EIGER: STRONGER SEMANTICS FOR LOW-LATENCY GEO-REPLICATED STORAGE

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen

Presenter: MeiXing Dong
OUTLINE

• Background
• Eiger
• Evaluation
• Conclusion
GEO-REPLICATED STORAGE

- Backend of massive websites

[Facebook logo]
[Amazon logo]
[Reddit logo]
GEO-REPLICATED STORAGE

- Want to serve requests quickly
- 100 ms of latency costs Amazon 1% in sales!
GEO-REPLICATED STORAGE

• Impossible to have both low latency and strong consistency!
• Choose to optimize for latency or consistency
• Eiger
  • have low latency AND consistency stronger than eventual-consistency (causal consistency)
  • rich column-family data model (built on open-source Cassandra distributed system)
  • atomic updates and transactions
COLUMN-FAMILY DATA MODEL

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Town</th>
<th>Friends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337</td>
<td>Alice</td>
<td>NYC</td>
<td>[2664, …]</td>
</tr>
<tr>
<td>2664</td>
<td>Bob</td>
<td>LA</td>
<td>[1337, …]</td>
</tr>
</tbody>
</table>
COLUMN-FAMILY DATA MODEL

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Town</th>
<th>Friends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337</td>
<td>Alice</td>
<td>NYC</td>
<td>[2664, …]</td>
</tr>
<tr>
<td>2664</td>
<td>Bob</td>
<td>LA</td>
<td>[1337, …]</td>
</tr>
</tbody>
</table>

![Table and diagram](image)

**Figure 1**: An example use of the column-family data model for a social network setting.
Figure 1: An example use of the column-family data model for a social network setting.
COLUMNS-FAMILY DATA MODEL

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Town</th>
<th>Friends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337</td>
<td>Alice</td>
<td>NYC</td>
<td>[2664, ...]</td>
</tr>
<tr>
<td>2664</td>
<td>Bob</td>
<td>LA</td>
<td>[1337, ...]</td>
</tr>
</tbody>
</table>

**Figure 1:** An example use of the column-family data model for a social network setting.
### Column-Family Data Model

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Town</th>
<th>Friends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337</td>
<td>Alice</td>
<td>NYC</td>
<td>[2664, ...]</td>
</tr>
<tr>
<td>2664</td>
<td>Bob</td>
<td>LA</td>
<td>[1337, ...]</td>
</tr>
</tbody>
</table>

**Figure 1:** An example use of the column-family data model for a social network setting.
COLUMN-FAMILY DATA MODEL

- Each location represented as compound key and a value - Alice:Assocs:Friends:Bob -> 3/2/11

Figure 1: An example use of the column-family data model for a social network setting.
CLIENT API

- `batch_mutate({key->mutation})`
- `atomic_mutate({key->mutation})`
- `multiget_slice({key, column_parent, slice_predicate})`

![Figure 1: An example use of the column-family data model for a social network setting.](image)
CLIENT API

- `batch_mutate({Alice->insert(UserData:Town=Paris), Bob->insert(UserData:Town=Paris)})`

- Doesn’t matter if the towns change at the same time

---

**Figure 1:** An example use of the column-family data model for a social network setting.
CLIENT API

- atomic_mutate({Alice->delete(Assocs:Friends:Bob), Bob->delete(Assocs:Friends:Alice)})

- Don’t want awkward situation if things don’t work out...

Figure 1: An example use of the column-family data model for a social network setting.
CLIENT API

- multiget_slice ({Alice, Assocs:Friends, (B, C, 10)})
- Just checking that Bob is gone
- Is read-only transaction

![Table]

**Figure 1:** An example use of the column-family data model for a social network setting.
COUNTER COLUMNS

- can be commutatively updated
- don’t need to read first, modify the original value, then write

Figure 1: An example use of the column-family data model for a social network setting.
**CAUSAL CONSISTENCY**

- Carol: *removes boss from friends list*
- Carol: “I hate my job! So glad I’m quitting tomorrow.”
CARASAL CONSISTENCY

- Carol: *removes boss from friends list*
- Carol: “I hate my job! So glad I’m quitting tomorrow.”
- Carol: “I hate my job! So glad I’m quitting tomorrow.”
- Carol: *removes boss from friends list*
CAUSAL CONSISTENCY

