IMPROVING MAIN-MEMORY DATABASE (MMDB) DISTRIBUTED TRANSACTIONAL CONCURRENCY

A Dissertation Presented

by

WEIWEI GONG

Submitted to the Office of Graduate Studies,
University of Massachusetts Boston,
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2014

Computer Science Program
IMPROVING MAIN-MEMORY DATABASE (MMDB) DISTRIBUTED
TRANSACTIONAL CONCURRENCY

A Dissertation Presented

by

WEIWEI GONG

Approved as to style and content by:

_________________________________________
Patrick E. O’Neil, Professor Emeritus
Chairperson of Committee

_________________________________________
Elizabeth J. O’Neil, Professor
Member

_________________________________________
Dan A. Simovici, Professor
Member

_________________________________________
Tim Kraska, Assistant Professor
Brown University
Member

_________________________________________
Dan A. Simovici, Program Director
Computer Science Program

_________________________________________
Peter A. Fejer, Chairperson
Computer Science Department
ABSTRACT

IMPROVING MAIN-MEMORY DATABASE (MMDB) DISTRIBUTED TRANSACTIONAL CONCURRENCY

December 2014

Weiwei Gong, B.M., Beijing Information Technology Institute
M.S., Renmin University of China
M.S., University of Massachusetts Boston
Ph.D., University of Massachusetts Boston

Directed by Professor Emeritus Patrick E. O’Neil

Concurrency control provides multi-user access to a database system, while ensuring concurrent transactional correctness. Classical RDBMS products such as Oracle, DB2 and SQL Server have Disk Resident databases (DRDBs) with designs from the 1970s, whose data can only be read or updated after being brought into memory. Great reductions in memory price and ever-faster compute cycles have made DRDBs much less desirable for all but the largest databases. Study of Main Memory Database Systems (MMDBs) and their concurrency control algorithms is a current research effort. An advantage of MMDBs cited by early researchers was that execution of a transaction on a CPU in memory from start to finish would save all the work of disk I/O, locking, latching and coping with deadlocks. But until recently there was very little discussion of distributed MMDB transactions, needed when not all data is accessible at one computer.
This dissertation analyzes the performance bottlenecks of distributed transactions in an open-source multi-CPU shared-nothing MMDB VoltDB, which is based on an earlier academic MMDB H-Store. On each shared-nothing node, the transactions execute serially. The waits for the messages involved in distributed transactions seriously impact performance. We developed a low-overhead concurrency scheme that runs Prepares serially on each node so that new Prepares could run while old Prepares were waiting to commit. Possible conflicts are detected using Write locks but no Read locks. We also provide an Ordered Escrow Method, a variant of original Escrow that speeds up transactions doing incremental updates. We implemented these serializable concurrency schemes on VoltDB, and named this modified system CVoltDB. CVoltDB is the first DBMS providing an Escrow method to the transaction programmer. We performed measurements comparing VoltDB and CVoltDB performance showing significant performance improvement.
ACKNOWLEDGEMENTS

To Pat and Betty O’Neil for their help and guidance in conducting this research.
To Vertica Systems for supporting my graduate work.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS .................................................. vi

LIST OF TABLES ....................................................... ix

LIST OF FIGURES ...................................................... x

CHAPTER

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION ............................................. 1</td>
<td></td>
</tr>
<tr>
<td>CVoltDB ....................................................... 5</td>
<td></td>
</tr>
<tr>
<td>2. TRANSACTION EXECUTION ............................... 8</td>
<td></td>
</tr>
<tr>
<td>Single-Threaded Database Systems Transaction Execution . 8</td>
<td></td>
</tr>
<tr>
<td>Sharded Main Memory Database Architecture for OLTP . 12</td>
<td></td>
</tr>
<tr>
<td>Decreased Throughput For Distributed Transactions In A Sharded Main-Memory Database System for OLTP . 15</td>
<td></td>
</tr>
<tr>
<td>Summary ....................................................... 18</td>
<td></td>
</tr>
<tr>
<td>3. A CONCEPTUAL DESIGN FOR CVoltDB ........... 19</td>
<td></td>
</tr>
<tr>
<td>Concepts and Rules .......................................... 19</td>
<td></td>
</tr>
<tr>
<td>Data Structure: Queues ...................................... 21</td>
<td></td>
</tr>
<tr>
<td>Enabling Concurrent Execution ........................ 22</td>
<td></td>
</tr>
<tr>
<td>Concurrency Strategies in CVoltDB .................. 24</td>
<td></td>
</tr>
<tr>
<td>Summary ....................................................... 37</td>
<td></td>
</tr>
<tr>
<td>4. CVoltDB IMPLEMENTATION WITH VoltDB .......... 38</td>
<td></td>
</tr>
<tr>
<td>Enable Concurrent Execution .......................... 39</td>
<td></td>
</tr>
<tr>
<td>Uses Only Needed Partitions ....................... 41</td>
<td></td>
</tr>
<tr>
<td>Early Fragment Distribution ....................... 43</td>
<td></td>
</tr>
<tr>
<td>Consecutive Prepares Execution .................. 50</td>
<td></td>
</tr>
<tr>
<td>The Ordered Escrow Method ....................... 58</td>
<td></td>
</tr>
<tr>
<td>Summary ....................................................... 65</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. TPC-B Tables Description</td>
<td>68</td>
</tr>
<tr>
<td>2. TPC-B TPS Speed-up Table</td>
<td>70</td>
</tr>
<tr>
<td>3. TPC-C Tables Description</td>
<td>76</td>
</tr>
<tr>
<td>4. Transactions in TPC-C</td>
<td>77</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Two transactions under <em>Serial Execution</em></td>
<td>9</td>
</tr>
<tr>
<td>2.</td>
<td>Two transactions under <em>Consecutive Prepares Execution</em></td>
<td>12</td>
</tr>
<tr>
<td>3.</td>
<td>Two transactions under <em>Consecutive Prepares Execution (Commit within Prepare)</em></td>
<td>12</td>
</tr>
<tr>
<td>4.</td>
<td>Messages Passing for a Distributed Transaction</td>
<td>16</td>
</tr>
<tr>
<td>5.</td>
<td>TPC-B on VoltDB by Distributed%, Single Host</td>
<td>17</td>
</tr>
<tr>
<td>6.</td>
<td>TPC-B on VoltDB by Distributed%, 4 Host</td>
<td>18</td>
</tr>
<tr>
<td>7.</td>
<td>Transaction Transit among Queues in CVoltDB</td>
<td>23</td>
</tr>
<tr>
<td>8.</td>
<td>Fragments Relationship in CVoltDB</td>
<td>50</td>
</tr>
<tr>
<td>9.</td>
<td>Data Structure of Row Journal</td>
<td>52</td>
</tr>
<tr>
<td>10.</td>
<td>Updated Data Structure of Row Journal</td>
<td>58</td>
</tr>
<tr>
<td>11.</td>
<td>TPC-B TPS by Distributed%, Single Host</td>
<td>71</td>
</tr>
<tr>
<td>12.</td>
<td>TPC-B TPS by Distributed%, 2 Host</td>
<td>72</td>
</tr>
<tr>
<td>13.</td>
<td>TPC-B TPS by Distributed%, 4 Host</td>
<td>73</td>
</tr>
<tr>
<td>14.</td>
<td>TPC-B TPS by count of hosts, 15% distributed</td>
<td>74</td>
</tr>
<tr>
<td>15.</td>
<td>TPC-B TPS by count of hosts, 99% distributed</td>
<td>75</td>
</tr>
<tr>
<td>16.</td>
<td>Timeline for NewOrder on VoltDB</td>
<td>77</td>
</tr>
<tr>
<td>17.</td>
<td>Timeline for NewOrder on CVoltDB</td>
<td>78</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>18</td>
<td>TPC-C TPS by nodes/host, 4 hosts</td>
<td>81</td>
</tr>
<tr>
<td>19</td>
<td>TPC-C TPS by counts of hosts, 6 nodes/host</td>
<td>82</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The premier relational database systems for On-Line Transaction Processing (OLTP), such as DB2, Oracle, and Microsoft SQL Server, are all disk-based systems that trace their design back to System R, developed at IBM in the early 1970s. These systems were designed for hardware characteristics much different than those of today. Computer processors are thousands of times faster and thousands of times less expensive today; disks and memory are also much bigger and less expensive [SMA07]. A 1984 paper [DKO84] said that although memory then cost $1500 per megabyte (million bytes), prices would go down over time so as to support much faster main-memory databases (abbreviated MMDB), which the paper explored. Nowadays, we can buy memory for about $10 per GByte.

OLTP processors designed in the 1970s accessed data on disk that needed to be brought into memory to be read and updated. To speed this up there were disk-based indexes to the data (B-trees after 1971), and while the popular upper nodes of a B-tree might be memory-resident, it would usually require at least two disk accesses to access any single piece of data, such as the bank record of a customer who provided his account number. We note that while CPUs and memory have become much faster and cheaper, the speed of bringing data from disk into memory has lagged dramatically, a speed-up of about four.
This means we must always have many transactions running at once as process threads so that a fast CPU has different things to do as it waits for the terribly slow disk accesses. As an analogy, if a few CPU instructions are considered enough time to write a character of a letter to be mailed and this letter were sent off to disk to bring a disk page into memory, the wait for the return is what a five minute letter sent by Voltaire from France to Moscow in 1780 would need for a reply: about three weeks. This gives an idea of the speed of CPU compared to Disk access. Because many transaction threads must run at once to keep the CPU busy, we have to deal with conflicts between threads, as when two transactions try to access the same bank balance (read or write) and we need locks held on bank balance rows to avoid errors from such conflicts (writes conflict with reads and other writes). A lock is held until the transaction that holds the lock commits. It can often happen that a large fraction of popular rows are locked, making it likely that transactions attempting writes (or reads) are blocked on rows that have been locked by other transactions, until there are only a few threads that can make progress.

This is not a fanciful problem. If there are popular database rows, then the number of transactions running concurrently must be limited to avoid such lock-up and this can mean we use only a fraction of CPU speed to perform useful work. Even if the CPU is more fully occupied, there are a number of tasks that take up CPU time on disk-based databases that do not involve useful work. Reference [HAM08] illustrates this with the time taken in disk-based access to perform typical transactions in the TPC-C database benchmark. Figure 6 for the TPC-C Payment Transaction shows 10.1% of the time is spent reading in B-tree nodes from disk, 29.8% of the time reading row data into memory buffers, 25.2% performing row locking, 12.2% in latching, a form of fast locking to guarantee two transactions don’t modify a disk page at the same time, e.g. to insert new rows, and 11.9%
writing logs out to disk so that aborts can succeed and commits will be remembered if the system crashes. This leaves only 4.7% of the time to actually carry out the steps of various transactions.

Clearly disk access time is a key bottleneck in Online Transaction Processing (OLTP) databases. And since main memories have become sufficiently large, most OLTP databases can reside entirely in memory, saving common disk access time.

Concurrency control ensures the serializability of transactions while actually executing multiple transactions simultaneously in a shared database. However, several research papers suggest that for some specialized databases, concurrency control may not be necessary [SMA07, GLV84, GS92, WSA97], especially for main memory databases without user stalls, i.e. user interaction during a transaction. A main memory database can avoid overheads described above, and this could improve performance significantly [JAM10]. Instead of traditional concurrency control, transactions are simple executed serially. Unfortunately, due to network stalls prevalent in distributed transactions, pure serial execution of these transactions limits their performance. Thus two schemes, speculation and locking, are introduced in [JAM10] to allow other transactions to work during network stalls of distributed transactions. Measurements show the performance of these schemes degrades as conflict rate increases and at high conflict rate, those two schemes do not outperform pure serial execution. Thus there is still a need for a scheme that can improve performance under high as well as low conflict rates.

In OLTP databases an UPDATE bottleneck of transactions accessing high-concurrency aggregate data can be minimized by use of the Escrow Method [ON86] to permit greater concurrency. The Escrow Method guarantees new updates can be performed on Escrow columns after any subset of prior updates while maintaining serializability. However, this
theoretical capability has not been employed in any traditional database. We show main memory databases can take advantage of this methodology, as long as we can encapsulate transactions in stored procedures, without any user interaction.

The research for this dissertation is rooted in [JAM10] and [ON86]. We’ve modified the open-source MMDB VoltDB to support interleaved distributed transaction execution with Escrow operations. VoltDB is based on an academic prototype called H-Store [KKN08]. VoltDB and H-Store execute transactions in strict Transaction-ID order. We allow concurrency, but we still ensure that all transactions prepare and commit/abort in their transaction order. We allow multiple prepares to occur before the first transaction commits or aborts, and so provide concurrency among them. We will show that our methods can improve distributed main memory database performance significantly, even at high conflict rates, by avoiding update conflicts where possible via Escrow operations and allowing maximum concurrency, while still guaranteeing serializability.

Main memory database systems store data entirely in main physical memory and provide very high-speed access. Many current commercial transactional MMDBs, including TimesTen and Microsoft’s Heketon, are designed as centralized systems, which limit their scale-up capability as new processors are added to the system. With today’s trend to ever-bigger data, the scale-up potential of distributed systems is compelling.

