JavaScript for Extending Low-latency In-memory Key-value Stores

Tian Zhang      Ryan Stutsman
Today's large scale key-value stores (e.g. Ramcloud, FaRM, etc.) are able to:

- Store TBs ~ PBs of data.
- 2~5μs end to end access time.
- Perform billions of operations per second.

Have today’s applications been able to properly leverage such systems?

Not yet.
Semantic Gap

- Implementing high level semantics with KVS APIs requires many roundtrips.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lchild 2, rchild 3</td>
</tr>
<tr>
<td>2</td>
<td>lchild 4, rchild 5</td>
</tr>
<tr>
<td>3</td>
<td>no child</td>
</tr>
<tr>
<td>4</td>
<td>no child</td>
</tr>
<tr>
<td>5</td>
<td>no child</td>
</tr>
</tbody>
</table>

Application Server

Key Value Storage

- Get() 2~5μs
- Get() 2~5μs
- Get() 2~5μs
- Get() 2~5μs
- Get() 2~5μs
Semantic Gap

- Implementing high level semantics with KVS APIs requires many roundtrips.
Existing Solution - Customized KVS

- Facebook has implemented TAO, a social graph data model in Memcache.
- Entities (e.g. people) are modeled as objects, their connections as associations.
- TAO stores objects and association lists, and provides APIs to operate on them.

<table>
<thead>
<tr>
<th>Memcache</th>
<th>TAO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hash Table</strong></td>
<td><strong>TAO API</strong></td>
</tr>
<tr>
<td>Key</td>
<td>object_add()</td>
</tr>
<tr>
<td>Value</td>
<td>object_get()</td>
</tr>
<tr>
<td></td>
<td>assoc_add()</td>
</tr>
<tr>
<td><strong>Default API</strong></td>
<td></td>
</tr>
<tr>
<td>get()</td>
<td></td>
</tr>
<tr>
<td>set()</td>
<td></td>
</tr>
<tr>
<td>delete()</td>
<td></td>
</tr>
</tbody>
</table>
Existing Solution - Customized KVS

- Other customized KVS:
  - Md-hbase with multi-attribute access support.
  - Comet with application-specific actions.
  - G-store with consistent multi-key access support.

Disadvantage: ad-hoc solutions for specific applications, not general.
Our Solution - Runtime Extensibility

- A more general solution is to allow pushing custom logic to KVS at runtime.
- The KVS can be dynamically reconfigured to support new applications.
More to Consider - Cloud Service

- Combining workloads improves utilization.
- Deploying the system on cloud to leverage the elasticity and scalability.
Challenge - Isolation with Low Overhead

• KVS is fast, server processes requests in 2 μs.
  - Its performance extremely sensitive to any overhead, even cache misses.

• Security isolation incurs 3 sources of overhead:
  - The cost of safer languages.
  - Context switches between protection domains.
  - Interactions with DB across protection domain boundaries.
Approaches

**SQL**
- Difficult to implement new operators or complex algorithms.
- Leading to ad-hoc extensions such as SimSQL, SciDB etc.

**Native/C++**
- Flexible. Need process isolation, interactions happen over IPC.

**JavaScript**
- Flexible. Embedding V8 engine in DB process.
C++ vs. JavaScript

• Our expectation: JavaScript may be slower than C++:
  - JIT compiler doesn’t optimize as aggressively as C++ compiler.
  - Less static type information.
  - Garbage collection.
C++ vs. JavaScript

• Experiment setup:
  - Compare same query logics written in C++ and JavaScript.
  - We also compare these queries written in C++ and compiled to asm.js.
  - Queries process 1 GB of records with varied selectivity and compute intensity.
  - We don’t consider GC in this experiment, assume procedures are often short.
C++ vs. JavaScript

- For our memory intensive query, JavaScript is 27% slower than C++.
- Performance of asm.js is just 2% slower than C++. 

Q2: SELECT A+A
WHERE A < x

<table>
<thead>
<tr>
<th>Language</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>100%</td>
</tr>
<tr>
<td>JavaScript</td>
<td>73%</td>
</tr>
<tr>
<td>asm.js</td>
<td>98%</td>
</tr>
</tbody>
</table>
C++ vs. JavaScript

- For our compute intensive query, JavaScript is faster than C++.
- Glibc’s pow implementation may be the cause of slower performance of C++.

Q4: SELECT SUM(A^B^C^D) WHERE A < x

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>JavaScript</th>
<th>asm.js</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>100%</td>
<td>220%</td>
<td>95%</td>
</tr>
</tbody>
</table>
Isolation Costs, Process vs. V8

- Isolation of C++ code is done using process.
  - DB APIs invoked over IPC.
- Isolation of JavaScript code is done with V8::Context.
  - DB APIs invoked in the same process through wrappers.
Isolation Costs, Process vs. V8

- Measured the time of process switch and V8 context switch.
- V8::Contexts switch is 11.4x faster than processes.
Isolation Costs, Process vs. V8

• Measured the time of invoking a DB API.

• Invocation from JS is 72x faster than over IPC.

![Diagram showing the comparison between client process and JS script data access.]

Client process

\[
\text{null()} \\
\text{func()} \\
\text{2,242 ns}
\]

DB process

\[
\text{null()}
\]

Data access 72x faster

JS script

\[
\text{func()} \\
\text{31 ns}
\]

DB

\[
\text{null()}
\]
Evaluation Summary

• We compare SQL, C++ & JavaScript for their suitability of implementing our idea.
• SQL is ruled out for its limited generality.
• C++ is ruled out for high isolation overhead.
• JavaScript is promising with generality, performance and low isolation overhead.
Why Not Software Fault Isolation

- For SFI, interactions require copying data between client procedure and DB.
- For data intensive procedures, that means huge overhead.

![Diagram showing interactions in SFI](image.png)
Design

- Leverage scatter-gather list & zero copy DMA
- Leverage kernel bypassing networking (DPDK)
- Eliminating garbage collection
- Expose low level database abstractions

```javascript
V8::Context 1
var raws = getTable('t');
...
});
```
Conclusion & Research Questions

• Conclusion:
  - We propose JavaScript for extending low latency in-memory KV store.
  - The challenge is to keep overhead under a small fraction of 2 μs.
  - Evaluation shows JavaScript as a promising choice with low isolation cost and good performance.

• Call for feedbacks:
  - What interesting APIs can be built?
  - Is there other potentially better approaches that we overlooked?