Scalable Transaction Management in Cloud Data Systems

NSF Award 1319333 (Principal Investigator: Anand Tripathi)

Graduate Students: Vinit Padhye, Tara Sasank Sunkara, Jeremy Tucker , Gowtham Rajappan, B. Thirunavukarasu, Varun Pandye, Rahul Sharma, Manu Khandelwal, Tanmay Mehta

REU Participants: Indrajith Premanath, Kris Lange, Alexander Cina, Wei-Wei Wu, Henry Hoang

http://ajanta.cs.umn.edu

Department of Computer Science & Engineering

University of Minnesota, Minneapolis MN 55455

Research Goals

- Provide scalable multi-row serializable transactions support for key-value based NoSQL systems
- How to utilize Snapshot Isolation model for scalable transactions
- Provide transaction support for geographically replicated data.
- Develop scalable transaction management model for partially replicated data in large-scale systems.
- Develop models and techniques for supporting different consistency models for an application.
- Utilize transaction model for data analytics applications on clusters.

Transactions in NoSQL Systems

- Transaction support is either not present (Amazon Dynamo, Yahoo PNUTs) or available with certain restrictions (Google Megastore)
- Strong consistency requires synchronous replication which incur higher latencies
- Due to this systems such as Dynamo and PNUTs adopted eventual consistency model
- Eventual consistency models may not be adequate for many applications which require stronger consistency guarantees.

Research Approach

- Transaction mangaement techniques investigated here are based on the Snapshot-Isolation (SI) model.
- Because the SI model does not guarantee serializable execution, this project has developed techniques to ensure serializability under the Snapshot Isolation model on NoSQL systems.
- Building upon the causal consistency model and Snapshot Isolation, this work has developed a model to simultaneiously support multiple cosnistency models, ranging from serializability to eventual consistency.
- Replication management model is based on asynchronous update propagation.

Publications

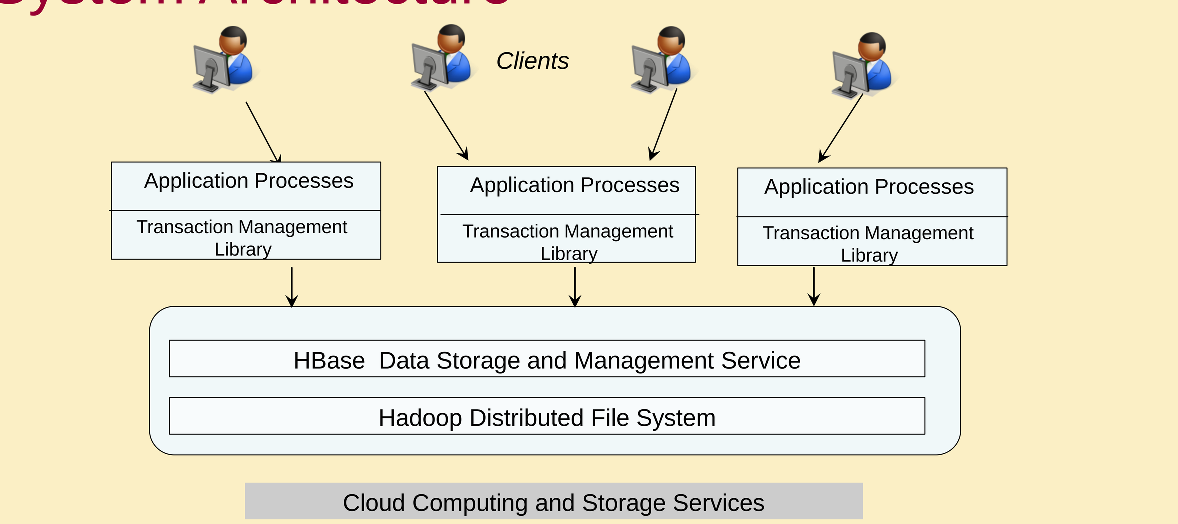
- *Scalable Transaction Management with Snapshot Isolation for NoSQL Data Storage Systems*, Vinit Padhye and Anand Tripathi, IEEE Transactions on Services Computing, 2013.
- *Transaction Management with Causal Snapshot Isolation in Partially Replicated Databases*, Vinit Padhye, Gowtham Rajappan and Anand Tripathi, in IEEE Symposium on Reliable Distributed Systems (SRDS'2014).
- *Beehive: A Framework for Graph Data Analytics on Cloud Computing Platforms*, Anand Tripathi, Vinit Padhye, Tara Sasank Sunkara, in Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing, held in conjunction with 2014 International Conference on Parallel Processing (ICPP'2014).
- *A Transaction Model for Management of Replicated Data with Multiple Consistency Levels*, Anand Tripathi and BhagavathiDhass Thirunavukarasu, in IEEE Conference on Big Data, 2015.
- A Transaction Model with Multilevel Consistency for Shared Data in Distributed Groupware Systems, Anand Tripathi, in Proceedings of IEEE 2nd International Conference on Collaboration and Internet Computing, 2016
- Scalable Transaction Management for Partially Replicated Data in Cloud Computing Environments, Anand Tripathi and Gowtham Rajappan, in Proceedings of IEEE 9th International Conference on Cloud Computing, 2016.
- *A Transactional Model for Parallel Programming of Graph Applications on Computing Clusters,* Anand Tripathi, Vinit Padhye, Tara Sasank Sunkara, Jeremy Tucker, BhagavathiDhass Thirunavukarasu, Varun Pandey, Rahul R. Sharma, IEEE CLOUD 2017.
- *Incremental Parallel Computing using Transactional Model in Large-scale Dynamic Graph Structures,* Anand Tripathi, Rahul R. Sharma, Manu Khandelwa, Tanmay Mehta, Varun Pandey, in the International Workshop on Big Graph Processing (BGP) 2017, in conjunction with ICDCS-2017 .
- Investigation of a Transactional Model for Incremental Parallel Computing in Dynamic Graphs, Anand Tripathi, Rahul R. Sharma, Manu Khandelwal, Tanmay Mehta, Varun Pandey, Charandeep Parisineti, Tech Report, Department of Computer Science & Engineering, University of Minnesota, May 2017.

