Data Management in the Cloud

Ryan Stern

stern@cs.colostate.edu

CS 655: Advanced Topics in Distributed Systems

Department of Computer Science

Colorado State University

Outline

- Today
  - Microsoft Cloud SQL Server
  - Pig and Pig Latin
- Next Week
  - H-Store

Adapting Microsoft SQL Server for Cloud Computing

Cloud SQL Server

- A relational database system
- Backed by Microsoft SQL Server
- Designed to scale out to cloud computing workloads

Shared-nothing Architecture

- A distributed computing architecture
- Each node in the system is independent of the others
- Each node in the system is self-sufficient
- The nodes do not share disk storage

Data Model

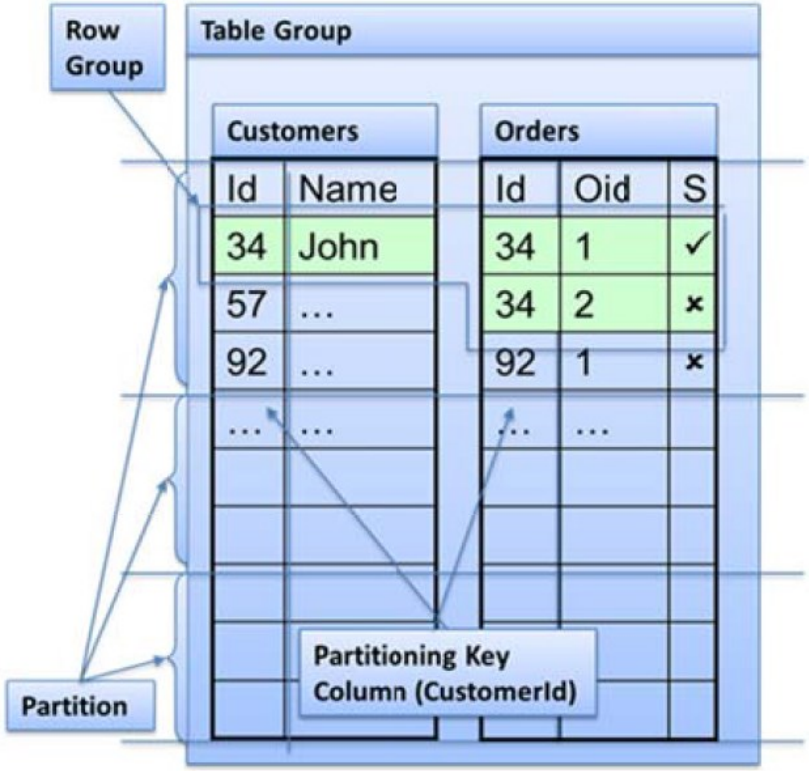
- Provides relational access to large datasets
- A logical database is called a table group
  - Table groups may be keyless or keyed
- Want to avoid running transactions across multiple nodes
  - Avoid the need for a two-phase commit
  - Participants would block in the event of a failure
  - Introduces messages with relatively random distribution patterns

Keyed vs. Keyless Tables

- Table groups can be keyed or keyless
- A keyed table group has an additional column, a partition key
  - All rows with the same partition key are part of the same row group
  - Transactions are limited to operations on only one row group
- A keyless table group does not have a partition key
  - Transactions can operate on any row in the table group
  - Tradeoff: The entire table group must fit on a single node

Data Partitioning

- Partition size is limited by the capacity of a single server
- Partitioning depends on the table type



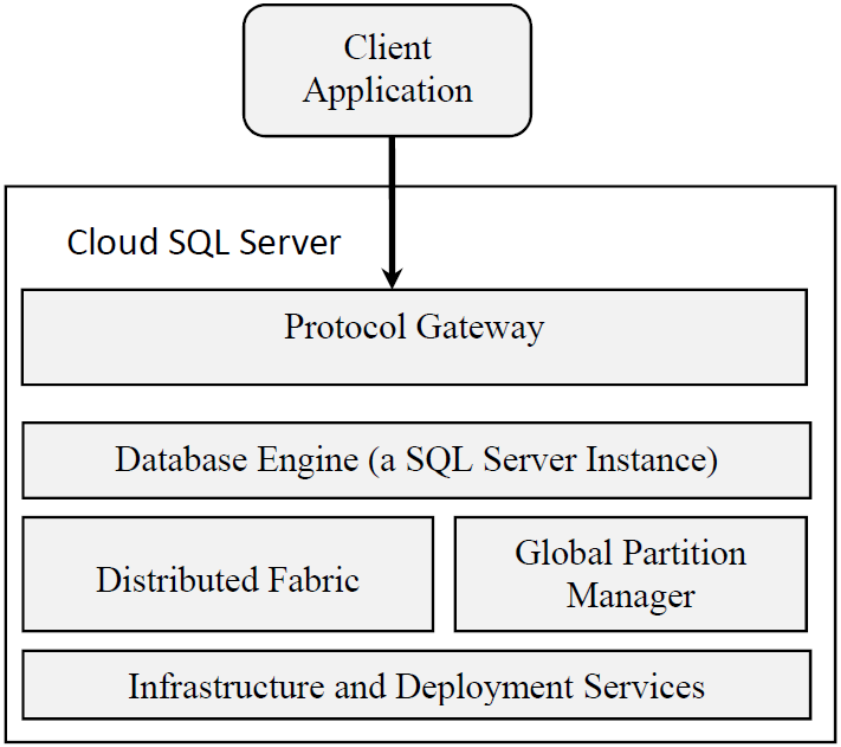
Partition Replication

- Partitions are replicated for high availability
  - A single partition is a failover unit
- No two replicas are placed in the same failure domain
  - Meaning not on the same rack or behind the same switch
- There is always one primary replica of a partition
  - Responsible for processing queries, updates, and data definition ops
  - Ships updates to secondary partitions
  - Similar to BigTable

