Transactions to support Consistent Multi-Key Access

Multi-key Access

Key-value Stores offer only limited consistency

They are good for

1. Scale out using commodity servers
2. Fault tolerance
3. High availability, etc….

Examples: PNUTS, Dynamo, BigTable, HBase

BUT: Atomic and consistent access at the granularity of single keys. What about applications needing multi-key access?

Application Scenario

Key value pairs

John
Novice
$100

Bob
Expert
$2000

Application Scenario

Multi-Key Applications

- KV stores provide consistent access to applications that only touch single keys.

- Many applications require consistent multi-key access
  - Online Gaming – application deducting money from John and crediting to John must be atomic!
  - Social Networks
  - Collaborative Applications
  - …
  - …

What we need – Run Multi-key Access applications as transactions!

Transaction is a fault tolerant consistent execution

- **Atomicity** (logging & commit protocols)
- **Consistency & Isolation** (concurrency control)
- **Durability** (logging)

Why Are Transactions difficult to implement in the Cloud?

- **Distributed transactions are expensive**
  - Especially for OLTP workloads (small, touch a few records)
  - 2 PC decreases throughput, increases latency, potentially leads to distributed deadlocks.

- **Constraints based on CAP Theorem**
  - Difficult to support ACID properties in Cloud Environment
  - Transactions → consistency, their implementation interferes with availability

Relax ACID Properties?

- **Significantly increases application complexity**
- Applications need to deal with failures, recovery
- Very difficult to reason about correctness and then write code to deal with all corner cases

- **Such consistency logic has to be duplicated for all applications**

- **Often leads to bugs and poor code with inefficiencies.**

Alternative Approach

- **Limit interactions to a single node**
  - –Allows systems to scale horizontally
  - –Graceful degradation during failures
  - –Obviate need for distributed synchronization

- **Limits the need for distributed synchronization**
  - –Only for data that needs strong guarantees.

- **Extensively explored by many systems**
  - –G-store, Megastore, Cloud SQL server, Relational Cloud, Hyder, … (We will see a few examples.)

Limiting Transactions to Single Nodes

- **Transaction aware data partitioning**
  - –Strive to co-locate data items touched by a transaction into a single node.
  - –Prevents the use of 2PC or distributed synchronization.

Co-locating Data versus Ownership

- **Co-locate data:**
  - –Data location no longer based on hash or range partitioning.
  - –Cannot treat the storage system as a black box.
- **Co-locating Ownership (instead of data):**
  - –Data may reside on one node but “owned” by another
  - –Ownership refers to permission to exclusive read/write access to data
  - –Partitioning ownership effectively partitions data
  - –Decoupling allows light weight ownership transfer
  - –Can treat storage system as a black-box.
- **But partitioning data has its own advantages**
  - –Many recovery algorithms like to store transaction state on pages (pageLSN, etc.) for efficiency.
- **We will see examples of both.**(*Ignore the issue for now*)

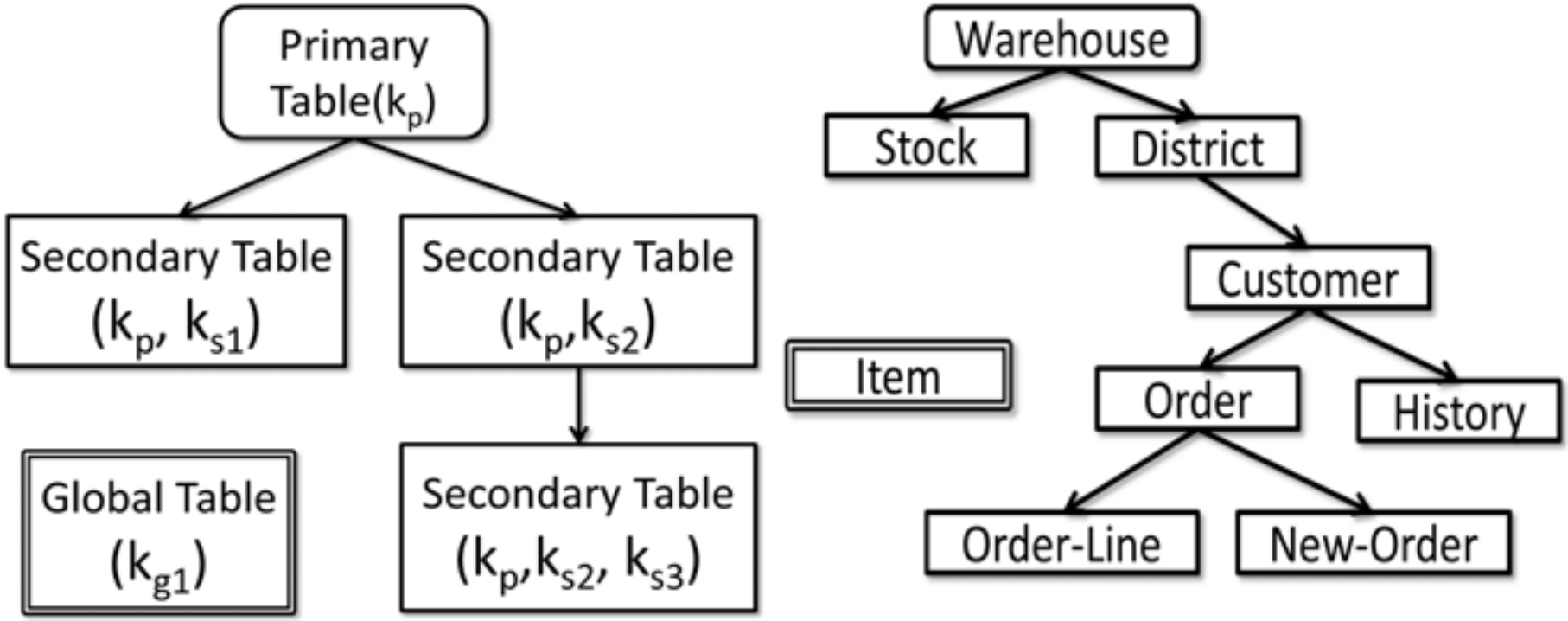
Co-locating Data/Ownership

- Schema based
- schema dictates the nature of multi-key transactions and hence can be used to co-locate the data

- **Workload driven**
- Explicitly evaluate the workload to decide on how to partition the data.

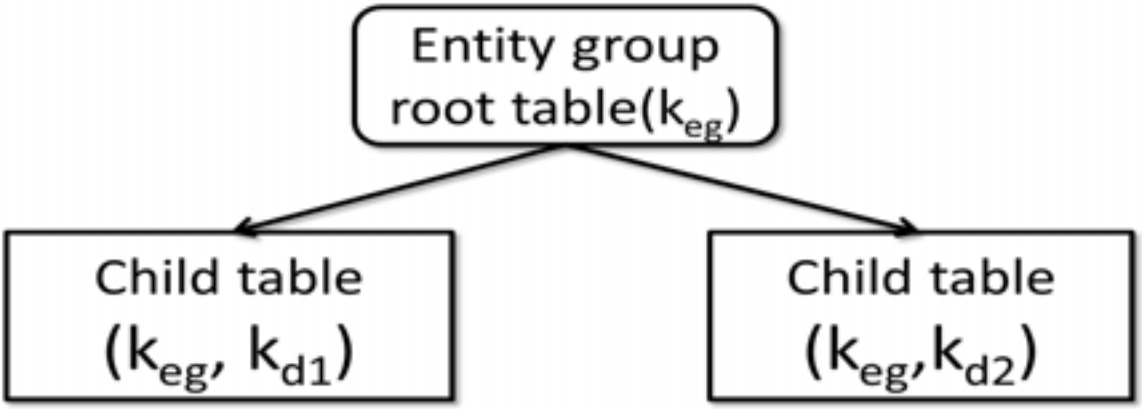
- **Dynamic Application driven**
- Empower users to define dynamic grouping of data based on emerging transaction needs

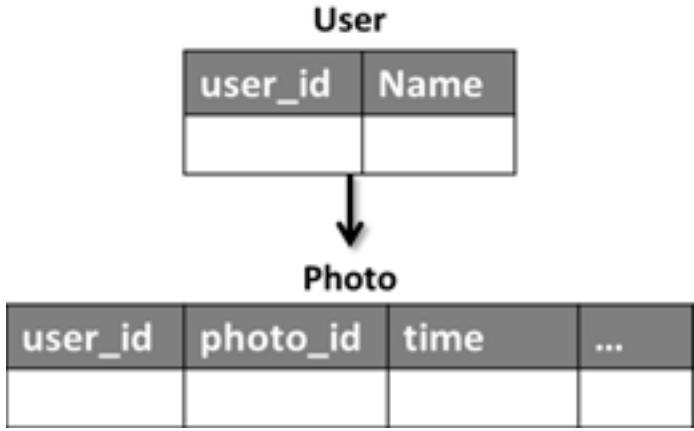
Tree Schema



- Secondary tables store primary keys of parent table as foreign keys
- A given key of the root table (along with all dependent rows from child tables form a “row group”
- Row group becomes the unit of co-location
- Transactions are limited to within a row group.
- Used in ElasTrans and H-Store

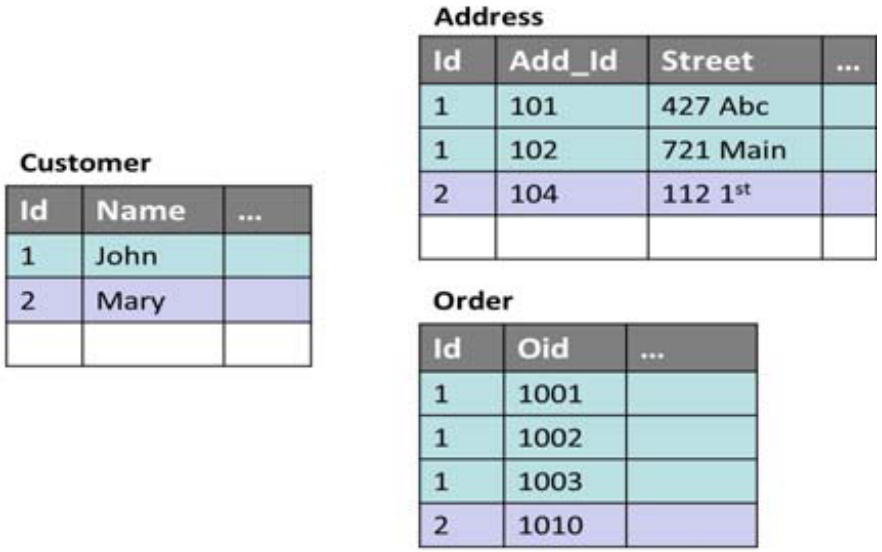
Entity Groups





- Entities are rows in table with properties.
- Each table is either entity group root table or child table
- Several root tables possible.
- A root entity along with all child entities form an entity group.
- Entity group unit of co-location and transactional access
- Used in Megastore

Table Group



Customer

Id	Name	...
1	John	
2	Mary	

Address

Id	Add_Id	Street	...
1	101	427 Abc	
1	102	721 Main	
2	104	112 1st	

Order

Id	Oid	...
1	1001	
1	1002	
1	1003	
2	1010	

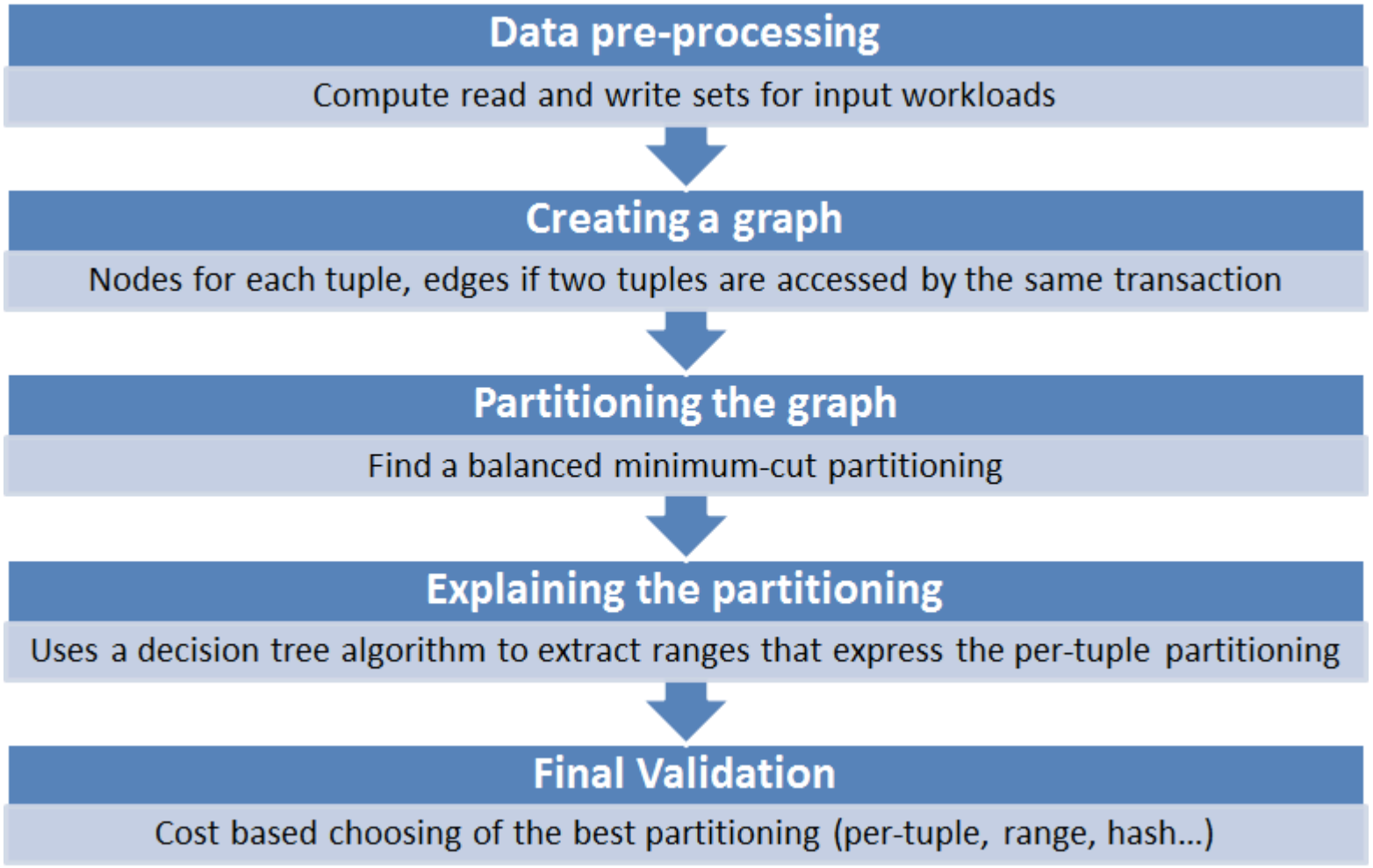
- Each table in a table group has a partitioning key.
- Rows in a table group with the same partitioning key = **row group**.
- Partition database based on row groups
- Transactions limited to row groups.
- Used in Cloud SQL server

Co-locating Data

- Schema based
- schema dictates the nature of multi-key transactions and hence can be used to co-locate the data
- Workload driven partitioning
- Explicitly evaluate the workload to decide on how to partition the data.
- Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. **Schism**: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endowment*, 3(1):48–57, 2010. Cited on page(s) 44, 45, 64
- Used in Relational Cloud – a database as a service system from MIT

- Dynamic Application driven
- Empower users to define dynamic grouping of data based on emerging transaction needs

Schism - Approach



Graph Representation

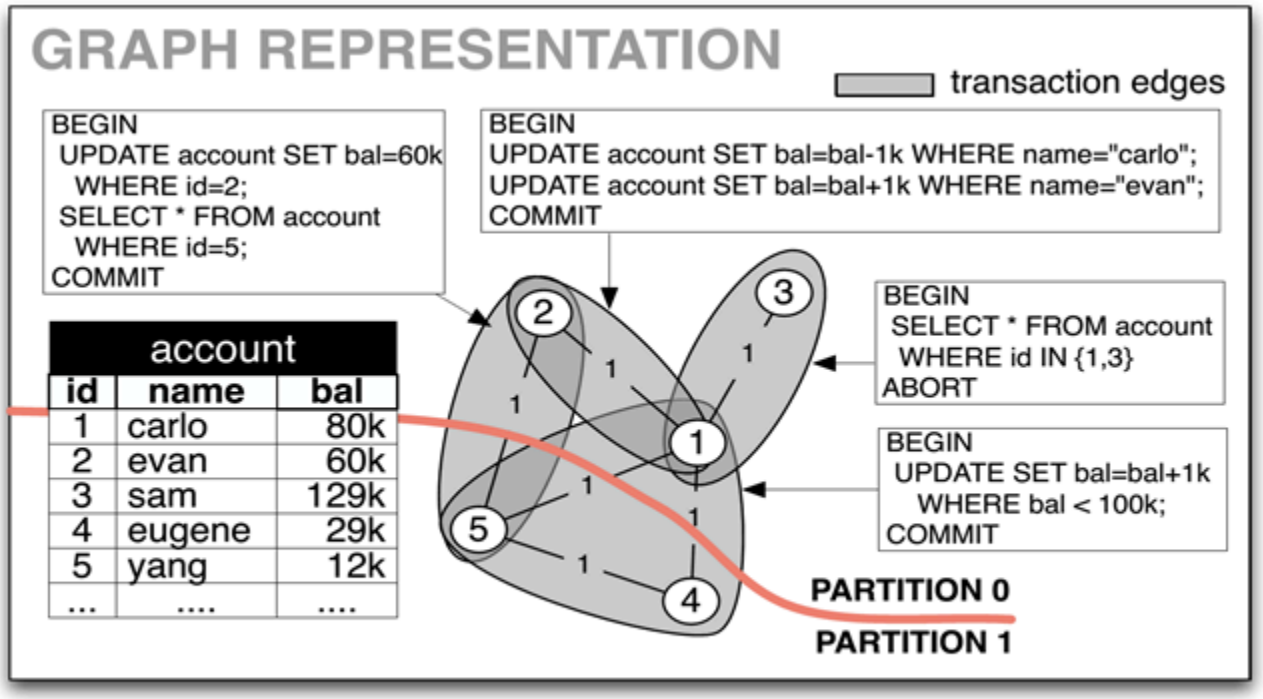


Figure 2: The graph representation

- Each tuple represented as a node
- Edges connect tuples that are used within the same transaction

