Concurrency Control In Distributed Main Memory Database Systems

Justin A. DeBrabant
debrabant@cs.brown.edu

BROWN

Concurrency control

- Goal:
  - maintain consistent state of data
  - ensure query results are correct
- The Gold Standard: ACID Properties
  - atomicity – "all or nothing"
  - consistency – no constraints violated
  - isolation – transactions don't interfere
  - durability – persist through crashes

BROWN

Why?

- Let's just keep it simple...
  - serial execution of all transactions
  - e.g. T1, T2, T3
  - simple, but boring and *slow*
- The Real World:
  - interleave transactions to improve throughput
    - …crazy stuff starts to happen

Traditional Techniques

- Locking
  - lock data before reads/writes
  - provides isolation and consistency
  - 2-phase locking
    - phase 1: acquire all necessary locks
    - phase 2: release locks (no new locks acquired)
    - locks: shared and exclusive
- Logging
  - used for recovery
  - provides atomicity and durability
  - write-ahead logging
    - all modifications are written to a log before they are applied

How about in parallel?

- many of the same concerns, but must also worry about committing multi-node transactions
- distributed locking and deadlock detection can be expensive (network costs are high)
- 2-phase commit
  – single coordinator, several workers
  – phase 1: voting
    - each worker votes “yes” or “no”
  – phase 2: commit or abort
    - consider all votes, notify workers of result

The Issue

- these techniques are very general purpose
  - "one size fits all"
  - databases are moving away from this
- By making assumptions about the system/ workload, can we do better?
  - YES!
  - keeps things interesting (and us employed)

Paper 1

- *Low Overhead Concurrency Control for Partitioned Main Memory Databases*
  - Evan Jones, Daniel Abadi, Sam Madden
  - SIGMOD '10

BROWN

Overview

- Contribution:
  - several concurrency control schemes for distributed main-memory databases
- Strategy
  - Take advantage of network stalls resulting from multi-partition transaction coordination
  - don’t want to (significantly) hurt performance of single-partition transactions
    - probably the majority