Partition Load Balancing

- Just balancing the number of primary and secondary partitions per node might not balance the load
- Dynamic rebalancing is possible by demoting the primary
  - This is only allowed if no transactions are in progress on that node
  - A secondary node is then promoted to primary
  - Managed by a global partition manager
- Keyed table groups can be partitioned dynamically
  - When the data exceeds the allowed size or when the node is loaded
  - Partitions are split using the existing replicas, no data movement
  - The secondary will become the primary for the subset of rows

System Architecture



System Architecture

- SQL Server Instance
  - Manages many individual databases
  - Different sub-databases are isolated
  - Saves memory on internal structures
  - Shares a common transaction log, improving performance

- Distributed Fabric
  - Runs on every node with an SQL server instance
  - Maintains the up/down status of servers
  - Detects server failures and recoveries
  - Performs leadership elections for various roles

System Architecture

- Global Partition Manager
  - Highly available
  - Knows the key range for each partition
  - Knows the location of all replicas
  - Knows the primary state and history
  - Decides whether to refresh, replace, or discard replicas
- Protocol Gateway
  - Understands the native wire protocol of SQL Server
  - Accepts inbound database connections and binds it to the primary replica
  - Masks some failures from clients
- Infrastructure and Deployment
  - Upgrades the cluster while it operational and enables new features

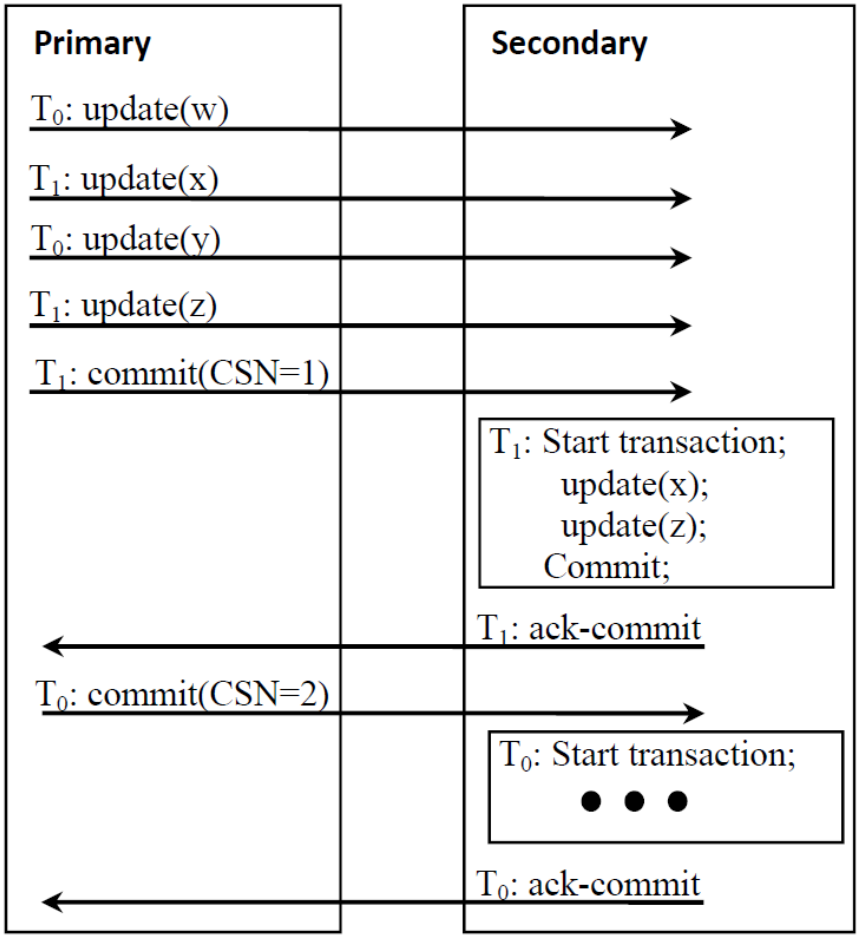
During a Transaction

- A transaction on the primary creates update records
  - An after-image of data changed by each update
  - These update records serve as redo records
- Secondaries can be used as read only copies of data if needed
  - Have an isolation level of read-committed
- Updates records are identified by table key and not page ID
  - SQL instances holding replicas do not need to be identical
- Primary streams updates to the secondaries
  - After-images of modified indices are sent as well
  - Deleted if secondary received abort from the primary

When a Transaction Commits

- The primary assigns the next commit sequence number
- The secondaries apply the updates to their databases in commit sequence number order
- The secondary sends an ACK back to the primary when finished
- When the primary obtains ACKs from a quorum of replicas, it writes a persistent commit record

Commits and Recovery



Recovery

- Update records that are lost by a secondary can be recovered
  - Primary sends a queue of updates to the secondary based on last CSN
  - If the secondary is too far behind, a fresh copy can be transferred
- Secondaries are always nearly up-to-date
  - Apply committed updates almost immediately
  - Act as hot standbys incase of primary failure
- In the event of primary failure
  - A quorum of secondaries are contacted, ensuring that no updates are lost
  - A leader is selected based on the secondary with the latest state
  - The leader propagates updates to the secondaries with older state
- Since a quorum of secondaries is used most failed secondaries are silently replaced in the background

