calvin。*IEEE Data Engineering Bulletin, 36, 2* (June 2013) 48-55

[14] O'Neil, P. E. 1986. The Escrow Transactional Method. *ACM Transactions on Database Systems* 11, 4, 406-430.

[15] O'Neil, P. E. 1991. Deadlock Prediction for Escrow Transactions. *Information Systems.* 16, 1 (1991), 13-20.

[16] Stonebraker, M., Madden, S., Abadi, D. J. , Harizopoulos, S., Hachem, N., Helland, P., 2007. The end of an Architectural Era: (It's Time for a Complete Rewrite). *Proc. VLDB*, 2007, 1150-1160.

[17] Stonebraker, M. and Weisberg, A. 2013. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin, 36, 2* (June 2013) 21-27.

[18] Transaction Processing Performance Council, www.tpc.org.

[19] Yalamanchi, A., Gawlick,D 2009. Compensation-aware data types in RDBMS. ACM SIGMOD 2009 931-938.