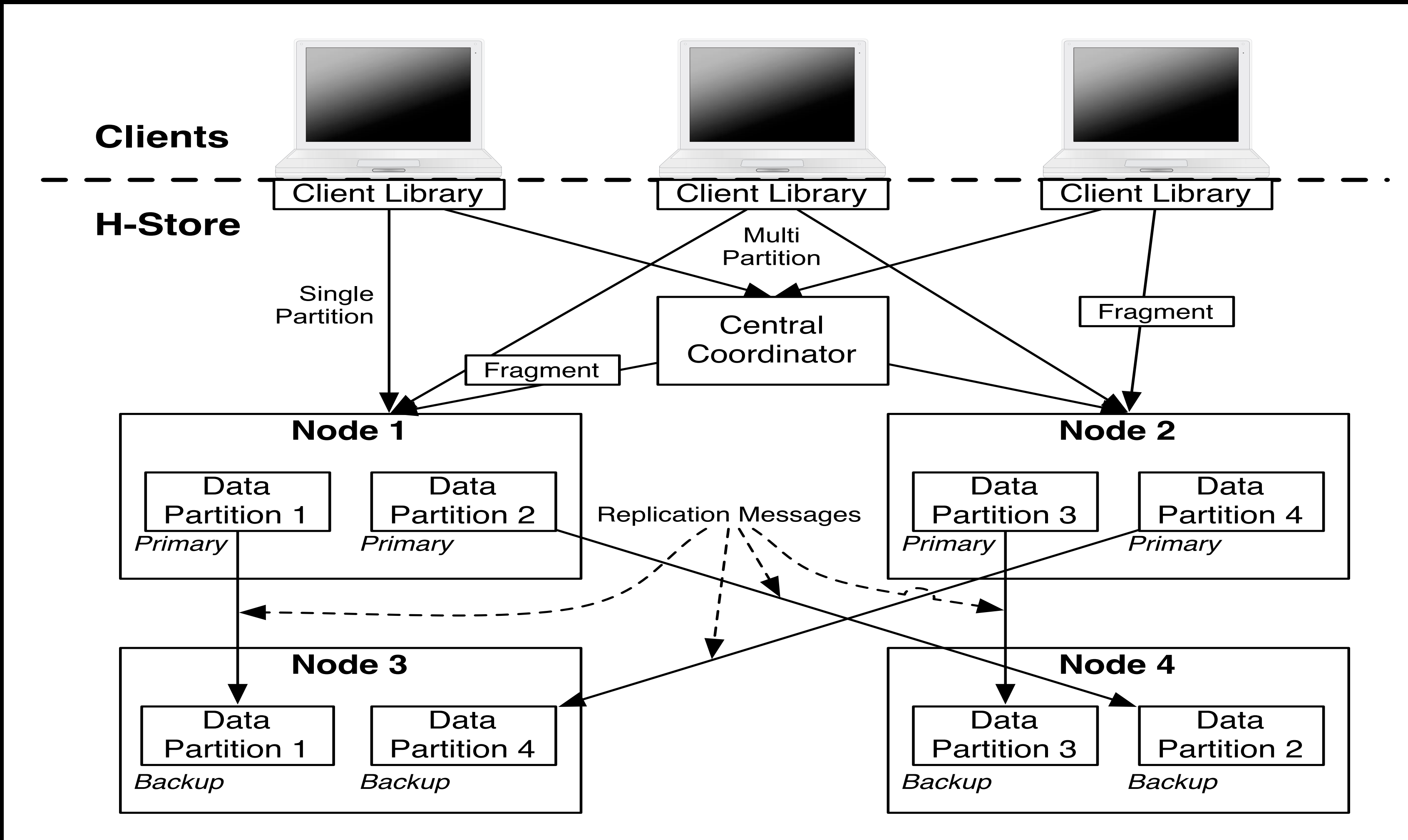
BROWN

System Model

- based on H-Store
- partition data to multiple machines
  - all data is kept in memory
  - single execution thread per partition
- central coordinator that coordinates
  - assumed to be single coordinator in this paper
    - multi-coordinator version more difficult

BROWN

System Model (cont'd)



Transaction Types

- Single Partition Transactions
  - client forwards request directly to primary partition
  - primary partition forwards request to backups
- Multi-Partition Transactions
  - client forwards request to coordinator
  - transaction divided into *fragments* and forwards them to the appropriate transactions
  - coordinator uses undo buffer and 2PC
  - network stalls can occur as a partition waits for other partitions for data
    - network stalls twice as long as average transaction length

Concurrency Control Schemes

- Blocking
  - queue all incoming transactions during network stalls
  - simple, safe, slow
- Speculative Execution
  - speculatively execute queued transactions during network stalls
- Locking
  - acquire read/write locks on all data

Blocking

- for each multi-partitioned transaction, block until it completes
- other fragments in the blocking transaction are processed in order
- all other transactions are queued
  - executed after the blocking transaction has completed all fragments

BROWN

Speculative Execution

- speculatively execute queued transactions during network stalls
- must keep undo logs to roll back speculatively executed transaction if transaction causing stall aborts
- if transaction causing stall commits, speculatively executed transaction immediately commit
- two cases:
  - single partition transactions
  - multi-partition transactions

Speculating Single Partitions

- wait for last fragment of multi-partition transaction to execute
- begin executing transactions from unexecuted queue and add to uncommitted queue
- results must be buffered and cannot be exposed until they are known to be correct

Speculating Multi-Partitions

- assumes that 2 speculative transactions share the same coordinator
  - simple in the single coordinator case
- single coordinator tracks dependencies and manages all commits/aborts
  - must cascade aborts if transaction failure
- best for simple, single-fraction per partition transactions
  - e.g. distributed reads