Benchmarks

- Used a variant of TPC-C
  - Reduced memory and disk I/O, to match cloud environment
- Used a database that fits in RAM

Benchmarks

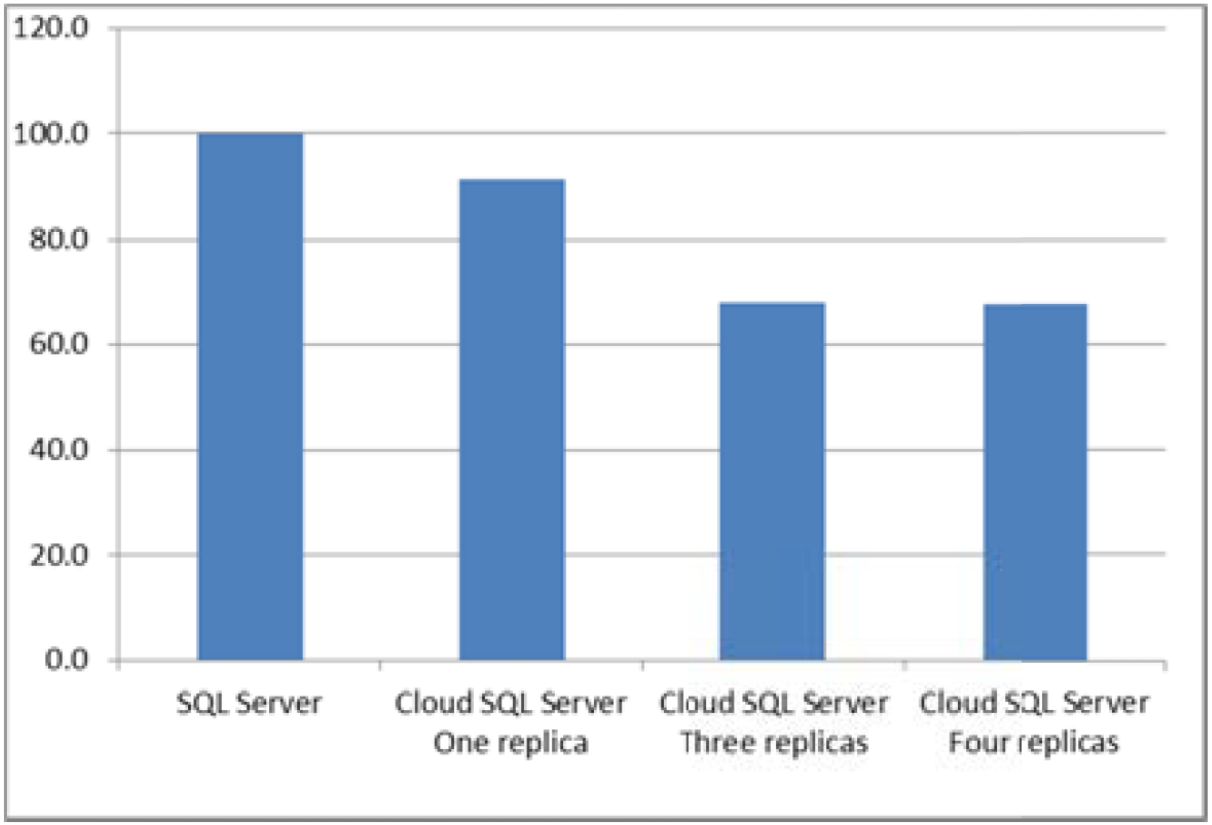


Figure 5 Relative throughput of SQL Server and Cloud SQL Server

Data Management in the Cloud: Limitations and Opportunities

Transactional data management systems

- Do not tend to scale well without limiting transactions
- It is difficult to maintain ACID guarantees due to required replication
  - Many systems choose to relax ACID guarantees
  - BigTable, SimpleDB, and PNUTS
  - Systems with eventual consistency
- Require storage of sensitive mission-critical data
  - Customer information, credit card numbers

Analytical data management systems

- Applications are mostly read-only
- ACID guarantees are not needed
  - Infrequent updates
  - Reading a snapshot of data is acceptable
- Sensitive data is often not required
  - Encrypted with an offsite key, left out

Hybrid Solution

- Requirements of a data management system
  - Efficiency
  - Fault tolerance
  - Ability to run in heterogeneous environment
  - Ability to operate on encrypted data
  - Ability to interface with existing business intelligence products
- Neither MapReduce nor parallel databases fully satisfy
  - Need a hybrid solution
  - Ease of MapReduce along with performance enhancing data structures

Pig and Pig Latin

Motivation

- There is often a need for ad-hoc analysis of very large datasets
  - Driven by innovation at a number of organizations
  - Data includes web crawls, search logs, and click streams
- Existing distributed databases tend to be prohibitively expensive at extremely large scales
- MapReduce works well for data analysis, but requires custom code for common operations available in database solutions
  - Projection
  - Grouping
  - Filtering

SQL vs. Pig Latin

- Encoding efficient dataflows in SQL is difficult
  - Procedural solution are easier to reason about
  - Automatic query optimization performs poorly in this environment
- Pig Latin is a middle ground between SQL and procedural code
  - A sequence of steps transforming the data
  - Each transformation is high level like SQL

