Improve Application Performance with SQL Auto-Caching, No Code changes

Roland Lee
Heimdall Data
Agenda

1) Database Proxy Introduction
2) Demo
Web scale Challenges

More users and applications implies
- Higher data volumes
- More tables, changes to schema
- Higher latency responses
# Database Proxy Vendors

<table>
<thead>
<tr>
<th>Feature</th>
<th>Heimdall Data</th>
<th>ScaleArc</th>
<th>MariaDB</th>
<th>ProxySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Failover</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SQL Read/Write Splitting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Automated Cache invalidation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduces network latency</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IMDG or Database Proxy

- Amazon ElastiCache
- GridGain
- Pivotal GemFire
- redis
- hazelcast
- HEIMDALL
- MariaDB
- ScaleArc

- Best scale & performance
- Greenfield applications
- Requires code changes

- Good scalability & performance
- Existing applications, small dev
- No code changes
Heimdall Data Software Options

Application
Vendor Database Driver
Heimdall Data Proxy

Application Server

JDBC driver, .jar file
Heimdall Data JDBC
Vendor JDBC Driver

Application Server

Runs as an agent
PostgreSQL
MySQL

Any JDBC data source
PostgreSQL
Oracle

SQL Server
Heimdall Data Distributed Proxy

1. Heimdall is installed on each application instance
2. Direct Heimdall to the grid-cache of your choice (e.g. Redis, Hazelcast)
3. Heimdall will auto-cache and auto-invalidate SQL results to the look-aside cache
<table>
<thead>
<tr>
<th>Use case</th>
<th>Customer Benefit</th>
</tr>
</thead>
</table>
| SQL Results Caching            | • Auto-caching / Auto-invalidate  
• NO code changes                  |
| Automated failover             | • Faster failover for MySQL & SQL Server AlwaysOn  
• PGPool-II Replacement       |
| Batch Processing               | • Improves write performance  
• Batches singleton DML operations  
• Removes Commit overhead |
| Auditing for Privacy Compliance| • Logs data access: Who, what, when  
• GDPR, SOX, PCI, HIPPA       |
Heimdall Data Architecture

Application Servers

Heimdall Central Console

- SQL Analytics
- Audit Logging

Heimdall Data

- You choose the storage
- Heimdall auto-caches
- Heimdall auto-invalidates

Automated DB Failover

ElastiCache

• You choose the storage
• Heimdall auto-caches
• Heimdall auto-invalidates
Heimdall Data for Microservices

Heimdall Data Solution
- DB Failover
- Connection Pooling
- Caching
- SQL Analytics
- Audit logging
Application Side – SQL Caching

- Heimdall auto-caches
- 2-Tier Look-aside SQL cache
- Not a write-through cache
- Not a read-through cache
- Auto-invalidation
What to Cache?

Uses real-time analysis and statistics on:

• Query frequency and variability
• Relative performance of Cache vs. Database

Provides:

• Auto-cache only if there is a performance benefit
• Cache recommendations and benefits
Heimdall Query Analytics

Very cacheable. 700 µs per query

1. Cache
   SELECT user_id, meta_key, meta_value FROM wp_usermeta WHERE user_id IN (?) ORDER BY umeta_id ASC
   Cache Time (s) = 13
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 697.2
   Result Retrieval Time (ms) = 730.7
   Average Result Size = 1k

2. Cache
   SELECT * FROM wp_users WHERE ID = ?
   Cache Time (s) = 13
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 693.7
   Result Retrieval Time (ms) = 733.2
   Average Result Size = 310.9

3. Cache
   SELECT * FROM wp_posts WHERE ID = ? LIMIT 1
   Cache Time (s) = 8.9
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 756.1
   Result Retrieval Time (ms) = 836.8
   Average Result Size = 3k

4. Cache
   SELECT option_name, option_value FROM wp_options WHERE autoload = ?
   Cache Time (s) = 7.1
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 1026.5
   Result Retrieval Time (ms) = 1123.9
   Average Result Size = 21.9

5. Cache
   SELECT option_value FROM wp_options WHERE option_name = ? LIMIT 1
   Cache Time (s) = 6.8
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 691.7
   Result Retrieval Time (ms) = 707.6
   Average Result Size = 6.9

6. Cache
   SELECT wp_posts.* FROM wp_posts WHERE YEAR(wp_posts.post_date) = ? AND MONTH(wp_posts.post_date) = ? AND DAYOFMONTH(wp_posts.post_date) = 1 AND wp_posts.post_name = ? AND wp_posts.post_type = ? ORDER BY wp_posts.post_date DESC
   Cache Time (s) = 5.4
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 773.0
   Result Retrieval Time (ms) = 811.6
   Average Result Size = 3k

7. Cache
   SELECT t1.* FROM wp_terms AS t1 INNER JOIN wp_term_taxonomy AS t2 ON t1.term_id = t2.term_id INNER JOIN wp_term_relationships AS r ON r.term_taxonomy_id = t2.term_taxonomy_id WHERE tt.taxonomy IN (?, ?, ?) AND t1.object_id IN (?) ORDER BY t1.name ASC
   Cache Time (s) = 5.1
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 804.0
   Result Retrieval Time (ms) = 733.1
   Average Result Size = 784.2