BROWN

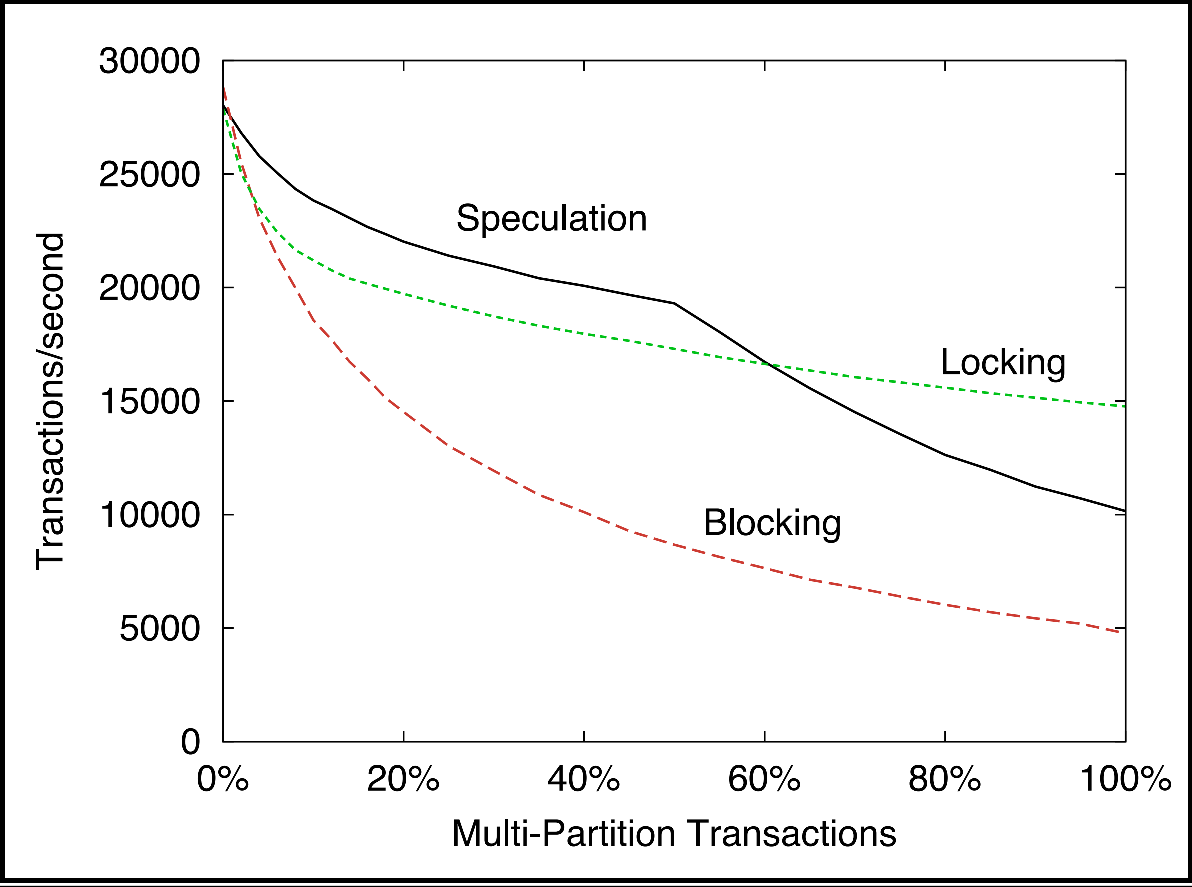
Locking

- locks allow individual partitions to execute and commit non-conflicting transactions during network stalls
- problem: overhead of obtaining locks
- optimization: only require locks when a multi-partition transaction is active
- must do local/distributed deadlock
  - local: cycle detection
  - distributed: timeouts

