Yesquel: scalable SQL storage for Web applications

Marcos K. Aguilera, VMware Research Group
Joshua B. Leners, UT Austin, NYU
Michael Walfish, NYU

Joy Arulraj, Reading Group Fall 2015

CARNEGIE MELLON
DATABASE GROUP

Motivation

- Web applications
  - *Application-specific caching/partitioning*
  - *NoSQL systems with less functionality*

Motivation

- NoSQL systems
  - *Shift complexity to the application*
  - *Comparing different feature sets hard*
  - *Specialized interfaces => lock-in problem*

Yesquel

- Yesquel
  - *All features of a relational system*
  - *Performance & scalability of NoSQL systems*
- SQL DBMS
  - *Query processor*
  - *Storage engine*

Key Question

How to scale-out the storage engine on high-contention workloads ?

High Level Idea

- Distributed balanced B+ tree
  - *Distribution adapts with the workload*
  - *Optimized to reduce network round trips*
  - *Strong consistency and fault tolerance*

NoSQL Systems

- Comparison
  - *Data Model*
  - *Durability*
  - *Distributed Txns*
  - *Secondary Indexes*
  - *Joins*
  - *Aggregation*

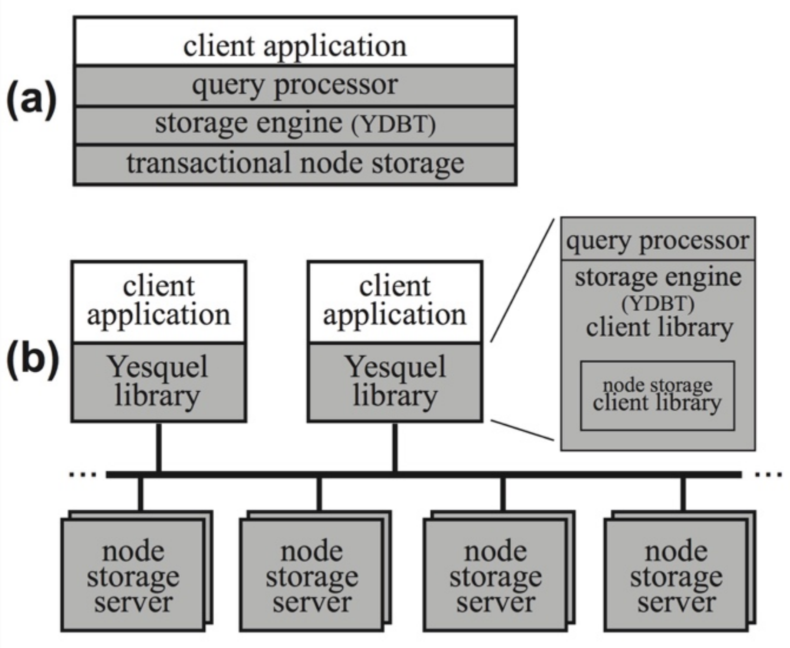
system	data model	dura-bility	distributed transactions	secondary indexes	joins	aggre-gation
Sinfonia	bytes	yes	lim	no	no	no
COPS/Eiger	kv	yes	lim	no	no	no
Dynamo	kv	yes	no	no	no	no
Hyperdex	kv	yes	yes	yes	no	no
Memcached	kv	no	no	no	no	no
Riak	kv	yes	no	yes	no	no
Scalaris	kv	no	yes	no	no	no
Redis	kv	yes	no	no	no	no
Voldemort	kv	yes	no	no	no	no
Walter	kv	yes	yes	no	no	no
AzureTable	table	yes	no	no	no	no
BigTable	table	yes	no	no	no	no
Cassandra	table	yes	no	lim	no	no
HBase	table	yes	no	no	no	no
Hypertable	table	yes	no	lim	no	no
PNUTS	table	yes	no	no	no	no
SimpleDB	table	yes	no	yes	no	yes
MongoDB	doc	yes	no	yes	no	lim