SQL vs. Pig Latin

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10000000
```

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY
      COUNT(good_urls)>10000000;
output = FOREACH big_groups
      GENERATE category, AVG(good_urls.pagerank);
```

Dataflow

- Pig Latin is compiled into many MapReduce jobs
- These jobs then execute on Hadoop

- Operations do not need to be executed in order
  - Some operations may be completed concurrently
  - Filters can be performed in an order that returns results efficiently

```
spam_urls = FILTER urls BY isSpam(url);
culprit_urls = FILTER spam_urls BY pagerank > 0.8;
```

Features

- Directly run on datafiles, no need to import into a database
- User defined functions are allowed
  - Written is Java
- Nested data is allowed, as well as nested operations
  - Would require multiple tables in a traditional database
  - Sets, maps, and tuples can be read from a single field

Data Model

- There are four basic data types
  - Atom: a single value such as a sting
  - Tuple: a sequence of fields of any data type
  - Bag: a collection of tuples
  - Map: a collection of data items where each has a key

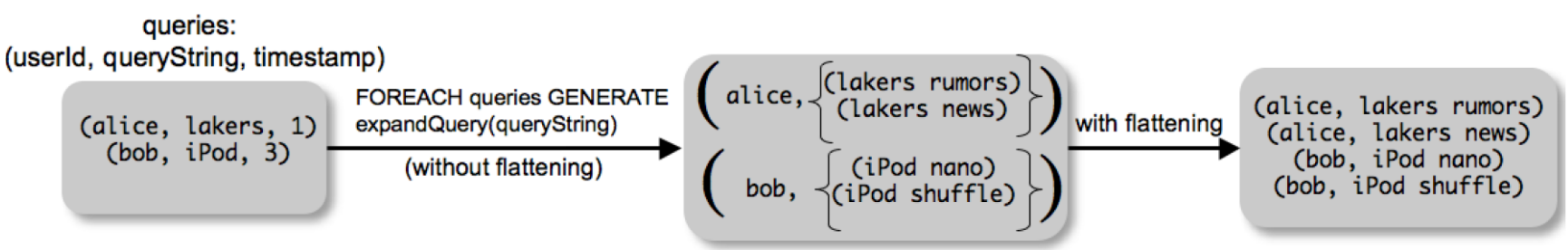
$$t = \left(\text{'alice'}, \left\{ \begin{array}{c} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20]\right)$$

Let fields of tuple t be called f1, f2, f3

Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Operations

- LOAD: specify the file and schema
- FOREACH: perform an operation on each tuple
- FILTER: discards unwanted data
- ORDER: sorts the data
- STORE: write the output to a file

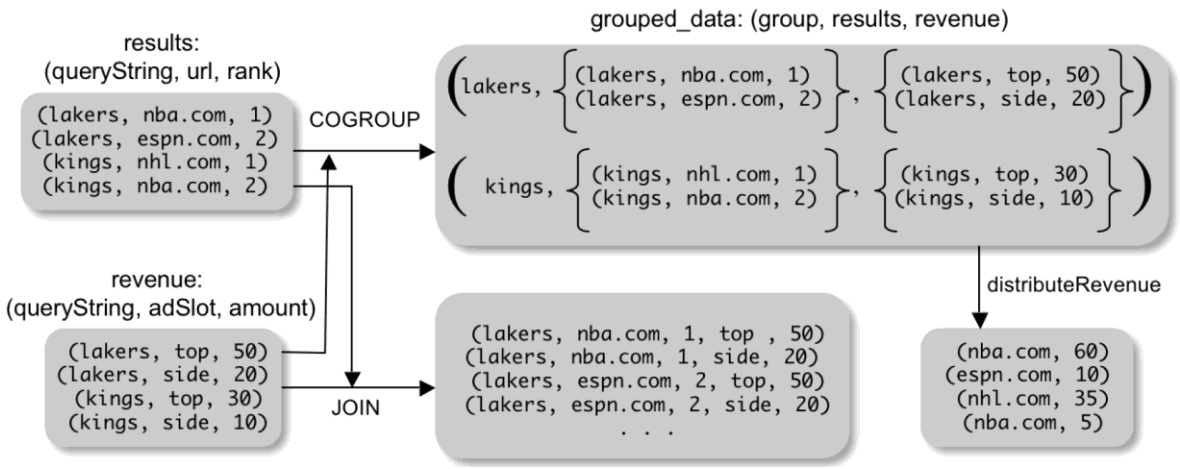


Grouping Operations

- COGROUP: groups tuples into nested bags
- GROUP: special case of COGROUP for one data set
- JOIN: shortcut for COGROUP followed by flattening
- CROSS: performs the cross product of two or more bags
- UNION: union of two or more bags