Microbenchmark Evaluation

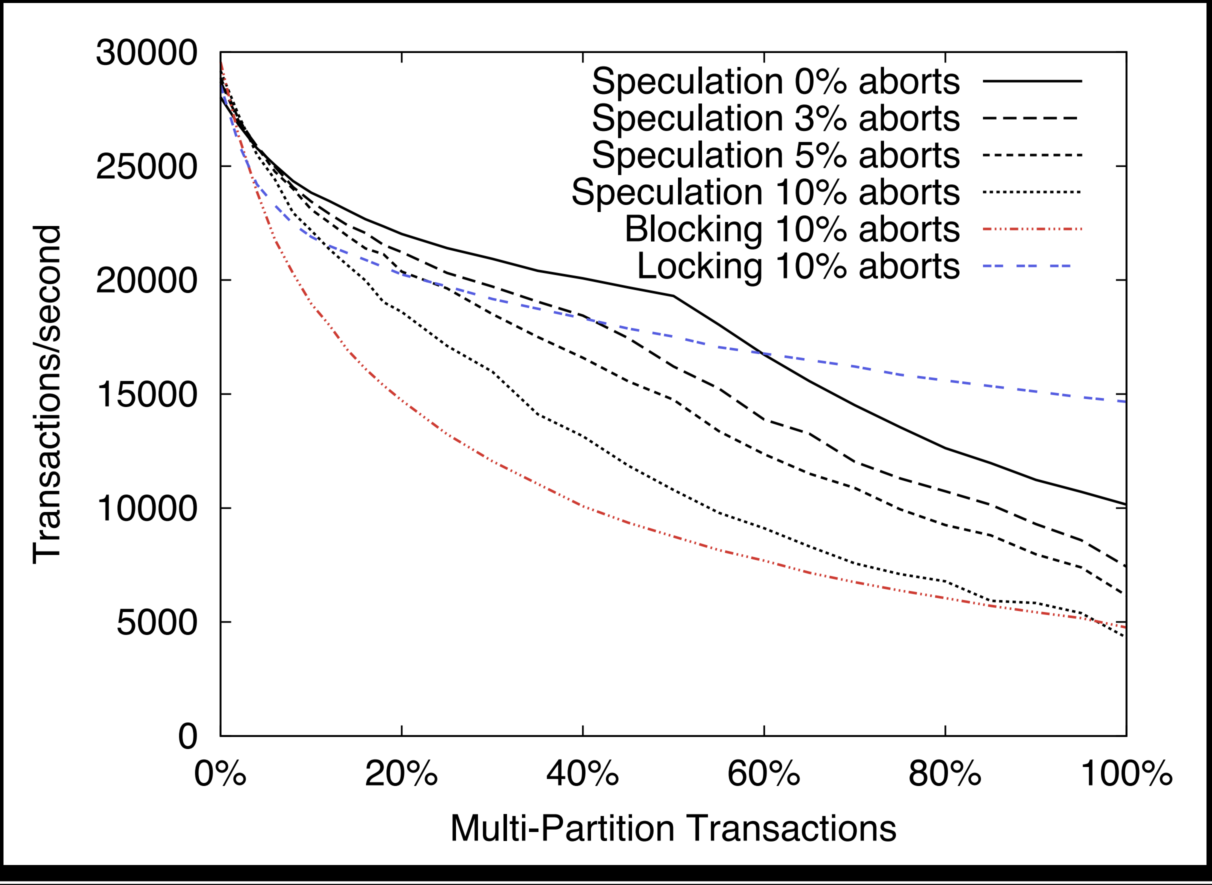
- Simple key/value store
  - keys/values arbitrary strings
- simply for analysis of techniques, not representative of real-world workload