Thus, distributed transactions play a very important role in large transactional MMDBs. The 2013 Bulletin on Data Engineering [SW13] had eight articles on current transactional MMDBs and only three articles described distributed database systems. Of the three only two are full-feature MMDBs, VoltDB (open-source) and SAP’s HANA (proprietary), and both use shared-nothing partitioning, i.e. sharding. VoltDB uses single-threaded nodes.
HANA uses multi-threaded nodes and distributed snapshot isolation, supporting long-lived read-only transactions on a snapshot derived from OLTP updates.

In MMDBs, the major bottleneck, disk I/O, is eliminated, so there are no waits for data retrieval from disk. Thus non-distributed transactions that need data only from one shard (i.e., one node) can be executed one after another from the beginning to the end without interruption. This serial execution provides very good transaction processing performance on perfectly partitioned workloads and was adopted by H-Store and VoltDB, but another bottleneck arises. In distributed MMDBs, even a few distributed transactions cause a performance problem because of the network delay.

The network delays for nodes connected by a high-speed local area network are much shorter than the disk delays they are replacing: 30μs vs. 3ms. Thus the amount of concurrency needed to fill in the idle time on nodes is smaller than in disk-based systems, and we do this with Consecutive Prepares as explained below in Section 2.1. A form of limited concurrency should be sufficient and should be chosen with an eye towards small overhead. We have studied one such form of limited concurrency. With concurrency in place, we can utilize the Escrow Method to further speed up access to aggregated values.

1.1 CVoltDB

This dissertation presents CVoltDB, a modification from VoltDB 2.1.3 to support concurrent distributed transactions and thus improve performance.

- CVoltDB provides a lightweight locking concurrency method, adding WRITE Locks but no READ Locks. This has the minimal overhead needed for concurrent execution.
• CVoltDB integrates the **Ordered Escrow Method**, a variant of the original Escrow Method [ON86], to handle high-frequency updates of aggregate data and provides this functionality to transaction programmers for the first time.

• CVoltDB ensures ACID and full serializability, i.e. the execution of a set of valid transactions is serializable on each node, and on the entire system.

• Throughput of CVoltDB, measured in transactions per second by benchmarks TPC-B and TPC-C, greatly exceeds that of VoltDB, while CVoltDB has very good scalability.

The remainder of this dissertation is organized as follows: Chapter 2 presents a detailed overview of transaction execution. We define concurrent execution in centralized and distributed environments and explore similarities and differences among different levels of concurrent executions. Chapter 2 also shows the performance bottleneck of distributed transactions on VoltDB and summarizes the architecture for CVoltDB.

Chapter 3 presents a conceptual system design for CVoltDB. This chapter contains the discussion of four strategies we applied to improve concurrency to VoltDB: *Uses Only Needed Partitions, Early Fragment Distribution, Consecutive Prepares*, and the *Ordered Escrow Method*.

Chapter 4 shows detailed CVoltDB implementation, including fundamental data structure and algorithms upon which CVoltDB is built. This chapter describes how we took the theoretical ideas in Chapter 3 to implement a CVoltDB prototype based on VoltDB 2.1.3.
Chapter 5 gives the experimental evaluation that compares the performance of original VoltDB 2.1.3 and the effects brought by four strategies. Chapter 6 describes areas for future work and concludes this dissertation.
Chapter 1 introduced main memory database systems and the current systems using a Serial Execution scheme. Chapter 2 explores these topics in depth. We will give some definitions used in later chapters, and argue that pure serial execution is not suitable for all OLTP workloads, and introduce the basic architecture of CVoltDB.

Multi-threaded databases were introduced to overcome disk I/O waits by executing another transaction while the first waited, but multiple threads also bring overhead since it is difficult to manage and maintain transactions to be both isolated and concurrent. In partitioned MMDBs, without disk waits, we need less concurrency, and can save on the overheads by adopting single-threaded execution on each node. Thus, we focus on single-threaded systems in this chapter and only bring multi-threaded systems into discussion when needed. More precisely, a single thread is used for transactional execution. Other threads may execute other work such as network communication.

2.1 Single-Threaded Database Systems Transaction Execution

A schedule is a sequence of actions or operations in order by time, performed by a set of transactions. In a distributed environment, a transaction consists of Prepare and Commit phases. We extend this specific case to a common, more general case, that a transaction
always contains two phases: *Prepare* and *Commit*. However, in a non-distributed transaction, the *Prepare* phase is usually immediately followed by the *Commit* phase.

**Definition 2.1 (Serial Execution):** In a *Serial Execution*, transactions are executed strictly serially. For transactions $T_i$ and $T_j$ where $i \neq j$,

$$\text{[start}(T_i), \text{commit}(T_i)] \cap [\text{start}(T_j), \text{commit}(T_j)] = \emptyset$$

This means that only one transaction is running in the system at a time, as shown in Figure 1.

*Serial Execution* of one transaction at a time gives several benefits, for example, the execution is always isolated and for each operation execution is fast with no blocking caused by data conflicts. However, the disadvantage of *Serial Execution* is that it represents inefficient processing especially while waiting for disk. This leads to low CPU utilization, while a transaction waits for disk I/O, or for another transaction to terminate.

The aim of database system is to utilize resources efficiently, limit waiting time and preserve database consistency. The only way is to execute transactions concurrently, unless the disk bottleneck and communication wait is eliminated such as in a centralized main-memory database system.
**Definition 2.2** (*Concurrent Transactions*): Two transactions are concurrent *iff* their lifetime overlap. For example, for transactions $T_i$ and $T_j$ where $i \neq j$,

$$[\text{start}(T_i), \text{commit}(T_i)] \cap [\text{start}(T_j), \text{commit}(T_j)] \neq \emptyset$$

The concept of overlapped lifetime can happen in more than one way, so we further define the *Concurrency Levels* for concurrent transactions. There are three levels:

**CL1. Serial Execution**

**CL2. Prepare-interleaved Execution (Consecutive Prepares);** at this level, *Prepares* are interleaved, not overlapped, but a *Commit* can occur within the *Prepare* of a later transaction, defined precisely below

**CL3. Operation-interleaved Execution**

An example scheduling of interleaved operations can be:

$$H_{2,1} : s_1, r_1(A), w_1(A), s_2, r_2(B), c_2, c_1$$

For *Operation-interleaved Execution*, the thread completes some but not all operations of one transaction, and works on some operations of another transaction, and then may continue to execute operations of the previous transaction.

The scheme of *Consecutive Prepares Execution* satisfies **CL2** above:

- Operations of different transactions cannot be interleaved;

- Transactions are concurrent, since *Prepare* and *Commit* phases of transactions can be interleaved.

An example scheduling can be:

$$H_{2,2} : s_1, r_1(A), w_1(A), \quad \# \text{prepare of } T_1$$
\begin{align*}
s_2, r_2(B), w_2(B), & \quad \# \text{prepare of } T_2 \\
c_1, & \quad \# \text{commit of } T_1 \\
s_3, r_3(B), & \quad \# \text{prepare of } T_3 \\
a_2, c_3 & \quad \# \text{abort of } T_2 \text{ and commit of } T_3
\end{align*}

A more formal definition is as following:

**Definition 2.3 (Consecutive Prepares Execution):** Concurrent transactions under this execution schema satisfy:

\[
[\text{start}_\text{prepare}(T_i), \text{end}_\text{prepare}(T_i)] \cap [\text{start}_\text{prepare}(T_j), \text{end}_\text{prepare}(T_j)] = \emptyset
\]

and

\[
[\text{start}_\text{commit}(T_i), \text{start}_\text{commit}(T_i)] \cap [\text{start}_\text{commit}(T_j), \text{start}_\text{commit}(T_j)] = \emptyset
\]

and

if \( \text{start}(T_i) < \text{start}(T_j) \), then

\[
[\text{start}_\text{commit}(T_i), \text{start}_\text{commit}(T_i)] \cap [\text{start}_\text{prepare}(T_j), \text{end}_\text{prepare}(T_j)] = \emptyset
\]

or

\[
[\text{start}_\text{commit}(T_i), \text{start}_\text{commit}(T_i)] \subseteq [\text{start}_\text{commit}(T_j), \text{start}_\text{commit}(T_j)]
\]

Figure 2 shows two transactions running under *Consecutive Prepares Execution*, the *Prepare* of the second transaction occurs between the *Prepare* and *Commit* of the first transaction; and Figure 3 shows the *Commit* of the first transaction happens within the second transaction’s *Prepare*.

Note that to avoid anomalies and achieve an equivalent serial scheduling, we must apply technologies to ensure the correctness, such as locking, or multi-version concurrency...
control, which can utilize the disk I/O (or network) wait time, but also bring new overheads to the database system.

2.2 Sharded Main Memory Database Architecture for OLTP

Main Memory Database Systems store data in main memory and provide very high-speed access. Since the access is so fast, we can expect transactions to complete more quickly in a main memory database system, especially in OLTP applications with their short transactions with little table-scanning or other heavy operations.

There are two approaches to store the data for main memory database systems. Centralized MMDBs put all data in one single big machine; in order to handle complex applications, it has to deploy expensive CPUs, lots of memory. Further, the maximum size
of memory on one single machine is limited, so it is possible that eventually a centralized MMDB cannot fit all its data. In contrast, sharded MMDBs partition data on many cheap machines. Those machines, each with CPU and memory, act as nodes of a big supercomputer, and it is possible to adjust to applications’ needs by adding or removing nodes. Thus, such PC clusters are becoming a more and more popular solution for large-scale computing. So we concentrate on a sharded MMDBs, i.e. MMDBs with large tables horizontally partitioned across the nodes. Some small tables can be replicated, that is, maintained with separate copies on all nodes.

Traditional disk-based database systems frequently use locking to achieve serializability among concurrent transactions. However, if we replace disk with main memory, locks will not be held as long, and the lock contention is not as important as it is when the data is disk resident. So we can chose to lock on the entire database in the extreme case, which is equivalent to Serial Execution and entirely eliminates concurrency control. There are some prototypes and commercial products that abandon well developed but complicated concurrency control schemas and obtain decent performance for some specific workloads just using simple Serial Execution on each node, and we call this Local Serial Execution.

**Definition 2.4 (Local Serial Execution):** In a distributed database system, at each node, Local Serial Execution means that incoming transactions are executed sequentially, i.e. transactions are executed serially on each node.

From the view of one node, all incoming transactions are executed sequentially; but from the system view, if no distributed transaction exists in the system at this time, many local transactions may be running on the various nodes. Moreover, when symmetric, i.e., not
master-slave replication is in use, a certain transaction order is needed to ensure that transactions are executed in the same order on all replicas.

While using *Local Serial Execution*, a sharded main memory database can generate a global deterministic execution order for all transactions to follow. For example, with some ordering protocol, they could assign a unique identifier to each transaction and then order them based on those numeric identifiers. This approach is theoretically simple but requires a non-trivial network protocol in a distributed environment.

For a distributed transaction, one node is chosen to be the coordinator of the transaction, and it handles the request and coordinates with other (participant) nodes via network round-trips, sending out SQL statements to do and receiving back results.

Finally, consider the representation of transactions. A transaction can be represented as a manual input via users, or connection via ODBC or JDBC, or a stored procedure, and etc. In OLTP systems, transactions are usually very lightweight, so usually no human interruption or communication with external servers is allowed in order to eliminate unnecessary delays, and this type of transactions are called deterministic. A stored procedure provides necessary business logic and sophistication, and it has no compilation overhead at execution time since it is already stored in the database dictionary directly. So it is chosen as a good vehicle for OLTP transactions in MMDB.

To sum it up, the properties of the OLTP-targeted architecture described as above are:

1. A sharded MMDB, i.e. distributed and horizontally partitioned.

2. A transaction provides some logic, such as a stored procedure.

3. All transactions are deterministic.
4. Transaction execution is single threaded on each node.

5. Each node can act as a coordinator or as a participant, or both, for an individual transaction.

6. Each participant handles RPC-like calls from the coordinator to do its part of the transaction work, and maintains undo logs for possible abort handling.

7. All transactions are made to execute in Transaction-ID order.

We call this architecture *Sharded Local Serial Execution MMDB Architecture*, H-Store and its commercial product VoltDB satisfy these conditions, and additionally provide durability via replication.

Though *Local Serial Execution* could lead to great performance when the data and the workload are partitioned exactly the same way, distributed transactions lead to a sharp decrease in throughput, even on a single machine without additional network delays.

### 2.3 Decreased Throughput For Distributed Transactions In A Sharded Main-Memory Database System for OLTP

Consider the *Local Serial Execution* of a simple distributed transaction with no replication and simplified transaction initialization in a sharded MMDB.

