Consistency-preserving Caching of Dynamic Database Content

Niraj Tolia
M. Satyanarayanan

Carnegie Mellon University
Motivation

- Easy to geographically distribute web and app. servers
- Harder to distribute databases
  - Pick Two: Consistency, Availability, Tolerance to Partitions
- How can you optimize the use of the WAN?
Outline

- Motivation
- Ganesh
  - Overview
  - Design and Implementation
- Evaluation
- Conclusion
High-level Overview of Ganesh

- Optimizes WAN usage for database accesses
  - Without interpreting queries or results
  - In a database-independent manner
- Uses a Content Addressable cache to
  - Detect similarity between query results
  - Eliminate redundancy over WAN link
- Tradeoff increased computation for network savings
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

Web and App. Server

SELECT queries

Database Server
Finding Result Similarity
(A 10,000 foot view)

Web and App. Server

SELECT queries

Database Server
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)
Finding Result Similarity

(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

Web and App. Server

SELECT queries

Database Server
Finding Result Similarity
(A 10,000 foot view)

Web and App. Server

Database Server

SELECT queries
Finding Result Similarity
(A 10,000 foot view)

SELECT queries

UPDATE queries are not optimized
Design Goals

- Transparent
  - To both the application server and the database
- Does not weaken consistency
- Efficiently detects similarity
Design - Transparency

- Doesn’t require modifications to the application and database server
Design - Transparency

- Doesn’t require modifications to the application and database server
Design - Transparency

- Doesn’t require modifications to the application and database server
Design - Transparency

- Doesn’t require modifications to the application and database server
Design - Transparency

- Doesn’t require modifications to the application and database server
Proxy-based Architecture

- Ganesh JDBC Driver
  - Thin but smart -- conservative network use
  - Contains in-memory cache
    - Caches previous results at different granularities
    - Uses a LRU replacement policy

- Ganesh Proxy
  - Forwards queries to the database
  - Eliminates redundancy in results
Detecting Similarity

- Ganesh uses Content Addressability
  - Hash value is a globally unique identifier
    - Independent of any particular system
    - Infeasible to find another object with same hash
    - If hash values are equal, so are the source objects
  - Any small change in source completely changes hash

Diagram: Result → Cryptographic Hash → Hash
Exploiting Structure

Exploit structure in results – they look like tables

```
SELECT name, address, zip, email FROM USERS
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>T1L1J4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>
Exploiting Structure

- Exploit structure in results – they look like tables

```
SELECT name, address, zip, email FROM USERS
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>TILL4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>

Result Hash

---

28
Exploiting Structure

- Exploit structure in results – they look like tables

```
SELECT name, address, zip, email FROM USERS
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>T1L1J4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>
Exploiting Structure

- Exploit structure in results – they look like tables

```
SELECT name, address, zip, email FROM USERS
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>T1L1J4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>

- Row Hash
- Result Hash
Exploiting Structure

Exploit structure in results – they look like tables

SELECT name, address, zip, email FROM USERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>T1L1J4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>

Row Hash₁

Row Hash₂

Result Hash
Exploiting Structure

- Exploit structure in results – they look like tables

```sql
SELECT name, address, zip, email FROM USERS
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>T1L1J4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>

Row Hash\(_1\), Row Hash\(_2\), Row Hash\(_3\), Result Hash
Exploiting Structure

- Exploit structure in results – they look like tables

```
SELECT name, address, zip, email FROM USERS
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Zip</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>412 Avenue</td>
<td>15213</td>
<td><a href="mailto:jd2@eg.com">jd2@eg.com</a></td>
</tr>
<tr>
<td>Mary Major</td>
<td>821 Lane</td>
<td>15232</td>
<td><a href="mailto:mm@eg.com">mm@eg.com</a></td>
</tr>
<tr>
<td>John Stiles</td>
<td>701 Street</td>
<td>TILLJ4</td>
<td><a href="mailto:js@eg.com">js@eg.com</a></td>
</tr>
</tbody>
</table>

- This process does not interpret data
Putting it all together
Putting it all together

Web and App. Server → Query₁ → Database Server
Putting it all together

Web and App. Server

Query

0x2fd4e1c6
0x8c3aea3b

Database Server
Putting it all together
Putting it all together
Putting it all together
Putting it all together
Putting it all together