Microbenchmark Evaluation



BROWN

Microbenchmark Evaluation

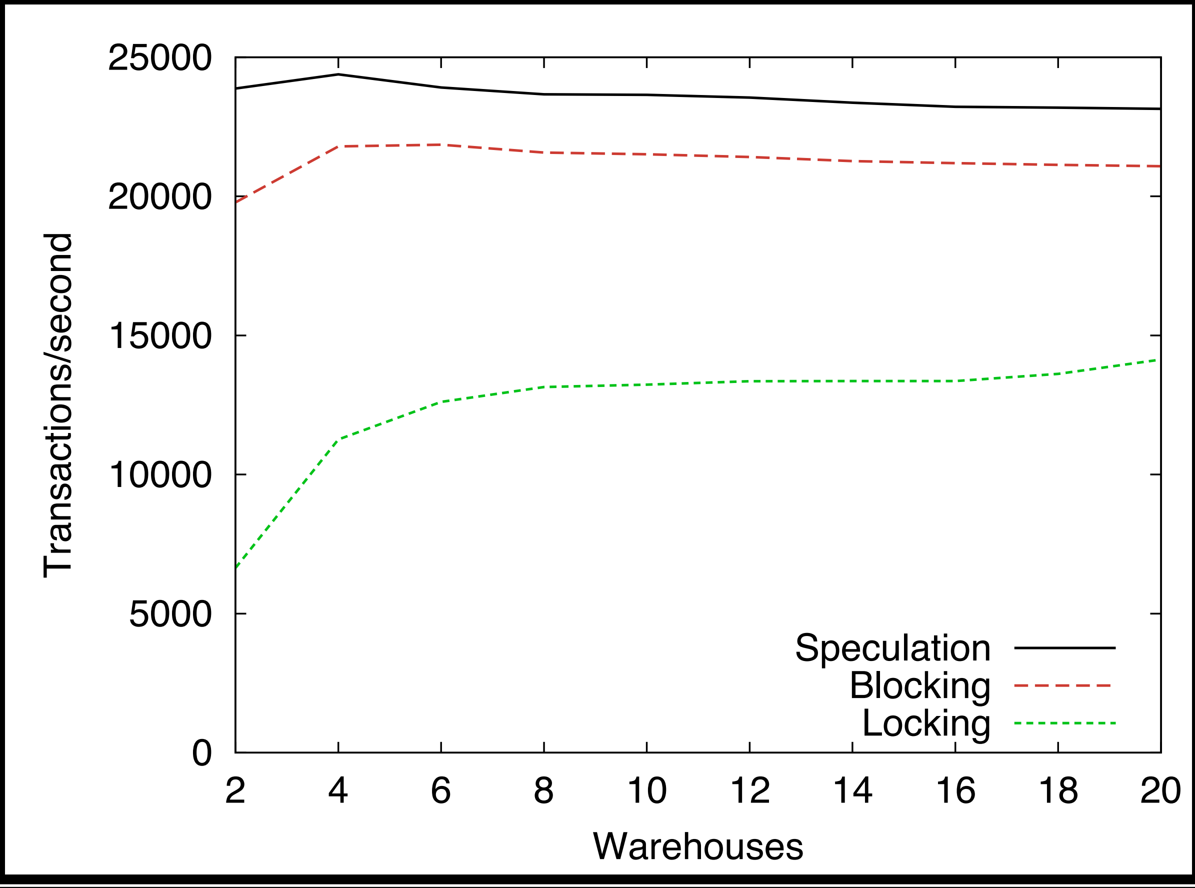


BROWN

TPC-C Evaluation

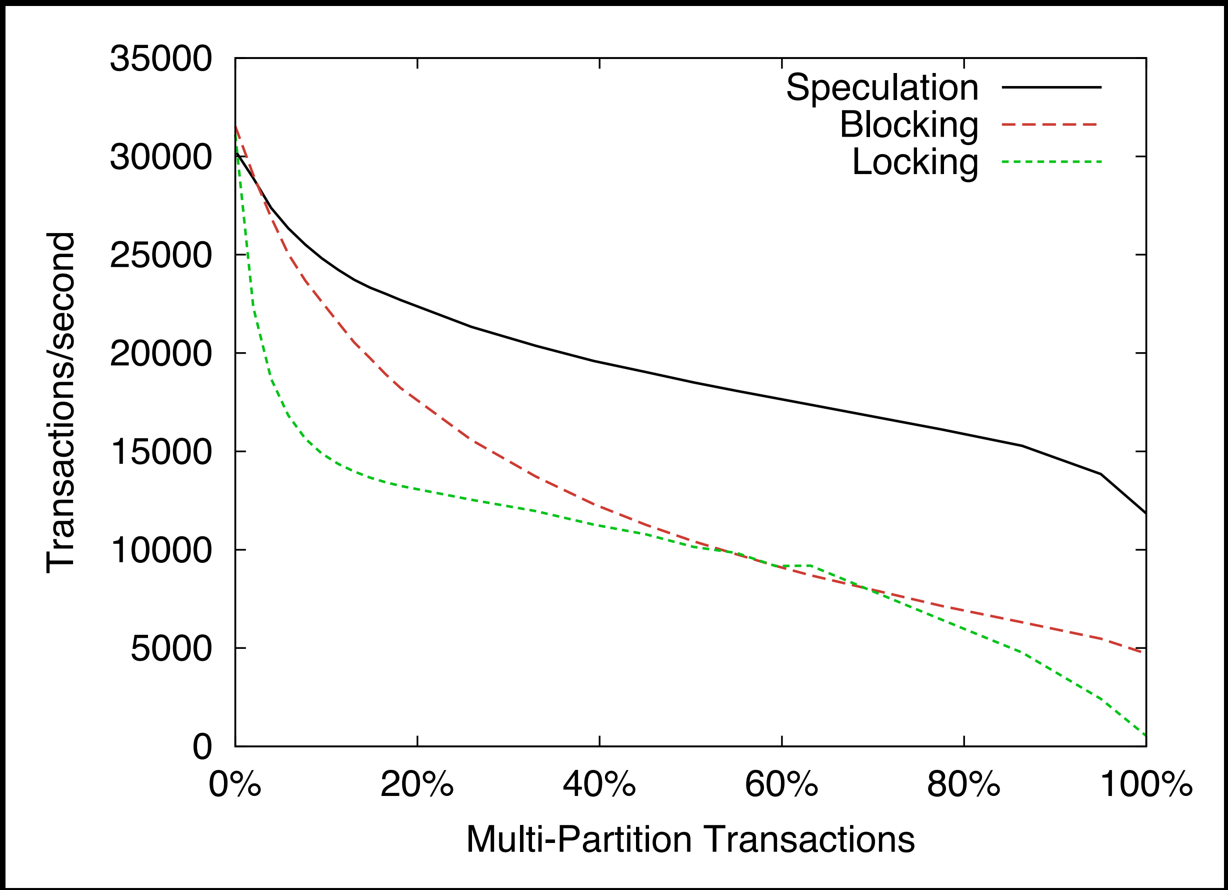
- TPC-C
  - common OLTP benchmark
  - simulates creating/placing orders at warehouses
- This benchmark is a modified version of TPC-C