The client initiates the transaction by sending the request to the system. Suppose the transaction needs data from more than one node. The system sends the request to the coordinator node, which sends needed transactional work to all other nodes in the system. Then each node executes its part of the transaction and sends back intermediate results to
Figure 4: Messages Passing for a Distributed Transaction

1: client sends user request to server, which sends it to the coordinator node
2: the coordinator forwards the transaction’s work to all other participants
3: participants return intermediate results to the coordinator after Prepare
4: the coordinator sends the commit/abort decision to participants again
5: participants send acknowledges back to the coordinator after executing the decision
6: the coordinator finally sends back result to the client

the coordinator. The coordinator assembles all results and follows the Two-Phase Commit protocol to commit the transaction, and finally returns the result to the client.

In this execution, there are two message round trips between coordinator and all participants as described in Figure 4. In more complicated cases, there may be additional round trips to do additional transactional work before the final round trip that finalizes the commit / abort (phase 2 of Two-Phase Commit).

The experiments in [HAM08] show these network stalls are at least the same length as the whole Prepare time of the transaction, which means the network stall takes up more than half of the execution time for the distributed transactions.
To evaluate the impact of distributed transactions to performance, we tested the performance of modified version of VoltDB that satisfies all architectural requirements in section 2.2, using TPC-B benchmark. TPC-B measures throughput in terms of how many transactions per second a system can perform. With 1% distributed transaction, shown in Figure 5, and the throughput decreases by half in single host case, and drops to 20% in multi-host case, shown in Figure 6.

It is clear that simple Local Serial Execution is unacceptable even when distributed transactions occur occasionally, i.e. when the data and workload are not perfectly partitioned.
2.4 Summary

This chapter defined different transaction execution schemes for centralized database systems and distributed systems, then introduced the *Sharded Local Serial Execution MMDB Architecture* aiming for high performance, and points out the performance problem of distributed transactions using this architecture.

Chapter 3 will present a conceptual design for CVoltDB, which adds concurrency to the architecture by implementing *Consecutive Prepares Execution* and a modified *Escrow Method* [ON86] to obtain high throughput for both local transactions and distributed transactions. Chapter 3 covers concepts without focusing on detailed implementation. Chapter 4 will present implementation of this design based on VoltDB, a sharded MMDB for OLTP with *Local Serial Execution*. 
CHAPTER 3

A CONCEPTUAL DESIGN FOR CVOLTDB

This chapter describes a conceptual design for CVoltDB and necessary architecture changes. We will focus on the high-level features, and leave the low-level details for the implementation discussion in next chapter. This chapter explains four strategies to enable transactions' concurrent execution. Concurrent execution is not new. Most modern database systems execute transactions concurrently in order to allow the CPU do useful work while accessing disks, so that they can achieve good throughput performance. However, as we described in the last chapter, H-Store and VoltDB give up this complex concurrency control and simply execute transactions serially on each node without all the overhead of concurrency control. Local Transactions can run concurrently on sites for different partitions.

3.1 Concepts and Rules

Before the discussion of CVoltDB, let’s introduce concepts used through this chapter first.

- **Execution Mode**: two modes in CVoltDB for execution on a certain node. Serial Execution Mode is defined as executing transactions serially, i.e. applying Local Serial Execution, the same as VoltDB does; Concurrent Execution Mode is
introduced and only available in CVoltDB, it is the opposite mode in which multiple transactions can run concurrently with overlapping lifetime, i.e. Consecutive Prepares Execution as defined in Definition 2.3.

- **Transaction States**: the life cycle of a transaction has five states; Invisible, Ready to Start, Started, Preparing, and Prepared. These states apply to the transaction’s execution at each node. The full transaction can be said to be prepared when it becomes prepared at its coordinator site, and committed or aborted when this happens at its coordinator site.

  - **Invisible**: the transaction is invisible until it can be guaranteed to be the next transaction in Txn ID order
  
  - **Ready to Start**: visible but not started
  
  - **Started**: the transaction is chosen to start. Its executable tasks (fragments) have been sent to nodes it needs to access if it is distributed. This is a state only defined for CVoltDB, needed to support Early Fragment Distribution.
  
  - **Preparing**: the transaction is running its transactional work, also known as its Prepare phase.
  
  - **Prepared**: all transactional work has completed and, on the coordinator, the stored procedure has made its commit / abort decision. For a distributed transaction’s coordinator, the transaction waits for the transaction-completion response message from participants; for a distributed transaction’s non-coordinator, it waits for the transaction-completion message from the coordinator. After any message wait, the prepared and now-decided
transaction waits for its turn to commit or abort. This state is also known as the
Commit phase.

Following all the rules introduced in section 2.2, CVoltDB has several new rules:

1. A node is either in *Serial Execution Mode* or *Concurrent Execution Mode*.

2. Only one transaction is executing in its Prepare phase at a time at each node, but
   several transactions may be in their Commit phases

3. No table definition changes and no changes to unique columns are allowed while
   the node is in *Concurrent Execution Mode*.

These rules are important and essential to ensure the correctness of concurrent execution
of CVoltDB.

### 3.2 Data Structure: Queues

There are three transaction queues and one transaction variable named
CurrentPreparingTransactionState that can be thought of as a single-element queue. A
transaction is on only one of these four queues at each point in time, at each node.

- **TransactionQueue**: priority queue with all transactions not yet started. Each
  becomes visible in this queue when it can be guaranteed to be the next transaction in
  Txn ID order. Transactions sitting on this queue can be in *Invisible* state or *Ready to
  Start state*, but when a transaction is dequeued, it must be in *Ready to Start* state.

- **StartedTxnQueue**: contains all started but not yet preparing transactions. Starting
  work involves distributing work requests for a distributed transaction (*Early
Fragment Distribution done by coordinator, or nothing at all. Transactions on this queue are all in Started state.

- CurrentPreparingTransactionState: the one transaction doing its Prepare phase (if any), the transaction is in Preparing state.

- FinishingTxnQueue: on-coordinator transactions waiting for some response messages in order to finish, or waiting for earlier transactions to finish, and for off-coordinator transactions, waiting for the message from coordinator in order to finish, or also has earlier transactions to finish. Transactions on this queue are in Prepared state.

In the life cycle of one transaction, shown in Figure 7, it first enters the TransactionQueue, after it is Ready to Start it is moved to the StartedTxnQueue, and then waits its turn to become CurrentPreparingTransactionState to execute all its transactional work, and when it finishes its Prepare phase, it will stay in the FinishingTxnQueue until Commit or Abort and return of results to the client. We will describe state transitions of concurrent execution later this chapter.

In VoltDB and CVoltDB, no matter under which execution mode, the last-sent results from participants contain their votes for Two-Phase Commit. Those votes are naturally made based on the success / failure of the returned results.

### 3.3 Enabling Concurrent Execution

We add an annotation in each stored procedure to enable CVoltDB execution, which requests Concurrent Execution in the server. This annotation is either ON or OFF for any
user-defined transaction, and the default value is OFF. And all system procedures have this annotation disabled.

The annotation looks like following, TRUE in isCVoltDBExecution turns on the annotation, otherwise, the annotation is OFF.

\[
isCVoltDBExecution = \text{TRUE} / \text{FALSE}
\]
The programmer is responsible for setting this annotation correctly when writing an application. To gain higher throughput, when there are no calls to system procedures to change schema and no changes to unique columns, this annotation should be turned ON. Note that the server is smart enough to switch between two execution modes on each node to always ensure serializable results.

A node is either in **Serial Execution Mode** or **Concurrent Execution Mode**. With preparing a transaction with this annotation ON, the node must be in **Concurrent Execution Mode**; otherwise, it’s in **Serial Execution Mode**. When the node is in one mode, only transactions marked as that mode can run (i.e., be in state **Preparing** or **Prepared**). Transactions of the other mode can be in state **Started**, since only non-transactional work is involved there. And when a transaction with different mode comes in, it will eventually change the execution mode of the site when it gets to its turn to prepare in transaction order.

Each node is independent and self-managed, thus they may be in different execution modes, and so the system does not require all nodes to be in the same mode at the same time. However, a distributed transaction involving all sites can synchronize the mode of all nodes.

In following sections, we will introduce four concurrency strategies one by one, and assume each strategy contains changes from previous ones.

### 3.4 Concurrency Strategies in CVoltDB

#### 3.4.1 Uses Only Needed Partitions

First strategy to improve concurrency in CVoltDB is to use only needed partitions. This was provided in VoltDB for local transactions by directing the user request to the one
partition it needs. For distributed transactions, as described in Chapter 2, transaction code executes on all nodes in VoltDB. This is unnecessary, since some nodes have no relevant data in many cases.

Therefore, instead of transmitting the request to all nodes, we will find out which partitions are needed for the transaction when possible, and only send the requests to them, leaving the other nodes to do other useful work.

When isCVoltDBExecution is turned on, it will automatically apply *Use Only Needed Partitions* when there is a list of partition parameters are provided. Here is the definition of a new annotation for the programmer to turn on this option and extra usage information:

\[
\text{isCVoltDBExecution} = \text{TRUE} / \text{FALSE}
\]
\[
\text{partitionInfo} = \text{PARTITION\_TABLE:PARTITION\_COL}
\]
\[
\text{partitionParams} = \text{INTEGER\_ARRAY}
\]

*Use Only Needed Partitions* is folded in the isCVoltDBExecution; following two parameters are required when this switch is ON. partitionInfo provides the name of the partitioned table and its partition key to look up in the system dictionary for the transaction. partitionParams is the key information to locate the partitions by the values of the input parameters.

From Rule 2 and 3 in section 2.2, represented by a stored procedure, the transaction is deterministic: before execution, we know all SQL statements contained in the transaction, pattern of arguments for each SQL statement. However, we don’t know the values of the parameters at compile time, thus, partitionParams does not contain the real values of the parameters, but the index values of parameters for the stored procedure’s argument list.
Knowing the indexes, it’s easy to find out the values at run time when the transaction is initiated. When executing a transaction, the same work request generated from a batch of SQL statements will be sent out to all participating sites at the very beginning of the transaction execution and possibly another batch will be sent later, etc. The current CVoltDB system doesn’t calculate needed partitions for each SQL statement. Instead it works from the information in the annotations and the parameter values to determine the set of partitions for each transaction execution. The possibility of composing such annotations requires sufficiently well-behaved transaction programs such as those found in the TPC-B and TPC-C benchmarks.

The programmer should be aware if each SQL statement in the transaction has the partitioning key as its argument, then the Use Only Needed Partitions should be turned ON to use the key value to determine which partitions it belongs to.

However, sometimes the value of the partitioning key is dynamically calculated at runtime. For example, the value is a returned result of previous SQL statements. It’s too difficult to predict the needed partitions for this case, and we will simply send the request to all nodes instead. At this time, the programmer should know that the transaction can’t take advantage of Uses Only Needed Partitions and should not add the annotation in the transaction.

By applying Uses Only Needed Partition, the transaction execution is guided by the programmer-provided annotation information along with the stored procedure arguments. From one transaction’s perspective, nodes are divided into two sets: involved nodes, and non-involved nodes. And concurrency exists between these two sets, meaning another transaction could run at the same time on those disjoint nodes. However, with just this
feature, at one node at any time, there is no overlapping of life cycles of any two transactions.

3.4.2 Early Fragment Distribution

The second strategy is called Early Fragment Distribution. The purpose of it is to fill the idle time of a node with non-transactional work to avoid unnecessary wait at transaction start. The life cycles of transactions are somewhat overlapped, but it must still obey Rule 2 in section 3.1, that the overlapped life cycle cannot include overlapped Prepares.

Fragment distribution is not transactional work, because it only involves sending the work request for future execution (actual transactional work) to all participant nodes. Thus, this work can be overlapped with another ongoing Prepare.

Early Fragment Distribution is performed when the node is idle between the executions of two transactions or in the gap of the execution of one transaction, e.g., when the preparing transaction is waiting for message communication.

This work is nonexistent for a local transaction, since it only involves one partition and the work set must already be on the partition. For a distributed transaction, if it’s in the state of Ready to Start, the coordinator will send the work requests (fragments) to all participants of the transaction. All the non-coordinator participants receive the same work requests via the network. If the coordinator is also a participant, it also receives the same work request (not using the network), except for some message modifications involving optimized reads of replicated tables.
Communication networks play a vital role in a distributed system. By sending available messages ahead, no node would be in an unnecessary idle status waiting for incoming work requests. Thus, network delays are decreased and the performance is improved.

3.4.3 Consecutive Prepares Execution and Blocking

To further overlap the lifetimes of transactions, Prepare phases and Commit phases of transactions can be interleaved and furthermore a Commit phase can occur within another Prepare phase as long as no Prepares overlap. However, transaction Prepare phases and Commit phases must still be executed in their transaction identity order. A Commit phase (itself very lightweight) can execute during a later transaction’s Prepare phase, as well as between Prepare phases.

Example 3.1 (Interleaved Transaction Execution in CVoltDB): $P_i$ stands for the Prepare phase for Transaction $T_i$, and $C_i / A_i$ stands for the Commit / Abort phase for $T_i$. The following execution sequence is allowed:

$P_1 \quad P_2 \quad C_1 \quad P_3 \quad A_2 \quad C_3$

Note that the lifetimes of $T_1$ and $T_2$ overlap, and so do those of $T_2$ and $T_3$.