Multi-Row ACID Transactions on HBASE

Goal is to develop scalable techniques for serializable multi-row transactions on Haddop/HBase

- Transaction management functions are decoupled from the data storage/management service
- Transactions are performed entirely outside storage system without affecting scalability of the storage system
- Cooperative recovery model: Any application process can execute recovery actions on behalf of a failed transaction process to complete its commit/abort.
- Both the transaction management data and the application data are stored in the global key-value based storage service.

System Architecture



Row Key (Trans ID)	Snapshot Timestamp TS	Commit Timestamp TSc	Write-Set	Out-Edges	Status (Trans State)
tid1	TS = 100	TS = 150	List of Item IDs	Outgoing Dedpendency Edges	Active, Validation, Commit-Incomplete, Commited, Aborted
tid2					

Transaction Table

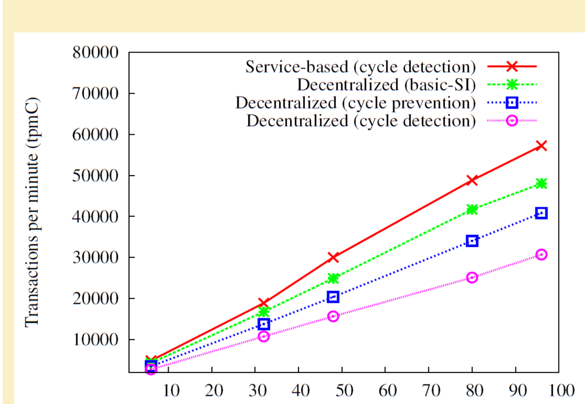
Row Key (Item ID)	Data Column	Commited Version	read-ts	rlock	wlock	readers
Item1	Value 1 tid1 Value 2 tid2 Value 3 tid3	tid1 80 tid2 100 tid3 120	Time Stamp	tid/null	tid/null	List of Transaction IDs
Item2						

Storage Table

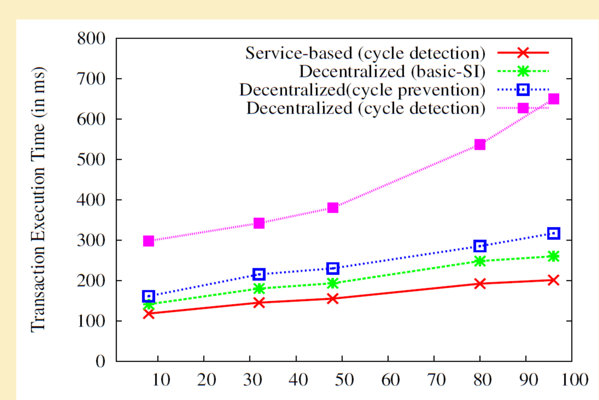
Two Approaches for Transaction Validation

We developed and evaluated two system architecture models for transaction validation and for detecting/preventing cycles to make SI-based transactions serializable:

- Fully decentralized (each application process performs checks)
- Service-Based validation (dedicated service performs checks)



Transaction Throughput

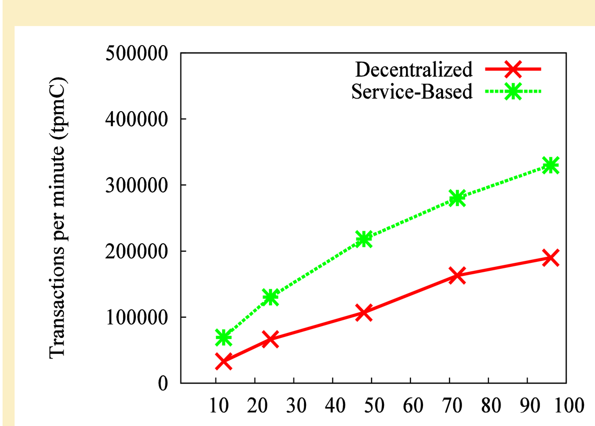


Transaction Response Times

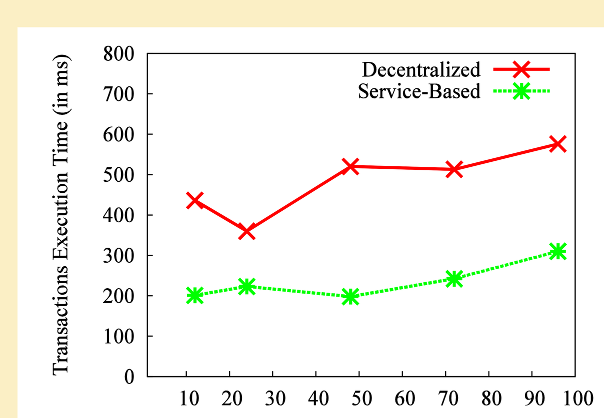
Scale-out of System Performance on 100 nodes (8 cores per node)