TPC-C Evaluation



BROWN

TPC-C Evaluation (100% New Order)



BROWN

Evaluation Summary

		Few Aborts		Many Aborts	
		Few Conflicts	*Many Conflicts*	*Few Conflicts*	*Many Conflicts*
Few multi-round xactions	*Many multi-partition xactions*	Speculation	Speculation	Locking	Locking or Speculation
	Few multi-partition xactions	Speculation	Speculation	Blocking or Locking	Blocking
Many multi-round xactions		Locking	Locking	Locking	Locking

Paper 2

- *The Case for Determinism in Database Systems*
  - Alexander Thompson, Daniel Abadi
  - VLDB 2010

Overview

- Presents a deterministic database prototype
  - argues that in the age of memory-based OLTP systems (think H-Store), clogging due to disk waits will be a minimum (or nonexistant)
  - allows for easier maintenance of database replicas

Nondeterminism in DBMSs

- transactions are executed in parallel
- most databases guarantee consistency for *some* serial order of transaction execution
  - which?...depends on a lot of factors
  - key is that it is not necessarily the order in which transactions arrive in the system

Drawbacks to Nondeterminism

- Replication
  - 2 systems with same state and given same queries could have different final states
    - defeats the idea of "replica"
- Horizontal Scalability
  - partitions have to perform costly distributed commit protocols (2PC)

Why Determinism?

- nondeterminism is particularly useful for systems with long delays (disk, network, deadlocks, …)
  – less likely in main memory OLTP systems
  – at some point, the drawbacks of nondeterminism outweigh the potential benefits

How to make it deterministic?

- all incoming queries are passed to a preprocessor
  - non-deterministic work is done in advance
    - results are passed as transaction arguments
  - all transactions are ordered
  - transaction requests are written to disk
  - requests are sent to all database replicas