Grouping Operations

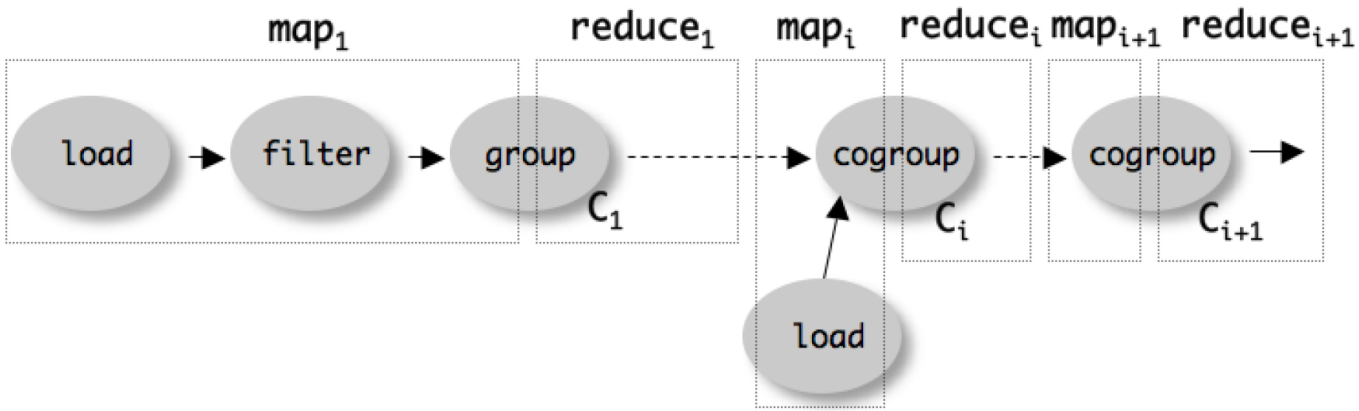
```
join_result = JOIN results BY queryString,
      revenue BY queryString;

temp_var = COGROUP results BY queryString,
      revenue BY queryString;
join_result = FOREACH temp_var GENERATE
      FLATTEN(results), FLATTEN(revenue);
```



Conversion to MapReduce

- Commands are parsed and a logical plan is built for each bag
- Lazy: Processing starts once a STORE command is issued
  - Allows filter reordering and other optimizations

- FOREACH: multiple MapReduce instances are started
- GROUP: mappers assign keys, reducers process each group
- ORDER: first job samples input, second range partitions based on first job



H-Store
Concurrency Control

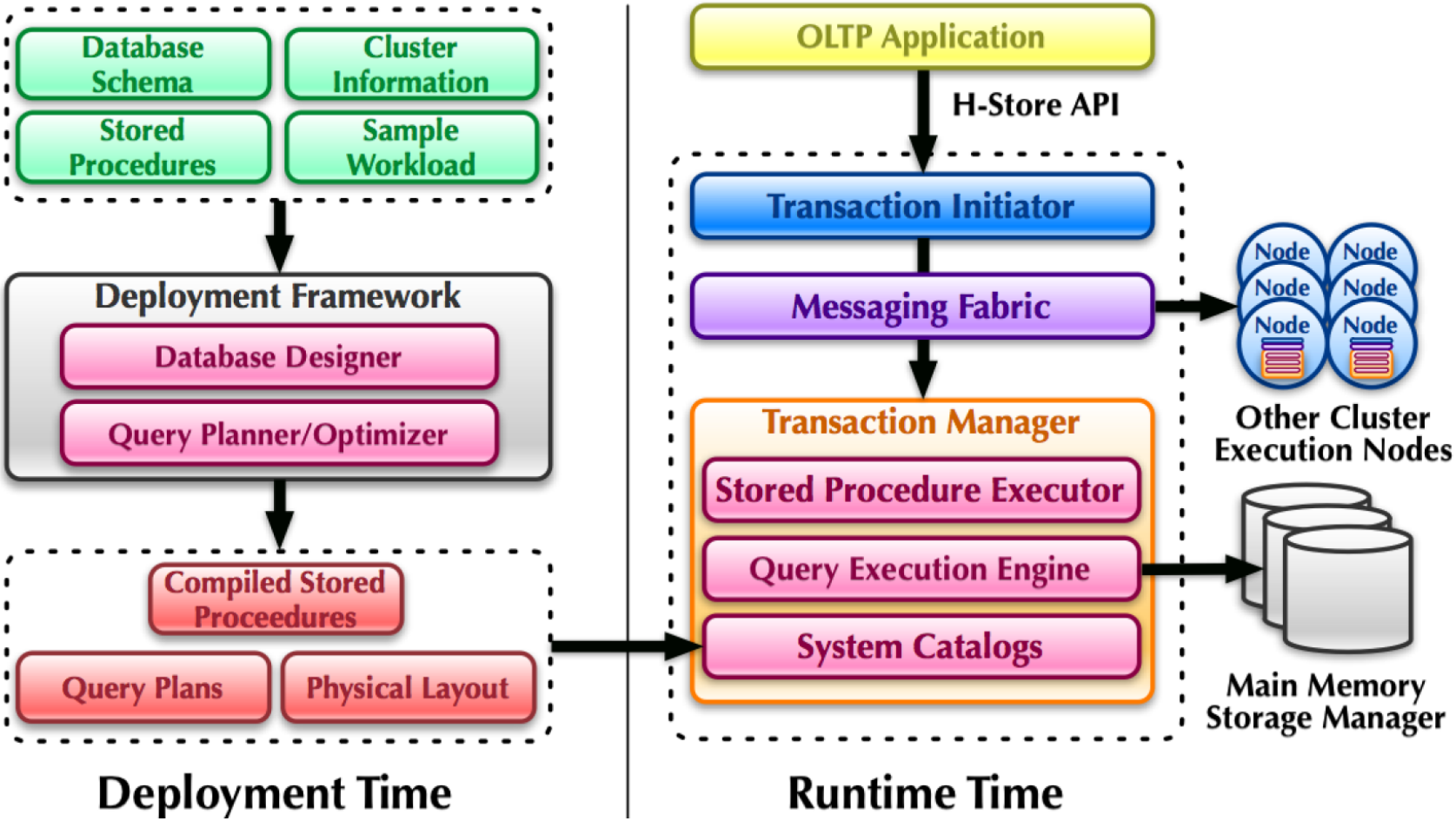
H-Store

- Distributed cluster of shared-nothing machines
- All data resides in main memory