We evaluated two schemes:

Service-based and Decentralized Cycle-Prevention Schemes



Transaction Throughput



Transaction Response Times

Partitioned Causal Snapshot isolation (PCSI)

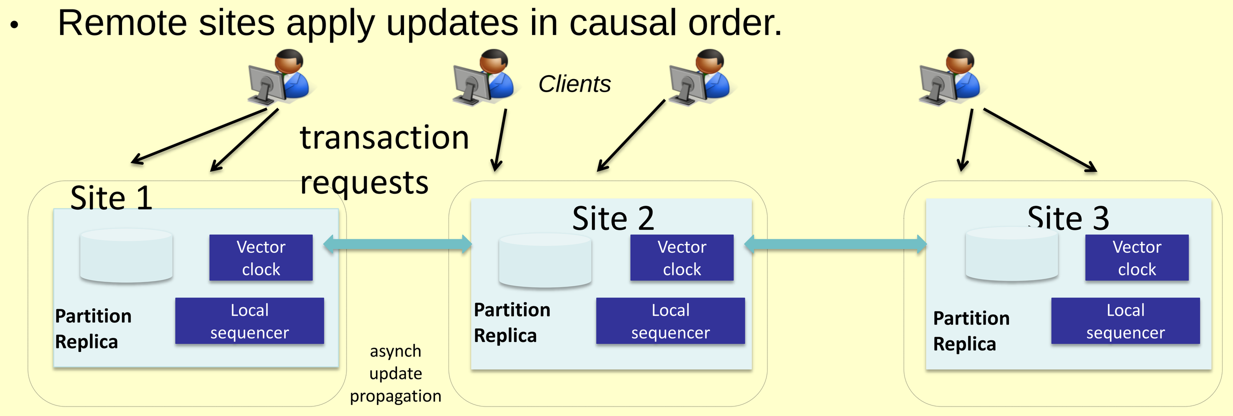
- Goal of this work is to provide a scalable model for transaction management in systems with **partially replicated data**.
- A key-value based data store is sharded into partitions, and each partition is replicated on a subset of the sites.
- A partition is not required to be replicated at all sites.
- A site may contain replicas of any subset of the partitions.
- Goal is to provide a weaker yet useful model based on causal consistency and snapshot isolation.

PCSI Model

PCSI model is based on Snapshot Isolation with causal consistency guarantees.

Properties of the PCSI Model:

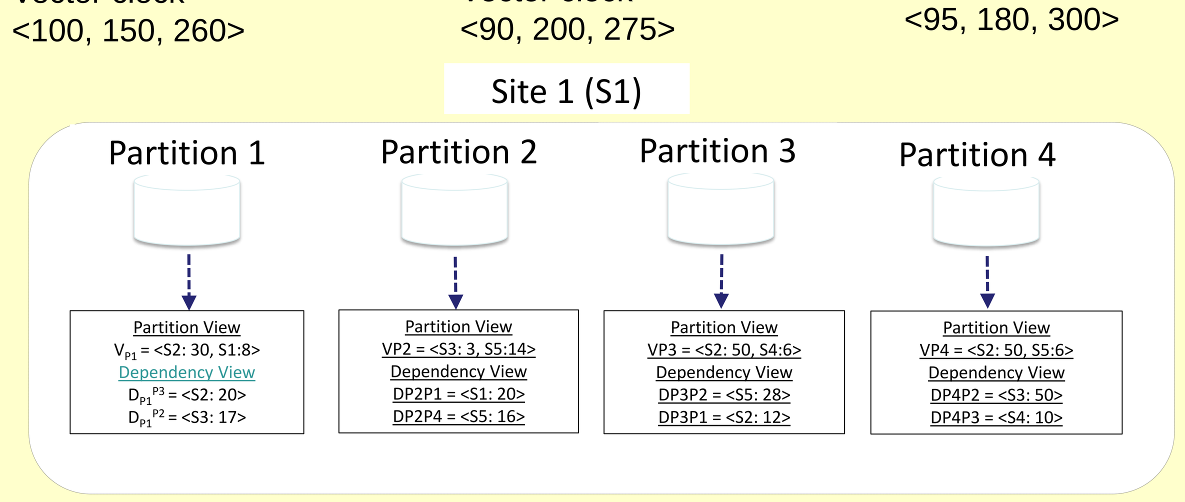
- Transaction Update Ordering:
  - Causal ordering of transaction updates
  - Global ordering of per-item updates
  - Replica-based ordering of per partition updates
- Globally Consistent Snapshot
  - A transaction can read data from any set of partitions (local or remote using a snapshot which is globally consistent.
  - A snapshot is causally consistent
  - Atomicity of transaction updates (all or none of the updates of a transaction are visible in the snapshot)
- Updates are propagated asynchronously to remote sites storing the updated partitions.
- Remote sites apply updates in causal order.



Sequencer = 100
Vector clock = <100, 150, 260>

Sequencer = 200
Vector clock = <90, 200, 275>

Sequencer = 300
Vector clock = <95, 180, 300>



Each site Si maintains for **each partition p**:

- A sequencer to assign monotonically increasing sequence numbers to the local update transactions.
- A vector clock called ***partition view*** (V), with one entry for each replica site.
- A set of vector clocks called ***partition dependency view*** (D)

Snapshot time of a transaction is a set of vector clocks, with one vector clock value for each of the partition to be accessed.

