Caching: Turbocharging Your Application Workloads

Damon LaCaille, Sr. Solutions Architect
Zach Gardner, Sr. Solutions Architect
Table of contents

- In-memory Datastore Fundamentals
- Caching Concepts
- Caching Demo
- Lazy Loading Pattern
- Write-Through Pattern
- Amazon ElastiCache for Redis
- Best Practices
- Q&A
In-memory Datastore Fundamentals
## Datastore Trait Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Disk</th>
<th>In-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes / Reads</td>
<td>Disk</td>
<td>Memory</td>
</tr>
<tr>
<td>Engine latency</td>
<td>Milliseconds (ms)</td>
<td>Microseconds (µs)</td>
</tr>
<tr>
<td>Performance bottleneck</td>
<td>Disk</td>
<td>Network</td>
</tr>
<tr>
<td>Throughput</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Data</td>
<td>Data models</td>
<td>Rich data structures</td>
</tr>
</tbody>
</table>
Why Performance Matters

“A 100-millisecond delay in website load time can hurt conversion rates by 7 percent.”

"A two-second delay in web page load time increases bounce rate by 103 percent."

– 2017 Akamai Study

Why Performance Matters

Customers

Slow site?

https://www.businessnewsdaily.com/15160-slow-retail-websites-lose-customers.html
The Need for Speed

**FAST**: Memory is at least 50x faster than SSDs

**PREDICTABLE**: Key-based index, no disk seek time

μs is the new ms

Amazon ElastiCache
Caching Concepts
Caching Concepts

- Improves query response time
- Relieves pressure from services
- Allows new workloads on existing services
Each page access requires multiple database queries.
Caching Concepts - Redis Data Structures

ElastiCache for Redis Node

- List
- Hash
- Set
- Sorted Set
- Stream
- Geospatial
- HyperLogLog

Client

© 2020, Amazon Web Services, Inc. or its Affiliates.
Caching Concepts - Redis Data Structures

Most commonly used data structures for caching use case

- **String**: "Hi there" 1 .. 2 .. 3 .. n
- **Hash**: 
  `{ A:"hello", B:"wonderful", C:"world" }`
- **List**: 
  `[ A -> B -> C -> D -> E ]`
- **Set**: 
  `{ D, A, E, C, B }`
- **Sorted Set**: 
  `{ A:1, B:2, C:3, D:4, E:5 }`
- **Geospatial**: 
  `{ A:(32.1,34.7), B:(51.5,0.12) }`
- **HyperLogLog**: 
  `01101110 00100010 01101101`
- **Stream**: 
  `... msg1 ... msg2 ... msg3`
## Caching Concepts - Redis Data Structures

**Hash** represents a table structure  
Useful with Java HashMap, Python Dictionary, Database Row

```bash
> hgetall app1:customer:1
1) "first_name"
2) "Carlos"
3) "last_name"
4) "Salazar"
```

**String** is binary safe  
Useful with Strings, Numbers, Serialized Objects

```bash
> get app1:my_key_name
"\x80\x03(K\xa\x00\x00\x00..."
```

**Sorted Set** tracks unique, ordered entries  
Useful with any sorted Set-type structure

```bash
> zrevrange items_by_zip 0 -1
1) "78723"
2) "500"
3) "78701"
4) "329"
5) "78705"
6) "303"
```

**Set** tracks unique, unordered entries  
Useful with any Set-type structure

```bash
> smembers app1:set:valid_zips
1) "98105"
2) "98101"
3) "98109"
```
Demo
Lazy Loading Pattern
Lazy Loading

1. Read from cache
2. Read from source (if miss)
3. Write to cache

Advantages
- Avoids unnecessary data in cache
- Cache can be repopulated at anytime
- Immediate benefit

Disadvantages
- Cache miss may be expensive
- Data "freshness" factor
Lazy Loading

```
> SELECT
  customers.name, SUM(orders.num_items)
FROM
  customers, orders
WHERE
  customers.id = X AND
  orders.customer_id = customers.id
```

MD5 Hash
```
"ecf361704f15aa0e26e3b24e1ce6d1d6"
```

Key
```
ecf361704f15aa0e26e3b24e1ce6d1d6
```

Value
```
"\x80\x03M\xd2\x06cdecimal\nDecimal\n..."
```
def fetch(sql):
    key = get_md5_hash(sql)

    if r.get(key) is not None:
        return pickle.loads(value)
    else:
        cursor = m.cursor()
        cursor = execute(sql)
        value = cursor.fetchall()
        r.setex(key, TTL, pickle.dumps(value))
        return value

1. Read from cache
2. Read from source (if miss)
3. Write to cache
def fetch(sql):
    key = get_md5_hash(sql)

    if r.get(key) is not None:
        return pickle.loads(value)

    else:
        cursor = m.cursor()
        cursor = execute(sql)
        value = cursor.fetchall()
        r.setex(key, TTL, pickle.dumps(value))
        return value

SELECT COUNT(*) FROM users WHERE . . .

