Determinism in High-Throughput Distributed Databases

John Hugg

VOLTDB

2/1/2013
Run the Same SQL Against Two Databases
CREATE TABLE BDAYS (ID INTEGER, D DATE);
INSERT INTO BDAYS (1, '1943-10-11');
INSERT INTO BDAYS (2, '1976-08-04');
UPDATE BDAYS SET D = '1944-08-17' WHERE ID = 2;
SELECT * FROM BDAYS ORDER BY D LIMIT 1
INSERT INTO BDAYS (1, '1943-10-11');
INSERT INTO BDAYS (2, '1976-08-04');
UPDATE BDAYS SET D = '1944-08-17' WHERE ID = 2;
SELECT * FROM BDAYS ORDER BY D LIMIT 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1943-10-11</td>
</tr>
<tr>
<td>2</td>
<td>1944-08-17</td>
</tr>
</tbody>
</table>
INSERT INTO BDAYS (3, CURDATE());
INSERT INTO BDAYS (4, CURDATE());
INSERT INTO BDAYS (3, CURDATE());
INSERT INTO BDAYS (4, CURDATE());
Sources of Non-Determinism
(aka “Pure Evil”)

[Image of a scene from a TV show with two characters in shock, one holding an empty oven]
• Wall Clock Time
• Randomness

insert into table
values (rand(), now());
Depending on Storage Engine Design:

Tuple Order
Depending on Storage Engine Design:

Tuple Order
DELETE FROM BDAYS WHERE ID IN (SELECT ID FROM BDAYS ORDER BY D LIMIT 1);

<table>
<thead>
<tr>
<th></th>
<th>BDAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1943-10-11</td>
</tr>
<tr>
<td>2</td>
<td>1944-08-17</td>
</tr>
<tr>
<td>3</td>
<td>2013-01-31</td>
</tr>
<tr>
<td>4</td>
<td>2013-01-31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>BDAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1943-10-11</td>
</tr>
<tr>
<td>2</td>
<td>1944-08-17</td>
</tr>
<tr>
<td>3</td>
<td>2013-01-31</td>
</tr>
<tr>
<td>4</td>
<td>2013-01-31</td>
</tr>
</tbody>
</table>
External Systems

Begin Transaction
Read my id
Ping the NOAA for the weather
Write (id, wind speed) to db
Commit Transaction
Why?
Logical Logging
(Replication)

• Log operations in a fixed order
• Load a transactional snapshot and replay a log
• Truncate the log on snapshot for disk
• Stream the log continuously for replication
Logical Logging Done Right

- Deterministic Operations in the Log
- Serializable Isolation
  - Serializable is easy for basic KV/Document stores that have a serializable ordering and deterministic functions
  - Serializable is harder for RDBMS with multi-statement, multi-row transactions
So...

Asynchronous Replication/Logging

Logical or Binary Log?
# Logging Throwdown

<table>
<thead>
<tr>
<th></th>
<th>Logical</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Efficiency</td>
<td>When modified data is larger than the command</td>
<td>When modified data is smaller than the command</td>
</tr>
<tr>
<td>Execution Speed</td>
<td>Less overhead</td>
<td>Faster to recover</td>
</tr>
<tr>
<td>Latency</td>
<td>Can log anytime</td>
<td>Must log after completion</td>
</tr>
<tr>
<td>Determinism</td>
<td>Yes please</td>
<td>Meh</td>
</tr>
</tbody>
</table>
Synchronous Replication/Logging

Logical Wins!
Why Logical Log for Synchronous Replication?

- Latency - can start logging when the operation is ordered, rather than when it completes.
- Bounded size - If the operations fit on a gigabit connections as they come in, you can log them at $\leq$ gigabit rates.
- Determinism required though
Logical Log-Based Replication & Deterministic Stored Procedures

Tristan & Isolde
Romeo & Juliet
Stonebraker & Ellison
Knead and toss dough
Spread sauce on dough
Spread cheese on sauce
Sprinkle oil and oregano on cheese
Add sliced ham on top
Add olives on top
Heat at 450° until cheese melts and crust turns slightly brown
Put in pizza box
Look up the URL and the USER
Find all potential ads, sorted by bid price for this user/url combo
Filter out ads that don’t have sufficient balance or have met their impression or click-through budget
Pick an ad to show
Debit the client’s ad budget

displayAd(user, url);

Meet H-Store Researchers in Cambridge, MA

“We’re ready to party!”
Logical Log-Based Replication & Deterministic Stored Procedures

How VoltDB can be fast and safe at the same time
Connect Four
THE VERTICAL FOUR-IN-A-ROW CHECKERS GAME
3 SP calls arrive → 5 SP calls are ordered → 5 SP calls are executed → 5 SP responses collected → 3 SP responses sent

2 SP calls arrive → 5 SP calls are executed → 5 SP responses collected → 2 SP responses sent
So what?

- 3-node cluster with 3-copies of all data
- Run >100k transactions per second
- Synchronously
- <1ms latency pertxn
- Add a few ms / txn to get sync disk log
- Add more logic + SQL to each stored proc with reckless abandon
Calvin: Fast Distributed Transactions for Partitioned Database Systems