In the original design, for updating a non-local partition by a transaction a "ghost replica" was created at the execution site.

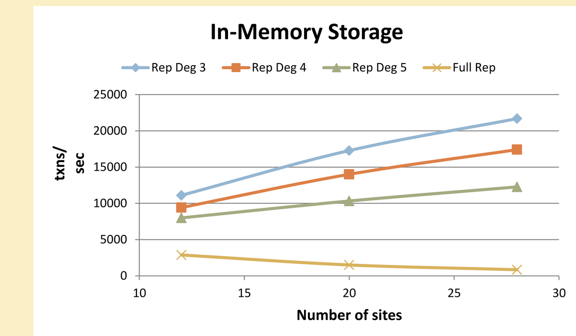
- A ghost replica does not store any data but contains a local sequencer and vector clock based dependency information.

Scalability of the initial design was limited to about 30 nodes.

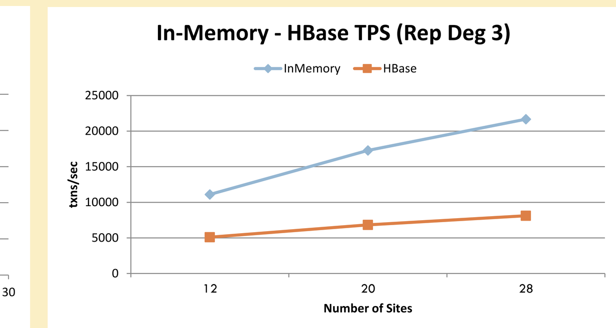
- Overheads imposed by vector clock operations increased with the creation of "ghost replicas", which were used for assigning update sequence number for a non-local partition update.

Performance Scaling of the PCSI Protocol

Scalability evaluations were conducted for both in-memory data storage as well storing data in HBase at each site.



Throughput (Txn/sec) for In-Memory Data Store



Throughput (Txn/sec) for HBase Data Store

- The scalability limitation of the original design motivated us to develop a new protocol for transactions updating data non-local partitions.

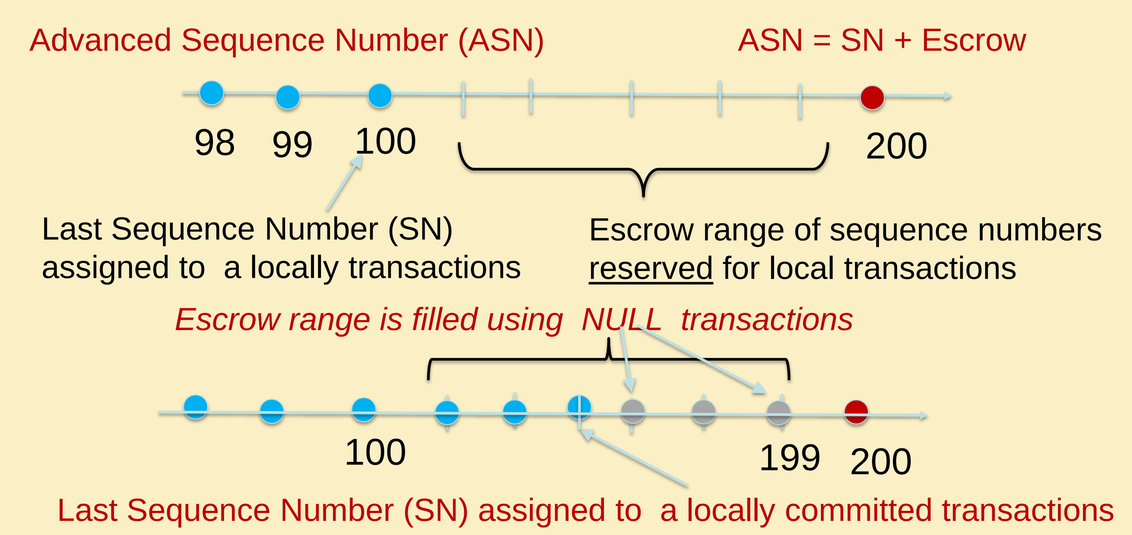
Design Revision

Approach: A transaction updating a non-local partition obtains the update sequence number from a remote replicas of the partition

Challenges:

- Stalling of local transactions at the site issuing sequence number to a remote transaction, because all update transactions on a partition replica are applied in their sequence number ordering.
- Potential of deadlocks when sequence numbers are obtained from multiple remote partition replicas because of conflicting ordering implied by two or more such numbers.

Solution: Escrow-based sequence numbers to remote transactions.