SELECT customers.customer_id, reviews.review_id
FROM customers, reviews
WHERE . . .
Amazon RDS Caching Example

```python
import pymysql, redis, pickle, hashlib
m = pymysql.connect('rds_host', ...)
r = redis.Redis(host='elasticache_endpoint', ...)

### Pass in SQL string
sql = "SELECT ... FROM ..."

### Convert SQL string into a shorter, unique hash for use as Redis key
key = hashlib.sha224(sql.encode('utf-8')).hexdigest()

### Check for value in Redis
value = r.get(key)

if value is None:
    print("Cache Miss")
    # Fetch result set from RDS
    cursor=m.cursor()
    cursor.execute(sql)
    value = cursor.fetchall()
    # Store full result set in Redis
    r.setex(key, pickle.dumps(value))
    return value

else:
    # Return cached result set from Redis
    print("Cache Hit")
    return pickle.loads(value)
```

Step 1: Query cache

Step 2: Read from source

Step 3: Write to cache
Amazon S3 Caching Example

import boto3
import redis

r = redis.StrictRedis(host="elasticache_endpoint", port=6379)
s3 = boto3.resource('s3')

### Pass in s3_bucket_name and s3_object_key and check to see if value is in Redis
value = r.get(s3_bucket_name + ':' + s3_object_key)

if value is None:
    print("Cache Miss")
    ### Get data from S3
    obj = s3.Object(s3_bucket_name, s3_object_key)
    data = obj.get()['Body'].read().decode('utf-8')
    ### Store the data into Redis
    r.set(s3_bucket_name + ':' + s3_object_key, data)
else:
    print("Cache Hit")
    print("Data retrieved from redis = " + value)
Write-Through Pattern
Write-Through

Example 1

1. Write to source
2. Triggers AWS Lambda function
3. Lambda function updates cache

Advantages
- Data is never stale
- Write latency better tolerated by customers vs. read latency

Disadvantages
- Unnecessary data, cache churn
- Delayed vs. immediate benefit
**Write-Through**

**Example 2**

1. Write to source
2. Write to cache

**Advantages**
- Data is never stale
- Write latency better tolerated by customers vs. read latency

**Disadvantages**
- Unnecessary data, cache churn
- Delayed vs. immediate benefit
Write-Through - Python example

```
def update_location(name, new_loc):
    # Write Through – Step 1: Update DB
    cursor = m.cursor()
    sql = "UPDATE... WHERE userid = " + str(name) + "...
    cursor.execute(sql)
    db.commit()

    # Write Through – Step 2: Update Cache
    r.hset("userid:" + str(name), "location", new_loc)

update_location("mike", "DFW")
```

<table>
<thead>
<tr>
<th>Hash key name</th>
<th>Field name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;userid:mike&quot;</td>
<td>&quot;location&quot;</td>
<td>&quot;DFW&quot;</td>
</tr>
</tbody>
</table>
Write-Through - Python example

userid:mike

- first: Michael
- team: ElastiCache
- location: DFW

> HSET userid:mike location DFW
OK

> HGETALL userid:mike
1) "first"
2) "Michael"
3) "team"
4) "ElastiCache"
5) "location"
6) "DFW"
# Caching Strategies - Summary

<table>
<thead>
<tr>
<th>Lazy Load</th>
<th>Write-Through</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only caches data that is read</td>
<td>Caches data that may never be read</td>
</tr>
<tr>
<td>Cache populates when reads occur</td>
<td>Cache populates when updates occur</td>
</tr>
<tr>
<td>Immediate benefit</td>
<td>Delayed benefit</td>
</tr>
<tr>
<td>Latency from cache miss is expensive</td>
<td>Write latency better tolerated vs. read latency</td>
</tr>
<tr>
<td>Data can get stale</td>
<td>Data is never stale</td>
</tr>
<tr>
<td>Often uses String *</td>
<td>Often uses Hash *</td>
</tr>
</tbody>
</table>

You can combine these strategies!