Challenges & Solutions

CHALLENGE	SOLUTION
Locality	Distributed Balanced Tree
Network Trips	"Back-down search"
Load Balancing	Load-based node splits, "Replits"
Fault Tolerance	Replication
Performance	Fewer round trips

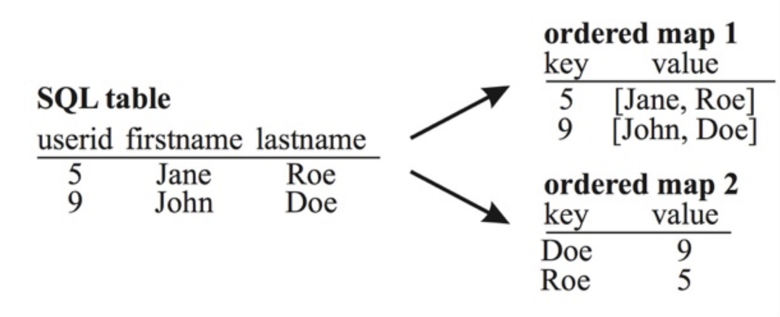
Architecture

- Logical Architecture
  - *Yesquel DBT (YDBT)*
- Physical Architecture
  - *Node storage system*



Architecture

- YDBT Ordered Map
  - *Tables*
  - *Indexes*



- Distributed node storage system
  - *Multi-version*
  - *Distributed transactions at lowest logical layer*

YDBT Interface

- Transactional API
- Traversal API
  - *Ordered iterators*
- Data API
  - *Create Indexes*
  - *Insert, Delete keys*

Operation	Description
	[Transactional operations]
start()	start a transaction
rollback/commit()	rollback or try to commit transaction
	[Traversal operations]
createit()	create an iterator, return *it*
seek(it, ix, k)	move *it* to *k* in *ix*
first/last(it, ix)	move *it* to smallest or largest key in *ix*
next/prev(it)	move *it* to next or prev key
	[Data operations]
createix(db)	create index in *db*, return *ix*
destroyix(ix)	destroy index
insert(ix, k, v)	insert (k, v) in *ix*
delete(ix, k)	delete $(k, *)$ from *ix*
deref(it)	dereference *it*, return (k, v)

YDBT Ideas

- Speculate and validate
  - *Clients cache tree nodes without coherence*
  - *Execute speculatively*
  - *Validate results before commit*

Back-down search

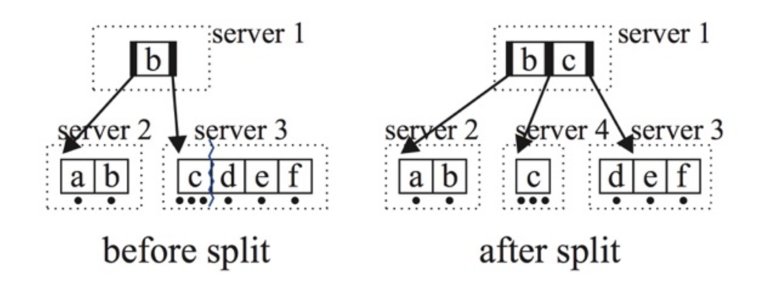
- Cache search optimization
  - *High-level tree nodes mostly in cache*
  - *Concurrent clients can modify lower-level nodes*
  - *Detect stale nodes using "fence intervals"*
  - *Interval of keys that a node is responsible for*

Back-down search

- Back-down search
  - *If key not inside fence => something wrong*
  - *Back phase : Backtrack upwards to a node where the key within the fence*
  - *Down phase : Go down the tree again till you find the leaf*
  - *Reduces read load on higher-level nodes*

Load Splits

- B+tree splits nodes based on size
- YDBT splits nodes based on load



Replits

- Combine replication and splitting
  - *Split popular key into 2 replicas*
  - *Append key with "r" bits*
  - *Old key – all bits are zeroes*
  - *New key – random bits (another server)*
  - *Search key – random bits*