This Consecutive Prepares Execution constitutes a form of concurrency and brings data conflicts into consideration.

Two operations conflict if they operate on the same data and one of them changes the data. Since data are horizontally partitioned across sites, a data conflict only involves one node, and we need to block the preparing transaction when it happens.
**Example 3.2 (Two Interleaved Transactions causing blocking in CVoltDB):** With the same notation from Example 3.1, when \( T_2 \) is trying to read the content written by \( T_1 \), we have the following sequence:

\[
P_1 \quad P_2 \quad (\text{blocked}) \quad \cdots \quad C_1 \quad P_2 \quad C_2
\]

In the above example, the *Prepare* phase of \( T_2 \) is split into two parts, both denoted \( P_2 \), and the *Commit* phase of \( T_1 \) happens in between to unblock \( T_2 \). We allow previously prepared transactions to commit / abort during the *Prepare* phase of a later transaction. However, no other transaction may start preparing until the blocked transaction unblocks, because of the rule that only one transaction at a time can prepare at a node.

The mechanism to implement *Consecutive Prepares Execution* and provide serializability is locking. Our schema is a lightweight write locking. It only tracks essential data dependencies:

- **Write-Write Data Dependency:** lock on written data item, second *Write* needs to wait first *Write* commits / aborts
- **Write-Read Data Dependency:** lock on written data item, because we don’t allow dirty read (we support full serializability)
- **Read-Write Data Dependency:** not tracked, a *Read* followed by a *Write* in a different transaction does not need a read lock if transactions are prepared in Txn ID order because the serialization order is the same as the operations order. The *Read* will only return the committed value due to previous transactions, and (like read-locked data) this value cannot be changed by other transactions during this transaction’s *Prepare*. 
When any Write-Write or Write-Read data dependency is detected, a noticeable data conflict is raised and latter transaction with larger Txn ID is blocked and waits until the prior transaction commits or aborts.

Consecutive Prepares Execution could greatly increase the transaction concurrency when multiple transactions need to access common partitions but without overlapping data set. In OLTP systems, typically each transaction access only a few rows, and multiple transactions are rarely waiting for the same data item, except for some special hot spot data items. The following section describes an approach to handle a class of hot-spot data conflicts.

3.4.4 The Ordered Escrow Method

**Definition 3.3 (Escrow Column):** A column can be designated as an Escrow Column if its data type is numeric and only undergoes incremental changes by transactions running in Concurrent Execution Mode. It works like a normal table column for transactions running in Serial Execution Mode. No index is allowed on the Escrow column.

**Definition 3.4 (Escrow Update Operation):** update of an existent column value of an Escrow Column by an increment, possibly negative. A single row may have multiple Escrow Updates.

**Definition 3.5 (Escrow Request):** An Escrow Request is a request for an Escrow Update Operation on an Escrow Column value involving an incremental change, possibly negative. The Escrow request can succeed or fail (we say it is granted or not granted). A transaction can make multiple Escrow Requests. An Escrow Request targets a single row.
CVoltDB can perform increment-decrement updates on a numeric column that mitigates conflicts between transaction prepares, using special Escrow Columns [ON86]. Escrow columns are designated in the schema by a column naming convention. Update requests to increment (by positive or negative amounts) can be done without locking up the actual value until commit time. Instead, an Escrow Journal is created in memory that allows the eventual increment to be there at commit time if this transaction commits. The Escrow Update succeeds only if the increment keeps the column value within permitted bounds (e.g., ensures there is sufficient money in an account to permit a given withdrawal). The exact rule for success of an Escrow Request is given below in Rule 3.6.

**Rule 3.6: Escrow Update Out-of-Bound Rule**

a). 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 \(INC > 0\).

b). If \([INF+INC, SUP+INC]\) is completely outside of \([MIN, MAX]\), the request fails and the requesting transaction Aborts.

c). Otherwise, there are points of \([INF+INC, SUP+INC]\) both inside and outside of \([MIN, MAX]\). The requesting transaction blocks until all previous transactions Commit or Abort. At this point, the current \([INF+INC, SUP+INC]\) is a 0-length interval, inside or outside \([MIN, MAX]\). The request is granted in case a) and aborts in case b).

The value of Escrow is that UPDATE requests can be made by the Prepare phases of multiple transactions on an Escrow Column prior to Commit by the first of these transactions. The failure of an Escrow Request is quite rare, usually caused by a low
balance in an account or an item stocked for sale. Of course there are non-Escrow columns as well, normal single valued data such as text, where a prepared but uncommitted transactional update on the column will block reads or updates of that row by a later transaction prepare.

Consider the Escrow Column layout below. If the column represents money in a bank account, the largest value allowed (MAX) might be $250,000 (for more, one needs a special account) and the smallest value (MIN) might be $10 (minimum value to maintain the account). We explain below how MIN and MAX limit possible Escrow Updates. VAL is the current committed value of the column. The layout for a logical Escrow Column in a table has three numeric column quantities in the so-called "column". Any numeric column type can be used, but integers are most common (representing pennies in bank balances).

**Escrow Column Layout (3 actual database columns)**

\[
\text{MIN} = 10, \text{MAX} = 250000, \text{VAL} = 1000
\]

As increment and decrement update operations occur, a chain-linked list of Escrow Journals for each operation is created, with the chain-list header containing two new values, INF and SUP, which are quantities of the same type as VAL. After multiple operations, INF is the smallest VAL possible if all grants with positive increments abort and all grants with negative increments commit, while SUP is the largest value if all negative increments abort and positive increments commit. When Escrow Requests would bring the INF value below MIN or SUP value above MAX (even allowing for possible aborts by transactions with increments having a different sign), the operation fails and causes an abort of the transaction. If the success of an Escrow Update is in doubt (requiring aborts of transactions with an opposite sign) then the preparing transaction
blocks the node from further prepares, but already prepared transactions will continue to commit or abort and eventually resolve whether any blocked prepares can complete.

As each new Escrow Request is applied to the Escrow Column with Column Index ColIdx, a new Escrow Journal is placed in memory with the following layout.

**Escrow Journal Layout**

<table>
<thead>
<tr>
<th>ColIdx</th>
<th>TxnID = txn-id</th>
<th>INC = delta</th>
</tr>
</thead>
</table>

Hashing on the row pointer allow us to access these Escrow Journals. A sequence of updates to Escrow Column ColIdx is given below (in a simplified form (with no chain-links shown)).

Start

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

**After Escrow Request 1 for T₁**

| ColIdx = 1 |
| MIN = 10, MAX = 100000 |
| INF = 1000, VAL = 950, SUP = 1000 |

| ColIdx = 1 |
| TxnID = T₁ |
| INC = -50 |

**After Escrow Request 2 for T₂**
Note that if both transactions in the above example had aborted, everything would return to its original state: INF = 1000, VAL = 1000, SUP = 1000. Note too that the two Escrow Requests shown above could commit or abort in any order, a feature in some systems, but we need to commit in successive Txn-ID order for CVoltDB replication to work properly.
Increment/Decrement operations are so common in business applications that in 1976 IBM put out a product called IMS Fast Path [GK85] with data in memory for maximum speed of changes that couldn’t wait for regular database handling.

**Reads of Escrow Columns.** We permit Reads of an Escrow column value (VAL) by blocking until existing requests have all committed or aborted. The transaction in *Prepare* phase that performs the Read will of course be blocked until that time. We don’t normally need to read an *Escrow Column* with a large number of updates by outstanding transactions, so our wait is not usually very long. The Read of *Escrow Column* is not permitted in [ON86], but the rule that prepares and commits in Txn-ID order makes this possible.

Here we give the syntax of *Escrow UPDATE* for use in stored procedure code.

**Definition 3.7 (Escrow Update):** An UPDATE statement using following syntax

\[
\text{UPDATE tab SET ecol += / -= expr [WHERE \cdots]}
\]

where ecol must be an *Escrow Column*.

Thus, the Non-Escrow UPDATE is the regular UPDATE not using above syntax. Note that if the expression expr or the WHERE clause contains ecol, the execution will involve an *Escrow Read*, causing a wait for previous *Escrow Updates* to commit/abort. For fast *Escrow Updates*, it is important to avoid using ecol in expr or the WHERE clause.

### 3.4.5 Serializability Proof

1) **Local Serializability.** On each node, transactions are prepared in the global Txn-ID order. In CVoltDB, though we allow the interleaves of *Prepare* and *Commit* (*Commit* can be even within *Prepare*), but only one *Prepare* at a time on a node. Thus, on each node,
transactions are naturally serializable because all possible data dependencies in the Precedence graph are from earlier transaction to later transaction, and the serializable schedule is always the schedule in Txn-ID order.

2) **Global Serializability.** From 1), we know that the sub-Precedence graphs on each node do not have a cycle, then we need prove that the global Precedence graph cannot contain any cycle as well. Since all edges in all sub-Precedent graphs are from transaction with lower Txn-ID to transaction with higher Txn-ID. It will not change in the global Precedent graph, thus, no cycle can exist.

3) **Taking replication into consideration.** On each replica, in order to ensure consistency, the system always uses the same schedule, though time of operations can be different. Following the same reasoning in 2), transactions without Escrow operations are always serializable. 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 of a certain *Escrow Request*) on different replicas. 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. By Rule 3.6, we see that both replicas have the same outcome of the *Escrow Request*.

### 3.5 Summary

This chapter presented CVoltDB’s conceptual design and fundamentals. This chapter began with the definitions of concepts and terms used in VoltDB/CVoltDB, including two execution modes, different transaction states, data structures presenting the state of the transaction. Then we described the design of four features in CVoltDB, in actual implementation order, each feature is based on the assumption of previous ones.

*Uses Only Needed Partitions* frees up unrelated partitions in the system to handle other transactions if any while the current one is only executing on needed partitions.

*Early Fragment Distribution* takes advantage of the idle time on each node to execute future transactions’ non-transactional work, distributing the work requests from the coordinator to the participants. Thus the participants are less likely to need to wait for the work requests to arrive.

In order to support concurrent transaction execution, *Consecutive Prepares Execution* re-introduces the locking mechanism but with minimal overheads; and the *Ordered Escrow Method* is used to handle data conflicts on aggregate data items and speeds up the processing of OLTP transactions.

Chapter 4 will describe how these designs were implemented on VoltDB, a real working and well-known database system. And we believe any MMDB with *Local Serial Execution* can benefit from the four features we described in this chapter.
CHAPTER 4

CVOLTDB IMPLEMENTATION WITH VOLTDB

This chapter describes our implementation of CVoltDB derived from VoltDB 2.1.3. VoltDB is a good starting point for this project, given that it is open sourced and has high performance. Starting from standard VoltDB 2.1.3, our prototype work required two phases of development:

1. Adding support for concurrent execution, which involves implementing the first three strategies presented in Chapter 3.

2. Implementing the Ordered Escrow Method, a variant of original Escrow method, described as the forth strategy also in Chapter 3.

In VoltDB and CVoltDB, the code running on a node is layered with the upper layer, the Execution Site, written in Java, being responsible for all inter-node communication and interaction with the stored procedure, and the lower layer, the data engine, written in C++, doing the transactional work on the node’s data residing in memory. The Execution Site holds the transaction objects in the various queues discussed in Chapter 3. The data engine executes work fragments passed to it from the Execution Site and tracks changes to rows by the transactions, allowing data conflict detection.
4.1 Enable Concurrent Execution

VoltDB only supports serial execution of transactions on each site, and we still want to keep this execution mode for all system operations, including updating catalog, retrieving system information, initiating snapshotting, and etc. Thus, our first task was to teach VoltDB to distinguish two execution modes at each site, by adding extra information both in transaction and for the site.

The changes are straightforward. In the transaction, i.e. a stored procedure, an annotation is added, the syntax becomes:

```java
@ProcInfo {
    Boolean concurrentExecutionMode
        // is the procedure meant to run under concurrent execution mode?
    Boolean singlePartition
        // is the procedure meant for a single partition?
    String partitionInfo
        // information needed to direct this procedure call to proper
        // partition(s) if singlePartition is turned on
}
```

As mentioned in section 3.4.1, the information from the annotations is processed at transaction initiation time. Transaction initiation is carried out for the system by one thread on each host, the initiator, which accepts client connections on a certain port, interprets the stored procedure id and arguments, and looks up the procedure’s `procInfo`. The code for this thread is part of the Java layer, but separate from the code for the
Execution Site. In the *InitiateTaskMessage* sent from the initiator to all participating nodes at the very beginning of the transaction, a field is added:

Boolean \texttt{m\_isCVoltDBExecution}

The same field is added to *MultiPartitionParticipantMessage*. The CVoltDB flag is needed (at both the coordinator and participants) so that we do not have to look up catalog information on a stored procedure to determine if we need to switch from executing CVoltDB transactions to VoltDB transactions or vice versa. This flag passes the execution mode set by the user into the server. Inside the server, a variable \texttt{m\_isCVoltDBExecution} was added on each Execution Site. Only transactions with compatible execution mode can execute transactional work on each site, otherwise, blocking happens until incompatible transactions finish.