- Uses stored procedures for transactions
  - Identified by a unique name
  - Procedures are provided when cluster is deployed
  - Used to determine how data is partitioned and replicated
  - Ad-hoc queries are still possible

- Each partition is managed by a single thread
  - Transactions are queued and execute one at a time

H-Store Architecture



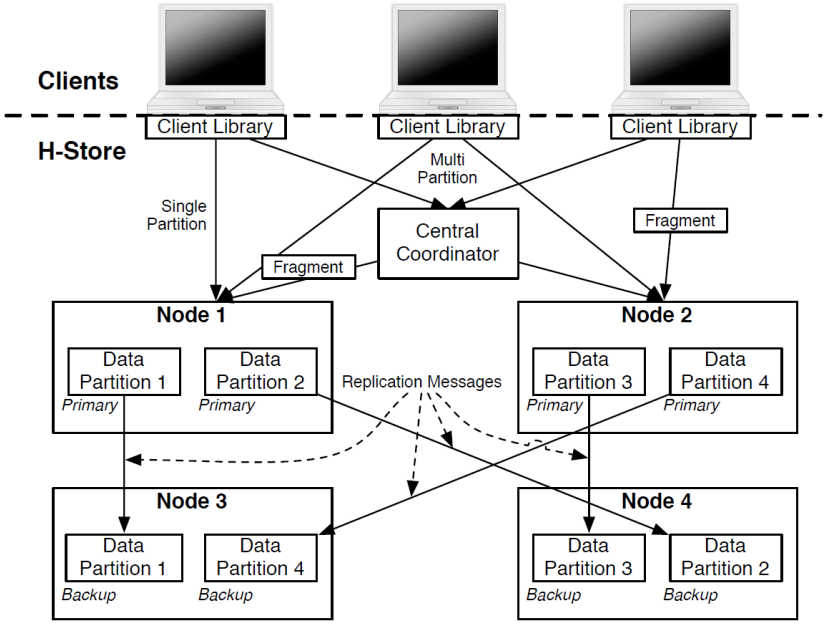
Assumptions

- Designed for a partitioned main-memory database

- Transactions are stored procedures
- All data fits into main memory at the node
- Most of the transactions should access a single partition

Components

- Process for:
  - Each partition, replicated
  - A central coordinator



When a Client Connects

- The client downloads parts of the system catalog
- Downloaded information contains:
  - The available stored procedures
  - The locations of each partition
  - Details on how data is distributed
- Allows clients to direct queries to the appropriate process

Transactions

- Available as stored procedures
  - A mixture of control code and SQL operations
- Transactions can be divided into fragments
  - A fragment is a unit of work that can be executed on one partition
- Single partition transactions:
  - Request sent directly to the primary partition for the data
  - Primary forwards requests to replicas
  - Primary executes the transaction
  - Waits for acknowledgement from replicas

Multi-Partition Transactions

- All multi-partition transactions can be sent through a central coordinator
  - Only needed when using the speculative concurrency scheme
  - Assigns a global order so that there will be no deadlocks
  - Downside: Limits the rate of multi-partition transactions
- Use two-phase commit with an undo buffer
  - The in-memory buffer is discarded when the transaction commits
- Transactions may need to wait for replies from other nodes
  - Could be doing useful work during this time

Concurrency Control Schemes

- Blocking
  - The simplest scheme
  - Limits system throughput
- Speculative Execution
  - Execute other transactions while waiting
  - Rollback if there are conflicts
- Locking
  - Utilizes read and write locks

Speculative Execution

- Multi-part transactions must wait on coordinator for commit
  - Most of the time, the transaction will commit
  - Solution: Execute queued transactions speculatively
- When the blocked transaction commits
  - Speculative transactions immediately commit
  - Results from speculative transactions are returned to the clients
- Multi-partition transactions can be speculated as well
  - Results returned to coordinator early, noting the blocked transaction

Speculative Execution

Transaction Fragment Arrives

```
if no active transaction:
  if single partition:
    execute fragment without undo buffer
    commit
  else:
    execute fragment with undo buffer
else if fragment continues active multi-partition transaction:
  continue transaction by executing fragment
  if transaction is finished locally:
    speculate queued transactions
else if tail transaction in uncommitted queue is finished locally:
  execute fragment with undo buffer
  same_coordinator ← false
  if all txns in uncommitted queue have same coordinator:
    same_coordinator ← true
  if transaction is multi-partition and same_coordinator:
    record dependency on previous multi-partition transaction
    send speculative results
else:
  queue fragment
```

Commit/Abort Decision Arrives

```
if abort:
  undo and re-queue all speculative transactions
  undo aborted transaction
else:
  while next speculative transaction is not multi-partition:
    commit speculative transaction
    send results
execute/speculate queued transactions
```