BROWN

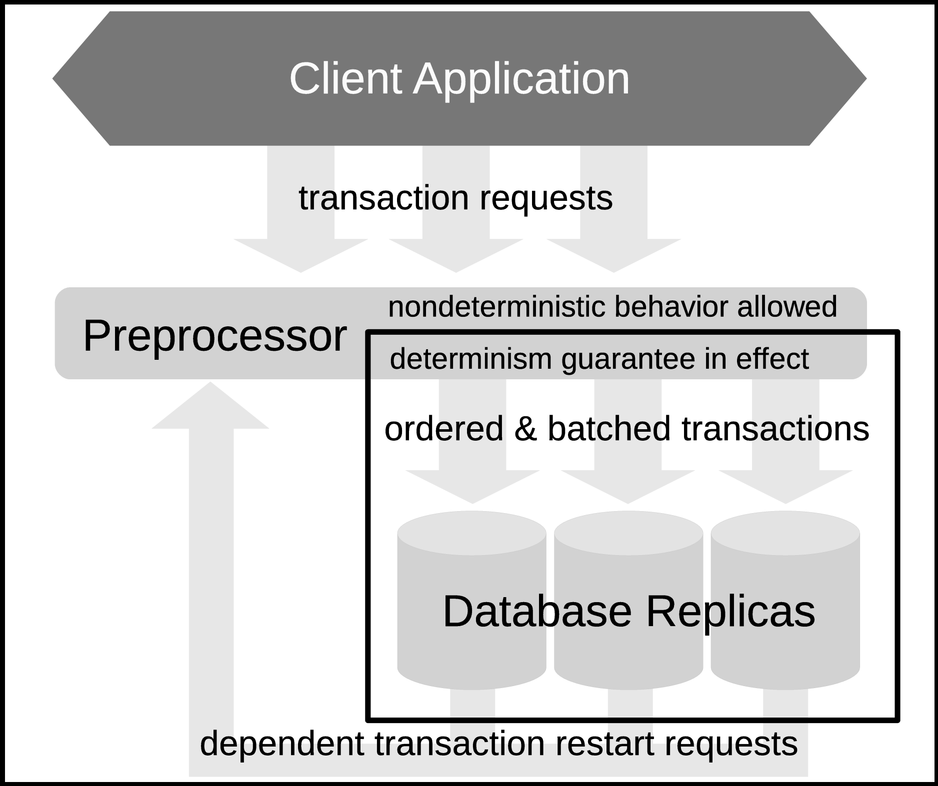
A small issue…

- What about transactions with operations that depend on results from a previous operation?
  - y ← read(x), write(y)
    - x is the records primary key
- This transaction cannot request all of its locks until it knows the value of y
  - …probably a bad idea to lock y's entire table

Dealing with "difficult" transactions

- Decompose the transaction into multiple transactions
  - all but the last are simply to discover the full read/write set of the original transaction
  - each transaction is dependent on the previous ones
- Execute the decomposed transactions 1 at a time, waiting for results of previous

System Architecture



Evaluation

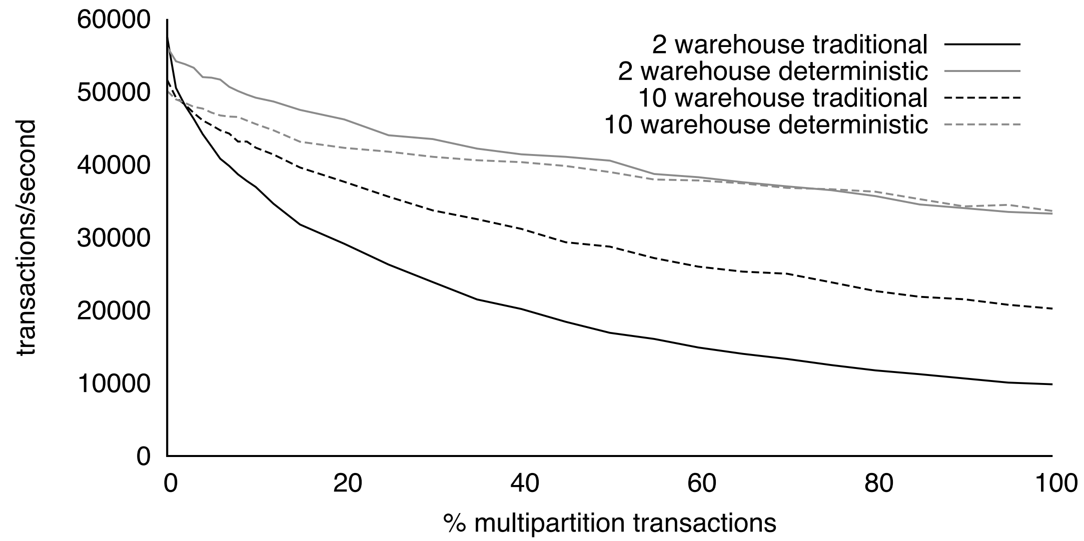


Figure 3: Deterministic vs. traditional throughput of TPC-C (100% New Order) workload, varying frequency of multipartition transactions.

Evaluation Summary

- In systems/workloads where stalls are sparse, determinism can be desirable
- Determinism has huge performance costs in systems with large stalls
- bottom line: good in some systems, but not everywhere

Paper 3

- *An Almost-Serial Protocol for Transaction Execution in Main-Memory Database Systems*
  - Stephen Blott, Henry Korth
  - VLDB 2002

BROWN

Overview

- In main memory databases, there is a lot of overhead in locking
- naïve approaches that lock the entire database suffer during stalls when logs are written to disk
- main idea: maintain timestamps and allow non-conflicting transaction to execute during disk stalls

BROWN

Timestamp Protocol

- Let transaction *T1* be a write on *x*
- Before *T1* writes anything, issue new timestamp TS(*T1*) s.t. TS(*T1*) is greater than any other timestamp
- When *x* is written, WTS(*d)* is set to TS(*T1*)
- When any transaction *T2* reads *d*, TS(*T2)* is set to max(TS(*T2)*, WTS(*d)*)