Graph Replication

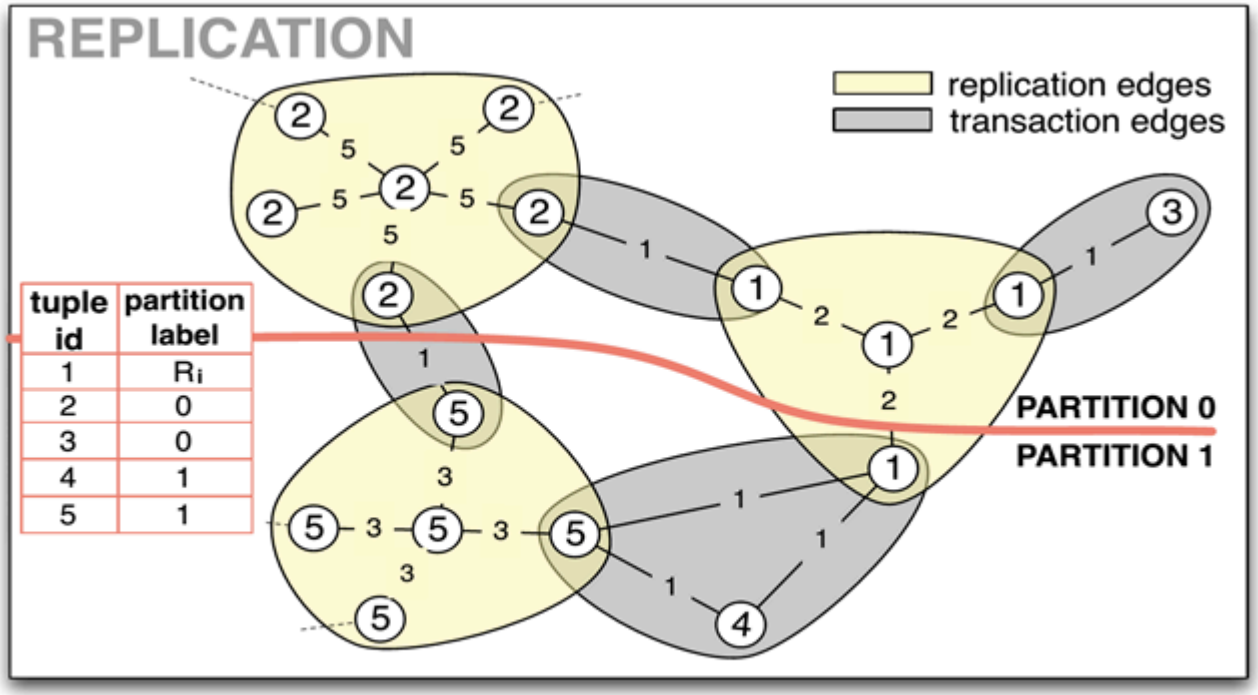


Figure 3: Graph with replication

- Extension: explode nodes to star-shaped configuration of n+1 nodes (n: number of transactions that access)
- Replicated edge weights: #of transactions that update the tuple (why?)
- Node weights:
  - Data-size balancing: tuple size in bytes
  - Workload balancing: #tuple access

Graph Partitioning Method:

- Split graph into k non-overlapping partitions
- **Weight of a partition:**
  - sum of weights of nodes assigned to that partition.
- **Optimization Criteria:**
  - Minimize overall cost of the cut edges(finding a Min Cut)while keeping the weight of partitions within a constant factor of perfect balance
- **Unified representation decides**
  - whether it is better to replicate a tuple across multiple partitions and pay the price of distributed updates
  - or place it in a single partition and pay the price of distributed transaction.
- NP-complete -> heuristics

Storing the partition information

- Unlike hash/range partitioning, since partition based on transaction, we need to store partition information for routing

- **Fine Grained Partitioning:**
  - Store the output of the partitioner in a **lookup table**.
  - Lookup tables are stored as indexes, bit-arrays or bloom-filters.
  - With look-up tables, new tuples are inserted into a random partition until, the partitioning is re-evaluated.

Explanation Phase

- The explanation phase attempts to **find a compact model** that captures the (tuple, partition) mappings produced by the partitioning phase
- Tries to find a set of rules (=ranges)

$$\begin{aligned} (id = 1) &\rightarrow \quad partitions = \{0, 1\} \\ (2 \leq id < 4) &\rightarrow \quad partition = 0 \\ (id \geq 4) &\rightarrow \quad partition = 1 \end{aligned}$$

- One way to compact representation is to learn decision trees using a small sample of (tuple, partition) pairs (**coarse grain partitioning**)
- Data organized based on coarse grained partitioning.
- May result in larger number of distributed transactions

Final Validation

- Compare different solutions and select the final partitioning scheme between:
  - Fine-grained per tuple partitioning
  - Range-predicate partitioning
  - Hash
  - Full-table replication
- Select based on optimization criteria.

Scalability and Robustness

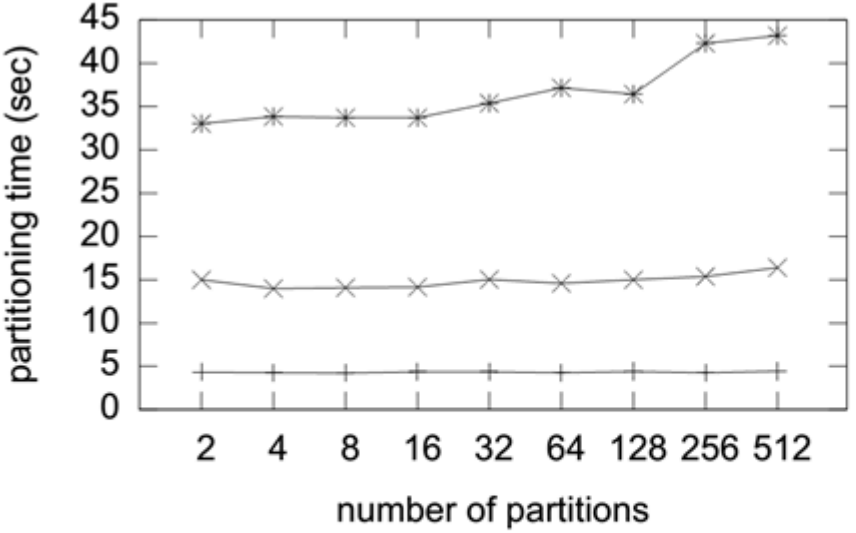


Table 1: Graph Sizes

Tuples	Transactions	Nodes	Edges
2.5M	100k	0.6M	5M
25.0M	100k	2.5M	65M
2.0M	100k	3.0M	100M

Figure 5: METIS graph partitioning scalability for growing number of partitions and graph size.

=> Cost of partitioning increases only slightly with the number of partitions, but strongly with graph size

Optimizations (Heuristics)

- Runtime of graph partitioner increases with graph size -> keep size low
  - Sampling (Transaction and Tuple Level)
  - Blanket-statement filtering (discard statements that scan large portions of a table)
  - Remove tuples that are accessed rarely
  - Connect replica nodes in star-shape
  - Tuple-coalescing

Performance

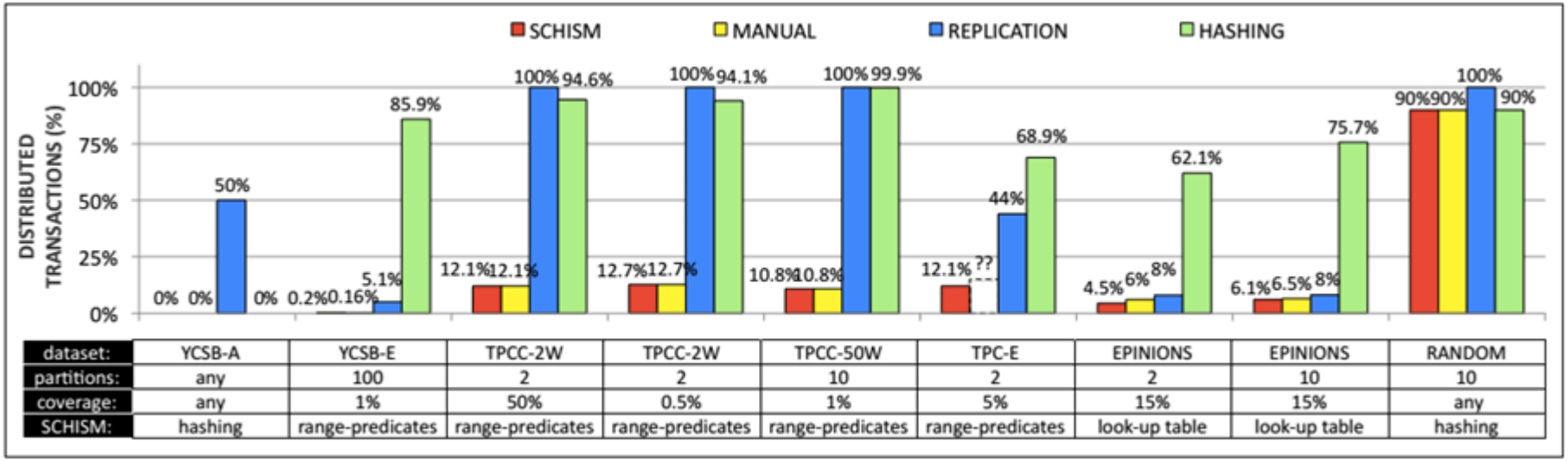


Figure 4: Schism database partitioning performance.

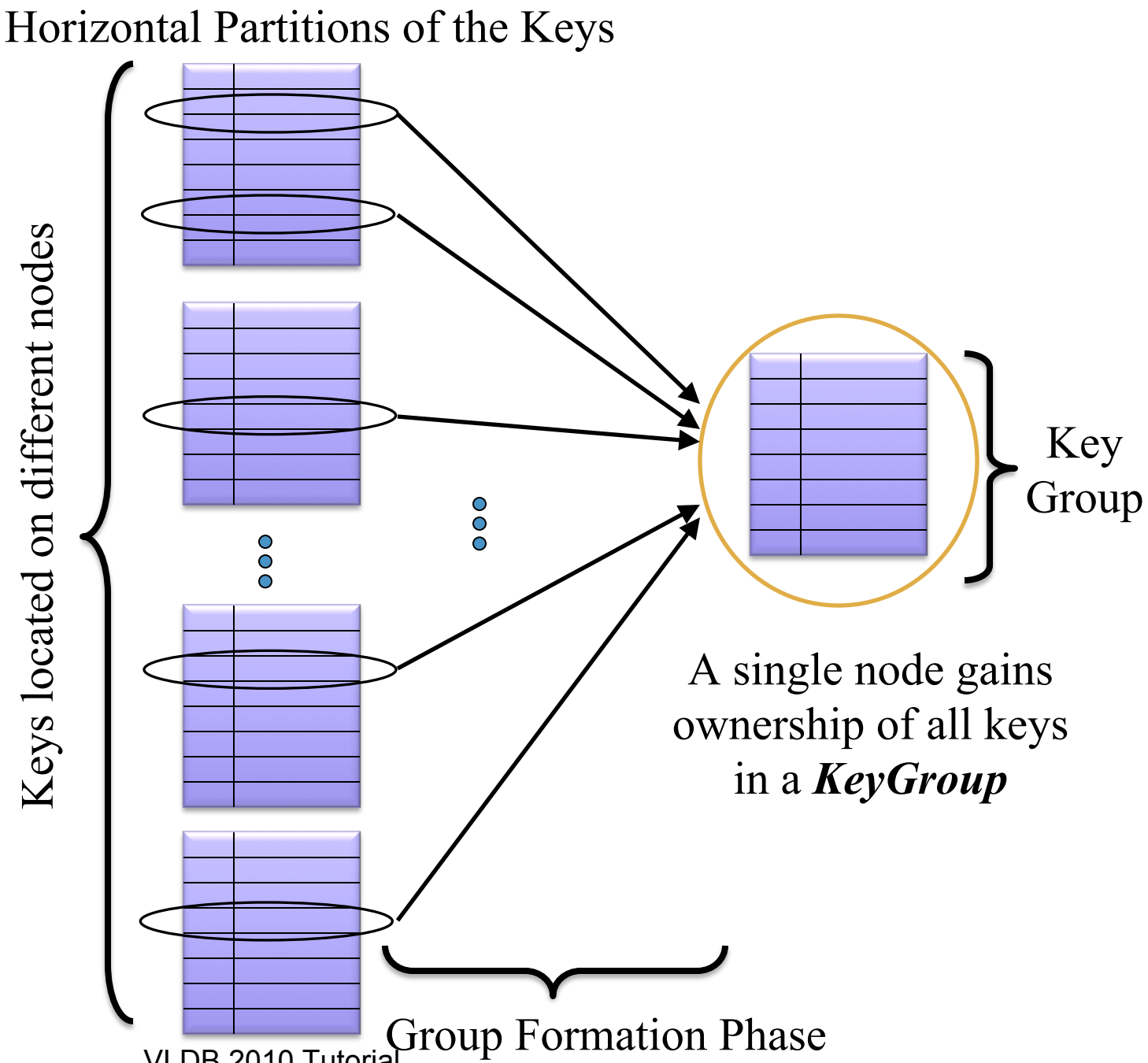
=> Schism is practical, demonstrating **modest runtimes**, **excellent partitioning performance** and ease of integration into existing databases

Co-locating Data

- Schema based
- schema dictates the nature of multi-key transactions and hence can be used to co-locate the data
- Workload driven partitioning
- Explicitly evaluate the workload to decide on how to partition the data.
- Dynamic Application driven
- Empower users to define dynamic grouping of data based on emerging transaction needs
- Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In *Proc. 1st ACM Symp. on Cloud Computing*, pages 163–174, 2010b.

Key Group Abstraction

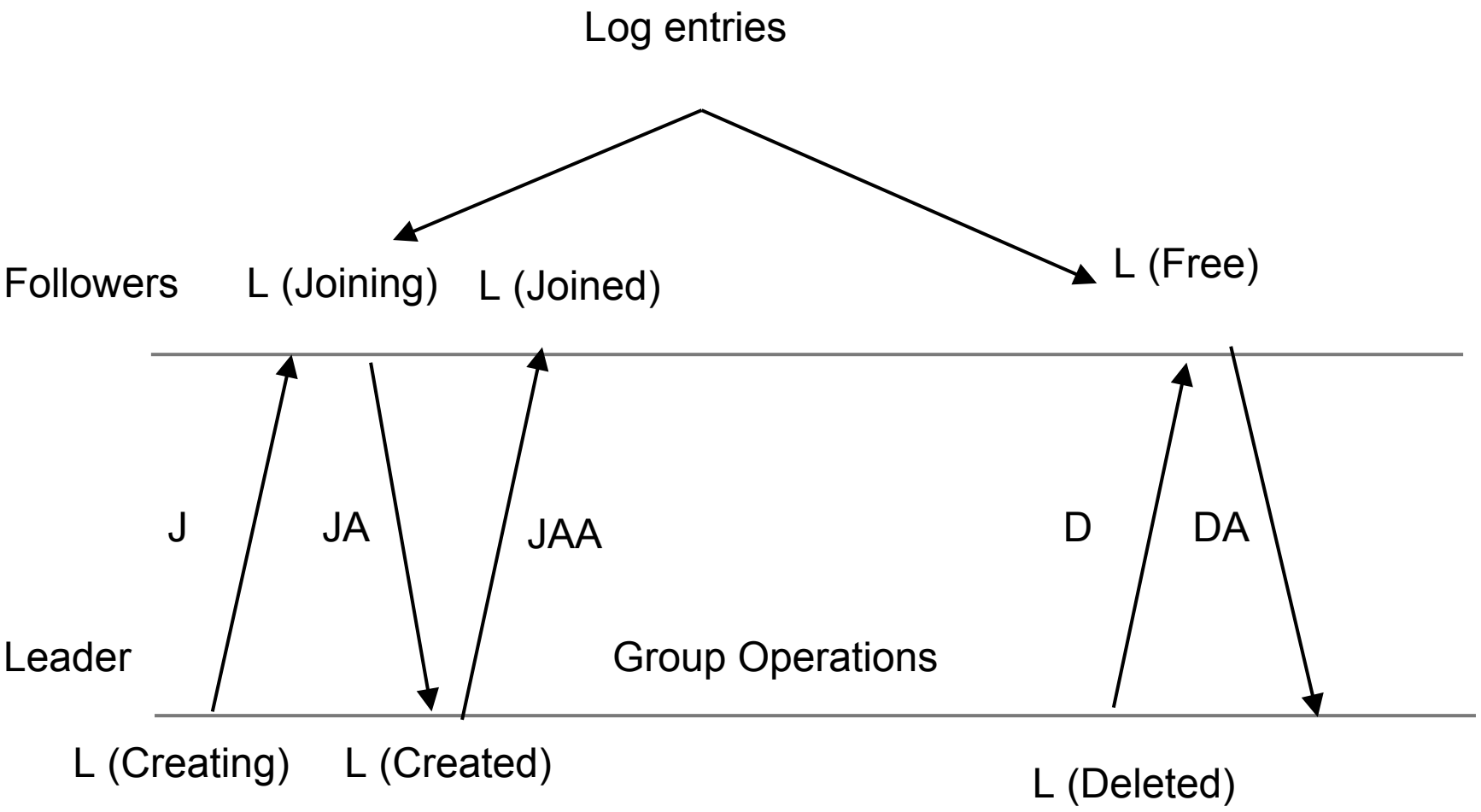
- Define a granule of on-demand transactional access
- Applications select any set of keys to form a group
- Data store provides transactional access to the group
- Non-overlapping groups