Web and App. Server

Database Server

0x2fd4e1c6
0xda39a3ee

0x2fd4e1c6
0x8c3aea3b
Putting it all together

Web and App. Server
0x2fd4e1c6
0xda39a3ee

Database Server
0x2fd4e1c6
0x8c3aea3b
Putting it all together

Web and App. Server

Query$_2$

Database Server

0x2fd4e1c6
0xda39a3ee

0x2fd4e1c6
0x8c3aea3b
Putting it all together
Putting it all together
Putting it all together
Putting it all together
Putting it all together
Putting it all together
Putting it all together

Web and App. Server

Database Server

0x2fd4e1c6
0xda39a3ee

✓ 0x2fd4e1c6
× 0x8c3aea3b
Putting it all together
Putting it all together

Web and App. Server

Database Server

0x2fd4e1c6
0xda39a3ee
0x8c3aea3b
Putting it all together

- No explicit cache-coherence algorithm needed
Outline

- Motivation
- Ganesh
  - Overview
  - Design and Implementation
- Evaluation
  - Benchmarks and Experimental Setup
  - Results
  - Did exploiting structure help?
- Conclusion
Evaluation – Benchmarks

- **AUCTION (RUBiS)** – Models eBay
  - Browsing, bidding, add auctions and feedback, …

- **BBOARD (RUBBoS)** – Models Slashdot
  - Reading articles, adding comments, moderating, …

- Benchmarks have RW and RO variants

- **Performance metrics**
  - Throughput (Requests/sec)
  - Latency (Average request response time)
Experimental Setup

- Two configurations:
  - Native: Unmodified Setup
Experimental Setup

- Two configurations:
  - Native: Unmodified Setup
  - Ganesh: Content-based optimizations used
Experimental Setup

- Two configurations:
  - Native: Unmodified Setup
  - Ganesh: Content-based optimizations used

- Evaluation Parameters
  - 5 Mbit/s + 66ms, 20 Mbit/s + 33ms, 100 Mbit/s
  - 400, 800, 1200, and 1600 clients
AUCTION – Throughput

Read-Only

Native  Ganesh
Higher is better

Requests / sec

5 Mbit/s  20 Mbit/s  100 Mbit/s

Test Clients

400  800  1200  1600

400  800  1200  1600

400  800  1200  1600

60
AUCTION - Latency

Read-Only

Lower is better

Avg. Resp. Time (sec)

5 Mbit/s

20 Mbit/s

100 Mbit/s

Test Clients
BBOARD - Throughput

Read-Write

Requests / sec

Native
Ganesh
Higher is better

Test Clients

5 Mbit/s
20 Mbit/s
100 Mbit/s

400
800
1200
1600
**BBOARD - Latency**

*Read-Write*

- **Avg. Resp. Time (sec)**
  - **Native**
  - **Ganesh**
  - Lower is better

<table>
<thead>
<tr>
<th>Test Clients</th>
<th>5 Mbit/s</th>
<th>20 Mbit/s</th>
<th>100 Mbit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Could we have used Rabin fingerprinting?
- Extensive use in storage systems
- Chunks data using a stochastic process
- Works well for in-place updates, deletes, insertions
- Does not work well for query results
  - Reordering of data (ORDER BY)
  - Also hard to pick average chunk size
Rabin vs. Exploiting Structure

BBOARD - Read-Only

<table>
<thead>
<tr>
<th>Test Clients</th>
<th>5 Mb/s</th>
<th>20 Mb/s</th>
<th>100 Mb/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Norm. Throughput

- 0.0
- 0.2
- 0.4
- 0.6
- 0.8
- 1.0

5 Mb/s  20 Mb/s  100 Mb/s
Related Work

- Caching Dynamic Database Content
  - DBCache, DBProxy, MTCache, …
  - Per-application consistency model [Gao03, GlobeDB]
  - Backend-scalability [C-JDBC, SSS, Ganeymed]

- Content-Addressable Systems
  - TCP-level duplicate elimination [Riverbed, Spring00]
  - P2P backup [Pastiche], Storage [Venti, SiS]
  - Distributed File Systems [LBFS, CASPER, PAST]
Conclusion

- Ganesh: Optimizes database access over the WAN
  - Transparent
  - Does not weaken consistency
- Prototype built around Java and the JDBC interface