Locking

- Transactions acquire read and write locks
  - Block for a conflicting lock request
- Allows non conflicting single-partition transactions
  - Execute during the network stalls of multi-partition transactions
- Multi-partition transactions are sent directly to partitions
  - Not forwarded from central coordinator
  - Two-phase locking ensures transactions have a serializable order
  - More efficient when there are no conflicts
- Deadlock is possible in this scheme
  - Cycle detection is used locally
  - Timeouts are used for distributed deadlocks

Performance

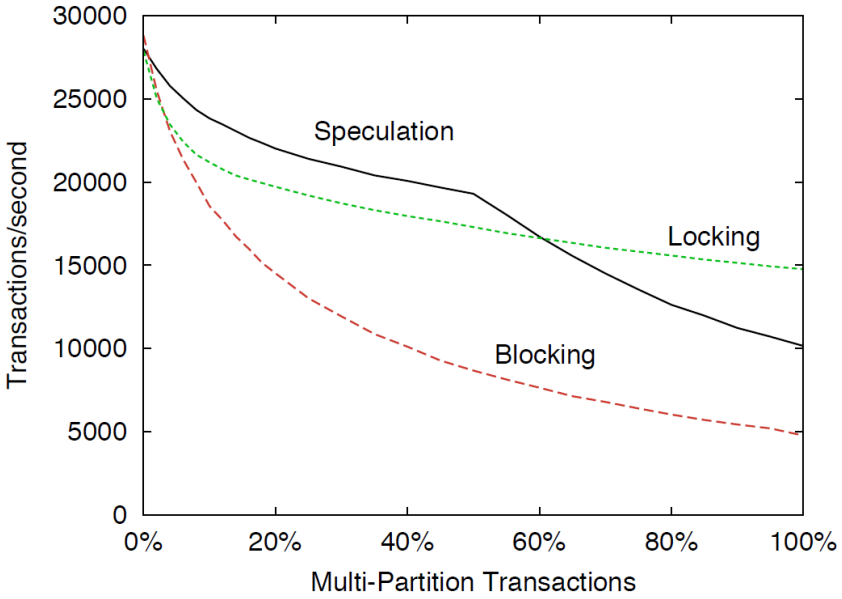


Figure 4: Microbenchmark Without Conflicts

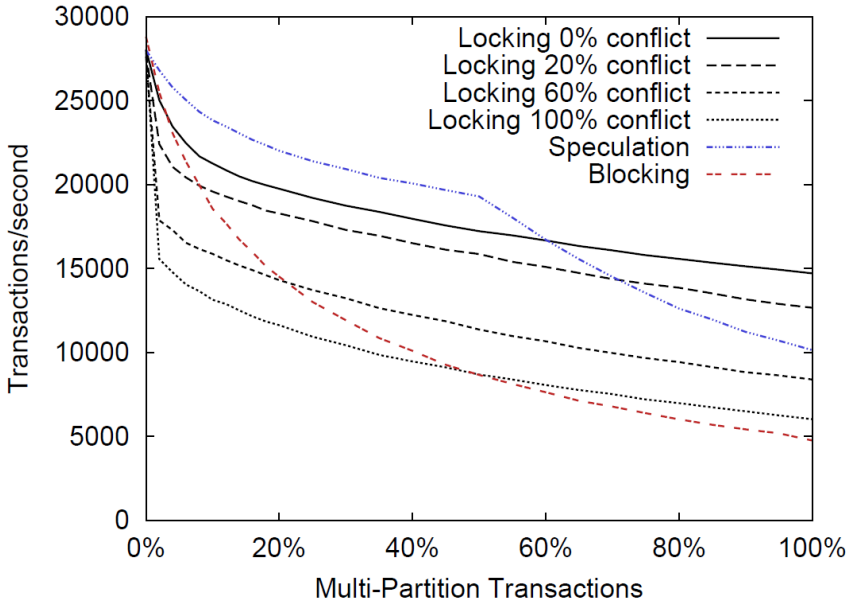


Figure 5: Microbenchmark With Conflicts