Horizontal Partitions of the Keys
Keys located on different nodes
Key Group
A single node gains ownership of all keys in a KeyGroup
Group Formation Phase

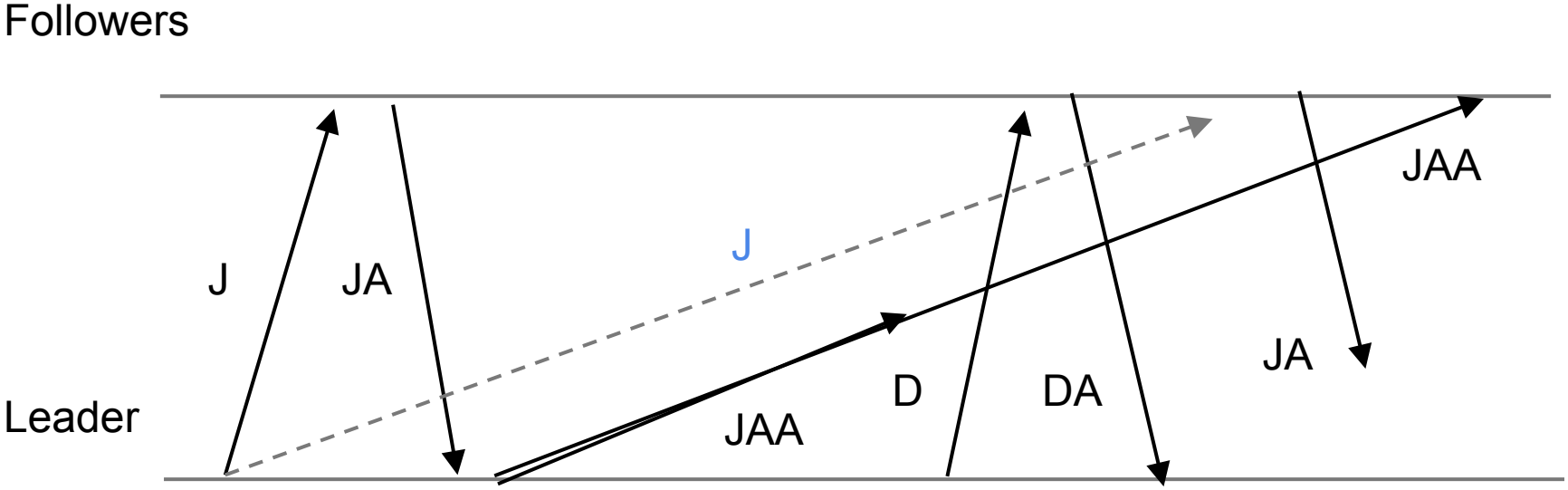
Key Grouping Protocol

- Conceptually akin to “locking”
- Allows collocation of ownership at the **leader**
- Leader is the gateway for group accesses

Key Grouping Protocol (2)



Need to be careful -- Ghost Groups!



G-Store Paper takes care of such problems…

- Group Creation
  - J (verify, log, retry)
  - JA (verify, yield-id, retry)
  - JAA (verify, log)
- Group Deletion
  - D (verify, log, retry)
  - DA (verify, log)

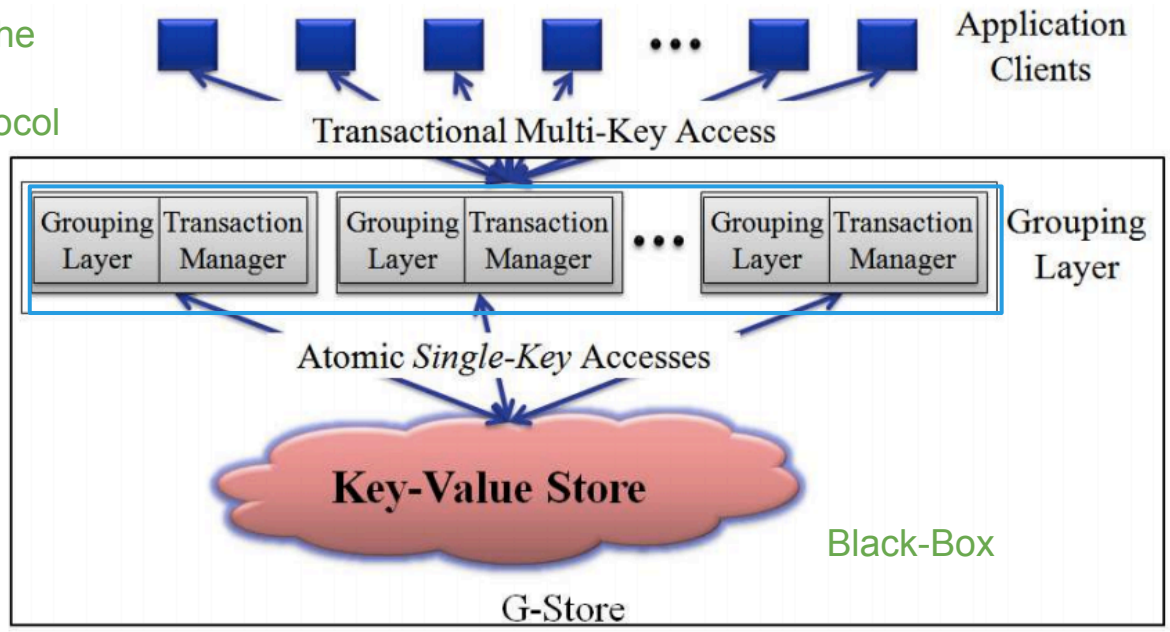
Other issues

- Message Loss, reordering or duplication
  - Timers and retries
  - Yield id (unique within group)
  - Group id
- Concurrent Group deletes
- Recovery from Node failures
  - Write-ahead logging for recovery and durability
  - Group safety
  - Extended availability

G-store Implementation
(Client Based Implementation)

1) GL executes the Key-Store Grouping protocol

2) Acquire ownership for access to the group

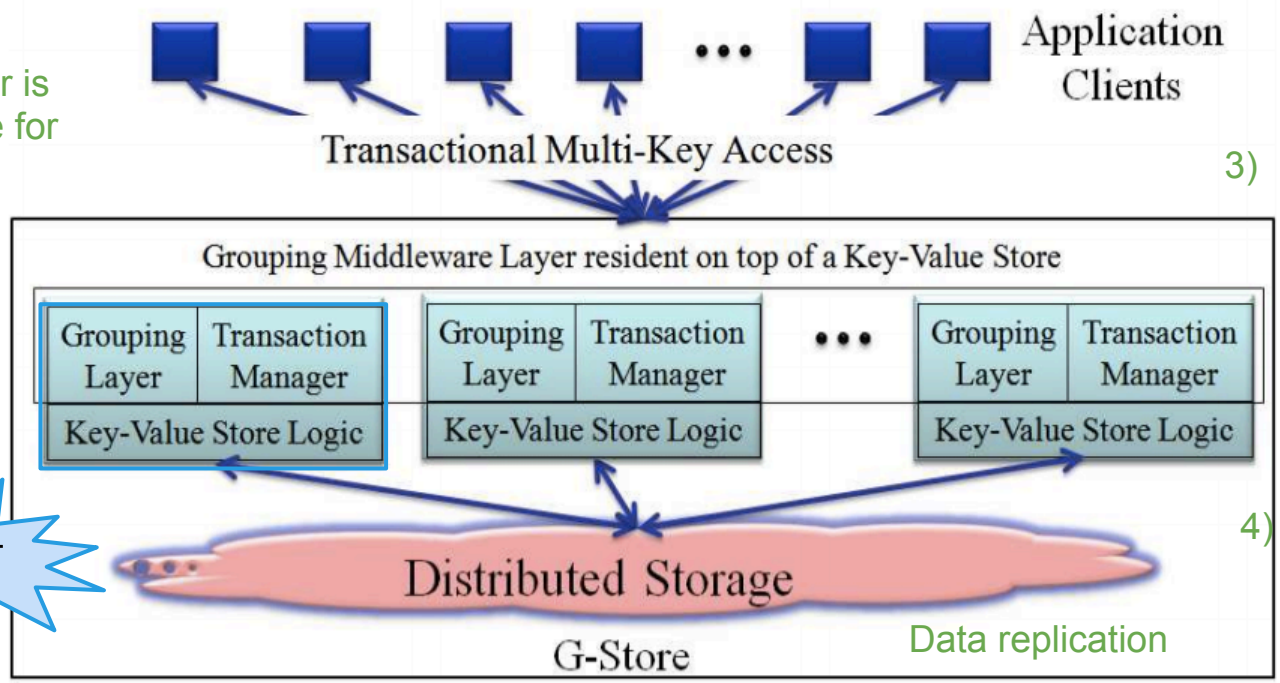


4) TM caches the contents of the keys and executes transactions on the group

3) If the key is available, test and set operation obtains the lock and stores the group id

G-Store Implementation – (Middleware Based Implementation)

2) Each server is responsible for certain key ranges



1) Tablet server

HBase-clone

3) TM provides access to the keys guaranteeing different levels of consistency

4) Logic does not need to know about the existence of groups

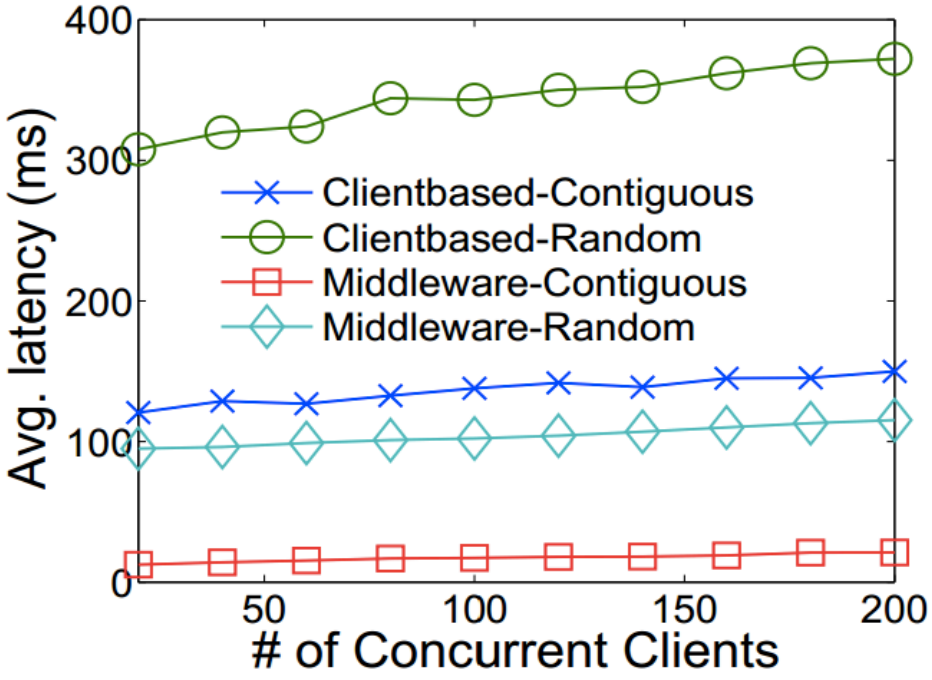
Data replication

Experimental Setup

- Key Grouping protocol as well as a G-Store has been implemented in Java
- Used HBase as the Key-Value store
- An application benchmark using online multi-player gaming application
- HBase cluster nodes : 10
- **# of concurrent clients : 20 to 200**
- **# of keys in a group : 10 to 100**
- Data size : ~1TB

Group Creation Latency

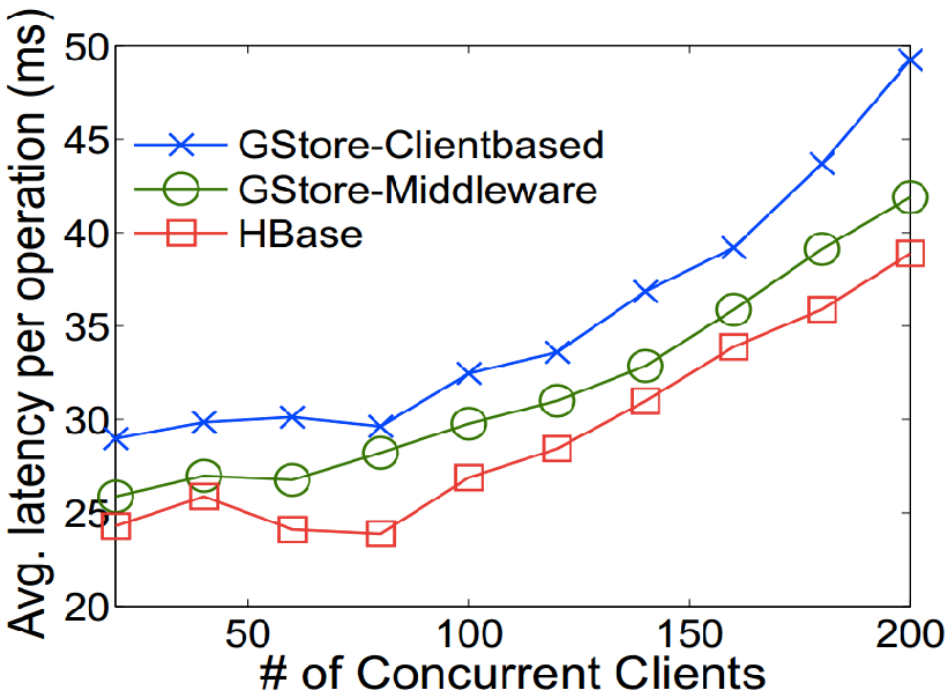
- 100 keys per group
- Random selection of keys
- The middleware based approach outperforms the client based approach owing to the benefits of batching the protocol message directed to the same node



★ Contiguous : keys in a group are selected such that they form a range of keys
★ Random : keys are selected in any arbitrary order

Group Operation Latency

- 100 keys per group
- Avg.latency includes group formation, deletion and operations
- The grouping layer introduces very little overhead(11-30%) when compared to the baseline



G-store Summary

- Key Group as the granule of on-demand transactional access
- Grouping Protocol is resilient to various types of failures
- G-store can be client-based on middleware based
- Overhead of executing G-store is minimal

Multi-Key Transactional Systems ..

- **Collocating data to avoid distributed transaction**s.
- Many aproaches: Schema based, Access pattern based, Dynamic groups

- **Many systems based on these approaches**
- G-Store, Megastore, Cloud SQL server, Relational Cloud, Hyder, …

- **Other design choices that differentiate these systems:**
- **How they implement transactions** – locking protocol, multiversion concurrency, optimistic techniques, logging strategy used, ..
- **I**
- **nterface with storage system** – coupled (classic design choice), or decoupled (ownership for implementing transactions separated from underlying data management).

- **Replication strategy used** – primary copy-based or multi-master, synchronous or asynchronous, explicit or implicit (at the storage layer level)

Storage Services

Coupled Storage

- Data items co-located within a transaction are also physically co-located at the server serving transactions on the partition

Decoupled Storage

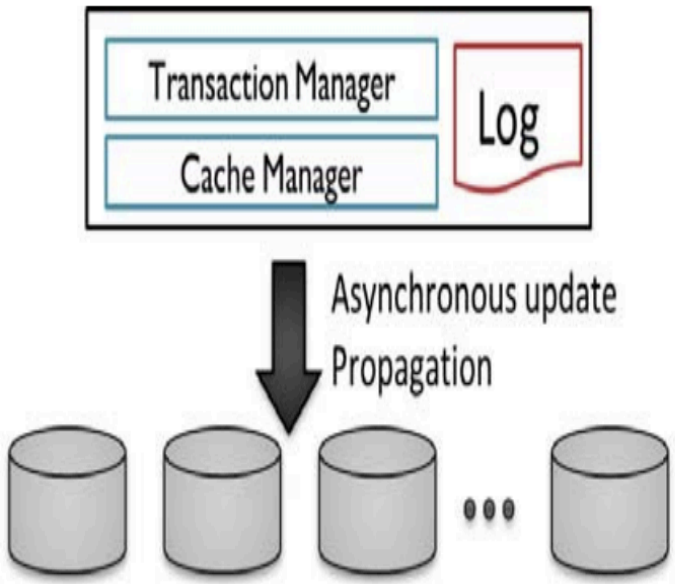
- Data ownership is decoupled from the physical storage of data

Many Systems decided for the decoupled solution and use an underlying Key-value store

Decoupled Storage

- G-store (Decoupled)
  - Uses standard Key Value Stores. Version built on H-base
- Deuteronomy (Decoupled)
  - Data Component: atomic record operations
  - Transaction Component: ACID properties
- CloudTPS (Decoupled)
  - Implemented on BigTable and SimpleDB
- Megastore (Decoupled)
  - Built on top of BigTable

Update propagation



Replication Strategies

Explicit Replication

Design the transaction manager to be cognizant of replication such that updates made by the transactions are explicitly replicated while transactions are executing

Implicit Replication

- The transaction execution logic is unaware of any replication protocol
- G-Store, Deuteronomy, CloudTPS, Megastore

Concurrency Control

- optimistic concurrency control – G-store
- Multi-version concurrency control + modified Paxos fore replica consistency – Megastore
- Locking – Cloud SQL server