Scalability Evaluations

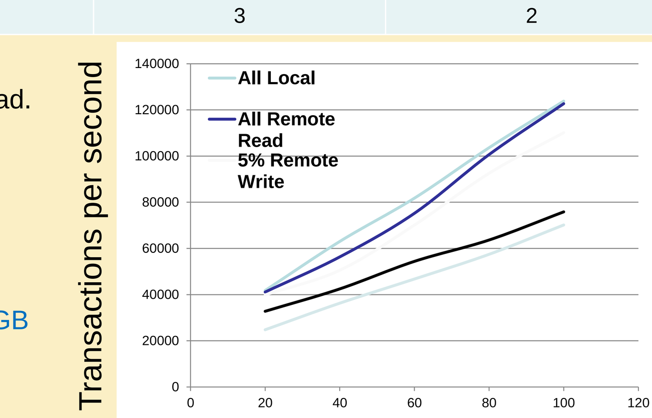
Used three workloads for evaluations

Workload	Number of Read Partitions	Number of Update Partitions	Number of Remote Read/Update Partitions
Workload A	2	1	1
Workload B	2	2	1
Workload C	2	3	2

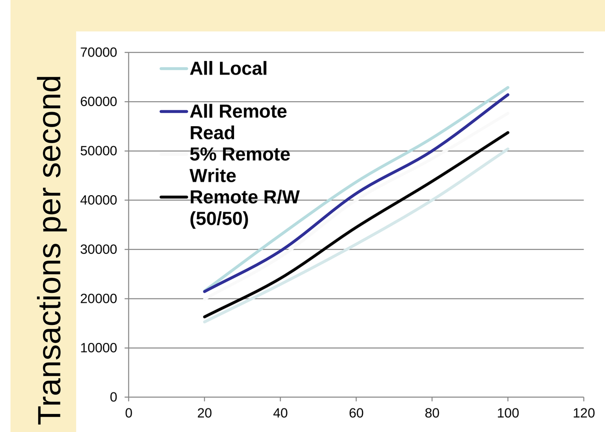
From a Read partition, 4 items were read. In an Update partition: 2 items were updated.

Five load mixes for each workload with varying remote access were used with each of the three workloads

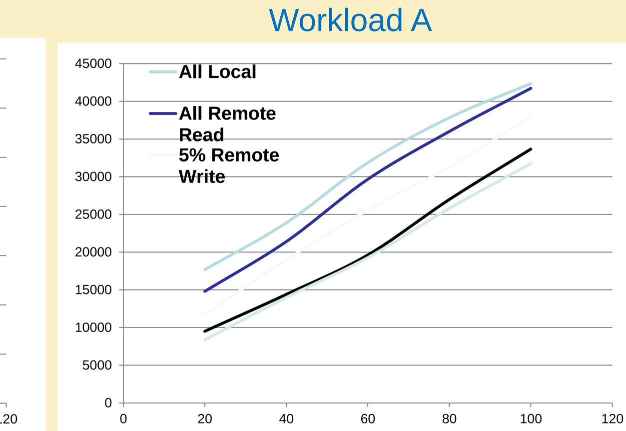
Each node is 8-core 2.6 GHz CPU, 22GB memory



Workload A



Workload B



Workload C

Acknowledgements: This work was conducted with support from NSF grant 1319333 and experiments were conducted on the resources provided by the Minnesota Supercomputing Institute

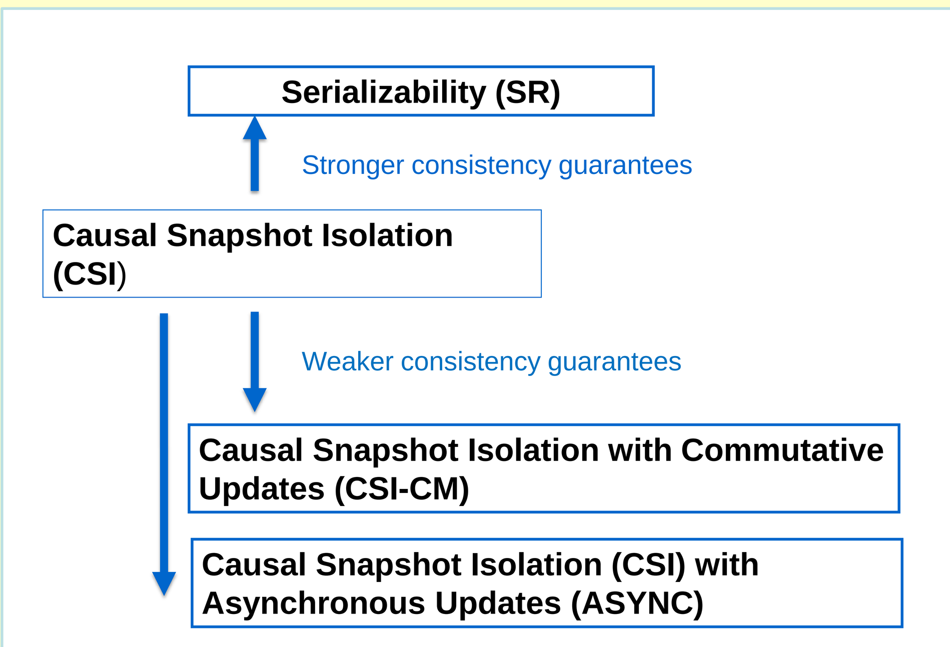
Scalable Transaction Management in Cloud Data Systems

NSF Award 1319333 (Principal Investigator: Anand Tripathi, University of Minnesota, Minneapolis)

Multilevel Transactional Consistency Model

The goal of this work is to simultaneously support transactions with different levels of consistency guarantees.