Performance

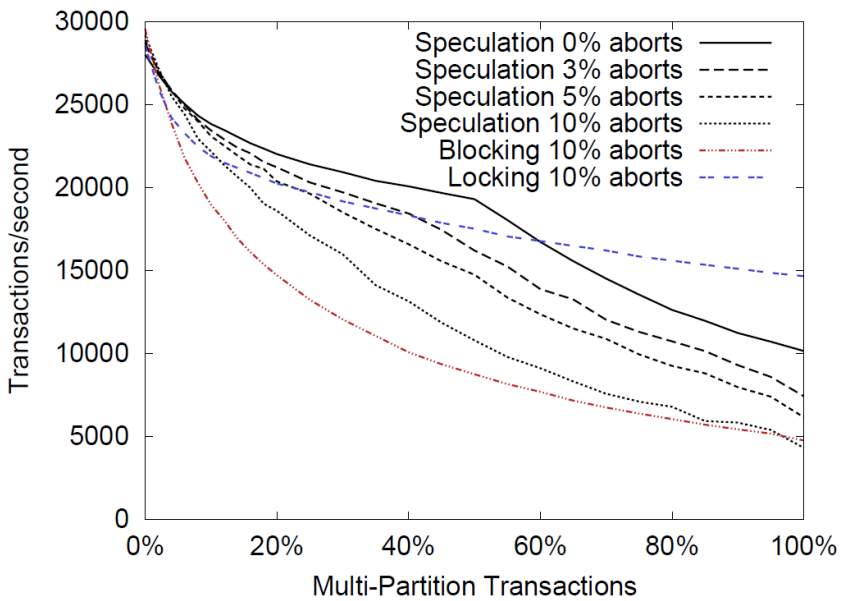


Figure 6: Microbenchmark With Aborts

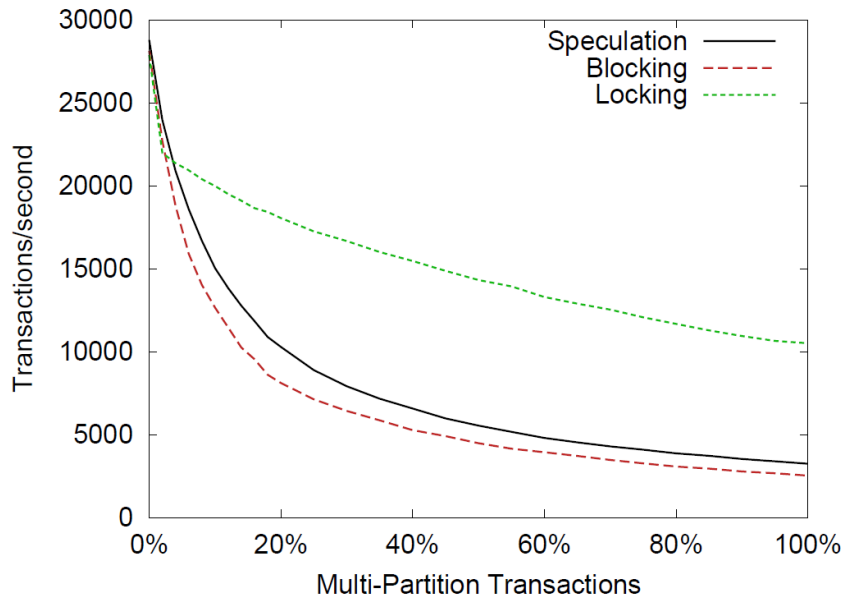


Figure 7: General Transaction Microbenchmark

Performance

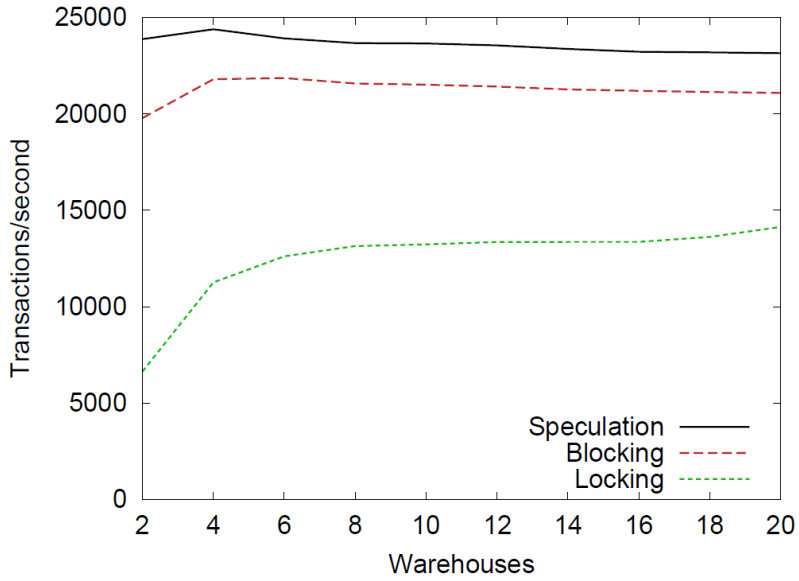


Figure 8: TPC-C Throughput Varying Warehouses

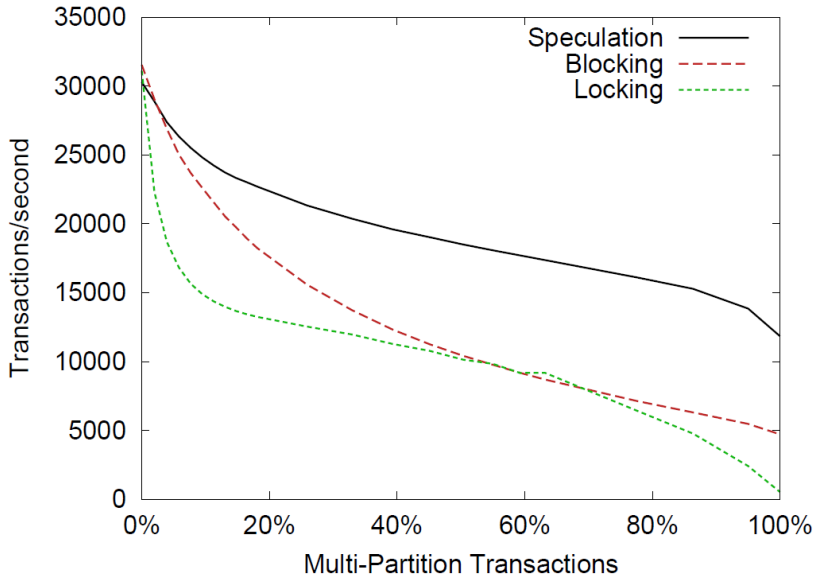


Figure 9: TPC-C 100% New Order

Concurrency Control Schemes

		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

Issues with H-Store

- Data is volatile
  - Assume that data will be recoverable from replicas
  - What happens when the data center loses power?
- Requires transactions as stored procedures
  - No ad-hoc queries
- Does not work well with long transactions

Questions?