Lots of variants to the basic protocols and tweaks to make it better

Example Systems

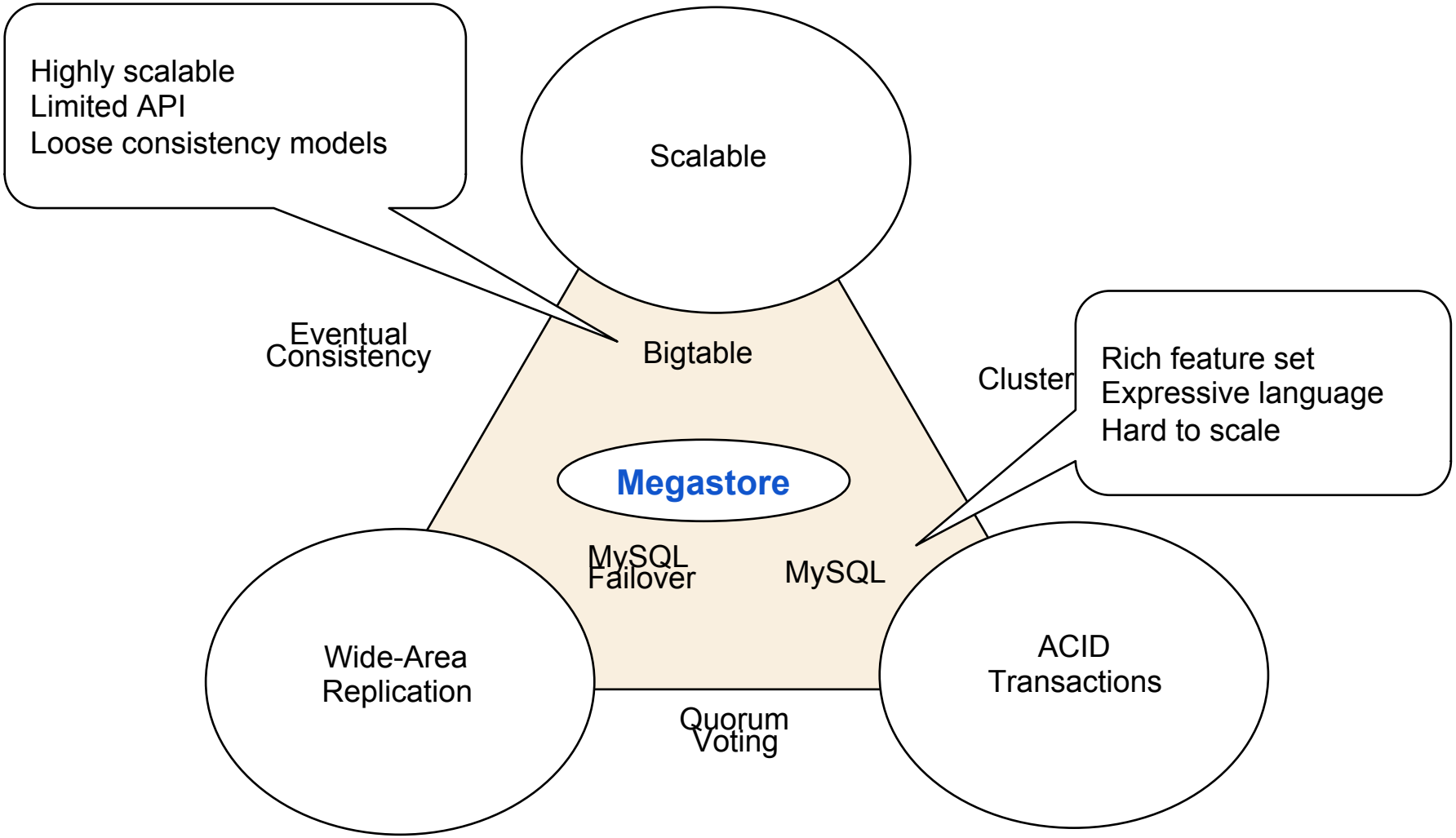
Megastore, Deutronomy, hyder, Cloud TPS, Cloud SQL server

Google Megastore – Motivation

- Storage requirements of today's interactive online applications
  - –Scalability (a billion internet users)
  - –Rapid Development
  - –Responsiveness (low latency)
  - –Durability and Consistency (never lose data)
  - –Fault Tolerant (no unplanned/planned downtime)
  - –Easy Operations (minimize confusion, support is expensive)

- These requirements are in conflict!

Technology Options



Megastore

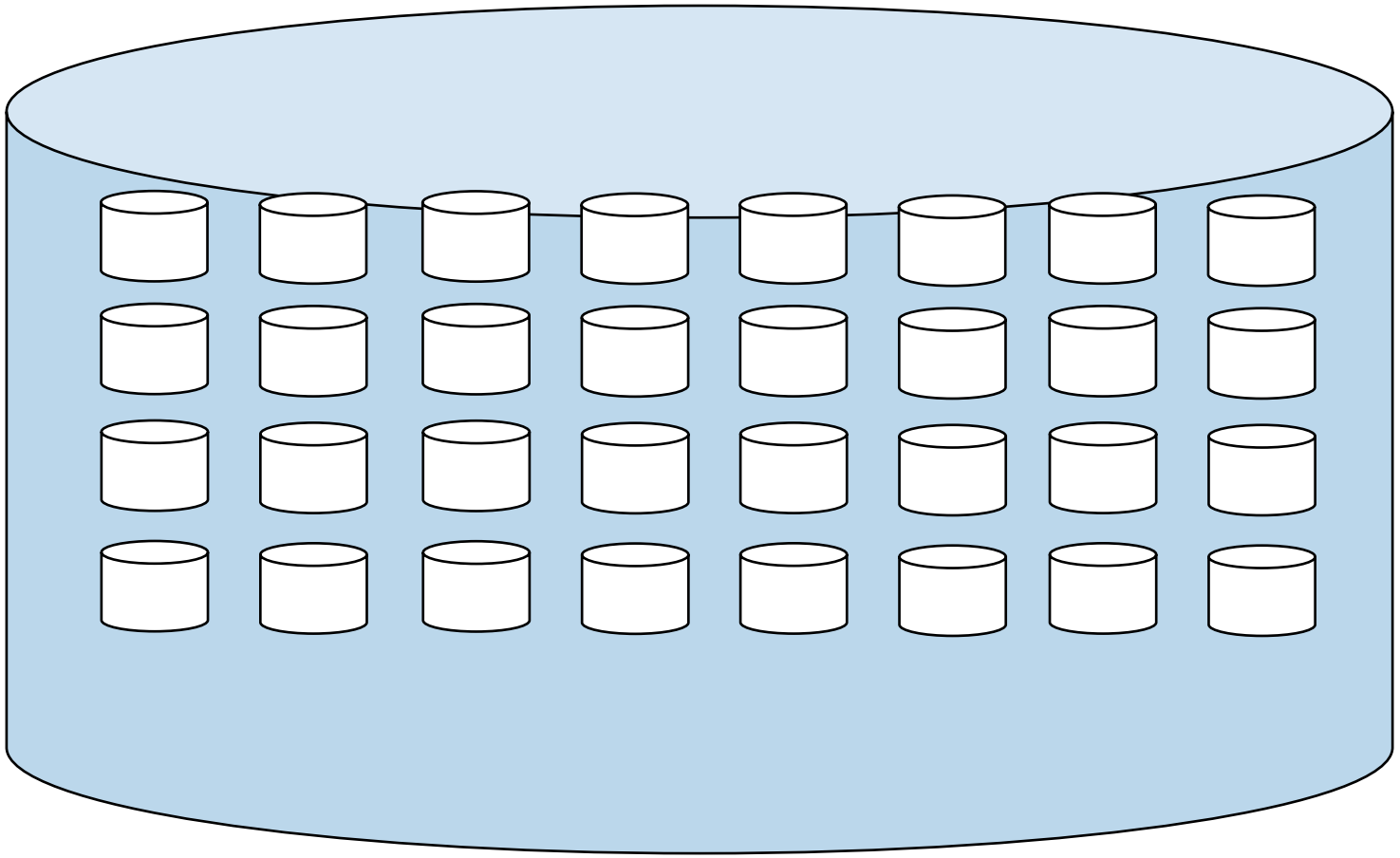
- Started in 2006 for app development at Google
- Service layered on:
  - –Bigtable (NoSQL scalable data store per datacenter)
  - –Chubby (Config data, config locks)
- Turnkey scaling (apps, users)
- Developer-friendly features
- Wide-area synchronous replication
  - –Partition by “Entity Group”

Data Model

- Between abstract tuples of RDBMS and concrete row-column storage of NoSQL
- Tables are *entity group root tables* or *child tables*
- Entity Group – consists of a *root entity* along with all *child entities*
- There can be several root tables – leading to several classes of *Entity Groups*

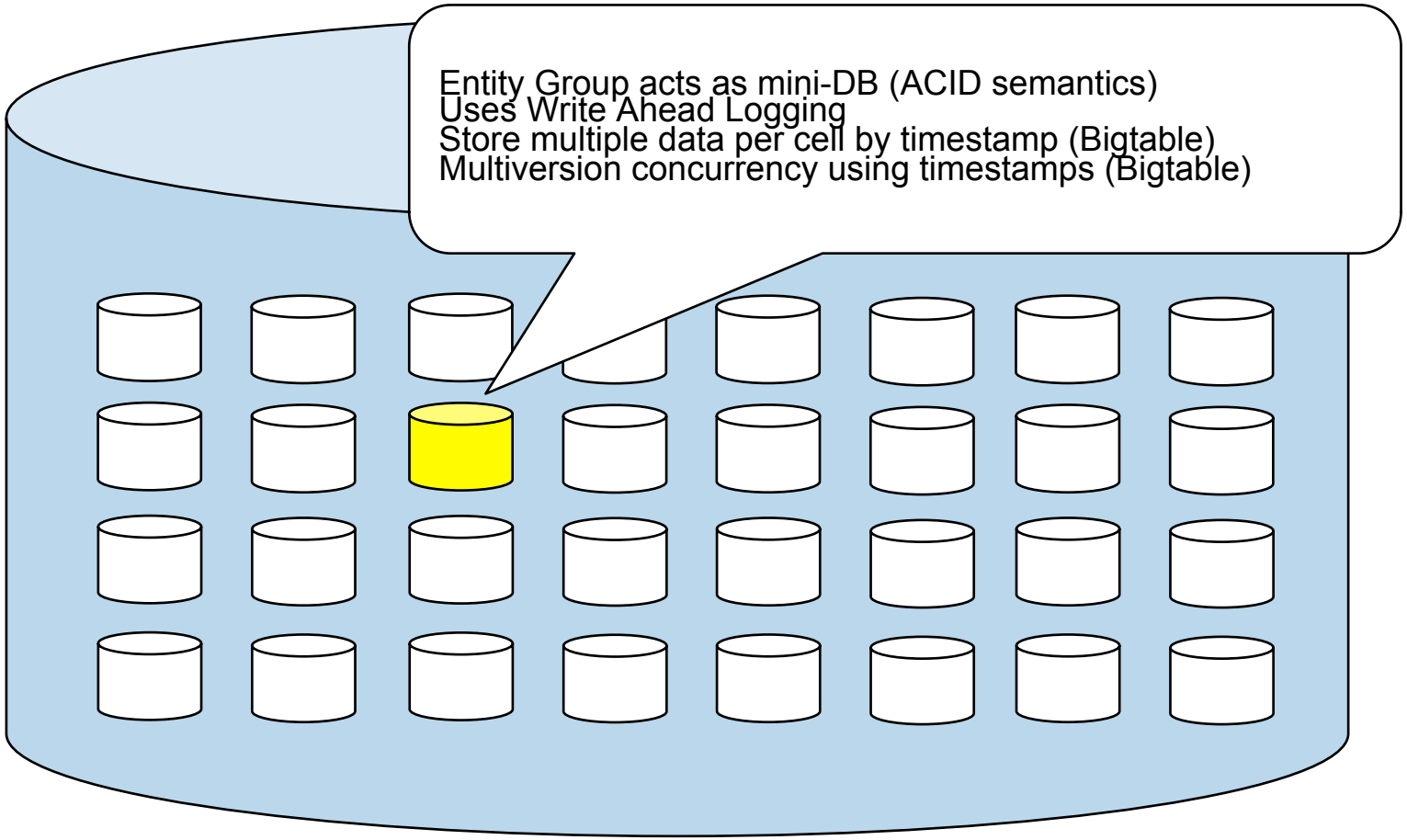
Entity Groups

- Entity groups are sub-databases



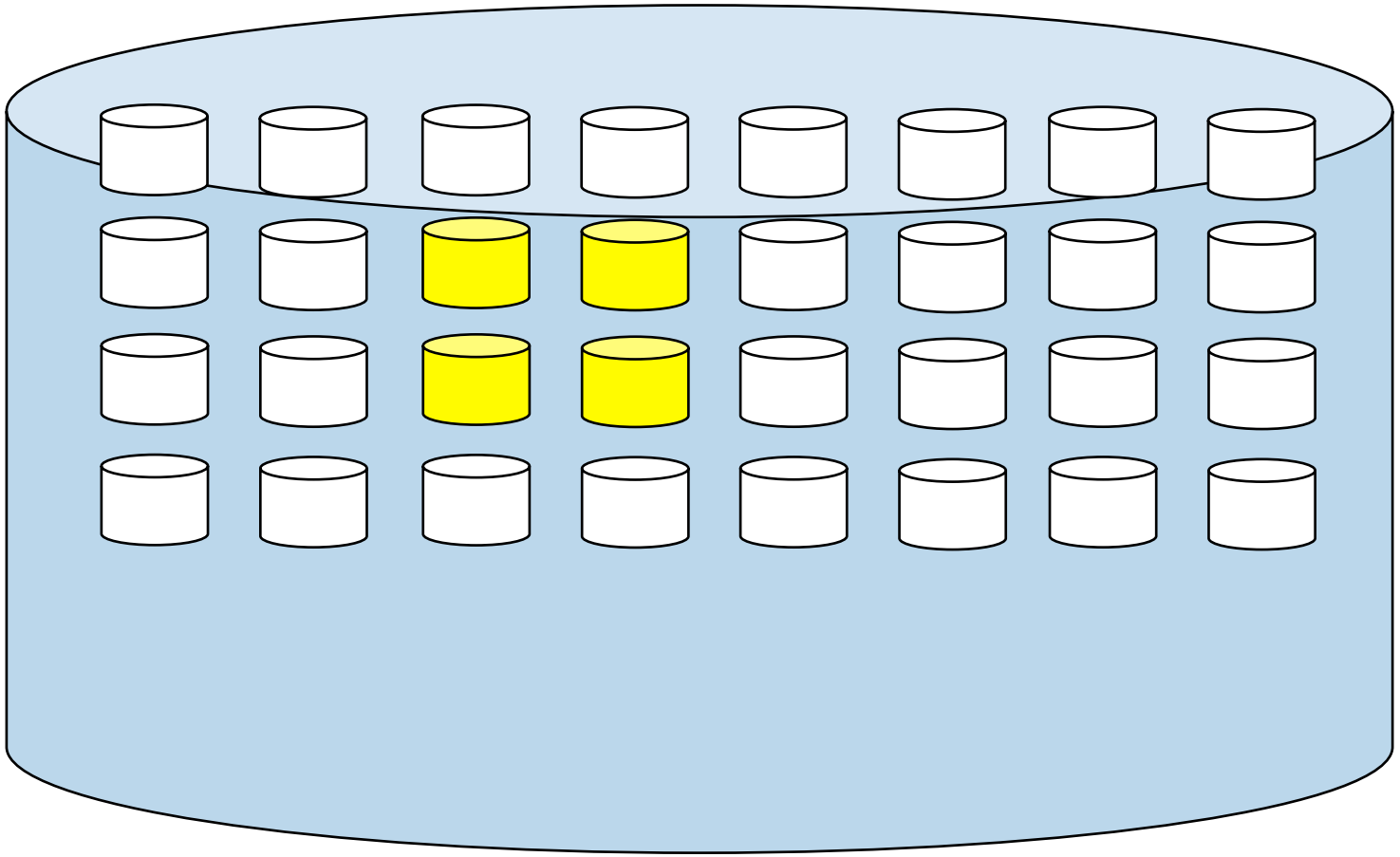
Entity Groups

- Cheap transactions within an entity group (common)



Entity Groups

- Expensive or loosely-consistent operations across Entity Groups (rare)

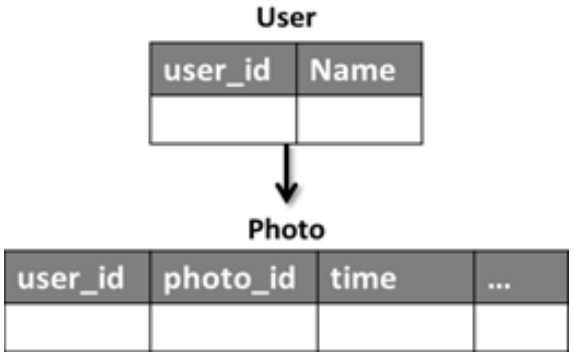


Entity Group Examples

Application	Entity Groups	Cross-EG Ops
Email	User accounts	none
Blogs	Users, Blogs	Access control, notifications, global indexes
Social	Users, Groups	Messages, relationships, notifications

Mapping Entity Group to BigTable

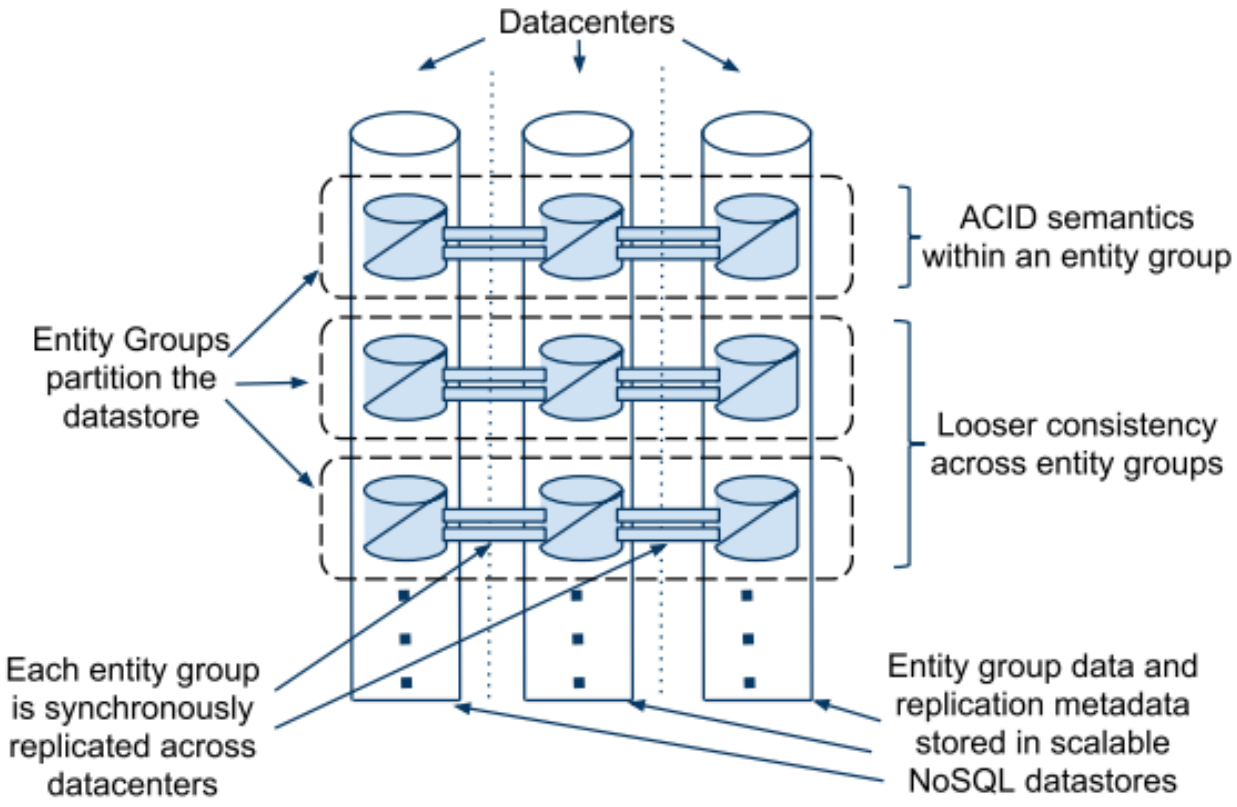
	Row key	User.Name	Photo.Time	Photo.Tag	Photo.URL
Entity group 1	501	John			
	501,101		12:34:56	Pisa, Italy	http://img.ur/asjkh
	501,102		12:56:34	Rome, Italy	http://img.ur/KGGsa
Entity group 2	551	Jane			
	551,151		11:22:33	New York, USA	http://img.ur/hgFDF
	551,152		11:33:44	Santa Barbara, USA	http://img.ur/BBA7t
	551,153		11:44:55	Seattle, USA	http://img.ur/kajhs1



Hierarchical structure of Entity group exploited to co-locate entity group in big table.