We provide a model which simultaneously supports transactions with four different consistency levels:
- Serializability
- Causal Snapshot isolation (CSI)
- Causal Snapshot Isolation with commutative updates (CSI-CM)
- Asynchronous updates (ASYNC)

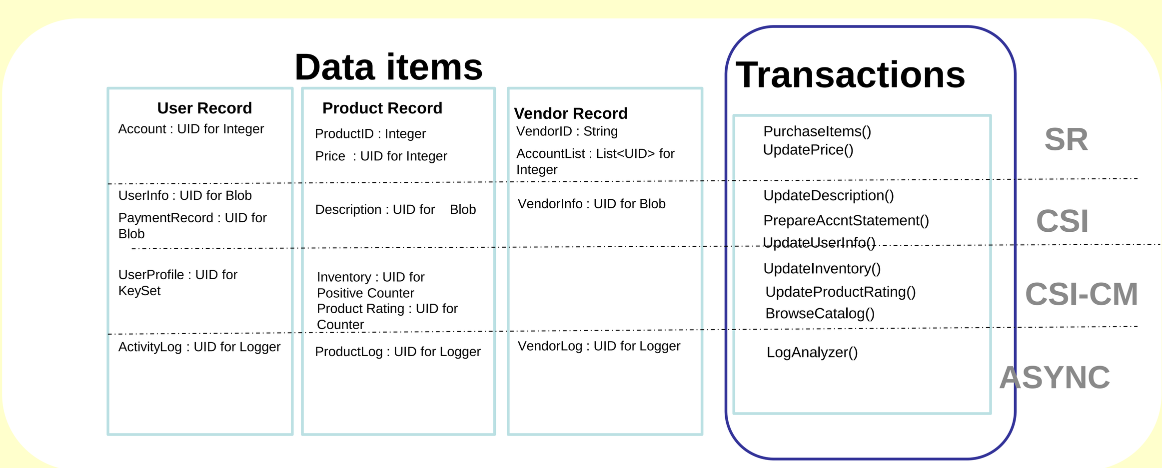


Model

Both data and transactions are organized along a hierarchy of consistency model.
- A data item can belong to only one level in this hierarchy.
- A transaction in the system is designated to execute at exactly one of the levels in this hierarchy

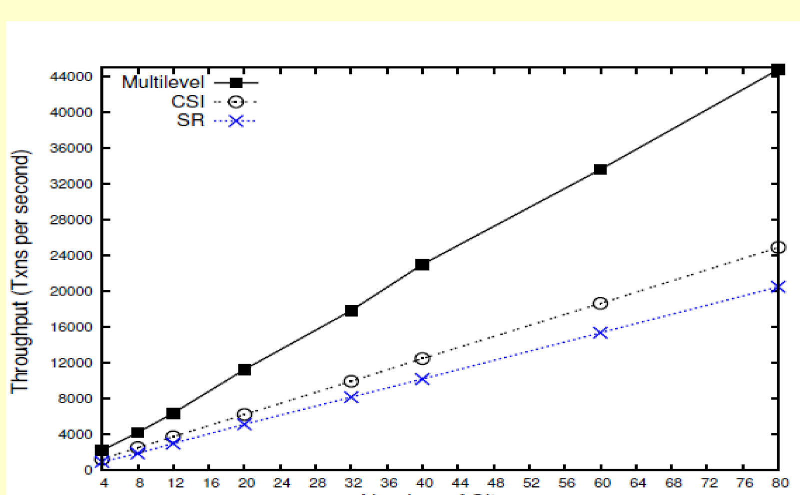
Read-up/Write-down Rule
- A transaction can read only data items that are at its execution level or at stronger consistency levels.
- A transaction can write only data items that are at its execution level or at weaker consistency levels.

Multilevel Modeling of an E-Commerce Application



Performance Evaluation

We used two benchmarks to evaluate the benefits of multi-level consistency model: E-Commerce and TPC-.



E-Commerce benchmark (repl =4) under Multilevel, SR, and CSI

In regard to throughput, Multilevel performs better than SR by a factor of 2.86, and 2.6 compared to CSI.
Average response time for Multilevel was found to be 30 msecs compared to 52 msecs for SR.

Transactional Model of Parallel Computing

Goal: Speculatively harness fine-grained amorphous parallelism in graph problems using optimistic execution techniques.

A parallel computation is defined as dynamic set of vertex-centric computation tasks which can be executed in parallel.