- Partial order over operations using potential causality
- 3 rules
  - Thread-of-Execution – operation performed by thread is causally after all of its previous ones
  - Reads-From – operation that reads a value is causally after the operation that wrote the value
  - Transitive-Closure – a after b, b after c -> a after c
CAUSAL CONSISTENCY

- only need to keep track of whether nearest dependencies were satisfied (applied in cluster)
CAUSAL CONSISTENCY

<table>
<thead>
<tr>
<th>User</th>
<th>Op ID</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>$w_1$</td>
<td>insert(Alice, “-,Town”, NYC)</td>
</tr>
<tr>
<td>Bob</td>
<td>$r_2$</td>
<td>get(Alice, “-,Town”)</td>
</tr>
<tr>
<td>Bob</td>
<td>$w_3$</td>
<td>insert(Bob, “-,Town”, LA)</td>
</tr>
<tr>
<td>Alice</td>
<td>$r_4$</td>
<td>get(Bob, “-,Town”)</td>
</tr>
<tr>
<td>Carol</td>
<td>$w_5$</td>
<td>insert(Carol, “Likes, NSDI”, 8/31/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>$w_6$</td>
<td>insert(Alice, “Likes, NSDI”, 9/1/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>$r_7$</td>
<td>get(Carol, “Likes, NSDI”)</td>
</tr>
<tr>
<td>Alice</td>
<td>$w_8$</td>
<td>insert(Alice, “Friends, Carol”, 9/2/12)</td>
</tr>
</tbody>
</table>

Figure 2: (a) A set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.
CAUSAL CONSISTENCY

<table>
<thead>
<tr>
<th>User</th>
<th>Op ID</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>w₁</td>
<td>insert(Alice, “-,Town”, NYC)</td>
</tr>
<tr>
<td>Bob</td>
<td>r₂</td>
<td>get(Alice, “-,Town”)</td>
</tr>
<tr>
<td>Bob</td>
<td>w₃</td>
<td>insert(Bob, “-,Town”, LA)</td>
</tr>
<tr>
<td>Alice</td>
<td>r₄</td>
<td>get(Bob, “-,Town”)</td>
</tr>
<tr>
<td>Carol</td>
<td>w₅</td>
<td>insert(Carol, “Likes, NSDI”, 8/31/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>w₆</td>
<td>insert(Alice, “Likes, NSDI”, 9/1/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>r₇</td>
<td>get(Carol, “Likes, NSDI”)</td>
</tr>
<tr>
<td>Alice</td>
<td>w₈</td>
<td>insert(Alice, “Friends, Carol”, 9/2/12)</td>
</tr>
</tbody>
</table>

Figure 2: (a) A set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.
CAUSAL CONSISTENCY

- only need to keep track of whether nearest dependencies were satisfied (applied in cluster)
- nearest dependencies – longest path of one hop to current operation
### Causal Consistency

<table>
<thead>
<tr>
<th>User</th>
<th>Op ID</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>$w_1$</td>
<td>insert(Alice, “-,Town”, NYC)</td>
</tr>
<tr>
<td>Bob</td>
<td>$r_2$</td>
<td>get(Alice, “-,Town”)</td>
</tr>
<tr>
<td>Bob</td>
<td>$w_3$</td>
<td>insert(Bob, “-,Town”, LA)</td>
</tr>
<tr>
<td>Alice</td>
<td>$r_4$</td>
<td>get(Bob, “-,Town”)</td>
</tr>
<tr>
<td>Carol</td>
<td>$w_5$</td>
<td>insert(Carol, “Likes, NSDI”, 8/31/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>$w_6$</td>
<td>insert(Alice, “Likes, NSDI”, 9/1/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>$r_7$</td>
<td>get(Carol, “Likes, NSDI”)</td>
</tr>
<tr>
<td>Alice</td>
<td>$w_8$</td>
<td>insert(Alice, “Friends, Carol”, 9/2/12)</td>
</tr>
</tbody>
</table>

**Figure 2:** (a) A set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.
CAUSAL CONSISTENCY

- only need to keep track of whether nearest dependencies were satisfied (applied in cluster)
- nearest dependencies – longest path of one hop to current operation
- one-hop dependencies – shortest path of one hop to current operation
- don’t need to store dependency info at servers
### Causal Consistency