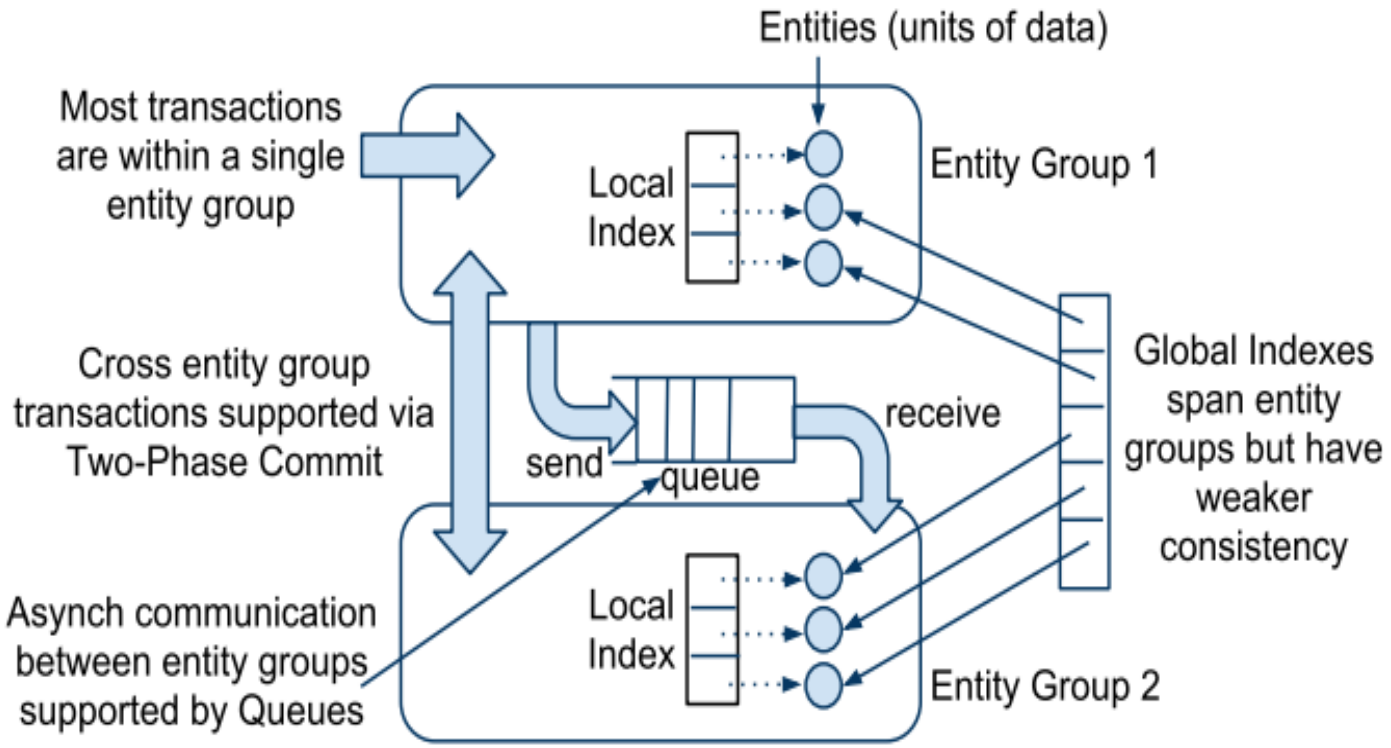
- **Column names in big table** = <table name + column name> in Megastore (***avoids collision***)
- **Row id for child group** = <row id of root entity + child row id> in Megastore (**ensures *locality of entity group within big table***).

Entity Groups are Co-Located to same partition



- **ACID transactions** using **MVCC** for transactions within entity group
- Each entity group has an associated **write ahead log**
- **Logs replicated** across datacenter synchronously using modified **Paxos**

Megastore supports both Transactions within Entity Group and across entity groups



How do transactions within Entity Group Work?

- Transactions acquire a read time stamp (***current position of the log***)
- Perform read and write operations (write operations not written yet to the underlying KV store).
- Once finished, transaction tries to commit by claiming the position in the log.

Commit Processing of a Transaction within Entity Group.

- If current log > transaction timestamp, ABORT.
- Transaction timestamp is log value when transaction began.
- Only one writer at a time, no concurrency amongst writers -- someone else beat this transaction for the log position.
- Else, transaction tries to commit by claiming the next log position and writing its log (using Paxos – discussed next).

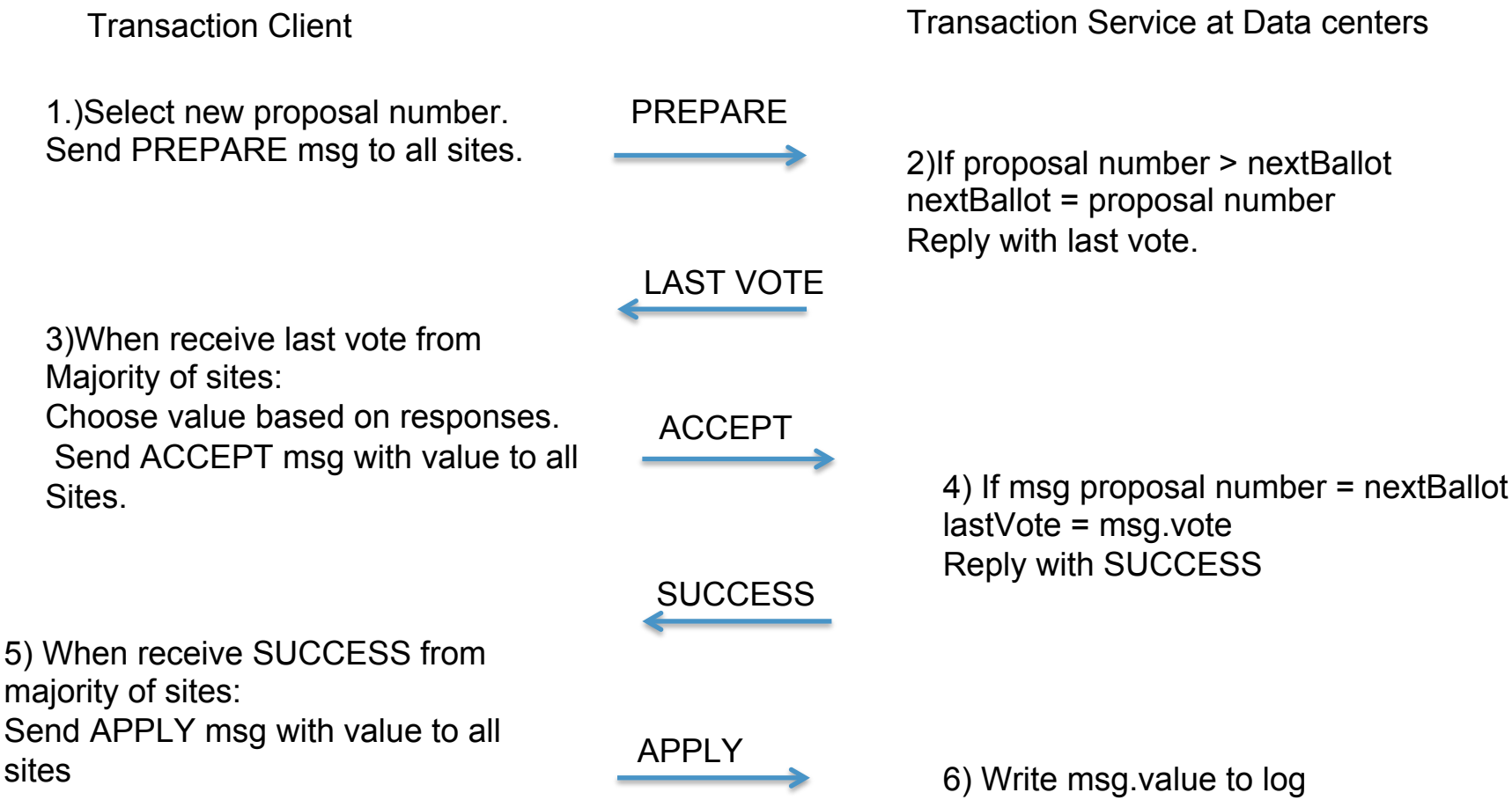
- If commit goes through, the log is written and it is reflected asynchronously to the KV store

- Read only transactions can read any version of the underlying data.

How does a transaction Commit?

- To commit, transaction tries to claim the log position to write the log record
- Other replicas at other data centers may be trying to concurrently claim the log position as well.
- Paxos used for agreeing on who gets to claim the log position.
- At the end of Paxos, the transaction learns the winner for the log position
- If the winner is the transaction itself, it commits, else, it aborts.
- Paxos protocol used for each log position

Basic Paxos for Agreeing on Log Position



Optimizations…

- Basic Paxos is too expensive (5 rounds of messages)

- Many optimizations: e.g., Mastering.
- one replica designation as master.
- client skips PREPARE phase, and simply sends proposed value to master
- Master decides if value accepted and in which order

- Megastore does not use a single master. Instead designates a leader per log position
- Leader of given log position is the site that won the last log position..

Updating the underlying KV store

Once a log position has been agreed and log record written to the replica, the transaction commits.

Updates in the log record asynchronously migrate to the underlying KV store

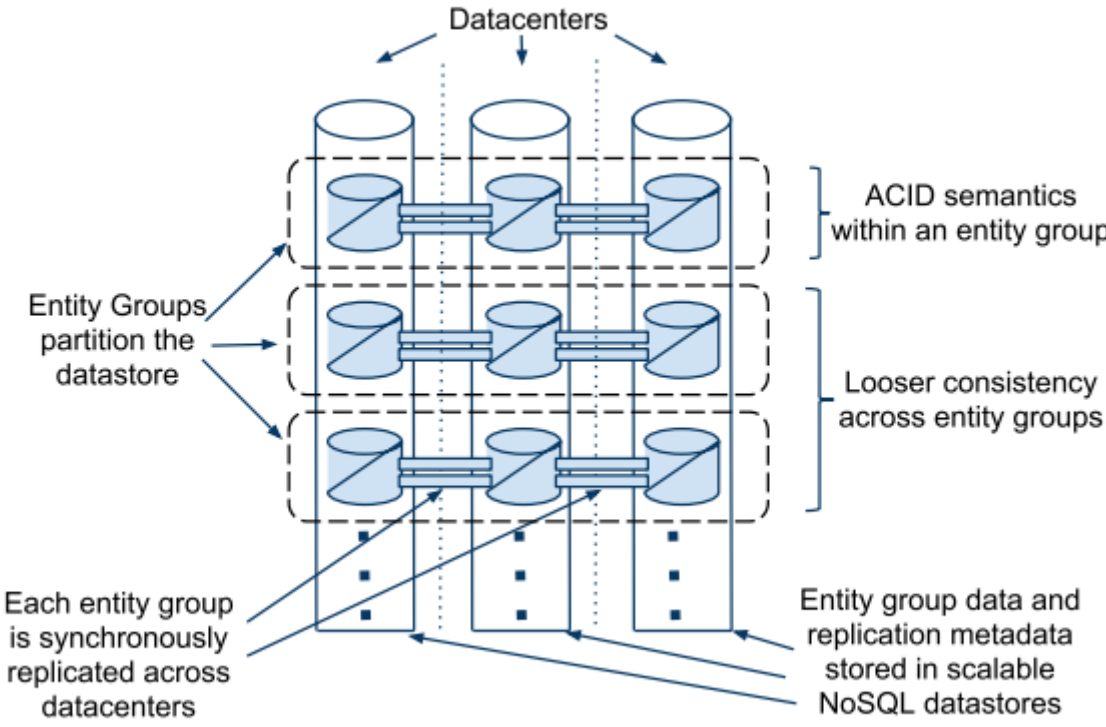
Note that if KV store out of synch with the log, readers will read stale data.

Read protocols designed to prevent such a situation in case readers need current data.

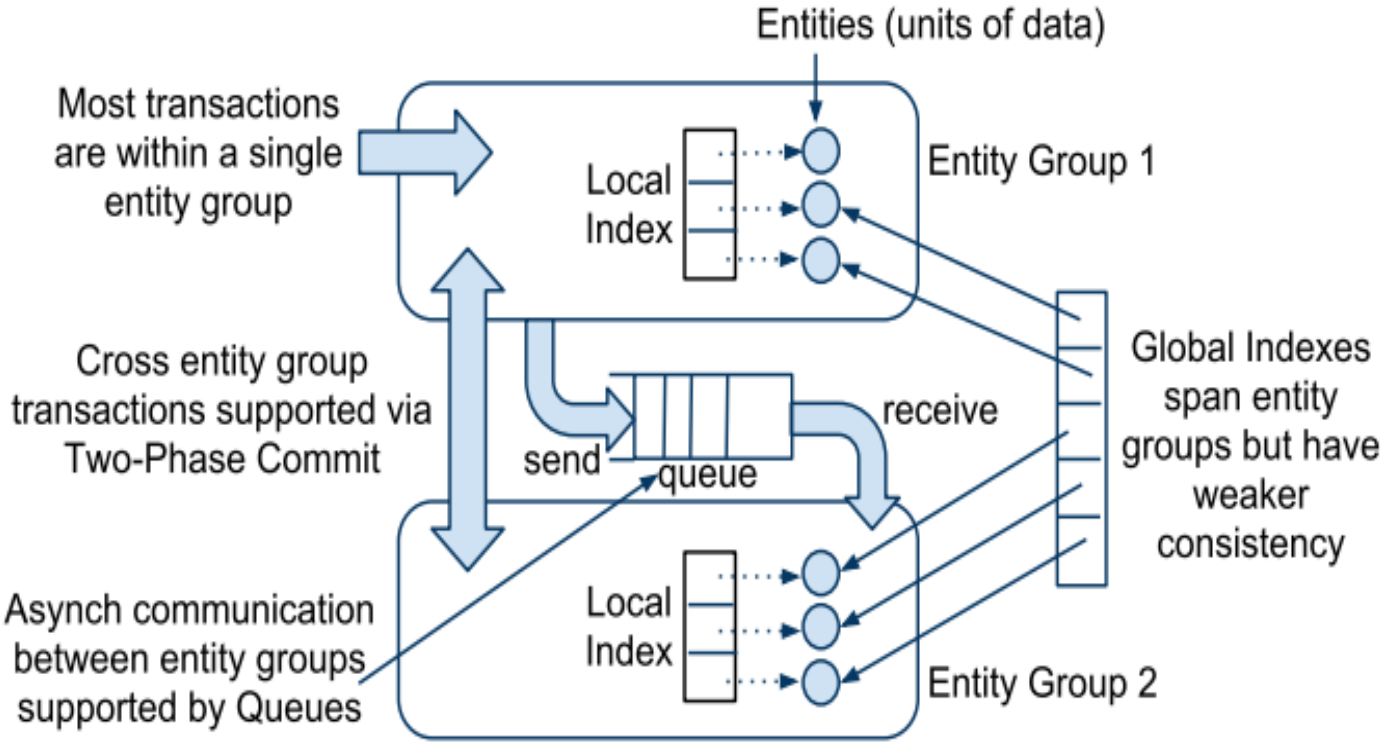
Achieving Technical Goals (I)

- Scalability
  - –Bigtable within datacenters – easy to add EGs (storage, throughput)
  - –Performance maximized by partitioning based on EGs
  - –Transactions within an EG – single phase using Paxos
  - –Transactions across entity groups – two phase using Asynchronous Message Queue