Improving Concurrency

- Multi-version concurrency control
  - *Free snapshots*
- Right node splits
  - *Keep the second half & move the first half*
  - *Reduce contention for autoincrement columns*
  - *Concurrent inserts and split*

Node Storage API

- Transactional API
- Node data API
  - *Commutative ops*
  - *Ordered key list*
- Whole node API

Operation	Description
[Transactional operations]	
start()	start a transaction
rollback()/commit()	rollback or try to commit
[Node data operations]	
insert(n, k, v, dir)	insert (k, v) into n
lookup(n, k)	lookup k in n, return value/pointer
delrange(n, k_1, k_2, dir)	delete keys in $[k_1, k_2]$ from n
setattr(n, at, v)/*getattr*(n, at)	set or get attribute *at* in n
[Whole node operations]	
read(n)	read n
create$(n, type, ks, vs, ats)$	create and populate node n
delete(n)	delete node n

Transactional Node Storage

- Read-only transactions grab no locks
- Clients run the commit protocol
  - *Transaction outcome is sole function of votes*
  - *Can recover without the coordinator (client)*
- Use clocks for performance, not safety
  - *Timestamp ordering*

Implementation Details

- SQLITE query processor
  - *Per-transaction node cache*
  - *Read keys without values*
  - *Deferred writes at client to reduce RPCs*
  - *Optimistic insert based on fence interval*

Evaluation

- Baselines
  - *Base : Sinfonia DBT with Optimistic CC*
  - *Base+ : Base and back-down searches*

DBT	description
BASE	Represents [2, 61]. Optimistic concurrency control (instead of multi-version concurrency control as in YDBT); no back-down searches, load splits, or delegated splits.
BASE+	Adds YDBT's back-down searches to BASE

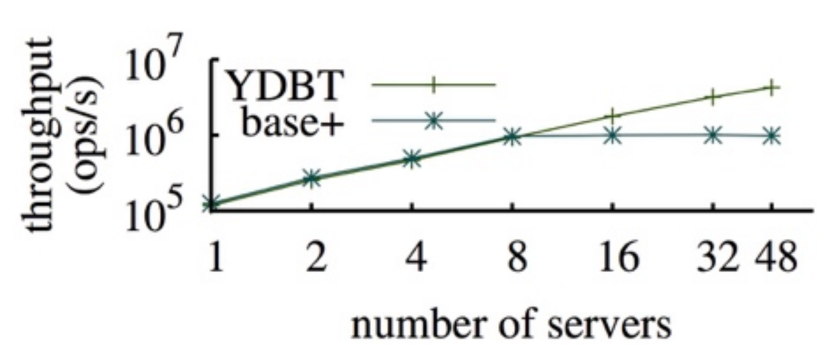
Root node load

- Benefit of back-down searches
  - *Fraction of ops accessing the root node*
  - *Node splits*

	update/read		insert	
# clients	**YDBT**	**BASE**	**YDBT**	**BASE**
1	0%	0%	0%	0%
5	0%	0%	0.02%	5.1%
10	0%	0%	0.03%	10.8%
20	0%	0%	0.06%	20.3%
30	0%	0%	0.10%	28.0%
40	0%	0%	0.13%	34.5%
50	0%	0%	0.16%	39.8%

