Indexing in RAMCloud

Ankita Kejriwal, Ashish Gupta, Arjun Gopalan, John Ousterhout

Stanford University
Introduction

● RAMCloud 1.0

● Higher-level data models
  ▪ Without sacrificing latency and scalability

● Secondary Indexes: lookups and range queries on attributes that are not the primary key
Key Design Issues

- API and RAMCloud object format
- Index placement / partitioning
- Index memory allocation
- Failure / Restoration
- Consistency
Key Design Issues

- API and RAMCloud object format
- Index placement / partitioning
- Index memory allocation
- Failure / Restoration
- Consistency
Index Placement

- Lookup on index
- Index Key “age”
- Table 1
- Data
- Table 1
- return data

Indexing in RAMCloud
Index Partitioning

Option 1: Co-locate with data:

- Index Key “age”
  - Table 1, Part 1
- Data
  - Table 1, Part 1

  return data

Option 2: Partition based on index key:

- Index Key “age”
  - Table 1, Part 1
  - “age”: <= 50
- Data
  - Table 1, Part 1

  Lookup in correct index partition

- Index Key “age”
  - Table 1, Part 1
  - “age”: > 50
- Data
  - Table 1, Part 1

  Multi read matched data

Multi lookup on index

return data

Indexing in RAMCloud
Index Partitioning

• Index lookup:
  ▪ Assume data + index on n servers
  ▪ Opt 1: multiLookup to n servers + local reads
  ▪ Opt 2: lookup to index server + multiRead to x servers
    ▪ $x \in [0, n-1]$
    ▪ For small n: expect $x \approx n-1$
    ▪ For large n: expect $x << n$
  ▪ Option 2 more scalable

• Index entry format:
  ▪ <index key, primary key hash>
Key Design Issues

- API and RAMCloud object format
- Index placement / partitioning
- Index memory allocation
- Failure / Restoration
- Consistency
Failure / Restoration

- **Tablet Server**
  - Doesn’t affect indexes
  - “Normal” RAMCloud data recovery

- **Index server**
  - Backup / Recover
  - No backup / Rebuild
## Failure/Restoration: Write Latency

<table>
<thead>
<tr>
<th>Failure Type</th>
<th>Latency</th>
<th># Mem writes</th>
<th># Backup writes</th>
<th># Msgs from data to index servers</th>
<th># Msgs to backups</th>
</tr>
</thead>
<tbody>
<tr>
<td>No indexing</td>
<td>15 us</td>
<td>1</td>
<td>R</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>Indexing w/ backup/restore</td>
<td>35 us</td>
<td>m+1</td>
<td>R*(m+1)</td>
<td>m</td>
<td>R*(m+1)</td>
</tr>
<tr>
<td>Indexing w/ no-backup/rebuild</td>
<td>25 us</td>
<td>m+1</td>
<td>R</td>
<td>m</td>
<td>R</td>
</tr>
</tbody>
</table>

- **Indexing in RAMCloud**
- **Slide 10**
Failure/Restoration: Restoration Time

- **Recovery**: Similar to RAMCloud data recovery: 1-2 s
- **Rebuild**: Cost analysis:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Index partition to be recovered</th>
<th>1 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size of index entries</td>
<td>50 B (42 for key + 8 for keyhash)</td>
</tr>
<tr>
<td></td>
<td>Num of index entries</td>
<td>2 * 10^7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data master</th>
<th>Max memory bandwidth</th>
<th>35 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory bw with overheads</td>
<td>20 GB/s</td>
</tr>
<tr>
<td></td>
<td>Hash table size (10% of total mem)</td>
<td>25 GB (for 256 GB machine)</td>
</tr>
<tr>
<td></td>
<td>Time to scan hash table</td>
<td>1.25 s</td>
</tr>
<tr>
<td></td>
<td>Time to compare hash info from bucket</td>
<td>negligible</td>
</tr>
<tr>
<td></td>
<td>Num objects to check if all match</td>
<td>2.5 * 10^9 (for 100B objects)</td>
</tr>
<tr>
<td></td>
<td>Cache miss time</td>
<td>0.5 * 10^9 cache miss / s</td>
</tr>
<tr>
<td></td>
<td>Total cache miss time</td>
<td>5.12 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>Bandwidth</th>
<th>1 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time to transfer over network</td>
<td>1 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Recovery Master</th>
<th>Time per object to insert</th>
<th>1.5 us</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total time to insert</td>
<td>30 s</td>
</tr>
<tr>
<td></td>
<td>Total time to insert with parallelization</td>
<td>1 s</td>
</tr>
</tbody>
</table>
Memory Benchmark

Random reads from array of $2 \times 10^8$ objects of size 64 B on rcmonster

Aggregate bandwidth in GB/s

Where $x$ is:
- 1
- 4
- 8
- 16

Reading $x$ objects in parallel
rcmonster: 2 x Xeon E5–2670@2.6GHz
Key Design Issues

- API and RAMCloud object format
- Index placement / partitioning
- Index memory allocation
- Failure / Restoration
- Consistency
At any time, data is consistent with index entries corresponding to it, if:

- If data X exists, X is reachable from all key indexes.
- Data returned to client is consistent with key used to look it up.

Provides linearizability

- Tradeoff with performance

Also desirable:

- Dangling pointers are not accumulating.
- Memory footprint will not increase beyond what is necessary.
Consistency

● **Simple solution:**
  - Lock indexes and tablets for the entire duration of index update – affects scalability and performance

● **Our solution: Key Idea:**
  - Writing object is the commit point

● **Interesting situations:**
  - For multi-threaded write/read, non-locking, no failures
  - For multi-threaded write/write, non-locking, no failures
  - Failure of an Index Server
  - Failure of Master Server
Consistency

- Multi-threaded write/read, non-locking, no failures: Object Update
- There exists time $x$, s.t.: at time $< x$, client can lookup old data; at time $\geq x$, it can lookup the new data.
Consistency

- Multi-threaded write/read, non-locking, no failures: Object Update
- There exists time $x$, s.t.: at time $< x$, client can lookup old data; at time $\geq x$, it can lookup the new data.

Indexing in RAMCloud
Multi-threaded write/read, non-locking, no failures: Object Update

There exists time $x$, s.t.: at time $< x$, client can lookup old data; at time $\geq x$, it can lookup the new data.

### Consistency

- **Data**
  - **Foo**: Bob Brown
  - **fname Index**: Bob, 4444444
  - **lname Index**: Brown, 4444444

- **Step 1**
  - **Foo**: Bob Brown
  - **fname Index**: Bob, 4444444
  - **lname Index**: Brown, 4444444

- **Step 2**
  - **Foo**: Sam Smith
  - **fname Index**: Bob, 4444444
  - **lname Index**: Smith, 4444444

Indexing in RAMCloud
Multi-threaded write/read, non-locking, no failures: Object Update

There exists time $x$, s.t.: at time $< x$, client can lookup old data; at time $\geq x$, it can lookup the new data.
Summary

- Secondary Indexes: lookups & range queries on attributes that are not the primary key

- Key design issues:
  - Index partitioning
    - Co-locate with data
    - Partition based on index key
  - Failure / Restoration
    - Backup / recover
    - No backup / rebuild
  - Consistency: Linearizability
    - Key idea: Writing object is the commit point
Thank you!