- Availability
  - Fault Tolerance through replication
  - Fault Tolerant log replication of logs (adapted from Paxos)



Achieving Technical Goals (II)



- ACID transactions
  - –Write-ahead log per Entity Group
  - –2PC or Queues between Entity Groups
- Wide-Area replication
  - –Paxos with tweaks for optimal latency

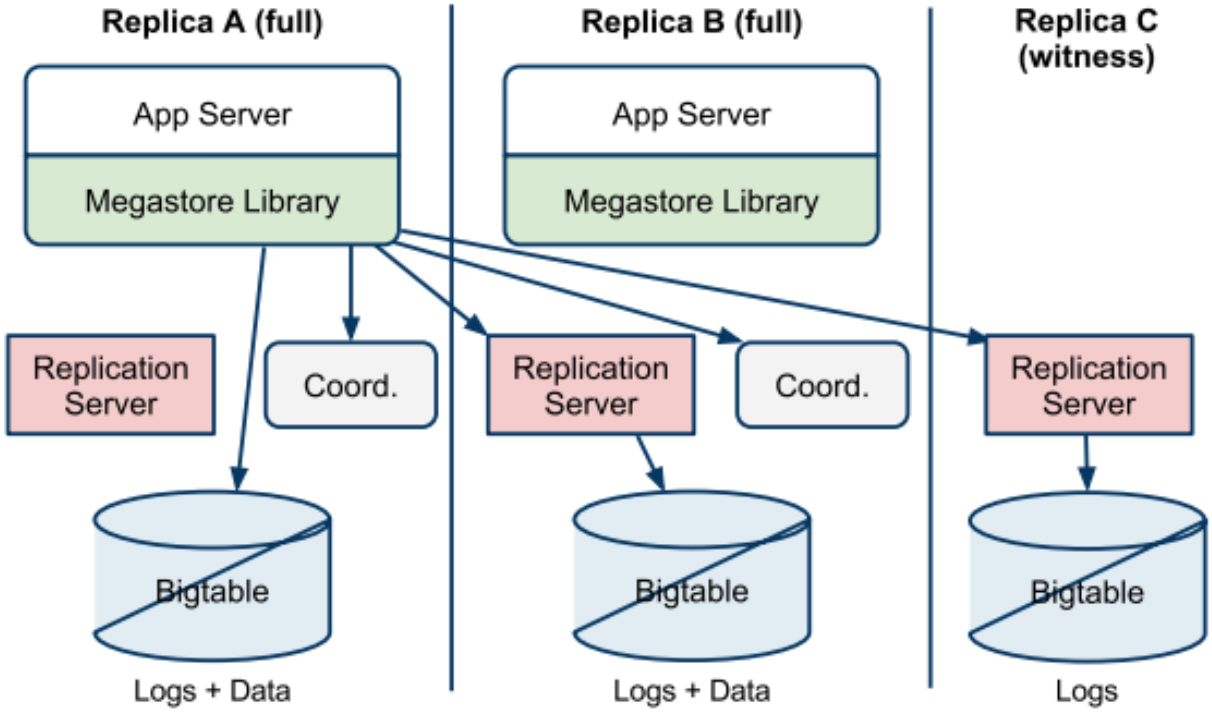
Paxos and Megastore

- Basic Paxos not used (poor match for high-latency links)
  - –Writes require at least two inter-replica roundtrips to achieve consensus (prepare round, accept round)
  - –Reads require one inter-replica roundtrip (prepare round)
- Approaches using a Master replica
  - –Master participates in all writes (state is always up-to-date)
  - –Master serves reads (current consensus state) *without additional comm*
  - –Writes are single roundtrip – piggyback *prepare* for next write on *accepted*
  - –Batch writes for efficiency
- Issues with using a Master
  - –Need to place transactions (readers) near master replica to avoid latency
  - –Master must have sufficient processing resources (side effect: replicas waste resources since they must be capable of becoming masters)
  - –Master failover requires lots of timers and a complex state machine (side effect: user visible outages)

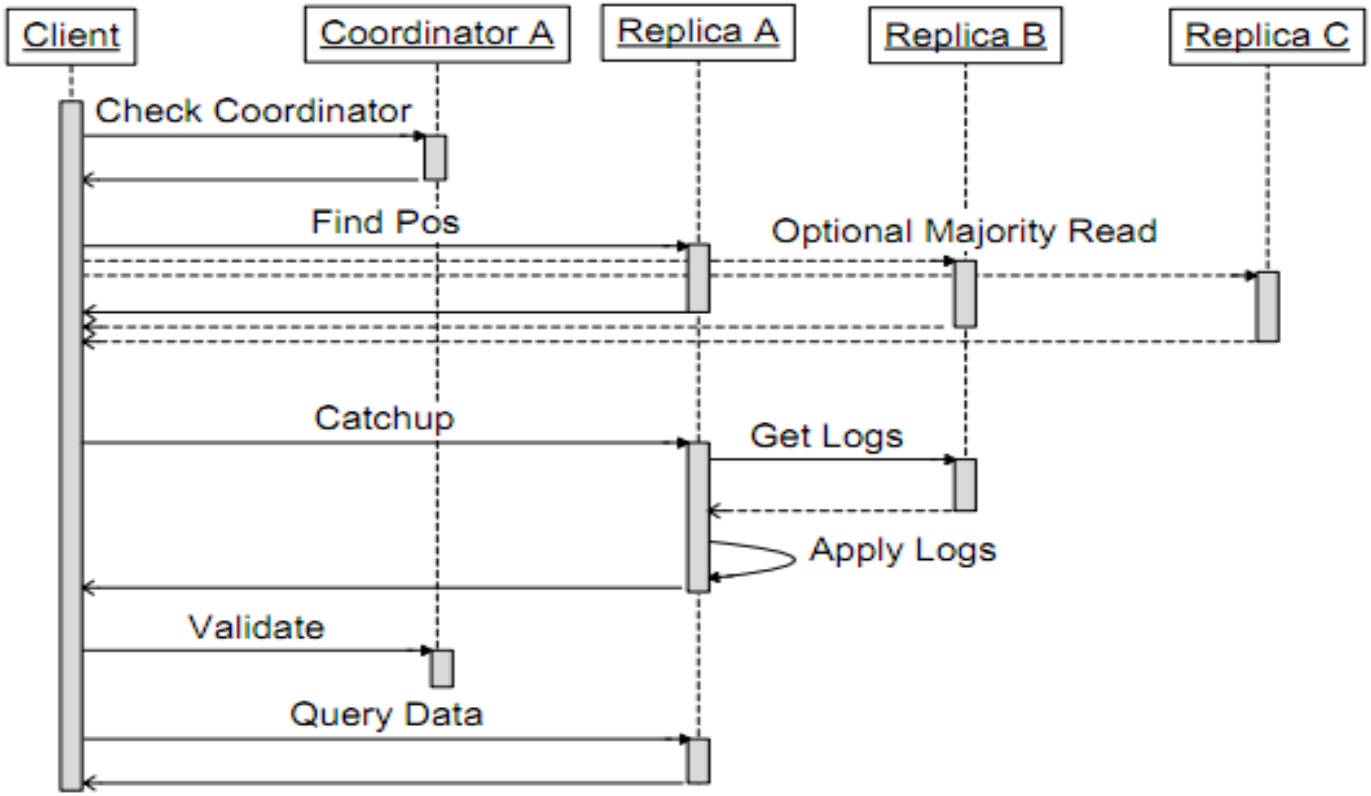
Megastore's Tweaks

- Coordinators
  - –Tracks set of entity groups for which its replica has observed all Paxos writes
- Fast Reads
  - –Local reads from *any* replica avoid inter-replica RPCs
  - –Yield better utilization, low latencies in all regions, fine-grained read failover, simpler programming experience
- Fast Writes
  - –Uses same pre-preparing optimization as Master approaches (*accepted* implies next *prepare*)
  - –Uses leaders (*coordinators*) instead of masters and runs a Paxos instance for each log position – leader arbitrates which writer succeeds
- Replica Types
  - –*Witness Replicas*: participate in voting (tie-breakers) and store log entries (no data)
  - –*Read-only Replicas*: non-voting replicas containing snapshots

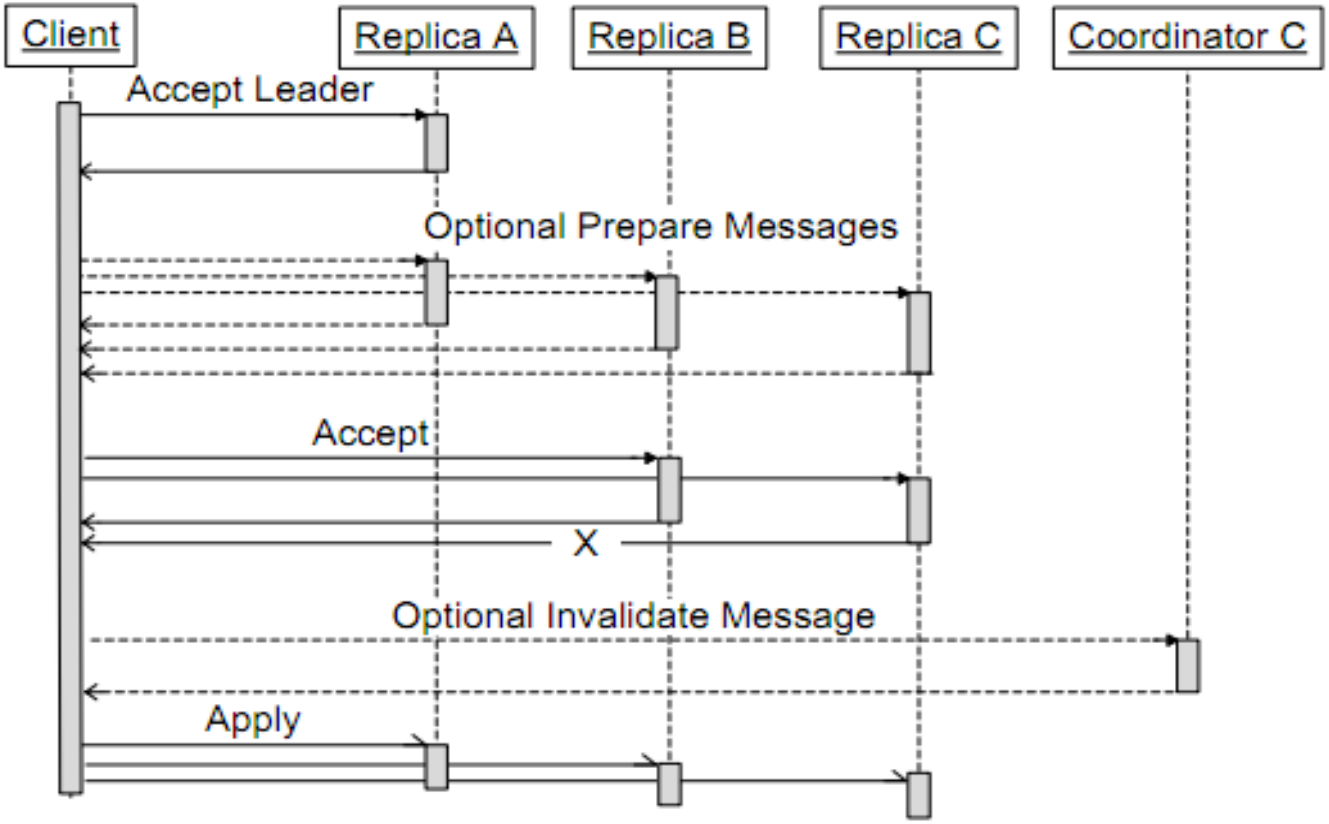
Megastore Architecture



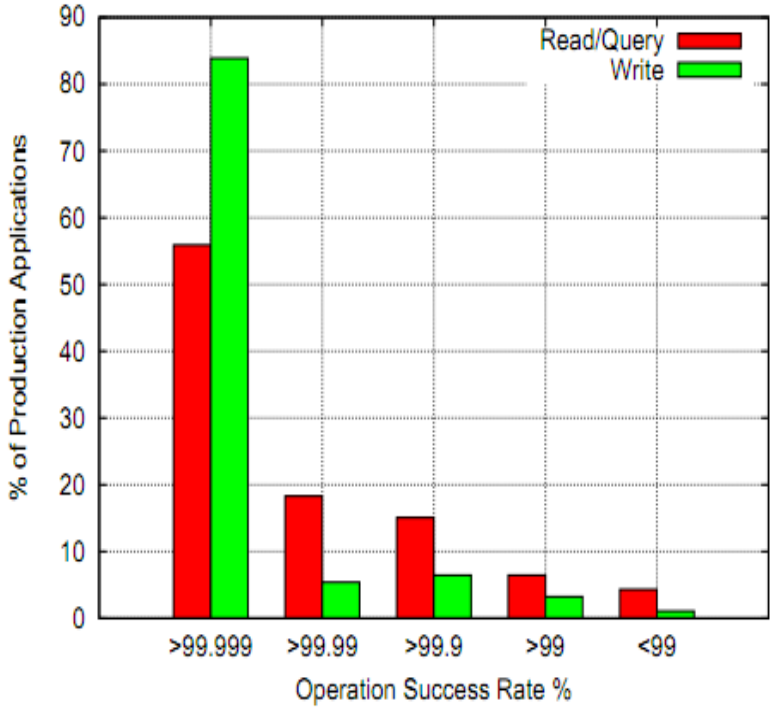
Megastore Reads

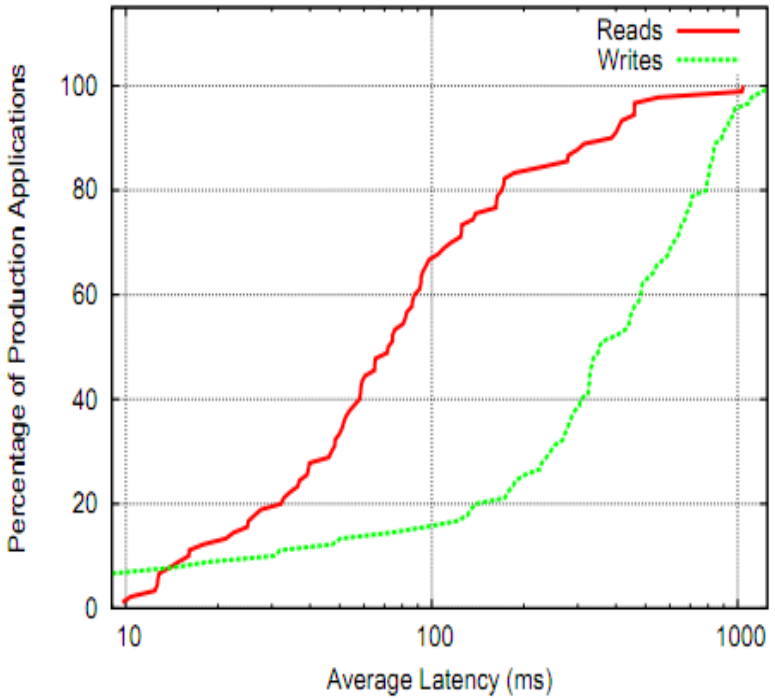


Megastore Writes



Availability and Performance





Benefits

- For admins
  - –Linear scaling, transparent rebalancing (Bigtable)
  - –Instant transparent failover
  - –Symmetric deployment
- For developers
  - –ACID transactions (read-modify-write)
  - –Many features (indexes, backup, encryption, scaling)
  - –Single-system image makes code simple
  - –Little need to handle failures
- For end Users
  - –Fast up-to-date reads, acceptable write latency
  - –Consistency

Summary

- Constraints acceptable for most apps
  - –Entity Group partitioning
  - –High write latency
  - –Limited per-EG throughput
- In production use for over 4 years
- No current query language
  - –Apps must implement query plans
  - –Apps have fine-grained control of physical placement
- Available on Google App Engine as HRD (High Replication Datastore)

Deuteronomy

Deuteronomy is a system which provides efficient ACID transaction for data anywhere, including the cloud.