#### Table of Operations

<table>
<thead>
<tr>
<th>User</th>
<th>Op</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>w₁</td>
<td>insert(Alice, “-, Town”, NYC)</td>
</tr>
<tr>
<td>Bob</td>
<td>r₂</td>
<td>get(Alice, “-, Town”)</td>
</tr>
<tr>
<td>Bob</td>
<td>w₃</td>
<td>insert(Bob, “-, Town”, LA)</td>
</tr>
<tr>
<td>Alice</td>
<td>r₄</td>
<td>get(Bob, “-, Town”)</td>
</tr>
<tr>
<td>Carol</td>
<td>w₅</td>
<td>insert(Carol, “Likes, NSDI”, 8/31/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>w₆</td>
<td>insert(Alice, “Likes, NSDI”, 9/1/12)</td>
</tr>
<tr>
<td>Alice</td>
<td>r₇</td>
<td>get(Carol, “Likes, NSDI”)</td>
</tr>
<tr>
<td>Alice</td>
<td>w₈</td>
<td>insert(Alice, “Friends, Carol”, 9/2/12)</td>
</tr>
</tbody>
</table>

#### Diagram

- **Figure 2:**
  - (a) A set of example operations;
  - (b) the graph of causality between them;
  - (c) the corresponding dependency graph; and
  - (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.
OUTLINE

• Background
• Eiger
• Evaluation
• Conclusion

• System Design
• Read-Only Transactions
• Write-Only Transactions
• Failure
SYSTEM DESIGN

• Assumptions
  • Keyspace is partitioned across logical servers
SYSTEM DESIGN

- Assumptions
  - Keyspace is partitioned across logical servers
  - Linearizability provided inside datacenters
SYSTEM DESIGN

• Assumptions
  • Keyspace is partitioned across logical servers
  • Linearizability provided inside datacenters
  • Keys are stored on logical servers, implemented w/ replicated state machines
TRACKING DEPENDENCIES

• based on operations (not values or key versions)
• decrease # dependency checks
TRACKING DEPENDENCIES

• based on operations (not values or key versions)
• decrease # dependency checks
• dependency consists of locator and unique id
  • tell other nodes which node to look at to see if operation has been committed
• single key
TRACKING DEPENDENCIES

• based on operations (not values or key versions)
• decrease # dependency checks
• dependency consists of locator and unique id
  • map dependencies to operations
  • operation’s timestamp
TRACKING DEPENDENCIES

• Check dependencies
  • send dep_check operation to owner of locator
  • owner checks if operation was applied
  • if yes: respond
  • if no: block dep_check until operation applied
• Wait for all dep_checks to return
CLIENT LIBRARY

- Clients access local Eiger datacenter using a client library
  - mediates access to nodes
  - executes read/write transaction algorithms
  - tracks causality and attaches dependencies to write operations.
- Client tracks dependencies on per-user basis
- When write is issued, library attaches dependencies on previous write and all observed writes
EIGER’S BASIC OPERATIONS

• Every machine maintains a logical clock
EIGER’S BASIC OPERATIONS

• Every machine maintains a logical clock
• Write ops – replace current column, timestamped w/ current logical time at the server applying write
EIGER’S BASIC OPERATIONS

- Every machine maintains a logical clock
- Write ops – replace current column, timestamped w/ current logical time at the server applying write
- Read ops – return current column (or empty column w/ deleted bit set)
EIGER’S BASIC OPERATIONS

- Every machine maintains a logical clock
- Write ops – replace current column, timestamped w/ current logical time at the server applying write
- Read ops – return current column (or empty column w/ deleted bit set)
- Replication – send writes to servers in other datacenters that own same keyspace
EIGER’S BASIC OPERATIONS

• Every machine maintains a logical clock
• Write ops – replace current column, timestamped w/ current logical time at the server applying write
• Read ops – return current column (or empty column w/ deleted bit set)
• Replication – send writes to servers in other datacenters that own same keyspace
• Last writer wins
READ-ONLY TRANSACTIONS

