**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
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
## Challenges & Solutions

<table>
<thead>
<tr>
<th>CHALLENGE</th>
<th>SOLUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality</td>
<td>Distributed Balanced Tree</td>
</tr>
<tr>
<td>Network Trips</td>
<td>“Back-down search”</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Load-based node splits, “Replits”</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Replication</td>
</tr>
<tr>
<td>Performance</td>
<td>Fewer round trips</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start()</td>
<td>start a transaction</td>
</tr>
<tr>
<td>rollback/commit()</td>
<td>rollback or try to commit transaction</td>
</tr>
<tr>
<td>createit()</td>
<td>create an iterator, return it</td>
</tr>
<tr>
<td>seek(it, ix, k)</td>
<td>move it to k in ix</td>
</tr>
<tr>
<td>first/last(it, ix)</td>
<td>move it to smallest or largest key in ix</td>
</tr>
<tr>
<td>next/prev(it)</td>
<td>move it to next or prev key</td>
</tr>
<tr>
<td>createix(db)</td>
<td>create index in db, return ix</td>
</tr>
<tr>
<td>destroyix(ix)</td>
<td>destroy index</td>
</tr>
<tr>
<td>insert(ix, k, v)</td>
<td>insert (k, v) in ix</td>
</tr>
<tr>
<td>delete(ix, k)</td>
<td>delete (k, *) from ix</td>
</tr>
<tr>
<td>deref(it)</td>
<td>dereference it, return (k, v)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start()</td>
<td>start a transaction</td>
</tr>
<tr>
<td>rollback()/commit()</td>
<td>rollback or try to commit</td>
</tr>
</tbody>
</table>

**[Transactional operations]**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(n, k, v, dir)</td>
<td>insert ((k, v)) into (n)</td>
</tr>
<tr>
<td>lookup(n, k)</td>
<td>lookup (k) in (n), return value/pointer</td>
</tr>
<tr>
<td>delrange(n, k1, k2, dir)</td>
<td>delete keys in ([k_1, k_2]) from (n)</td>
</tr>
<tr>
<td>setattr(n, at, v)/getattr(n, at)</td>
<td>set or get attribute (at) in (n)</td>
</tr>
</tbody>
</table>

**[Node data operations]**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(n)</td>
<td>read (n)</td>
</tr>
<tr>
<td>create(n, type, ks, vs, ats)</td>
<td>create and populate node (n)</td>
</tr>
<tr>
<td>delete(n)</td>
<td>delete node (n)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>DBT</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>Represents [2, 61]. Optimistic concurrency control (instead of multi-version concurrency control as in YDBT); no back-down searches, load splits, or delegated splits.</td>
</tr>
<tr>
<td>BASE+</td>
<td>Adds YDBT’s back-down searches to BASE</td>
</tr>
</tbody>
</table>
Root node load

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

<table>
<thead>
<tr>
<th># clients</th>
<th>update/read</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>YDBT BASE</td>
<td>YDBT</td>
</tr>
<tr>
<td>1</td>
<td>0% 0%</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>0% 0%</td>
<td>0.02%</td>
</tr>
<tr>
<td>10</td>
<td>0% 0%</td>
<td>0.03%</td>
</tr>
<tr>
<td>20</td>
<td>0% 0%</td>
<td>0.06%</td>
</tr>
<tr>
<td>30</td>
<td>0% 0%</td>
<td>0.10%</td>
</tr>
<tr>
<td>40</td>
<td>0% 0%</td>
<td>0.13%</td>
</tr>
<tr>
<td>50</td>
<td>0% 0%</td>
<td>0.16%</td>
</tr>
</tbody>
</table>
Load Split

• High skew workload
Insert Contention

- More clients added over time
Snapshots

• Benefits of MVCC
Comparison with Redis

• Redis
  – Hash table lookup

• MySQL
  – Centralized
  – Query processing
  – Multiple round trips
## Summary

<table>
<thead>
<tr>
<th>CHALLENGE</th>
<th>SOLUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality</td>
<td>Distributed Balanced Tree</td>
</tr>
<tr>
<td>Network Trips</td>
<td>“Back-down search”</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Load-based node splits, “Replits”</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Replication</td>
</tr>
<tr>
<td>Performance</td>
<td>Fewer round trips</td>
</tr>
</tbody>
</table>
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