Alexander Thomson
Yale University
thomson@cs.yale.edu

Thaddeus Diamond
Yale University
diamond@cs.yale.edu

Shu-Chun Weng
Yale University
scweng@cs.yale.edu

Kun Ren
Yale University
kun@cs.yale.edu

Philip Shao
Yale University
shao-philip@cs.yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Many distributed storage systems achieve high data access through-put by partitioning and replication. Each system has its own advantages and tradeoffs. In order to achieve high scalability, how-ever, today’s systems generally reduce transactional support, allowing single transactions from spanning multiple partitions. Calvin is a practical transaction scheduling and data replication layer that uses a deterministic ordering parameter to significantly reduce the normally prohibitive contention costs associated with distributed transactions. Unlike previous deterministic database system proto-types, Calvin supports disk-based storage, scales near-linearly, on a choice of commodity machines, and has no single point of fail-ure. By replicating transaction inputs rather than effects, Calvin is also able to support multiple concurrency levels—enabling pass-ive-based strong consistency across geographically distant replicas—at no cost to transactional throughput.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed databases; H.2.4 [Database Management]: Systems—concurrency, distributed database, transaction processing

General Terms
Algorithms, Design, Performance, Reliability

Keywords
deterministic, distributed database systems, replication, transaction processing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. 10.1145/2080406.2080427

1. BACKGROUND AND INTRODUCTION

One of several current trends in distributed database system de-sign is a move away from supporting traditional ACID database transactions. Some systems, such as Amazon’s Dynamo [13], Mon-goDB [34], Cassandra [17], and Casandra [17] provide no transac-tional support whatsoever. Others provide only limited transactional support. In contrast to single-row transactional updates (e.g. Bigtable [21]), these systems support ACID transactions whose accesses are limited to small subsets of a database (e.g., Azure [9], Maganize [7], and the Oracle NoSQL Database [25]). The primary reason that each of these systems does not support full ACID transactions is to provide linear out-put scalability. Other systems (e.g. Voldemort [18], HBase) support full ACID but either limit concurrent transaction execution, or prevent transactions from spanning multiple partitions.

Reducing transactional support greatly simplifies the task of build-ing linearly scalable distributed storage solutions that are designed to serve “embarrassingly partitionable” applications. For applica-tions that are not easily partitionable, however, the benefits of en-suring availability and isolation is generally left to the application programmer, resulting in increased code complexity, slower applica-tion development, and low-performance client-side transaction scheduling.

Calvin is designed to run alongside a non-transactional storage system, transmogrifying it into a shared-nothing (near-)linearly scal-able database system that provides high availability and full ACID transactions. These transactions can potentially span multiple parti-tions spread across the shared-nothing cluster. Calvin accomplishes this by providing a layer above the storage system that handles the scheduling of distributed transactions, as well as replication and network communication in the system. The key technical feature that allows for scalability in the face of distributed transactions is a deterministic locking mechanism that enables the elimination of distributed commit protocols.

In this paper we use the term “high availability” in the common colloquial sense found in the database community where a database is highly available if it can fail over to an active replica on the fly if it can fail over to an active replica on the fly if an active replica is non-functional. Calvin achieves this by ensuring that even minority replicas remain available during a network partition.
When Determinism Goes Wrong...
A Poisoned Log
How do you detect non-determinism?

- VoltDB hashes any DML run in a transaction and stores that hash with the results.
- Could also hash changed data or a redo-log
Crash Recovery

Let the user decide what to do

Snapshot

Replay to here

Let the user decide what to do
Async Replication

Master

When detected, kill replica state

Replica

Send full snapshot
Sync Replication

“VoltDB Style”

When detected,
Stop cluster
Snapshot both replicas
Let user sort it out
Successfully created voter.jar
Includes schema: ddl.sql

[MP][RW] CONTESTANTS.insert
   INSERT INTO CONTESTANTS VALUES (?, ?);

[MP][RO][Seq] Results
   [Seq] SELECT a.contestant_name AS contestant_name, a.contestant_number AS conte...

[SP][RW] VOTES.insert
   INSERT INTO VOTES VALUES (?, ?, ?);

[SP][RW][Seq] Vote
   SELECT contestant_number FROM contestants WHERE contestant_number = ?;
   SELECT state FROM area_code_state WHERE area_code = ?;
   SELECT num_votes FROM v_votes_by_phone_number WHERE phone_number = ?;
   INSERT INTO votes (phone_number, state, contestant_number) VALUES (?, ?, ?);
   [Seq] [NDO] SELECT * from votes;

-------------------------------------------------------------
NON-DETERMINISM WARNING:
The procedures listed below contain non-deterministic Procs
   voter.procedures.Vote

Using the output of these queries as input to write queries can result in differences between partitions at runtime, forcing VoltDB to shut down the cluster. Review the compiler messages above to identify the offending SQL statements (marked as "[NDO] or [NDC]"). Add a unique index to the schema or an explicit ORDER BY clause to the query to make these queries deterministic.
THANK YOU!
@johnhugg
jhugg@voltdb.com
http://voltdb.com