BROWN

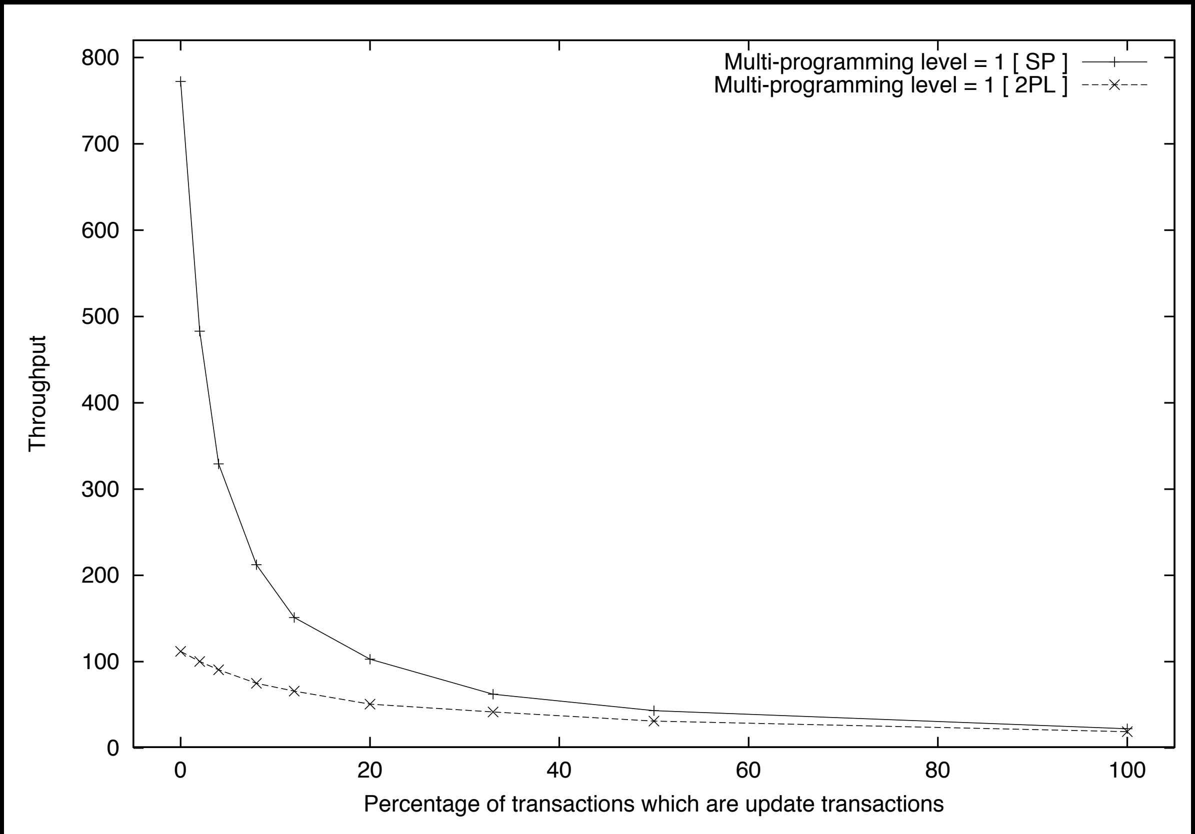
Transaction Result

- If *T* is an update transaction:
  - TS(*T*) is a new timestamp, higher than any other
- If *T* is a read-only transaction:
  - TS(*T*) is the timestamp of the most recent transaction from which *T* reads
- For data item *x:*
  - WTS(*x*) is the timestamp of the most recent transaction that wrote into *x*

The Mutex Array

- an “infinite” array of mutexes, 1 per timestamp
- Commit Protocol:
  - Update
    - *T* acquires database mutex, executes
    - When *T* wants to commit, acquire A[TS(*T*)], prior to releasing database mutex
    - *T* releases A[TS(*T*)] after receiving ACK that its commit record has been written to disk
  - Read-Only
    - release database mutex and acquire A[TS(*T*)]
    - immediately release A[TS(*T*)], commit

Evaluation



BROWN

General Conclusions

- As we make assumptions about query workload and/or database architecture, old techniques need to be revisited
- No silver bullet for concurrency/ determinism questions
  - tradeoffs will depend largely on what is important to the user of the system

Questions?