8. Cache
   SELECT post_id, meta_key, meta_value FROM wp_postmeta WHERE post_id IN (?) ORDER BY meta_id ASC
   Cache Time (s) = 4.7
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 680.7
   Result Retrieval Time (ms) = 702.0
   Average Result Size = 1.0

9. Cache
   SELECT wp_posts.* FROM wp_posts WHERE ID IN (?, ?, ?, ?)
   Cache Time (s) = 2.4
   Cache Hit (%) = 100
   Cache Size = 100
   Query Response Time (ms) = 813.8
   Result Retrieval Time (ms) = 915.2
   Average Result Size = 13.0

10. Cache
    SELECT wp_comments.* FROM wp_comments WHERE comment_id IN (?, ?, ?, ?)
    Cache Time (s) = 2.3
    Cache Hit (%) = 100
    Cache Size = 100
    Query Response Time (ms) = 351.0
    Result Retrieval Time (ms) = 771.2
    Average Result Size = 5.9

11. Cache
    SELECT wp_posts.ID FROM wp_posts WHERE ID = ? AND wp_posts.post_type = ? AND wp_posts.post_status = ? ORDER BY wp_posts.post_date DESC LIMIT 1
    Cache Time (s) = 2.3
    Cache Hit (%) = 100
    Cache Size = 100
    Query Response Time (ms) = 339.0
    Result Retrieval Time (ms) = 791.5
    Average Result Size = 811.0

SELECT SQL_CALC_FOUND_ROWS wp_comments.comment_id FROM wp_comments WHERE
Heimdall Cache Rules Analytics

Before: Query response time 700 $\mu$s per query. Significant repetitive database load.

<table>
<thead>
<tr>
<th>Rank</th>
<th>One Click Optimize</th>
<th>Query</th>
<th>Server Time (%)</th>
<th>Duplicate Query &amp; Response (%)</th>
<th>Cache Index</th>
<th>Cache Time (s)</th>
<th>Cache Hit (%)</th>
<th>Count</th>
<th>Query Response Time (μs)</th>
<th>Result Retrieval Time (μs)</th>
<th>Average Result Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cache</td>
<td>SELECT user_id, meta_key, meta_value FROM wp_usermeta WHERE user_id IN (?) ORDER BY umeta_id ASC</td>
<td>13</td>
<td>100</td>
<td>100</td>
<td>86400</td>
<td>0.0</td>
<td>2.1k</td>
<td>697.2</td>
<td>730.7</td>
<td>1k</td>
</tr>
<tr>
<td>2</td>
<td>Cache</td>
<td>SELECT * FROM wp_users WHERE ID = ?</td>
<td>13</td>
<td>100</td>
<td>100</td>
<td>86400</td>
<td>0.0</td>
<td>2.1k</td>
<td>693.7</td>
<td>733.2</td>
<td>310.0</td>
</tr>
<tr>
<td>3</td>
<td>Cache</td>
<td>SELECT * FROM wp_posts WHERE ID = ? LIMIT ?</td>
<td>8.9</td>
<td>100</td>
<td>100</td>
<td>86400</td>
<td>0.0</td>
<td>1.4k</td>
<td>756.1</td>
<td>836.8</td>
<td>2.9k</td>
</tr>
</tbody>
</table>

After: Cached response in 70 $\mu$s per query. Reduced database load.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Caching</th>
<th>SELECT user_id, meta_key, meta_value FROM wp_usermeta WHERE user_id IN (?) ORDER BY umeta_id ASC</th>
<th>1.8</th>
<th>100</th>
<th>100</th>
<th>86400</th>
<th>99.9</th>
<th>55k</th>
<th>68.4</th>
<th>68.4</th>
<th>1.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Caching</td>
<td>SELECT * FROM wp_users WHERE ID = ?</td>
<td>1.6</td>
<td>100</td>
<td>100</td>
<td>86400</td>
<td>99.9</td>
<td>54.9k</td>
<td>59.5</td>
<td>59.5</td>
<td>1.3</td>
</tr>
<tr>
<td>21</td>
<td>Caching</td>
<td>SELECT * FROM wp_posts WHERE ID = ? LIMIT ?</td>
<td>1.4</td>
<td>100</td>
<td>100</td>
<td>86400</td>
<td>97.8</td>
<td>35.4k</td>
<td>77.8</td>
<td>79.3</td>
<td>63.5</td>
</tr>
</tbody>
</table>
Other use cases
Other use cases
Use Case #1: Read / Write Splitting

1. Automate use of read replicas (read/write splits)
2. Replication lag detection to ensure data freshness
3. Scale out the database with no application changes!
Use Case #2: Async DML Batching

Benefits:
- Lower CPU overhead due to fewer commits
- Improved application response time
- Improved DML scale

Exceptions are logged, removed from batch, and transaction restarted
Use Case #3: SQL Analytics

Identifies performance bottleneck
- Application?
- Database?
- Network?
Thank you

Roland Lee
Heimdall Data