<table>
<thead>
<tr>
<th>Execution Site Execution Mode</th>
<th>Transaction Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{m_isCVoltDBExecution}</td>
<td>\texttt{Serial}</td>
</tr>
<tr>
<td>\texttt{m_isCVoltDBExecution}</td>
<td>\texttt{Concurrent}</td>
</tr>
</tbody>
</table>

There is one algorithms involved here: switching between two execution modes on one Execution Site, as described in Algorithm 1.

The process to synchronize Execution Mode among nodes is done automatically because of CVoltDB’s architecture. When there is a distributed transaction starting to *Prepare*, the Execution Mode of all involved nodes will eventually become consistent, however, this switching can happen at various times, depending on the situation of previous transactions’ execution on each node. During the *Prepare* of this distributed transaction, the eventual Execution Mode of the \(N\) involved nodes is the same, and this mode will hold
Algorithm 1: Switching Execution Site Execution Mode

at least until all participant Prepares have finished. The execution mode at the moment
that the last participant sent results to the coordinator will be the same for all nodes in the
transaction. This holds true until the first participant receives the CompleteTransaction
notification. Thus for at least a small interval, all involved nodes will have the same
Execution Mode. That is, the Execution Mode of an Execution Site will not change until
next Prepare. Therefore, a system procedure that involves all nodes, or a distributed
transaction involving all nodes, will synchronize the execution mode in the whole system
once it starts preparing.

4.2 Uses Only Needed Partitions

4.2.1 Partition Mapping

VoltDB holds the mapping information between partition keys and partitions, and between
partitions and sites. The first map knows which partition key value is represented in which
database partition; the second map is bi-directional, providing which partitions are held on which sites and vice versa. Partition keys are specified by user when creating the table and cannot be changed at runtime without stopping the database, recompiling the catalog, and reloading the data again.

### 4.2.2 Changes to ProcInfo

Currently, the programmer using VoltDB can specify in the stored procedure which partitions a Local Transaction runs on. It passes the information in partitionInfo in a string of the form "table.column:parameterindex". We expand this mechanism by allowing multiple parameters to specify which partitions are involved in a distributed transaction. The new form of partitionInfo is:

"table.column:parameterindex[, . . . , parameterindex]"

If the transaction needs to access data from two or more partitioned table, the programmer must make sure that all tables are partitioned based on the same key and needed partitions can be represented by only one partition table and key set. Otherwise, the transaction annotation should fall back to the simplest case, i.e., use all partitions.

### 4.2.3 Changes to Messages

Besides the changes described in section 4.1, there is an additional change to the messages needed. In InitiateTaskMessage, we append the list of participant Execution Site IDs.

InitiateTaskMessage: new fields int[] m_participantSiteIds
The participant site IDs specify the participant sites, not including the replicas of the coordinator. The coordinator will augment this list by the set of coordinator replicas to determine the sites to send fragments to.

4.2.4 Algorithms to lookup involved partitions

When a transaction is about to run, there are two parts of transaction program information; one is transactional work, which has been compiled from SQL statements and stored in the catalog in the system, the other is the Java stored procedure and runtime parameters’ values.

If partitionInfo is specified, the client combines received parameter values and partition mapping information to precisely calculate the needed partition subset in Algorithm 2. Note that we need to store the information of involved partitions in the transaction object held by the Execution Site, so that the transaction knows where to send the work requests.

This means the involved_partitions_list (computed in Algorithm 2) related to the Uses Only Needed Partitions feature not only plays a role when sending the transaction request, but also when sending the real transactional work fragments.

4.3 Early Fragment Distribution

Instead of starting work fragment distribution when the transaction on the coordinator node is next to execute, distribution work can be scheduled by the Execution Site as soon as the transaction on the coordinator is Ready to Start and the system is idle, shown in Algorithm 3.
1: procedure How $T_i$ finds its needed partitions
2: Input: $T_i$, a distributed transaction
3:   parameter_index_list = \{paramindex1, paramindex2, \ldots\}
4:   partition_map = \{(partitionkey1, partition1), (partitionkey2, partition2), \ldots\}
5: Output: involved_partitions_list = \{involvedP1, involvedP2, \ldots\}
6:   for each param_index $i$ in parameter_index_list do
7:     retrieve param$_i$ using paramindex$_i$
8:     if param$_i$.type is Array then
9:       for each entry subparam$_i$ in param$_i$ do
10:          look up partition $p$ in partition_map using subparam$_i$
11:          add $p$ into involved_partitions_list
12:       end for
13:     else
14:       look up partition $p$ in partition_map using param$_i$
15:       add $p$ into involved_partitions_list
16:     end if
17:   end for
18:   store involved_partitions_list in $T_i$.involved_partitions
19:   return involved_partitions_list
20: end procedure

Algorithm 2: Find Needed Partitions

Early Fragment Distribution is not transactional. It transfers work fragments, which are represented as executable code (compiled from SQL statements) and their runtime values, between nodes, but will never talk to the data engine and touch the data.

Participant nodes do not distribute fragments, only receive them, so when a transaction $T_i$ becomes Visible in TransactionQueue, it is simple added to the StartedTxnQueue without further processing.

However, when the participants receive work fragment of $T_i$ from the coordinator, they will add new received fragments to the current work fragment set of $T_i$, as described in
1: procedure Coordinate distributes fragments
2: Input: TransactionQueue, StartedTxnQueue
3: Output: modified TransactionQueue, StartedTxnQueue
4: dequeue next Visible transaction $T_i$ in TransactionQueue
5: if $T_i$ is a Local Transaction or only a participant execution then
6: add $T_i$ to StartedTxnQueue
7: else
8: create a new thread (proc-thread) for $T_i$
9: switch to the proc-thread
10: send work fragments to all nodes in $T_i$.involved_partitions
11: switch back to the main thread
12: add $T_i$ to StartedTxnQueue
13: end if
14: end procedure

Algorithm 3: Early Fragment Distribution (Coordinator)

Algorithm 4. At this time, $T_i$ can sit on either TransactionQueue or StartedTxnQueue, or be the current preparing transaction, waiting for the work request to arrive. $T_i$ sitting on TransactionQueue indicates that the Early Fragment Distribution work started earlier on the coordinator.

Fragment distribution for a distributed txn, i.e., distribution of work requests to participants, can happen as late as the start of the coordinator Prepare phase. See Algorithm 3. But we usually arrange to send out the fragments somewhat earlier, during an idle time of the coordinator node, so that the participant is less likely to need to wait for them to arrive. So the goal is that all participants receive needed fragments earlier than the start of the transaction prepare on the participant to avoid any unnecessary waiting by participants. The very best scenario is that all the participants start earlier than the coordinator Prepare and send back results before the coordinator starts Preparing, thus
1: **procedure** Participants handles received fragments
2: **Condition**: a set of fragments is received on a node
3: **Input**: work fragments F
4: **Output**: modified Tᵢ’s work fragment set FS
5: 
6: find transition Tᵢ which F belongs to
7: add F to Tᵢ.FS
8: **end procedure**

Algorithm 4: Early Fragment Distribution (Participant receives fragments)

avoiding early waits at both participants and coordinator. Clearly this best scenario can only happen in a fraction of the cases.

To send out the work requests, the coordinator looks up the stored procedure to run and executes it, and in the stored procedure determines the runtime values for the SQL operations, and composes the messages to send to participants. The early fragment distribution work is complete just after the send, but the Java stored procedure is only part way through its work at that point. A separate thread is created on the coordinator node for each distributed transaction, in order to keep the thread-wise context for the Java stored procedure to save it until later when this transaction is chosen to start preparing by the main thread. The main thread executes the fragments in the data engine and does all incoming message handling, but it does not do all the Prepare work because the Java stored procedure execution is part of that work. However, we can’t simply save a thread context and recreate it later in Java, thus, a separate thread is necessary for each distributed transaction. A thread pool is used as an optimization here.

On the coordinator, the new proc-thread will do the fragment distribution and then hand over to the main thread immediately. The main thread later chooses this transaction to
Prepare, and causes a switch back to its thread when results are ready for processing by the stored procedure. When the stored procedure returns, the created proc-thread no long exists, and the result is stored in the transaction object at that time. Then main thread goes on to send out commit messages to all participants. Later, when all acknowledges from participants are received, the transaction finishes.

There is no thread switch on the participant, so the main thread just executes the transaction work from beginning to the end and returns its local result to the coordinator one or more times, as directed from the coordinator, and eventually executes the decision from the coordinator with commit / abort.

4.3.1 Fragment Assembly

Fragment assembly provides sets of fragments for Early Fragment Distribution, and later sets of fragments for distributed execution of additional transactional work as needed by the transaction program. CVoltDB uses the same kinds of fragments and the same distribution system for them as VoltDB. The reason that we changed anything in this area is that there was a performance bug affecting distributed transaction performance that seriously interfered with our ability to measure distributed transaction performance of our system.

In this section, a dependency is a data table, either already filled with query results of one that is expected in the future. Note that even updates, deletes and inserts return tables of row counts. In fragment assembly, fragments are organized by dependencies and placed in a queue in specific order and then sent to different nodes to execute. For a SQL statement, usually two fragments are generated during compile time. The first fragment is the
distributed work executed on all involved nodes, and the second fragment is the accumulation work only executed on coordinator that assembles partial results returned by the participants. However, not all SQL statements require both fragments. For a replicated table, which is stored on all sites, it is enough to execute the SQL only on the coordinator (or any other single node) for a Read operation and only one fragment is generated. However, this is only true when there is no writes of replicated table are interleaved with the reads in VoltDB. There are two approaches VoltDB implemented to handle mixed replicated table Reads and Writes: the first is that they sent the replicated table Reads to all participants, which is unnecessary; the optimized approach is executing replicated table Reads locally on coordinator and dividing the batch into smaller sequences to make sure of the correct order of Reads and Writes.

The optimized approach causes additional network communication, and instead we arranged to execute replicated table Reads on coordinator in correct position with respect to the Writes and avoid breaking up the batch by composing a custom fragment sequence of the "distributed work" for the coordinator rather than trying to use the same sequence for both coordinator and non-coordinator sites.

We scan one batch of SQL statements one by one, track the dependency information for distributed work and accumulation work for Writes and non-replicated table Reads, and place replicated table Reads in proper place in this sequence in the fragment set bound for the coordinator called the localDistributedFrags. Here "distributed fragments" are the fragments collecting information from participants, and "local distributed fragments" are fragments collecting the information from the local participant, i.e. the coordinator site working as a participant. Similarly "remote distributed fragments" collect information
from non-coordinator sites. There are three different sets of fragments generated in
Algorithm 5:

```
1: procedure Assemble fragments for a batch of SQL statements
2: Input: \{S_1, S_2, \cdots, S_n\} a batch of SQL statements
3: Output: localDistributedFrags, coordinator distributed work
4: remoteDistributedFrags, non-coordinator distributed work
5: localFrags, accumulator work done on coordinator
6: for S_i in \{S_1, S_2, \cdots, S_n\} do
7:     if S_i contains one fragment f then
8:         put f into localDistributedFrags
9:         mark no input dependency for f
10:     else
11:         f1 is the first fragment:
12:         put f1 into localDistributedFrags
13:         put f1 into remoteDistributedFrags
14:         f2 is the second fragments:
15:         put f2 into localFrags
16:         mark f2 depends on the results of f1
17:     end if
18: end for
19: end procedure
```

Algorithm 5: Assemble fragments for a batch of SQL statements

- *localDistributedFrags* are fragments generated to execute the distributed task on the coordinator

- *remoteDistributedFrags* are fragments generated to send and execute distributed task on non-coordinator nodes

- *localFrags* are fragments performing accumulation work on the coordinator
In Figure 8, $S_1$ and $S_2$ are statements other than replicated reads, and each contributes fragments to $remoteDistributedFrags$ and $localDistributedFrags$, and their results (from all the participants) are routed to be accumulated by fragments in $localFrags$, and the final results are sent to output Results. Statement $S_i$ is a replicated read, so its fragment is inserted only in $localDistributedFrags$, and its results need no accumulation, and thus are sent directly to the output Results.

### 4.4 Consecutive Prepares Execution

Once the transaction prepare is executing at a node, each fragment will be passed into the data engine to execute. However, in VoltDB, the data engine on each site does not need to
know which transaction this fragment belongs to. Because of the Local Serial Execution of VoltDB, it does not need to track the data dependency because there won’t be any possible data conflicts. Therefore, in order to support Consecutive Prepares Execution, we must first introduce a mechanism to track the data dependencies inside the engine to detect conflicts.

Escrow UPDATE is the UPDATE using the syntax described in Definition 3.7, and the Non-Escrow UPDATE is the regular UPDATE not using the special syntax. We first consider non-Escrow operations, including INSERT, DELETE and Non-Escrow UPDATE.