- Transaction Algorithm
  - each data location marked with *earliest valid time* (EVT) – time when locally written
  - when queried, server responds with *latest valid time* (LVT) – time when value was read by server
  - values valid between EVT and LVT
READ-ONLY TRANSACTIONS

• Two-Round Read Protocol
  • round 1 – read all values
  • round 2 – if values not valid during round 1, client requests reads at effective time of transaction
    • minimum LVT (>= maximum EVT)
    • store values since last access time (in case asked for during round 2)
READ-ONLY TRANSACTIONS

- Two-Round Read Protocol

**Figure 4:** Examples of read-only transactions. The effective time of each transaction is shown with a gray line; this is the time requested for location 1 in the second round in (b).
WRITE-ONLY TRANSACTIONS

• split atomic_mutate request into one sub-request for each local server involved
• randomly choose one key as coordinator key
• use 2PC-PCI (two-phase commit with positive cohorts and indirection)
WRITE-ONLY TRANSACTIONS

• 2PC-PCI
  • always commits transaction once a vote is received from all cohorts
  • *indirects* request through coordinator if a cohort (replica) can’t answer a query
  • local: client library directly sends each participant its sub-request, is implicit PREPARE message
  • replicated: send to equivalent participants in remote datacenter
WRITE-ONLY TRANSACTIONS

• Local Write-Only Transactions
  • \textit{participant} server writes “pending” value
  • sends YESVOTE to coordinator
  • once all YESVOTE received, coordinator commits transaction
  • cohort replace “pending” with real value when receive COMMIT, then send ACK
  • coordinator cleans up after receiving all ACKs
WRITE-ONLY TRANSACTIONS

- Replicated Write-Only Transactions
  - each sub-request sent to equivalent server in remote datacenter
  - remote cohort sends NOTIFY w/ key count to its local coordinator
  - coordinator checks dependencies, waits for all NOTIFY messages, then sends PREPAREs and proceeds normally
FAILURE

- single server failure
  - single logical server composed of multiple physical servers
- meta-client (web browser) redirection
  - service providers detect redirection, redirect to original datacenter
- entire datacenter failure
  - permanent failure -> data loss
OUTLINE

• Background

• Eiger

• Evaluation

• Conclusion

• Setup

• Experiments
EXPERIMENTAL SETUP

- Shared VICCI testbed
  - distributed virtualized compute clusters and networking hardware
  - Linux VServer (virtual private server) instances
  - 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, 2x1 GigE network ports
- Datapoints are medians of 5+ trials
LATENCY MICRO-BENCHMARK

- Cluster of 2 machines at Princeton, Stanford and UW, run from UW on single client thread
- Compare read/write in Eiger vs Cassandra configs
  - Eventual (R=1, W=1)
  - Strong-A (R=3, W=1)
  - Strong-B (R=2, W=2)
  - R/W - # of datacenters involved in read/write
LATENCY MICRO-BENCHMARK

- Eiger – overall fast reads and writes, not as fast as Cassandra Eventual
WRITE TRANSACTION COST

Figure 7: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.
Figure 7: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.
Figure 7: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.
FACEBOOK WORKLOAD

Table 4: Throughput for the Facebook workload.

<table>
<thead>
<tr>
<th></th>
<th>Ops/sec</th>
<th>Keys/sec</th>
<th>Columns/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>23,657</td>
<td>94,502</td>
<td>498,239</td>
</tr>
<tr>
<td>Eiger</td>
<td>22,088</td>
<td>88,238</td>
<td>466,844</td>
</tr>
<tr>
<td>Eiger All Txns</td>
<td>22,891</td>
<td>91,439</td>
<td>480,904</td>
</tr>
<tr>
<td>Max Overhead</td>
<td>6.6%</td>
<td>6.6%</td>
<td>6.3%</td>
</tr>
</tbody>
</table>

- Eiger is close to Cassandra
FACEBOOK WORKLOAD

Figure 9: Normalized throughput of $N$-server clusters for the Facebook TAO workload. Bars are normalized against the 1-server cluster.
OUTLINE

• Background

• Eiger

• Evaluation

• Conclusion
CONCLUSION

- Eiger has low latency with causal consistency (stronger than eventual consistency)
- Comparable latency/throughput to Cassandra
- Read-only and write-only transaction algorithms
QUESTIONS?