Unbundling Transactions in the Cloud

[Lomet et al., CIDR 2009, CIDR 2011]

Transaction component: TC

Transactional CC & Recovery

At logical level (records, key ranges, …)

No knowledge of pages, buffers, physical structure

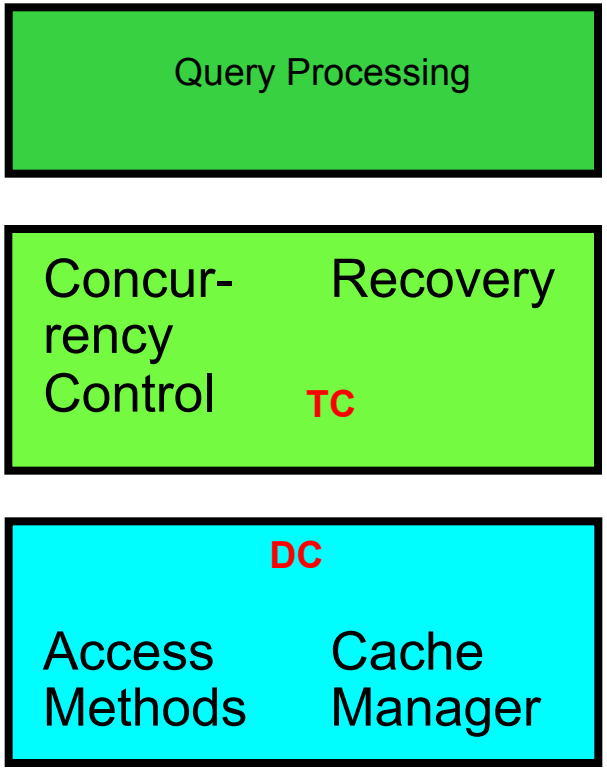
Data component: DC

Access methods & cache management

Provides atomic logical operations

Traditionally page based with latches

No knowledge of how they are grouped in user transactions

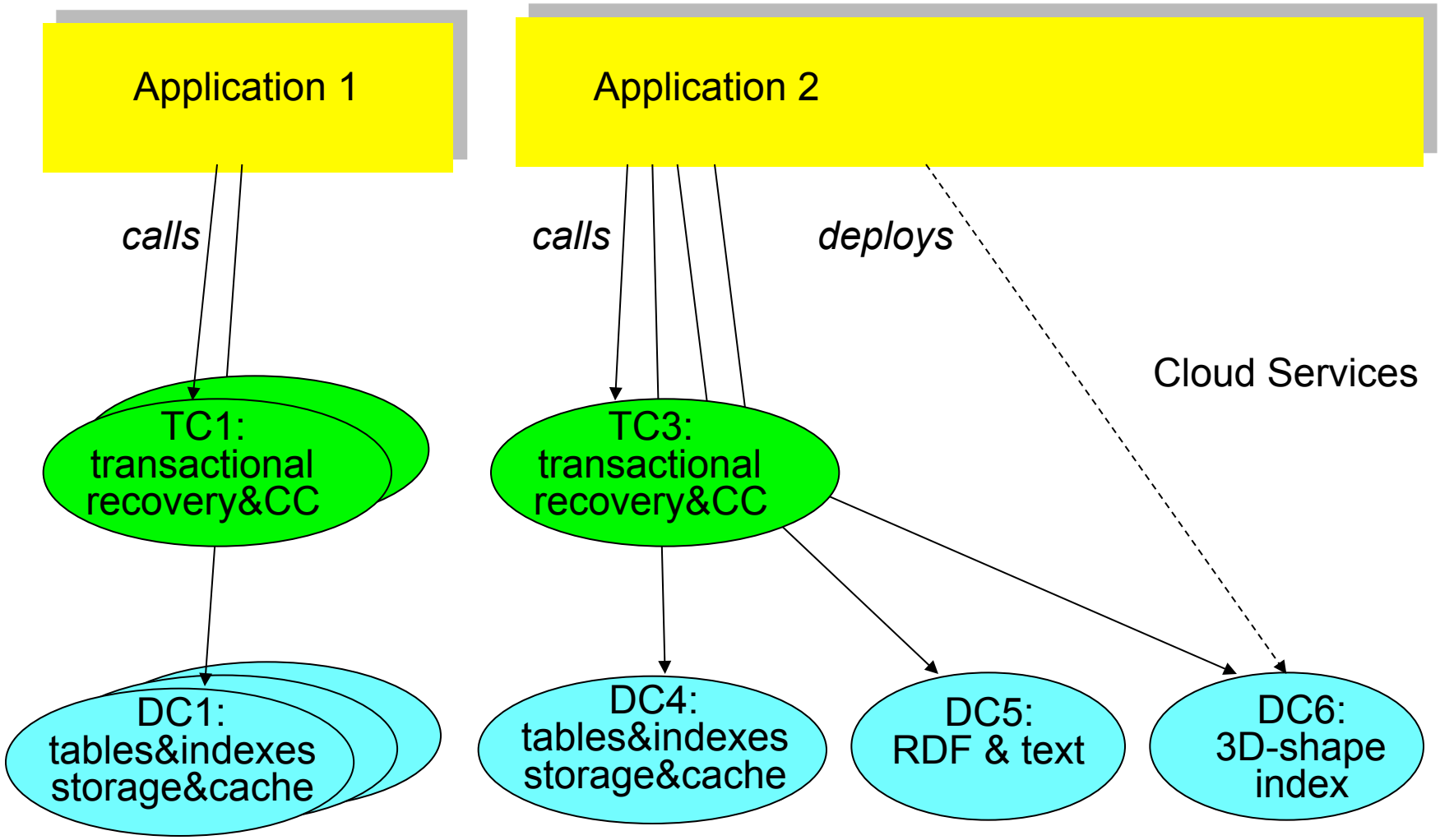


Why might this be interesting?

- Multi-Core Architectures
  - –Run TC and DC on separate cores
- Extensible DBMS
  - –Providing of new access method – changes only in DC
  - –Architectural advantage whether this is user or system builder extension
- Cloud Data Store with Transactions
  - –TC coordinates transactions across distributed collection of DCs without 2PC
  - –Can add TC to data store that already supports atomic operations on data

Slides adapted from authors' presentation

Extensible Cloud Scenario



Slides adapted from authors' presentation

Architectural Principles

View DB kernel pieces as distributed system

This exposes full set of TC/DC requirements

Interaction contract between DC & TC

Interaction Contract

Concurrency: to deal with multithreading

- no conflicting concurrent ops

Causality: WAL

- Receiver remembers request => sender remembers request

Unique IDs: LSNs

- monotonically increasing– enable idempotence

Idempotence: page LSNs

- Multiple request tries = single submission: *at most once*

Resending Requests: to ensure delivery

- Resend until ACK: *at least once*

Recovery: DC and TC must coordinate now

- DC-recovery *before* TC-recovery

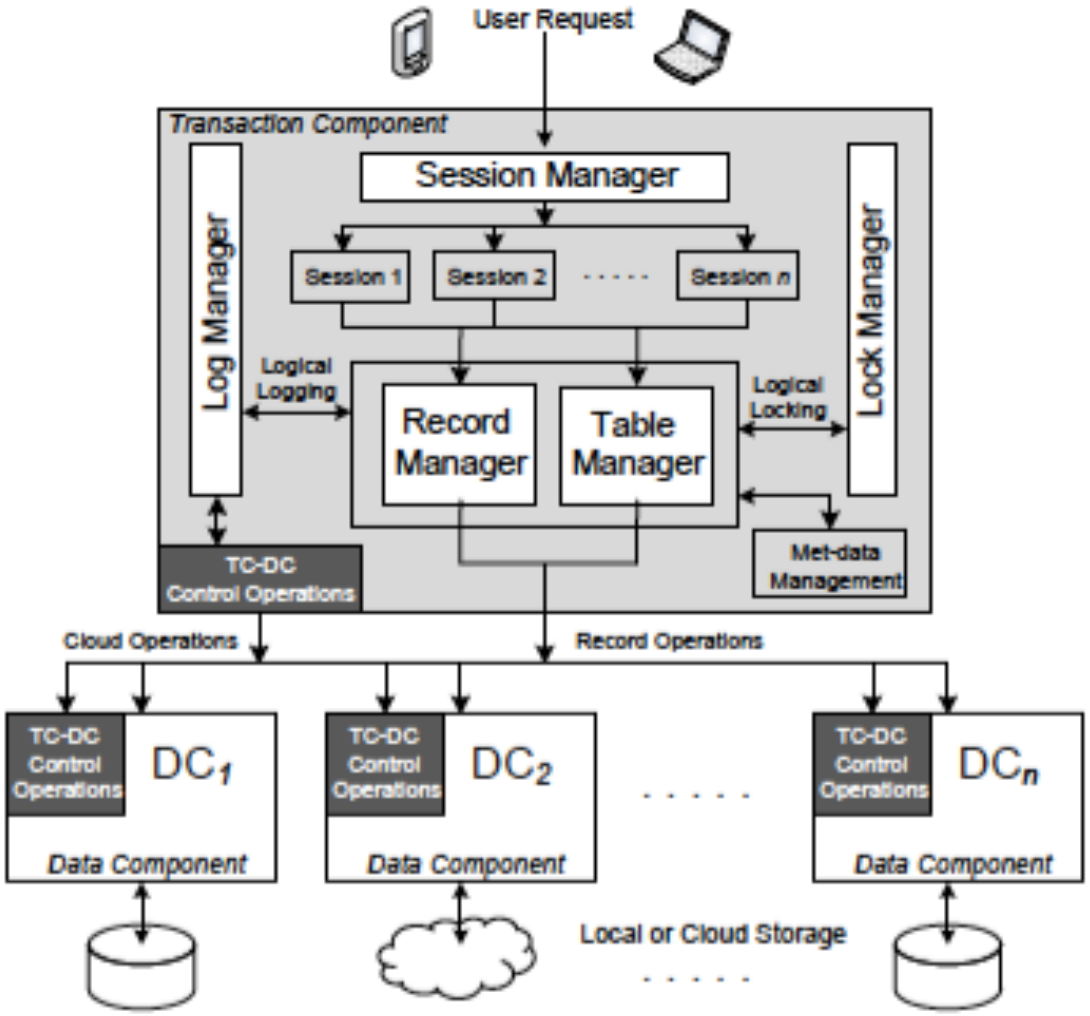
Contract Termination: checkpoint

- Releases resend & idempotence & causality requirements

TC Architecture

- 5 main components:
  - session, record, table, lock, log
- session manager:
  - manage sessions
  - requests flows through session managers
- record/table managers: record/table ops
- lock/log managers:
  - logical concurrency control and recovery
  - logical log/lock -> metadata mapping -> DC

Deuteronomy System Architecture



Transaction Component

1. Session Manager

- Provides interface to application and monitor execution of request.
- Maintains a thread pool, each active session is assigned one thread.

2. Record Manager

- Read operations get appropriate lock from Locker Manager, request then is forwarded to corresponding DC.
- Write operations request appropriate locks, get a log sequence number (LSN), send operation with the LSN to DC. After DC finishes the execution of the request, send the execution result (Record before and status of the request) back to TC. TC log the operation.

Transaction Component

3. Table Manager

- Metadata management including table catalog and column catalog.
- Logical Partition Creation, which is an intermediate granularity lockable resource for table.

4. Lock Manager

- Multi-granularity locking with three levels of resource: table, partition, record.
- Record Manager utilize partition to request locks on all partitions which overlap with the requested range.
- Generate LSNs which are used to log write operations.

Transaction Component

5. Log Manager

- Send two control operations to DC to enforce the write-ahead log protocol and to truncate the active part of the log via a checkpoint.
- EOSL: Permit DC to write cached data back to stable storage by sending an eLSN which indicates the end of Stable Log. Cached updates with LSN <= eLSN are persisted .
- RSSP: Send a rLSN to DC which indicates the desired Redo Scan Start Point. This operation requires DC to write cached updates to stable storage with LSN <= rLSN.

Transactional Optimizations

1. Fast Commit

- Allows a transaction to release all its lock before waiting for its commit record to be flushed to stable storage.

2. Group Commit

- Amortize the cost of a log by grouping transactions that commit close in time and issuing a single log flush instead one for each. The flush occurs either when the buffer is full or after a small time delta.

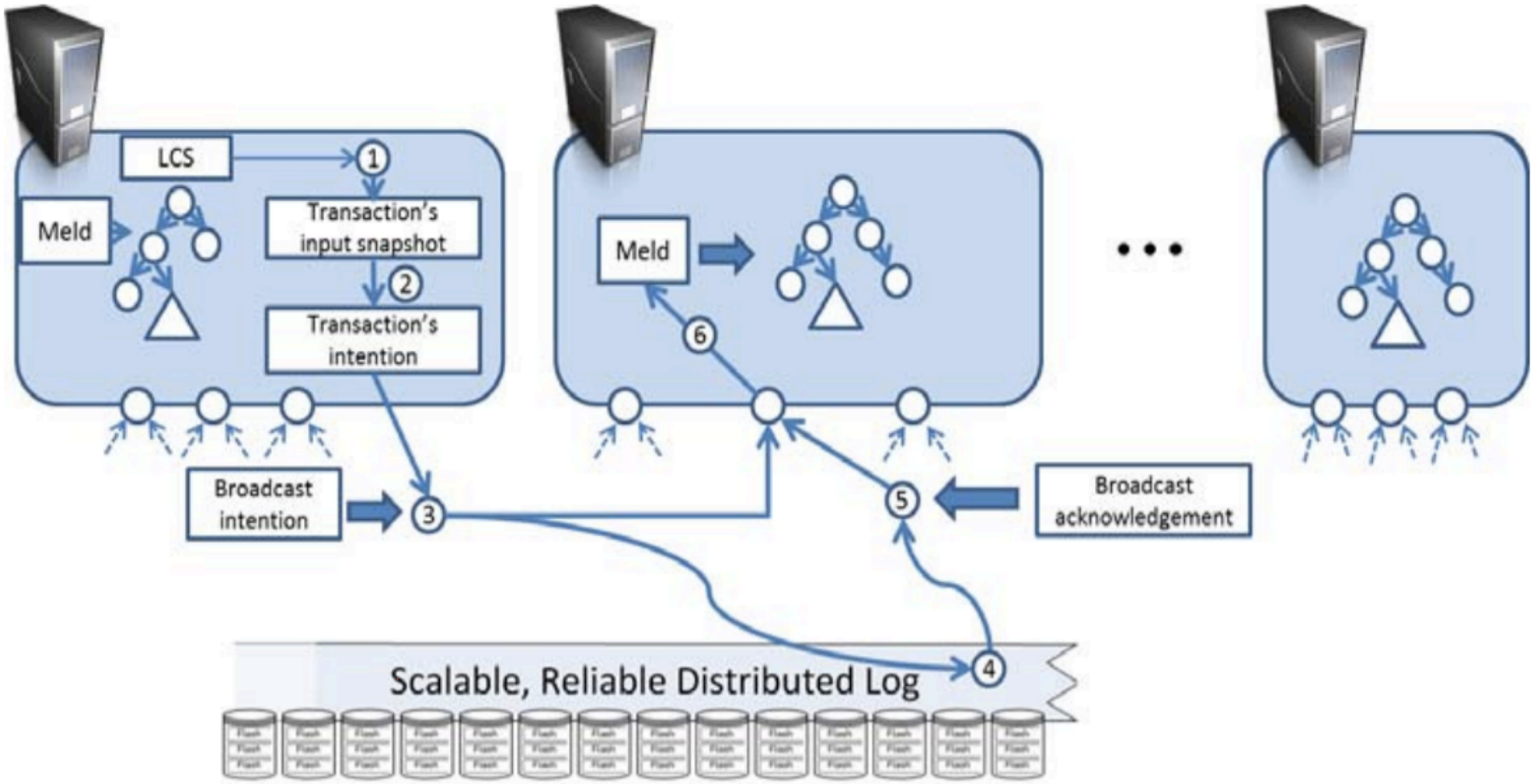
Hyder (No partitioning scheme)

- It is a transactional record manager for shared flash.
- It is designed to run on a cluster of servers that have shared access to a large pool of network-addressable raw flash chips.
- Log-structuring leverages the high random I/O rate of flash.
- Hyder uses a data-sharing architecture that scales out without partitioning the database or application.

Hyder (contd.)

- Each transaction executes on a snapshot, logs its updates in one record, and broadcasts the log record to all servers.
- the architecture of the overall system and its three main components:
  - the log,
  - the index
  - the roll-forward algorithm.
- Hyder can scale out to support high transaction rates.

Hyder


LCS
1
Meld
Transaction's input snapshot
2
Transaction's intention
Meld
6
Broadcast intention
3
5
Broadcast acknowledgement
Scalable, Reliable Distributed Log
4

Cloud SQL Server

- Cloud SQL Server is a relational database system designed to scale-out to cloud computing workloads.
- It uses Microsoft SQL Server as its core.
- To scale out, it uses a partitioned database on a shared-nothing system architecture.

Cloud SQL server

- Transactions are constrained to execute on one partition, to avoid the need for two-phase commit.
- The database is replicated for high availability using a custom primary-copy replication scheme.

CloudTPS

- CloudTPS is a scalable transaction manager providing full ACID properties for multi-item transactions even in the presence of server failures and network partitions
- Transaction span a small number of data items, thus low number of conflicts b/w transaction accessing same data.
- Read only query can return older but consistent version of data, thus executing complex read queries in the cloud rather in LTMs.

CloudTPS

- 2PC across all TLMs responsible for data items accessed for particular transaction.

- Transaction states and data items replicated backup TLMs during second phase to ensure atomicity in presence of failures.

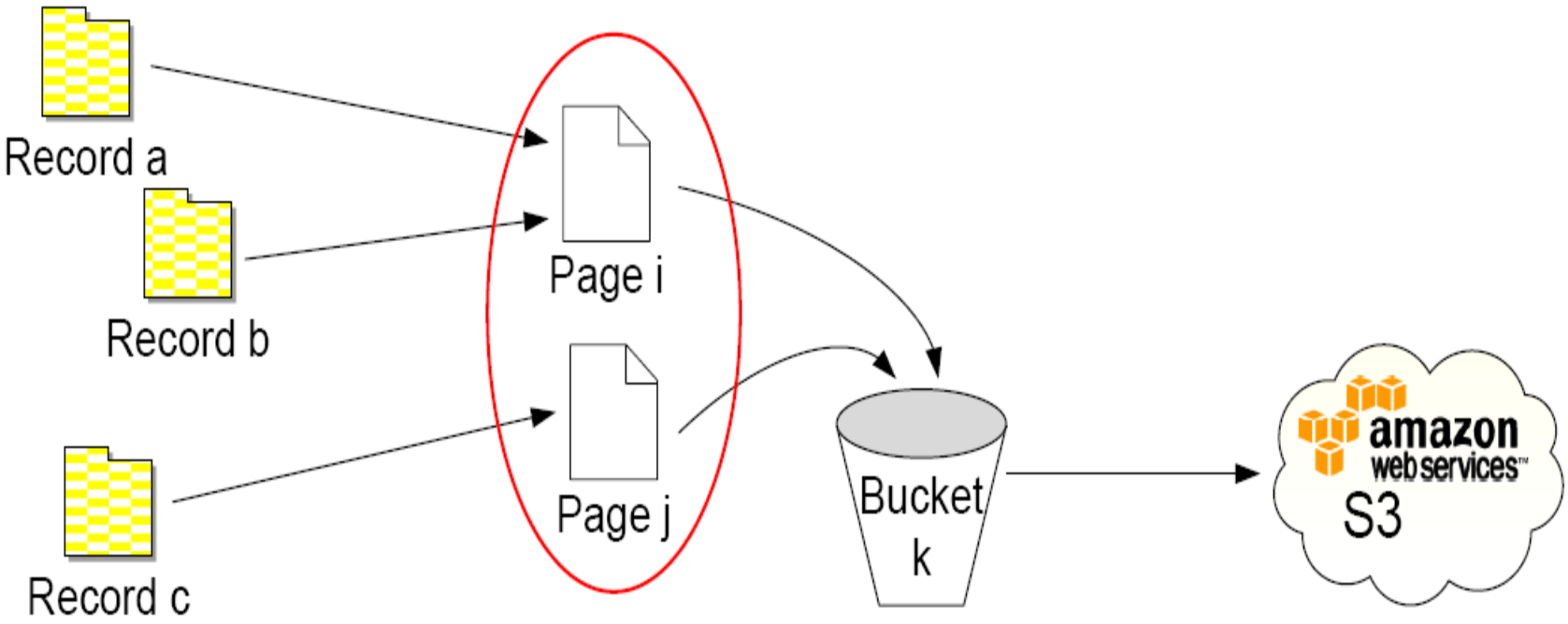
- Each transaction and data item has N+1 replicas, thus TPS can guarantee Atomicity under failure of N LTM servers.