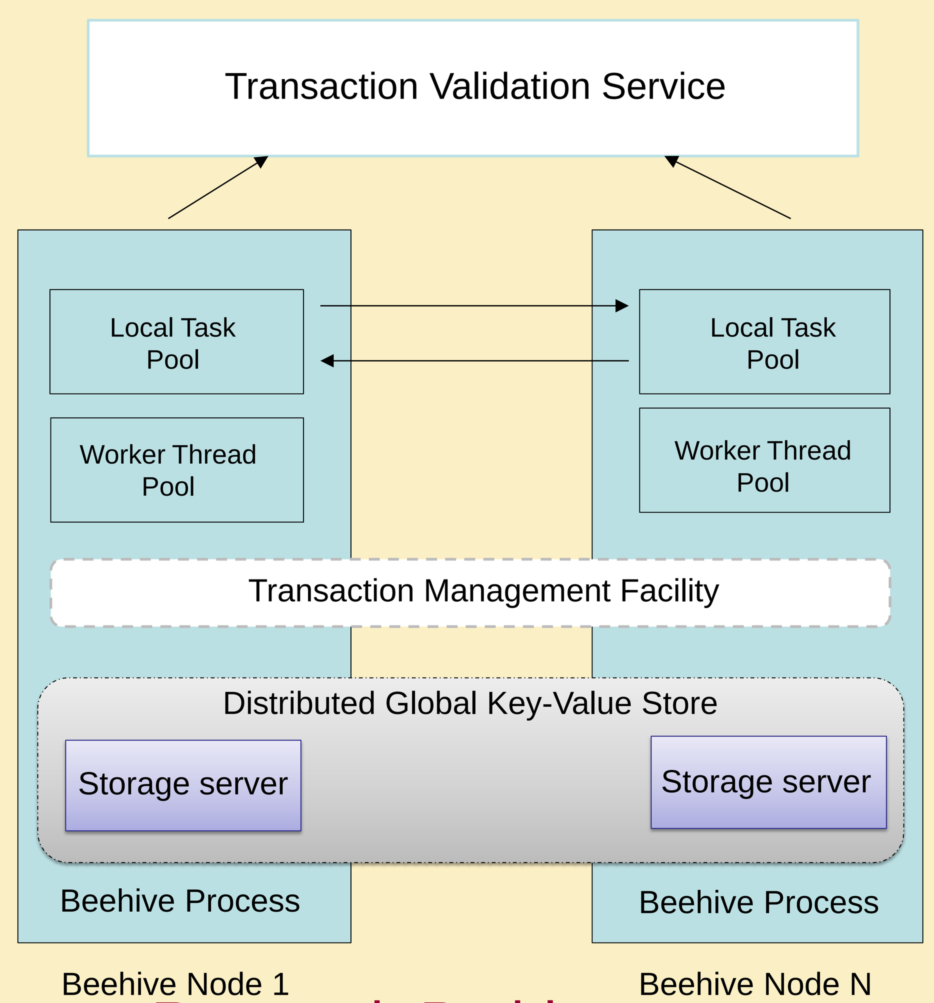
- A task performs computation related to the problem and the algorithm for solving it.
  - A task reads and updates some vertices
  - A task can create new tasks on its completion
- Each task is executed as a serializable transaction, based on optimistic concurrency control.
- After execution, a transaction performs validation before committing.
- In case of any read-write or write-write conflicts among parallel tasks, one of them commits and the others are aborted.
- On an abort, the task is re-executed as a new transaction
- On commit the updates are written to the shared global storage.

Beehive System Architecture

Beehive system executes on a collection of cluster nodes..

A Beehive process (called Beehive Node) executing on a cluster node contains the following components:

It has three key elements:
1. A distributed ***key-value*** based storage system which maintains graph data in the memory of cluster computing nodes.
2. Vertices and Edges are defined as Java objects, enabling representation of complex and rich relationships among vertices.
3. A distributed task-pool for parallel execution of tasks on cluster nodes
4. A pool of worker threads
5. The system contains a Global Transaction Validation Service for optimistic concurrency control



Research Problems
- Development of transaction programming primitives
- Efficient mechanisms for remote data access: caching, fine-grain remote data access and operations, aggregation of remote calls, Java RMI vs. Thrift based communication
- Task distribution strategies: Locality aware vs. Load aware
- Scalable Validation Service architecture for optimistic concurrency control
- Checkpointing and rollback mechanisms

Transactional Programming in Beehive

- Programmer extends the base Worker class to define the task computation in the *doTask* method, which is given a task to be executed.
- A worker executes a Task, which identifies the target vertex and any other parameters for the computation
- Example below shows the code for the Graph Coloring problem in which each vertex is assigned a color different from any of its neighbors' colors.

Base Worker Class

```
public class Worker extends Thread {
 Set<Node> readSet, writeSet;
 public void run() {
  while(true) {
        Task task  = Workpool.getTask();
         finished = false;
        while (!finished) {
             tnxnId = beginTransaction();
             readSet = new Set(); //read  objects
             writeSet = new Set(); //updated objects
             newTasks = doTask(task);
             status = validator.validate (txnId,   readSet,   writeSet);
             if (status == commit) {
                Workpool.commitTransaction (txnId, writeSet,  newTasks);
                 finished = true;
             }
             else { abortTransaction ( txnID ); sleep( delayInterval );  }
        }  }
 }
 abstract TaskSet doTask(Task t) { // Application defined implementation
 }
}
```

Worker Class for Graph Coloring Problem

```
public class GraphColorWorker extends Worker {
  public TaskSet doTask(Task task) {
        TaskSet newTasks = new TaskSet();
        Node u = storage.getNode( task.nodeId );
         // u is target node to be colored
        // Read all neighbor nodes of u
         Set<Node>  Nbrs = getNeighbors( u );
        Vector<Integer> NbrColrs = getNbrColrs(NBrs);
         Collections.sort( NbrColrs );
        int targetColor = 1;
        // Find smallest unassigned neighbor color
        foreach (Integer color in NbrColrs) {
              if (color > targetColor ) {
              break;
              } else if (color == targetColor) {  targetColor++;  }
        }
        u.color = color;
        writeSet.add(u);//add node u to write-set
        readSet.add(Nbrs);//add neighbors to read-set
        return null;
  }
}
```

We programmed several graph problems to evaluate the performance of the Beehive framework and its mechanisms

1. Graph Coloring
2. Max-Flow Problem using Preflow-Push Algorithm
3. Single Source Shortest Path (SSSP)
4. K-Nearest Neighbors (KNN)
5. Maximal Cliques in a Graph
6. Finding Connected Components
7. PageRank problem
8. All-Pairs Shortest Paths