In a database system using row locking, each data item in the data engine is exclusively controlled by its first changer transaction, i.e. the existing uncommitted transaction that performed or attempted to perform Insert, Non-Escrow Update, or Delete operation on this row. Once there is a change on that row, no change from other transaction is allowed to occur on the same data item. That is:

- For an Insert operation, only later operations (Read, Update, and Delete) by the same transaction are allowed.
- For a Non-Escrow Update operation, later operations (Read, Update, and Delete) by the same transaction are allowed; and Read operations on different columns by a different transaction are also allowed.
- For a Delete operation, it’s impossible to have later operations for the same transaction. Because a deleted row is never returned as selected row set for any operation.

Therefore, at least there are two kinds of information we need to remember: operation type, i.e. Insert, Update, or Delete, and the Txn ID.
Note that, an **Update** operation usually involves a few columns. Later **Reads** from other transactions should not be blocked if they are reading an unchanged column. So for **Update** operations, we also need to record the updated columns.

Figure 9 shows the data structure on the table level to track uncommitted transaction operations for each changed (newly inserted, updated, or going-to-be-deleted) row.

| Pointer to After Image of this row: PA |
| Pointer to Before Image of this row: PB |
| Pointer to row schema: PS |
| Changer Txn ID: CID |
| Changer Type: CT |
| Operation Count: OC |
| Updated Columns: UCOL |

Figure 9: Data Structure of Row Journal

We call this data structure *Row Journal*. Each table in the data engine maintains a map from pointer of every row involved in change, current or not, to its corresponding *Row Journal* pointer.

In order to share a single *RowJournal* between the current row and the before-image row used in a Non-Escrow **Update**, we use two maps in the implementation, one to house the *RowJournals* and another to map from either of the two row data pointers to the one *RowJournal* in storage.

The row pointer is the runtime identifier of a row in the data engine, and it is only occasionally changed during table compaction. We reflect the change in the
RowJournalMap easily by tracking it down during the table compaction, expanding on the VoltDB mechanism to update the row pointers in the index entries. Due to this possible unstable property of row pointer, we use the RowJournal pointer instead of the row pointer in some cases since it will never be changed during the execution, in addition to saving lookup time.

Algorithm 6 shows the creation of RowJournal for first Insert / Update / Delete operation operating on the row, and Algorithm 7 describes how the system works for later operations if the RowJournal already exists.

**Insert** In current VoltDB, a row insert adds all the needed index entries so that index scans will see new rows. There is no indication that the row is new, since none is needed with unmodified VoltDB’s one-txn-at-a-time execution. We need to create a RowJournal entry for an inserted row, with the non-escrow changer TxnID, type INSERT. There is no before image for an inserted row, only an after image.

**Non-escrow Update** In Algorithm 6, only one RowJournal is created for the Non-Escrow Update, but two mappings exist, one is from the old row, and the other is from changed row. Both rows are sitting in the table, and they are pointed to from all non-unique indexes. We are assuming no updates are made to unique columns and have only one pointer for both versions of the changed row in a unique index. In VoltDB, an update indexes the new column values, whereas we need the old values to show up in index scans too. So (unless this transaction inserted this row) we need two rows to represent the two possible future versions of the row, each with (secondary) index entries. Both rows are sitting in the table, but only one of them (the current version’s) has an entry in a unique index, to avoid duplicates. Here we are assuming no update is made to unique columns.
1: procedure Create a row journal when change the row in the first time
2: Input: New operation \( P \) in transaction \( T_i \) on row \( R \) (can be newly inserted) in table \( T \)
3: Condition: no row journal \( J \) for row \( R \) in \( T.\text{RowJournalMap} \); new operation type is not Read
4: create a row journal object \( J \)
5: \( J.PA = &R \) \( \triangleright \) after image points to \( &R \)
6: \( J.PS = T.\text{RowSchema} \) \( \triangleright \) the row schema is the same as the table
7: \( J.CID = T_i \) \( \triangleright \) the changer is \( T_i \)
8: \( J.CT = P.\text{type} \) \( \triangleright \) type is one of Insert, Update, or Delete
9: add \( J \) into \( T.\text{RowJournalStorage} \)
10: if \( J.CT == \text{Update} \) then
11: mark updated columns
12: copy \( R \) prior to change to \( R' \)
13: insert \( R' \) to the table \( T \) and its indexes
14: \( J.PB = &R' \) \( \triangleright \) before image points to \( &R' \)
15: \( \triangleright \) add mapping entry of before image row to \( \text{RowJournalMap} \)
16: Add(\( &R' \), \&\( J \)) into \( T.\text{RowJournalMap} \)
17: else
18: \( J.PB = \text{null} \) \( \triangleright \) no before image
19: end if
20: \( \triangleright \) add mapping entry of current row to \( \text{RowJournalMap} \)
21: Add(\( &R \), \&\( J \)) into \( T.\text{RowJournalMap} \)
22: end procedure

Algorithm 6: Row Journal Creation

The row pointer to the changed row is the after image, the one that lives on after commit or abort, while the other "before" image is considered of secondary importance, and is the one that gets discarded at commit or abort. We can detect Write-Write conflicts at non-Escrow update time by looking up the found row’s \( \text{RowJournal} \) entry and finding a non-Escrow changer \( \text{TxnId} \) different from our own \( \text{TxnId} \). In this case we throw a \( \text{DataConflictException} \) from the data engine back to the Execution Site, as discussed.
1: **procedure** Row Journal Maintenance
2: **Input:** New Insert / Update / Delete operation $P$ in transaction $T_i$ on row $R$ (can be newly inserted) in table $T$
3: **Condition:** existing row journal $J$ for row $R$ in $T$.RowJournalMap
4: 5: if $T_i$.ID $\neq$ $J$.CID then $\triangleright$ $T_i$ is not the changer in $J$
6: if $P$.type $\neq$ Read $\parallel$ $P$.cols in $J$.UCOL then $\triangleright$ cannot read a changed column
7: BLOCKING
8: end if
9: else $\triangleright$ operation from the same changer
10: if $P$.type == Insert $\parallel$ $J$.CT == Delete then
11: cannot happen $\triangleright$ cannot insert duplicated row or delete already deleted row
12: else
13: if $P$.type == Update then
14: $J$.UCOL = $J$.UCOL $\cup$ $P$.cols $\triangleright$ add new updated cols to $J$.UCOL
15: end if
16: if $P$.type == Delete then
17: $J$.CT=Delete
18: end if
19: end if
20: end if
21: **end procedure**

Algorithm 7: Row Journal Maintenance

further later. If the non-Escrow changer TxnId is our own TxnId, it should be an Insert or a Non-Escrow Update.

In the case that this transaction inserted the row, we handle the row as in current VoltDB, i.e., we just do the non-escrow update on the one current version of the row and continue to call it type INSERT. It continues to have only one row image in the table.
For a non-Escrow update that follows a non-Escrow update by the same transaction, with a second row version, we apply the updates to the current version of the row, thus continuing to have two versions of the row, and type UPDATE.

Delete

In VoltDB, a row delete action removes all the row’s index entries, so that currently no index scan will encounter deleted rows. We need to change this (and code doing scans) so we can detect phantoms. Also, in the case that this is the first operation on this row by this transaction, we need to create a RowJournal entry with the non-Escrow changer TxnID with type DELETE, and mark the tuple deleted as in current VoltDB. We can detect Write-Write conflicts by looking up the old row’s journal entry and finding a non-Escrow changer TxnID different from our own TxnID. In this case we throw a DataConflictException. If the non-Escrow changer TxnID is our own TxnID, its journal should show a Non-Escrow Update (with two row versions) or INSERT (with one row version). In the UPDATE case, where a non-Escrow update is followed by a delete by the same transaction, the operation becomes a DELETE, since that eclipses the insert or update for data conflict detection. The current tuple should remain as is and the before tuple is kept in the table, since we must represent this possible phantom. The type becomes DELETE, with two row versions. In the INSERT case, the one row version is marked deleted, as in current VoltDB, and the type becomes DELETE.

We can tell the three subclasses of DELETE by whether or not before image pointer is null, and whether or not Operation Count > 1:

a. If (Operation Count == 1), case of simple DELETE
b. If (Operation Count > 1 && Before Image == null), case of INSERT ··· maybe UPDATEs ··· DELETE

c. If (Before Image ≠ null), case of UPDATEs ··· DELETE

Therefore we do not need any other flags in the data structures to track state.

Note that the lifetime of a RowJournal is from the first non-Escrow change to a row until commit or abort of that transaction with no escrow data handling.

When blocking happens, a special type of exception DataConflictException is generated inside data engine and thrown back to the Execution Site to process, shown in Algorithm 8.

```
1: procedure Handling DataConflictException
2: Condition: DataConflictException is detected
3: Input: blocked transaction T_B conflicts with changes made by T_i
4: record T_i as Blocker
5: mark T_B as blocked
6: if current fragment f of T_B is not Read-Only then
7:    undo the change made by f
8: end if
9: while T_i is not committed do
10:   handle messages, commits / aborts of earlier transactions
11: end while
12: clear up Blocker
13: resume execution of T_B from f
14: end procedure
```

Algorithm 8: Blocking Handling
4.5 The Ordered Escrow Method

Based on the implementation of Consecutive Prepares Execution, the Ordered Escrow Method needs some more fields in the RowJournal data structure to record the change of Escrow Update, the new data structure becomes to Figure 10.

Recall from Section 3.4.4 that we designate columns as Escrow Columns using a naming convention. Thus in the engine we can determine that a particular update operation is acting on an escrow column while the system is executing a CVoltDB transaction, and thus classify it as an Escrow Update. Otherwise it is considered to be a non-Escrow
update. We have separate data structures for non-Escrow operations and Escrow
operations, so the row journal has two important pointers, one points to the Non-Escrow
Changer, which contains almost the same fields as the data structure in section 4.4, and
the other points to a map of Escrow Data, i.e. a map from column index to an Escrow
Data object. Each Escrow Data object represents the state of one Escrow column,
including column index, current value range, and all Escrow Journals, one for each
Escrow update on this column value. An Escrow Journal is the representation of one
granted Escrow Update Request on one column value: it contains the transaction identifier
of this Escrow Update, the value change delta for this column, and the pointer for next
Escrow Journal on the same row and same column.

If one Escrow Update operation changes multiple Escrow columns in one statement, then
multiple Escrow Journals will be created for this single Escrow Update operation.

Example 4.1 (Escrow Update in CVoltDB): Transaction T_{001} performs a Escrow Update:
UPDATE T SET X -= 100 WHERE PK = 200

1. Prepare Escrow Update Operation

Execution arrives in updateTuple for this PK= 200 row, with argued old row and
new row. The update is for an Escrow Column. The new row has a value of −100
for this column, meaning we should use an increment of −100. The old row (which
matches the after image of the RowJournal, if the RowJournal already exists) has
the committed value of X, which we’ll call the row_val. In the following, we
assume that the MIN is 0 for this Escrow Column.
(a) Look up the old row/s data pointer, get the RowJournal or start a new one for the row. Increment Operation Count OC for this row. Find the affected escrow columns (only one in the example), and for each:

i. Find the EscrowDate entry for this column, or start a new one, with
\[ \text{INF}=\text{SUP}=\text{row\_val}, \text{where row\_val is in the column value in current tuple.} \]

ii. Check if \( \text{SUP}-100 < 0 \) and if so, throw CONSTRAINT_ESCROW_OUT_OF_BOUND exception.

iii. Check if \( \text{INF}-100 < 0 \) and if so, we have a possible but not certain ESCROW_OUT_OF_BOUND, so a DataConflictExcpetion is thrown back to Execution Site to cause a Wait until the commits and aborts resolve the value.

iv. update EscrowData:
   - \( \text{INF} = \text{INF}-100 \)
   - \( \text{row\_val} = \text{row\_val}-100 \)
   - SUP unchanged

v. Create a new Escrow Journal entry with delta= \(-100\), TxnID=\(T_{001}\), and undoActionIdx=1 or more, copied from the current value of OC. Add the new Escrow journal to the list for the Escrow data.

(b) Current undo / release entry using a UPDATE undo entry. This entry holds a RowJournal pointer and a table pointer.

The undoActionIdx value is needed to support incremental undo, where we need to undo only certain undo actions for a transaction that has executed a
query via function `executeQueryPlanFragmentAndGetResults`. This may not be needed in VoltDB 3.1 where `executeQueryPlanFragmentAndGetResults` has been phased out.

2. **Abort / Commit Escrow Update Operation**

For **Abort**, we have two approaches, incremental undo and all-at-once undo (which is slightly more efficient). Incremental undo is needed for undoing only a subset of undo actions as is needed if the transaction has used `executeQueryPlanFragmentAndGetResults` and suffers a `DataConflictException` during that execution. We decide on incremental undo when we see that this transaction is undoing and no other intervening transaction has started to prepare. This means that some simple aborts of transaction without any intervening prepares will use incremental undo unnecessarily. **Commit** processing is always all-at-once.

**T₀₀₁** Commits or Aborts: We process the release or undo for this transaction

(a) Loop over undo entries.