Database on S3

[Brantner et al., SIGMOD 2008]

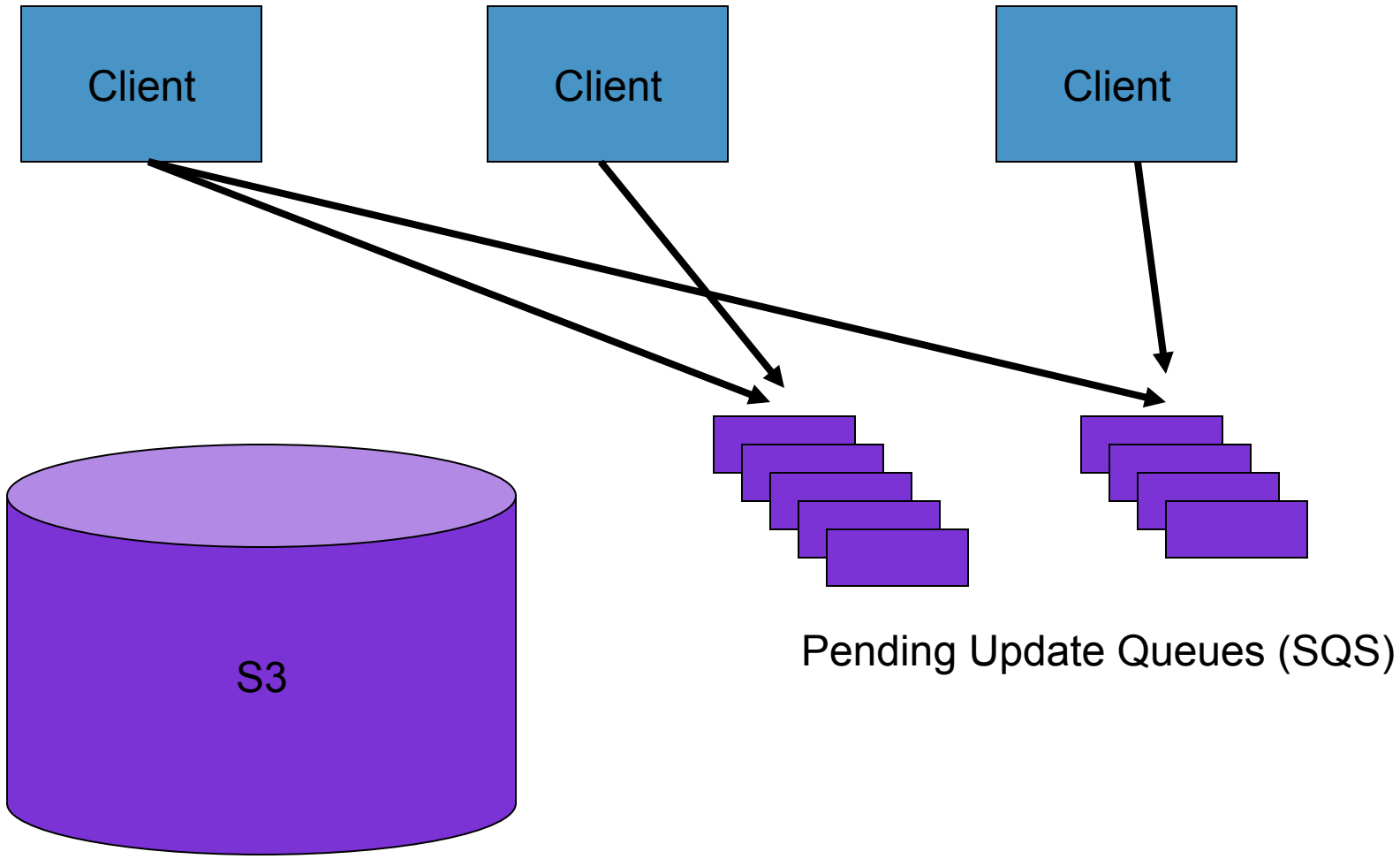
- Simple Storage Service (S3) – Amazon's highly available cloud storage solution
- **Use S3 as the disk**
- Key-Value data model – Keys referred to as records
- An S3 bucket equivalent to a database page
- Buffer pool of S3 pages
- Pending update queue for committed pages
- Queue maintained using Amazon SQS

Database on S3

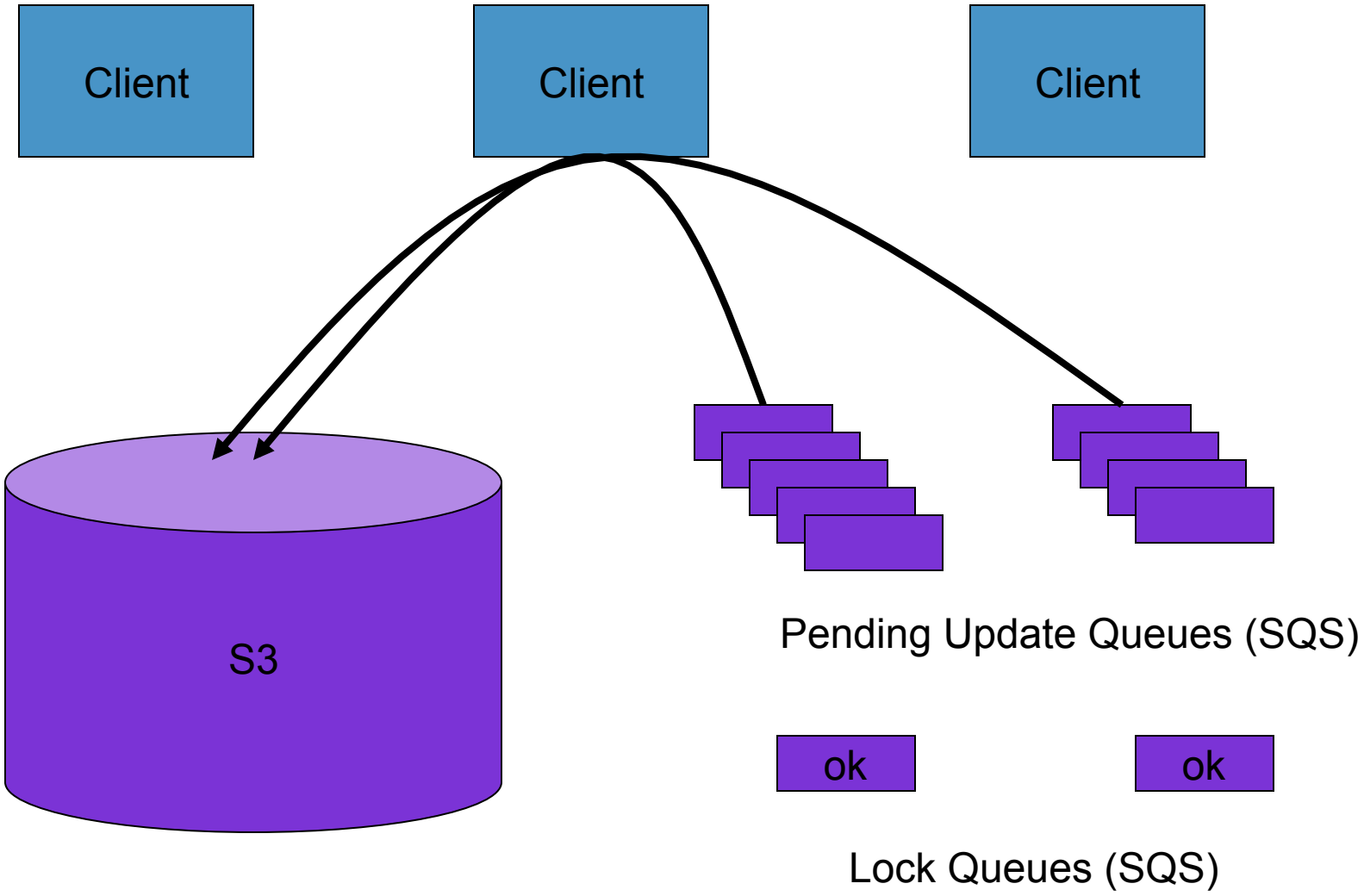


Slides adapted from authors' presentation

Step 1: Clients commit update records to pending update queues



Step 2: Checkpointing propagates updates from SQS to S3



Slides adapted from authors' presentation

Consistency Rationing

[Kraska et al., VLDB 2009]

- Not all data needs to be treated at the same level consistency
- Strong consistency only when needed
- Support for a spectrum of consistency levels for different types of data
- **Transaction Cost vs. Inconsistency Cost**
  - Use ABC-analysis to categorize the data
  - Apply different consistency strategies per category

Consistency Rationing

Classification

Category	Characteristics	Guarantees	Use Case
A-Data	**Inconsistencies are expensive and/ or cannot be resolved**	**Serializable (2PL)** •Pessimistic CC as conflicts are expected •No staleness: always up-to-date data	• Bank data • Atomic bomb
B-Data	**Violations might be tolerable**	**Adaptive guarantees** •Switches between A & C guarantees •Depends on some policy	• Product inventory • Tickets
C-Data	**No inconsistency cost and/or inconsistency cannot occur**	**Session consistency** •Practical •Still eventual consistent •Allows for aggressive caching	• Recom-mendations • Customer profiles • Products

In analogy to the ABC-analysis from Inventory Rationing

Slides adapted from authors' presentation

Adaptive Guarantees for B-Data

- B-data: Inconsistency has a cost, but it might be tolerable
- Often the bottleneck in the system
- Potential for big improvements
- Let B-data automatically switch between A and C guarantees

B-Data Consistency Classes

	Characteristics	Use Cases	Policies
General	Non-uniform conflict rates	Collaborative editing	General Policy
Value Constraint	•Updates are commutative •A value constraint/limit exists	•Web shop •Ticket reservation	•Fixed threshold policy •Demarcation policy •Dynamic Policy
Time based	Consistency does not matter much until a certain moment in time	Auction system	Time based policy

Slides adapted from authors' presentation

General Policy - Idea

- Apply strong consistency protocols only if the likelihood of a conflict is high
  - Gather temporal statistics at runtime
  - Derive the likelihood of an conflict by means of a simple stochastic model
  - Use strong consistency if the likelihood of a conflict is higher than a certain threshold

Slides adapted from authors' presentation

And the List Continues

- Cloudy [ETH Zurich]
- epiC [NUS]
- Deterministic Execution [Yale]
- …

Commercial Landscape
Major Players

- Amazon EC2
  - IaaS abstraction
  - Data management using S3 and SimpleDB
- Microsoft Azure
  - PaaS abstraction
  - Relational engine (SQL Azure)
- Google AppEngine
  - PaaS abstraction
  - Data management using Google MegaStore

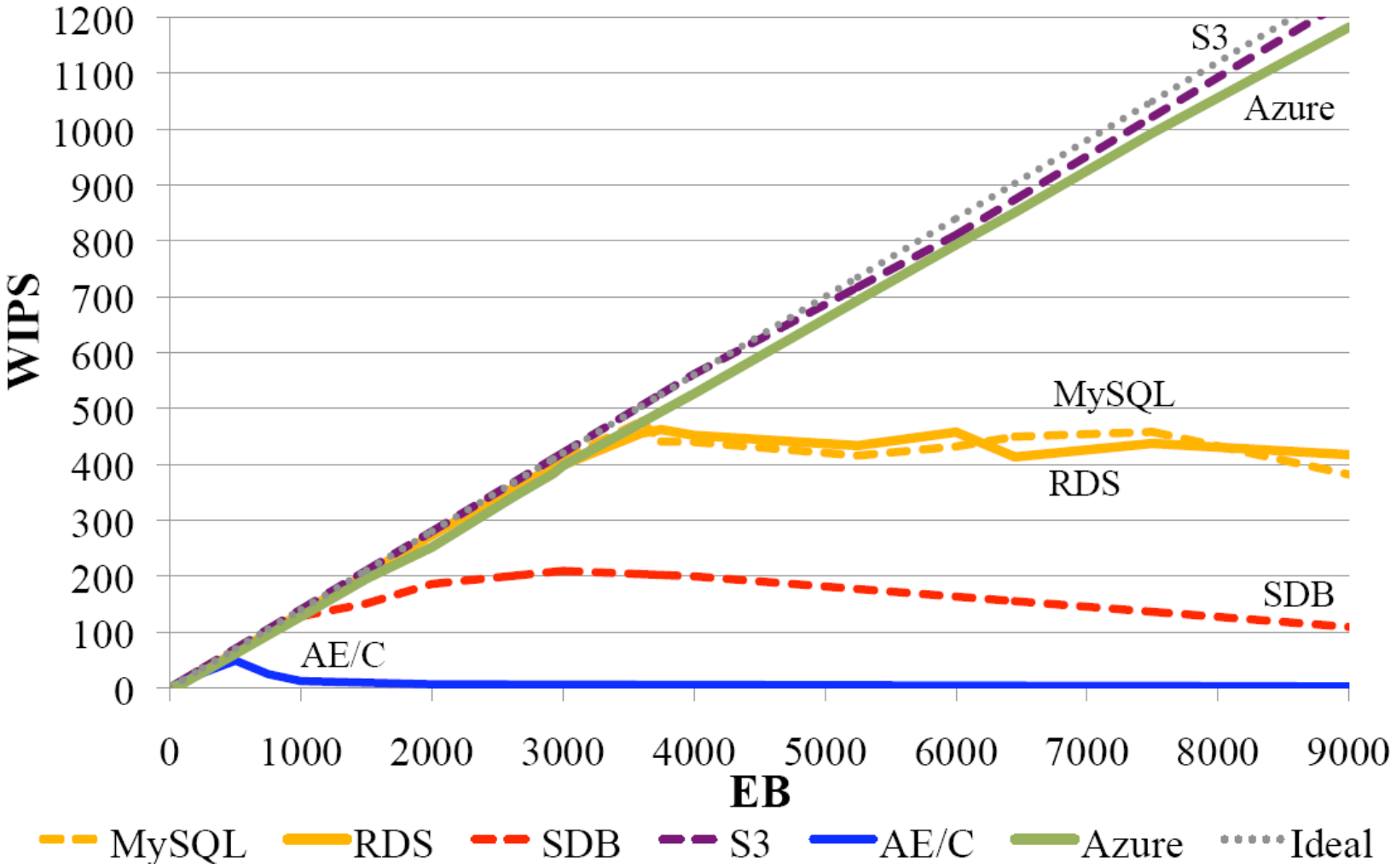
Evaluation of Cloud Transactional Stores
[Kossmann et al., SIGMOD 2010]

- Focused on the performance of the Data management layer
- Alternative designs evaluated
  - MySQL on EC2
  - AWS (S3, SimpleDB, and RDS)
  - Google AppEngine (MegaStore, with and without Memcached)
  - Azure (SQL Azure)

Scalability and Cost

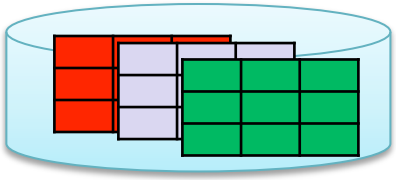
	Throughput (WIPS)	Cost/WIPS (m$)		Cost Predictability (mean $\pm s$)
		Low TP	High TP	
MySQL	477	0.635	***0.005***	0.015 ± 0.077
MySQL/R	454	2.334	***0.005***	0.043 ± 0.284
RDS	462	1.212	***0.005***	0.030 ± 0.154
SimpleDB	128	0.384	0.037	0.063 ± 0.089
S3	***>1100***	1.304	0.009	0.018 ± 0.098
Google AE	39	0.002	0.042	0.029 ± 0.016
Google AE/C	49	***0.002***	0.028	$\boldsymbol{0.021 \pm 0.011}$
Azure	***>1100***	0.775	***0.005***	0.010 ± 0.058

Scalability

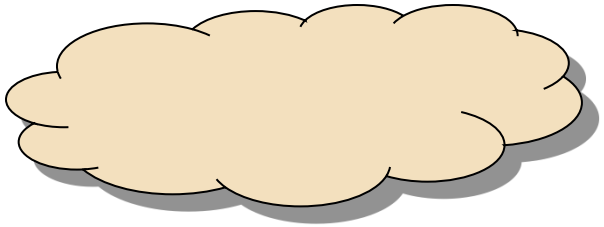


Slides adapted from authors'

Summary of Techniques for Transactions In the Cloud



RDBMS



Key Value Stores

Cloudify RDBMSs

Fusion of the architectures

Enrich Key Value Stores

RelationalCloud [CIDR ‘11]
SQL Azure [ICDE ’11]

Deutoronomy [CIDR ‘09, ‘11]
ElasTraS [HotCloud ’09, TR ‘10]
DB on S3 [SIGMOD ‘08]

MegaStore [CIDR ‘11]
G-Store [SoCC ‘11]
Vo et al. [VLDB ‘10]
Rao et al. [VLDB ‘11]