Transactional Model for Incremental Computations in Dynamic Graphs

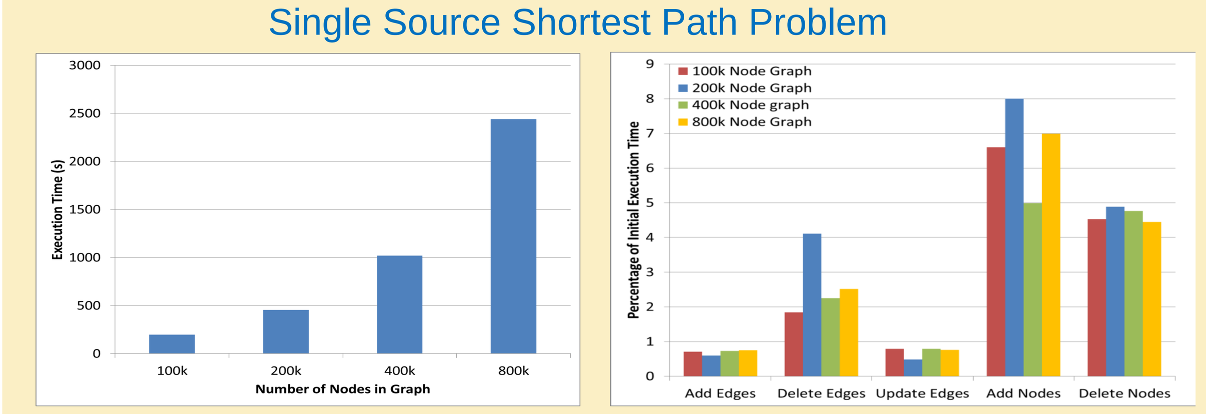
- The graph structure of a graph data analytics problem may change dynamically and evolve over a period.
- Periodic re-executions of an analytics program on a large-scale graph structure can become expensive.
- Incremental computing models can be utilized for supporting continuous queries on evolving graph structures.

Research Goals
- Eliminate the need of re-execution of an analytics program when the graph structure is modified with a *small set of updates* after the initial execution of the program.
- Evaluate how the transactional parallel computing model of Beehive can be utilized for supporting incremental computations.
- Evaluate the benefits of incremental computations for different amount of changes to the graph data.

Model for Incremental Computation
- An update to the graph after the execution of an analytics program for some problem may change certain properties of the result state.
- Updating the graph data would require some ***corrective tasks*** to be executed to restore such required properties.
- A transactional task performing a graph update operation creates a set of *cascaded tasks* for these incremental computations to be performed by the corrective tasks.
- Incremental computations can **progress concurrently** with a continuous stream of graph update transactions.
- When a stream of updates terminates and incremental computations are completed, one can query for results in a stable state.

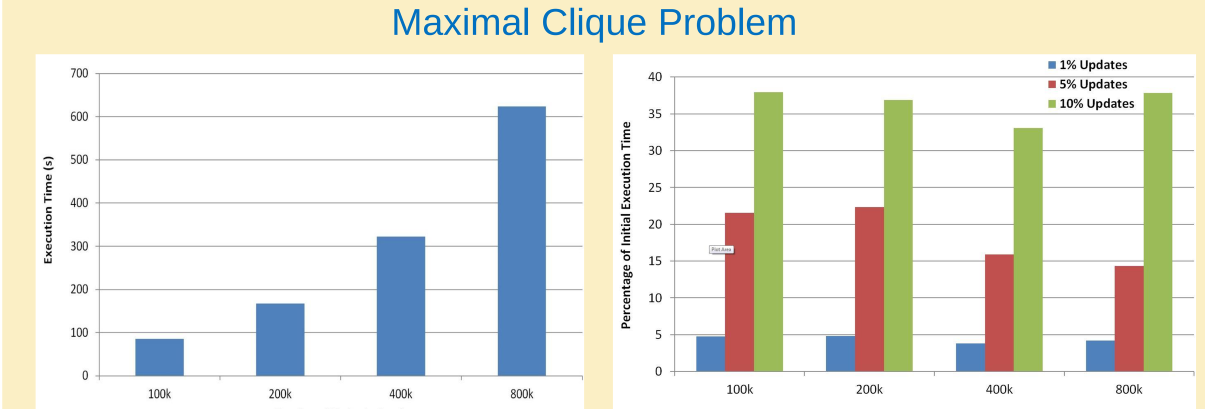
Experimental Evaluations
- We conducted experimental evaluations of this approach for several graph problems: Single Source Shortest Paths, Maximal Cliques, Graph Coloring, Connected Components, K-Nearest Neighbors.
- Experiments with 1% of edges or nodes added/deleted.

Single Source Shortest Path Problem



Initial Computation Time   Incremental Computation Time

Maximal Clique Problem



Initial Computation Time   Incremental Computation Time for Add-Edge

The benefits of performing incremental computations depend on
- Cost and complexity of the initial computation.
- Cost and complexity of performing the incremental computations for a specific type of update operation.
- Complexity depends on the problem and structure of the input graph
- For small number of updates, incremental computing approach has clear benefits in most of the cases..

Acknowledgements: This work was conducted with support from NSF grant 1319333 and experiments were conducted on the resources provided by the Minnesota Supercomputing Institute