(b) For one undo entry, using its `RowJournal` pointer to locate the target row

(c) Find all its **Escrow Data** objects, and for each

i. For incremental undo. Find **Escrow Journal** entries with `TxnID=T₀₀₁` and `undoActionIdx==OC`, and for each such entry, use its delta and `ColIdx` to update its **Escrow Data** and in one case

   • INF=INF+delta
   • `row_val=row_val+delta`
   • and keep SUP unchanged
• Delete this Escrow Journal entry
• Check if this was the last Escrow Journal entry and if so, remove the Escrow Data.

ii. For all-at-once undo / commit.
Find Escrow Journal entries with TxnID=T001, and for each such entry, use its delta and Colldx to update its Escrow Data and in one case, row_val:
• If Abort
  – INF=INF+delta
  – row_val = row_val + delta
  – keep SUP unchanged
• If Commit
  – SUP=SUP-delta
  – keep INF and row_val unchanged
• Delete this Escrow Journal entry
• Check if this was the last Escrow Journal entry and if so, remove the Escrow Data.

(d) Decrement OC. If the Non-Escrow Changer is null and OC==0, remove the RowJournal.

While doing Escrow Update, we always keep track two intervals: [MIN, MAX] and [INF, SUP]. MIN and MAX are the lowest and highest bounds set by user for one Escrow Column; INF and SUP are current range of lowest and highest possible values for this
1: procedure Escrow Update Handling
2:  Condition: pre-exam going-to-changed columns are of type Escrow Column
3:  Input: transaction Ti wants to update row &R by delta of table T
4:  if no row journal J in T.RowJournalMap for row R then
5:    create row journal J
6:    add J in T.RowJournalStorage
7:    add {&R, &J} in T.RowJournalMap
8:  else
9:    find row journal J for row R
10: end if
11: for each changed Escrow Column c do
12:    create an Escrow Journal EJ
13:      EJ.delta = delta  ▷ set delta
14:      EJ.TID = Ti  ▷ set the changer of this Escrow Journal
15:      EJ.NEJ = null  ▷ this is the first Escrow Journal
16:      check OutOfBound possibility  ▷ See Algorithm 10
17:      if Escrow Data ED for c not exists in J then
18:        create an Escrow Data ED
19:          ED.CI = c.getCid  ▷ set column index
20:          ED.INF = ED.SUP = R.c.val  ▷ initial INF, SUP are set to be the column value
21:          ED.EH = EJ  ▷ EJ is the head of chained list
22:          add (c.CID, ED) to J  ▷ add mapping entry to RowJournalMap
23:        else
24:          find ED in J using c.cid  ▷ already exist the Escrow Data
25:            EJ.NEJ = ED.EH  ▷ add new Escrow Journal at the beginning
26:            ED.EH = EJ  ▷ re-set the head of the list
27:        end if
28:      EJ.VAL = EJ.VAL + delta
29:    end for
30: end procedure

Algorithm 9: Escrow Update Handling
Escrow Column. Before placing an Escrow Update, we have to check Rule 3.6, which can BLOCK under certain "corner" conditions.

```
1: procedure Check OutOfBound Possibility
2: Input: [MIN, MAX] and [INF, SUP] on Escrow Column c, delta from operation P in transaction Ti
3:   if delta > 0 then
4:     if INF + delta > MAX then
5:       do not allow P, abort Ti
6:     else
7:       if SUP + delta > MAX then
8:         BLOCK
9:     end if
10:   end if
11: else
12:   if SUP + delta < MIN then
13:     do not allow P, abort Ti
14:   else
15:     if INF + delta < MIN then
16:       BLOCK
17:   end if
18: end if
19: end if
20: allow P
21: end procedure
```

Algorithm 10: Check OutOfBound Possibility

When a BLOCK occurs, a DataConflictException is thrown and returned to the Execution Site; the blocked transaction will be continued when blocking transaction commits / aborts, just as the same as the normal DataConflictException handling described in Algorithm 8.
With Escrow handling, the lifetime of a *Row Journal* is from the first transaction changing a row, either non-escrow change or escrow update, until all transactions on this row commit or abort. The *Row Journal* can have both non-Escrow changer and the array of *Escrow Data* or only one of them.

### 4.6 Summary

This chapter presented the detailed implementation of how CVoltDB can be implemented in the real-world database system, with adding concurrency to VoltDB. This is not a trivial effort, and the work touches many areas of the DBMS. The goal of this chapter was to highlight some of the challenges faced, and draw attention to some of the subtle issues encountered.
CHAPTER 5

EXPERIMENTS AND PERFORMANCE ANALYSIS

This chapter first presents a performance evaluation of CVoltDB. We use standard TPC benchmarks, TPC-B [TPC94] and TPC-C [TPC10]. TPC-B is developed following the defined standards, and TPC-C [TPC10] is a well-established industry-standard benchmark, and we adapted it from the less distributed implementation provided by VoltDB, except that our version is true to the distributed nature of the specification. We will first give a description of test environment, then present test results, and finally we will propose an analytic cost model and compare with the results.

5.1 Experiment Setup

We ran our tests using a four-machine cluster. All of them are identical Dell minitower systems. Each has a quad-core Core i5 – 3450 3.10GHz CPU, 16GB memory. The cluster is connected by a Gigabyte Ethernet switch, and run 64-bit Ubuntu Linux 12.04 LTS.

We will compare the performance of CVoltDB against VoltDB in two benchmarks, the TPC-B [TPC94] benchmark and TPC-C [TPC10] benchmark. There are four configurations we tested:
1. **Original VoltDB.** VoltDB 2.1.3, with one performance bug fix to avoid unnecessary shots. This follows *Local Serial Execution* only, allowing transactions to execute one by one at each node.

2. **Basic Features.** CVoltDB with *Uses Only Needed Partitions* and *Early Fragment Distribution*. This still runs only one transaction at a time on each node, but both *Local Transactions* and *Distributed Transactions* can be concurrently running at disjoint partition sets.

3. **Consecutive Prepares Execution.** CVoltDB with limited concurrency using Consecutive Prepares Execution, allowing the *Prepare* of T\(_j\) to immediately follow the *Prepare* of T\(_i\) on each node, etc.

4. **Escrow.** Full CVoltDB processing, with *Escrow Columns* and the *Ordered Escrow Method*.

We run experiments by 1 minute and 5 minutes, as long as all tables’ data still resides entirely in the memory.

### 5.2 Test Results

Our performance tests were run on [TPC94] and TPC-C [TPC10] benchmarks, both are OLTP type workload. We measure them with different data distribution and also test the system scalability.
5.2.1 TPC-B Experiments

TPC-B [TPC94] is a simplified version of the more realistic TPC-A Benchmark that avoids the need for terminal simulation and Teller think time, simply running TPC-A logic with a single Driver. The TPC-B benchmark measured performance of banking transactions when many database systems locked pages and had various other performance flaws. TPC-B defines four tables of which the first three have a specified numbers of rows, shown in Table 1.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Count of Rows</th>
<th>Row Size</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>N</td>
<td>100 bytes</td>
<td>B_ID</td>
</tr>
<tr>
<td>Teller</td>
<td>10N</td>
<td>100 bytes</td>
<td>T_ID</td>
</tr>
<tr>
<td>Account</td>
<td>100000N</td>
<td>100 bytes</td>
<td>A_ID</td>
</tr>
<tr>
<td>History</td>
<td>Varies</td>
<td>50 bytes</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: TPC-B Tables Description

TPC-B explicitly allows horizontal partitioning, and appropriate partitioning is by B_ID. For each successive transaction, a Driver sends to the SUT (System Under Test) four separate integers, Aid, Tid, Bid and Del (values for A_ID, T_ID, B_ID and a Delta Increment). At the end, the transaction returns to the Driver the Account Balance resulting from the transaction.

**Transaction Logic Profile**

Given Aid, Tid, Bid and Del (and a Timestamp TS needed by *CVoltDB*).

**BEGIN TRANSACTION**

```
    UPDATE Account SET Balance += Del WHERE A_ID = Aid;
```
– Escrow Update

SELECT Balance FROM Account WHERE A_ID = Aid;

– Read

UPDATE Teller SET Balance += Del WHERE T_ID = Tid;

– Escrow Update

UPDATE Branch SET Balance += Del WHERE B_ID = Bid;

– Escrow Update

INSERT INTO History VALUES (Aid, Tid, Bid, Del, TS);

– Insert

COMMIT TRANSACTION

Return Account Balance to Client;

Note that the Escrow Update syntax uses normal SQL updates in first three configurations.

The transaction is single-node if the Aid Branch (Aid/100,000) equals Bid; otherwise, if they are different, it is a distributed transaction involving two Branches, and since partitioning is by B_ID, these Branches are in two partitions and the transactions is distributed. There is a requirement that 85% of the transactions choose a branch on the local node and 15% choose a foreign branch chosen at random. The update increment Del is chosen randomly from a large negative-positive range and 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 (usually there are none, since there are 100,000 Accounts to a Branch) and return the value of the Account Balance. Note that the Account Balance, Teller cash and Branch
balance are all updated by the amount of the Account change, a feature known as triple entry bookkeeping to foil any attempts at fraud.

### 5.2.1.1 Increasing the percentage of distributed transactions

The TPC-B benchmark specifies that 15% of the transactions should involve two branches, thus two partitions, and the others just one. We have scaled this distributed mix value from 0 to 100% so we can see the effect of various levels of distributed transactions in the mix.

Figure 11, Figure 12 and Figure 13 show that VoltDB runs pure single-node transactions significantly faster than CVoltDB, especially in single host case, but as soon as even 1% of the transactions are distributed (Distributed = 1%), CVoltDB is much faster, about ten times faster on two hosts at 15%, the standard TPC-B multi-partition mix, and 20 times faster on four hosts at 15%.

Table 2 summarizes the speed up of Basic Features, Prepare-Interleaved Execution and Escrow against Original VoltDB.

<table>
<thead>
<tr>
<th>Distributed%</th>
<th>Basic Features</th>
<th>Consecutive Prepares Execution</th>
<th>Escrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.11 x</td>
<td>3.77 x</td>
<td>4.39 x</td>
</tr>
<tr>
<td>15</td>
<td>7.77 x</td>
<td>10.04 x</td>
<td>21.31 x</td>
</tr>
<tr>
<td>60</td>
<td>9.05 x</td>
<td>13.92 x</td>
<td>26.92 x</td>
</tr>
<tr>
<td>99</td>
<td>9.47 x</td>
<td>15.48 x</td>
<td>28.16 x</td>
</tr>
</tbody>
</table>

Table 2: TPC-B TPS Speed-up Table

Three basic optimizations, including Uses Only Needed Partitions and Early Fragment Distribution are very important to good performance. This Basic Features configuration
Figure 11: TPC-B TPS by Distributed%, Single Host

increases the throughput roughly by 2, 4, 8 times at 15% distributed transactions on single, two, and four hosts representatively.

Adding our limited form of on-node concurrency (Consecutive Prepares Execution, but not Escrow) improves performance of distributed transactions, but not by much in this benchmark. All transactions update the branch balance by a non-Escrow operation in this case, and thus each one blocks until the previous transactions Commit. For this benchmark, the concurrency support lays the groundwork for the Escrow Method to make a bigger difference. From Table 2, the throughput after adding Escrow is about two times more with reasonable distributed %.
5.2.1.2 Scale Up Experiment

The reason to choose shared-nothing architecture is for its good scalability. However, from Figure 14 and Figure 15, they show the throughput of Original VoltDB actually can’t take advantage of added resources with occurrence of distributed transaction.

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. Once multi-host is in use, on average, adding one more host increases throughput performance by about 30%, while Original VoltDB does increase throughput by about 15% for each additional host when no distributed transactions.
Figure 13: TPC-B TPS by Distributed%, 4 Host

5.2.2 TPC-C Experiments

The TPC-C [TPC10] benchmark is the oldest benchmark still supported by the Transaction Processing Performance Council, with a rather complex structure, having nine different tables and five different transactional profiles. It dealt with ordering, paying for, and delivering goods from warehouses. Table 3 lists the TPC-C Tables and Table 4 lists the transactions.

5.2.2.1 Transactions in TPC-C

**New Order Program.** A new order is placed for a Customer (a row in the Customer table). The home warehouse is the customer's warehouse, and its node is the optimal Coordinator under CVoltDB. Placing the order involves inserting a row in the Order table,
and also a row in the New-Order table, which basically points to the Order row. Then a series of five to fifteen rows are placed in the Order-Line table for that Order row, the individual Line Items that constitute a full order. Some Line Items may reference items held in remote warehouses, and stock-level adjustments (in the Stock table) for these Line Items require remote node access.

To accomplish its task, the New-Order program reads and/or updates a number of fields from the Warehouse table, District table, Customer table, Item table, and Stock table, but only two of these accesses have potential data conflicts: S_QUANTITY in the Stock table, the quantity of a stock item on hand, decremented by New-Order, and D_NEXT_O_ID in the District table, the next order number for the district, which New-Order uses and increments in a Non-Escrow manner, since this column cannot be further incremented by
Figure 15: TPC-B TPS by count of hosts, 99% distributed

another transaction until this transaction Commits; thus no "holes" in the D_NEXT_O_ID value sequence can occur. This may take long time and is no faster for Escrow than non-Escrow columns. (It is an unusual requirement to disallow "holes" in Order Numbers, but it is required by TPC-C.)