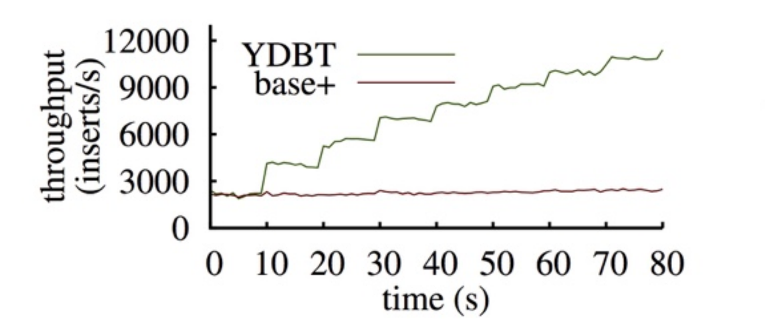
Load Split

- High skew workload



Insert Contention

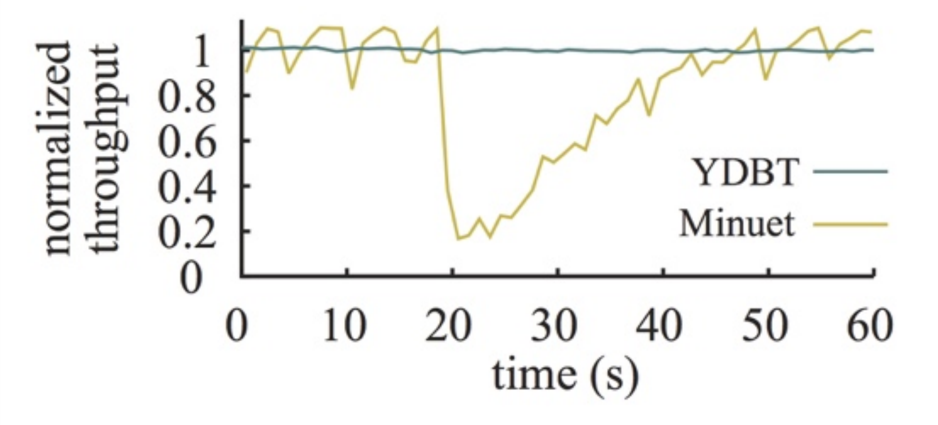
- More clients added over time



technique disabled	performance drop
delegated splits	55.0%
mostly commutative ops	80.0%
right splits	80.7%

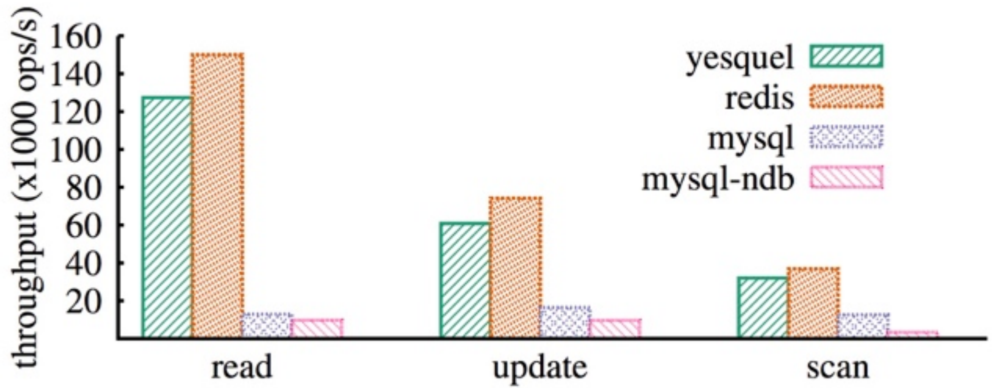
Snapshots

- Benefits of MVCC



Comparison with Redis

- Redis
  - *Hash table lookup*
- MySQL
  - *Centralized*
  - *Query processing*
  - *Multiple round trips*



Summary

CHALLENGE	SOLUTION
Locality	Distributed Balanced Tree
Network Trips	“Back-down search”
Load Balancing	Load-based node splits, “Replits”
Fault Tolerance	Replication
Performance	Fewer round trips

Limitations

- More network bandwidth
  - *Clients bring data to the computation*
  - *Not suitable for analytics*
- More client CPU
  - *Fundamental design choice*

Takeaways

- Shift complexity away from the application
- Provide functionality at the right layer
- Optimize for a class of applications
- Do not use MongoDB !

END

@jarulraj