Here are the work steps in the Prepare of the New-Order transaction.

1. An average of 10 stock items are decremented from quantity on hand in Stock tables (there is a 1% chance in each case of being remote). Locally (on the home warehouse), we retrieve data from the Item table (a table replicated on each node), and other tables. In the distributed case, this data goes back to the Coordinator and may require a new shot for step 2.
### Table 3: TPC-C Tables Description

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Count of Rows</th>
<th>Row Size</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warehouse</td>
<td>W</td>
<td>100 bytes</td>
<td>W_ID</td>
</tr>
<tr>
<td>District</td>
<td>10W</td>
<td>100 bytes</td>
<td>D_W_ID, D_ID</td>
</tr>
<tr>
<td>Customer</td>
<td>30K*W</td>
<td>600 bytes</td>
<td>C_W_ID, C_ID</td>
</tr>
<tr>
<td>History</td>
<td>30K*W+</td>
<td>50 bytes</td>
<td>n/a</td>
</tr>
<tr>
<td>New Order</td>
<td>9K*W+</td>
<td>20 bytes</td>
<td>O_W_ID, O_D_ID, NO_I_ID</td>
</tr>
<tr>
<td>Order</td>
<td>30K*W+</td>
<td>50 bytes</td>
<td>O_W_ID, O_D_ID, O_I_ID</td>
</tr>
<tr>
<td>Order Line</td>
<td>300K*W+</td>
<td>70 bytes</td>
<td>O_W_ID, O_D_ID, O_ID, OL_NUMBER</td>
</tr>
<tr>
<td>Stock</td>
<td>100K*W</td>
<td>350 bytes</td>
<td>S_W_ID, S_I_ID</td>
</tr>
<tr>
<td>Item</td>
<td>100K</td>
<td>80 bytes</td>
<td>I_ID</td>
</tr>
</tbody>
</table>

2. The Txn Aborts if any Item is not found (1% non-existence is required by the spec).

Locally, we insert Order, New-Order and Order-Line rows for which data was gathered in Step 1. Increment D_NEXT_O_ID locally (possibly blocking later Tx Prepares on the same district). In case the stock level for an item (possibly remote) falls below 10, increase it by 91. (In this case work step 2 would be a new shot.)

On unmodified VoltDB, Figure 16 shows the shots performed for these two steps and eventual Commit in the distributed case. We show when Java runs on the Coordinator (J) and executes the two sequences of SQL statements on VoltDB. The transactional executions of the sequences are listed 111 for work step 1 and 222 for the work step 2.

The exclusive period for VoltDB, the period on a node when no other transaction can execute, runs from Prepare time to Commit time, since no other Txns can start on the
<table>
<thead>
<tr>
<th>Txn Name</th>
<th>% in mix</th>
<th>% distributed</th>
<th>Partitions Involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Order</td>
<td>45%</td>
<td>10%</td>
<td>1-1.5</td>
</tr>
<tr>
<td>Payment</td>
<td>45%</td>
<td>15%</td>
<td>1-2</td>
</tr>
<tr>
<td>Delivery</td>
<td>45%</td>
<td>0%</td>
<td>1</td>
</tr>
<tr>
<td>Order Status RO</td>
<td>4%</td>
<td>0%</td>
<td>1</td>
</tr>
<tr>
<td>Stock Level RO</td>
<td>4%</td>
<td>0%</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Transactions in TPC-C

<table>
<thead>
<tr>
<th>Coordinator: J111</th>
<th>J222</th>
<th>J</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ / \ / \ /</td>
<td>\ /</td>
<td>\ /</td>
<td>\ /</td>
</tr>
<tr>
<td>Remote node:</td>
<td>111</td>
<td>222</td>
<td>C</td>
</tr>
</tbody>
</table>

Figure 16: Timeline for NewOrder on VoltDB

node until after *Commit*. We can see from Figure 16 that the exclusive period is 3RTT on the Coordinator and 2RTT on each non-Coordinator node.

Figure 17 shows the *CVoltDB* execution of the same TPC-C New-Order transaction. The order of operations has been changed from the TPC-C specification to optimize performance for *CVoltDB*, making the remote shot occur first so *Early Fragments Distribution* has an immediate effect. *Early Fragment Distribution* is shown by the J at the Coordinator well ahead of the first transactional execution ("111") at the Coordinator, which is running on the home warehouse partition, using the **Basic Features**. The second
work step occasionally needs remote access for a required update to add 91 to stock levels if they have fallen too low.

In Figure 17, note that the second work step execution, denoted as "J222", shows no remote access. This is possible on CVoltDB based on an extension of the on-Coordinator-only execution feature. This feature allows the stored procedure to indicate in a runtime that the submitted SQL sequence needs no data from remote partitions. In that case, no code is sent out to remote nodes, saving a shot.

The exclusive period on the Coordinator starts with the "111" and ends with the stored procedure execution that is represented by the last J in J222J. After that, another transaction may Prepare by CVoltDB rules. This exclusive period is possibly less than 1RTT, but if the remote node needs to finish earlier transactions before executing this one, may be longer. On the one remote node, the exclusive period extends to the Commit, roughly 1RTT.

**Payment Program.** The Payment program accepts a payment for a customer, possibly belonging to a remote warehouse, increments the customer balance (C_BALANCE) and also reflects this incremental information in the home warehouse W_YTD (Warehouse Year To Date) and District (D_YTD), all increments using Escrow. Then it inserts a record
of this payment to the History table in the home warehouse partition. Payment is the only program to access History. The payment is not associated with a particular order, so the Order table is not accessed. We note that Payment looks up the appropriate customer row by primary key \( (C_{ID}, C_{D_{ID}}, C_{W_{ID}}) \) 40% of the time, and otherwise looks up the row by lastname and firstname: \( C_{LAST} \), sorted by \( C_{FIRST} \) 60% of the time, selecting the median row in position \( \lceil (n/2) \rceil \).

**Delivery (Local Transaction).** The Delivery program delivers one outstanding order per transaction. Outstanding order are ones remaining in the New-Order table, and the logic of Delivery starts by searching for a row in New-Order with a specific \( NO_{W_{ID}} \), a specific \( NO_{D_{ID}} \), and the minimum (oldest undelivered) \( NO_{O_{ID}} \). If no such New-Order row is found, then Delivery for this district is skipped. Else, if a New-Order row is found, it is deleted, and Delivery retrieves the Order row with identical \( O_{W_{ID}} \), \( O_{D_{ID}} \), and \( O_{ID} \), reads the \( O_{C_{ID}} \) (Customer ID) and writes a value to the null \( O_{CARRIER_{ID}} \) (to reflect the "Carrier" performing delivery). Then Delivery retrieves all the Order-Line rows for the order and for each one sets \( OL_{DELIVERY_{D}} \) (Delivery Date) to the current system time, and aggregates a sum of all \( OL_{AMOUNT} \) (dollar charges). Finally, the Customer row for the order is retrieved and the Escrow column \( C_{BALANCE} \) is incremented by \( OL_{AMOUNT} \).

**Order-Status (Local Transaction).** The Order-Status program executes a read-only program to query the status of a customer’s last order, returning information from all Order-Line rows for this order: \( OL_{I_{ID}} \) (Item ID), \( OL_{DELIVERY_{D}} \) (delivery date), and others. To access the appropriate Order-Line rows, Order-Status starts by retrieving a Customer row using a Where clause with primary key \( (C_{ID}, C_{D_{ID}}, C_{W_{ID}}) \) 40% of
the time and C_LAST, sorted by C_FIRST 60% of the time, selecting the median row—in position ⌈(n/2)⌉. C_BALANCE, an escrow quantity, is read in this transaction but there are 30,000 customers involved so pending updates to C_BALANCE are uncommon. Then the row in the Order table with matching O_C_ID, O_D_ID, O_W_ID and largest existing O_ID (latest order) is retrieved, a few columns read, and the Order-Line rows with matching OL_O_ID retrieved, as explained above.

**Stock-Level (Local Transaction).** The Stock-Level program executes as a read-only Tx to determine which of the items ordered by the last twenty orders in a given warehouse and district have fallen below a specified threshold value. The Stock-Level program reads D_NEXT_O_ID to find the next order number to be assigned for this district, then accesses all Order-Line rows with OL_O_ID in range D_NEXT_O_ID-20 to D_NEXT_O_ID to learn OL_I_ID (all item IDs of the last twenty orders), then checks the quantity on hand in the STOCK table, S_QUANTITY, for this S_W_ID and S_I_ID to test it’s under the given threshold.

### 5.2.2.2 Scale Up Experiments

Just like what we have done with TPC-B, we show the size of cluster and the configuration of how many logical sites per host affect the performance.

Figure 18 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 (over 1.5 per CPU) provide better performance means the system is encountering significant delays on each node, although
Figure 18: TPC-C TPS by nodes/host, 4 hosts

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 Tx takes over the entire cluster.

Figure 19 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 many of the distributed transactions are multi-shot. Still, we are seeing significant improvement in performance, in a fairly realistic setting.
5.3 Summary

This chapter presented experimental results obtained from running TPC-B [TPC94] and TPC-C [TPC10] under four different configuration: Original VoltDB, Basic Features, Consecutive Prepares Execution, and full Escrow support. Basic Features improves the performance a lot in an easy way; Consecutive Prepares Execution is the necessary step to support Escrow which leads to another performance improvement when there is any distributed transactions.

The next chapter, Chapter 6, describes areas of future work and concludes the dissertation.
CHAPTER 6

FUTURE WORK AND CONCLUSION

This dissertation began with the discussion of transaction serial execution on MMDBs instead of complicated and costly traditional concurrency control mechanisms, such as locking and snapshot. With Serial Execution without the disk I/O bottleneck, data access is almost immediate, and most transactions can be run from the beginning to the end without interruption, especially for OLTP. However, Serial Execution obviously leads to poor performance for distributed transactions.

This dissertation proposes the design and implementation of CVoltDB, a modification of VoltDB, a database system with Local Serial Execution. CVoltDB preserves the advantage of VoltDB for Local Transactions, but provides much faster processing speed for Distributed Transactions by adding concurrency, that is by allowing execution of useful work during network stalls.

CVoltDB mainly consists of two parts, one is the addition of minimal locking to support concurrent transactions on each single node, and the other is the special handling aiming to speed up OLTP incremental operations, adapted from Escrow Method introduced in 1986 [ON86].

Before adding the locking back to the MMDB systems, some necessary and effective approaches are designed as well. The Uses Only Needed Partitions and Early Fragment...
Distribution are both non-transactional, and didn’t change the transaction’s Serial Execution of transactional work. Write locks are added in Consecutive Prepares Execution, but still no Read locks. Read locks are not needed since Prepares are executed one at a time on a node, and replica consistency follows from execution in Txn-ID order.

An Ordered Escrow Method for OLTP MMDB systems follows the lead of its original in [ON86], and is proved to maintain serializability and replica consistency. Moreover, Escrow Read is added in Ordered Escrow Method, and no deadlock detection is needed.

There are three important queue data structures in CVoltDB, and each transaction’s lifetime is comprised of five states. The state of the transaction can be mainly determined by which queue it current resides. These data structures are essential in the implementation of Chapter 4. Algorithms in Chapter 4 fall in two categories. One is logical execution of the transaction, happening on the system layer written in Java, which is called as ”front end” in VoltDB. The other relates to table data operations, occurring within the data engine, a lower layer written in C++.

The performance experiments are presented in Chapter 5. We conclude that CVoltDB performs similarly with VoltDB for a pure Local Transaction workload, but shows an unbeatable advantage if there exist any Distributed Transactions because concurrent execution is supported in CVoltDB and the Ordered Escrow Method solves the hot-spot data problem in OLTP systems.

However, besides the four strategies introduced in this dissertation, there are a few more strategies can be used to further CVoltDB or increase our understanding of its execution.

1. **What-if Query Support.** Blocking occurs since later transaction is waiting the execution of a previous one. The previous transaction can either Commit or Abort,
so if What-If query is supported, later transaction can execute on both scenario and later report the result based on the actual situation.

2. Auto-Increment Column. This is a special case of reducing number of shots in a transaction, when the value of the column in a new inserted row doesn’t depend on the results of previous query, but depends on current value of the column in some row. The network delay is huge between shots, and right now, none of the four strategies could eliminate any delay between shots. It combines a Read followed by an Insert as an atomic operation, and could reduce the number of shots.

3. Analytic cost model. An analytic analysis would be useful to research alternative system designs. We have worked on analysis to measure the benefits brought from the four strategies, but the synchronization between coordinator and participants turned to be very complex to characterize, so we are leaving this for future work.

In conclusion, we recognize that there are many areas that CVoltDB would benefit from future research, but we believe the concurrency scheme and strategies in CVoltDB has the potential to become attractive for OLTP MMDBs.