*Commonly used but not limited to this data type*
Amazon ElastiCache for Redis
Amazon ElastiCache for Redis

**Fully managed**
AWS manages all hardware and software setup, configuration, monitoring.

**Extreme performance**
In-memory data store and cache for sub-millisecond response times.

**Scalable**
Write and memory scaling with sharding. Non-disruptive scaling. Read scaling with replicas.

**OSS compatible**
Fully compatible with open source.

© 2020, Amazon Web Services, Inc. or its Affiliates.
Community

“Most popular key-value store”

<table>
<thead>
<tr>
<th>Rank</th>
<th>DBMS</th>
<th>Database Model</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Redis</td>
<td>Key-value, Multi-model</td>
<td>153.63</td>
</tr>
<tr>
<td>2.</td>
<td>Amazon DynamoDB</td>
<td>Multi-model</td>
<td>69.12</td>
</tr>
<tr>
<td>3.</td>
<td>Microsoft Azure Cosmos DB</td>
<td>Multi-model</td>
<td>33.54</td>
</tr>
<tr>
<td>4.</td>
<td>Memcached</td>
<td>Key-value</td>
<td>25.89</td>
</tr>
</tbody>
</table>

© 2020, Amazon Web Services, Inc. or its Affiliates.

– DB-Engines.com
https://db-engines.com/en/ranking/key-value+store

“Most loved database”

- Stack Overflow
Scaling Reads

Add replicas to scale read capacity

Clients

Amazon EC2

AWS Lambda

Shard

Primary

Replica

Replica

Asynchronous replication

Reads

Writes
Scaling Writes

Add shards to scale write capacity
High Availability

<table>
<thead>
<tr>
<th>Availability Zone A</th>
<th>Availability Zone B</th>
<th>Availability Zone C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary 0-5454</td>
<td>Replica 5455-10909</td>
<td>Replica 0-5454</td>
</tr>
<tr>
<td>Replica 10910-16363</td>
<td>Primary 5455-10909</td>
<td>Replica 5455-10909</td>
</tr>
<tr>
<td>Replica 10910-16363</td>
<td>Replica 10910-16363</td>
<td>Primary 10910-16363</td>
</tr>
</tbody>
</table>
Global Datastore

Fully managed, fast, reliable and secure cross-region replication

- Disaster Recovery
- Low latency reads
- Replication typically < 1s
Best Practices
## Best Practices - Evictions

Keys can be automatically removed when memory is full

<table>
<thead>
<tr>
<th>Policy Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allkeys-lru</td>
<td>Evicts the least recently used (LRU) regardless of TTL set</td>
</tr>
<tr>
<td>volatile-lru*</td>
<td>Evicts the least recently used (LRU) from those that have a TTL set</td>
</tr>
<tr>
<td>allkeys-lfu</td>
<td>Evict any key using approximated least frequently used (LFU)</td>
</tr>
<tr>
<td>volatile-lfu*</td>
<td>Evict using approximated LFU among the keys with a TTL set</td>
</tr>
<tr>
<td>volatile-ttl*</td>
<td>Evicts the keys with shortest TTL set</td>
</tr>
<tr>
<td>volatile-random*</td>
<td>Randomly evicts keys with a TTL set</td>
</tr>
<tr>
<td>allkeys-random</td>
<td>Randomly evicts keys regardless of TTL set</td>
</tr>
<tr>
<td>no-eviction</td>
<td>Doesn’t evict keys at all. This blocks future writes until memory frees up.</td>
</tr>
</tbody>
</table>

*Volatile policies only evicts keys with TTLs*
Best Practices - Time to Live (TTL)

Assign freshness factor / expiry
Explicitly assign to individual keys (they are not global)
Assign during creation / modification of key
Some keys should not have TTLs
Set TTL as relative (EXPIRE/PEXPIRE) or fixed time (EXPIREAT)
Add a random jitter (+/-)
Best Practices - Key Names

Decide on a standard separator convention like `appX:datatype:object:identifier`

> `get app1:string:customer:1` 
"\x80\x03(K+X\a\x00\x00\x00Amyq\x00X\x04\x00\x00\x00Zq\x01X\x05\x00\x00\x00Aun..."

Could be a hash of SQL query, API call, etc.

> `get app1:sqlquery:6c7d12836902ef9b2ac83dd54d714e8e` 
"\x80\x03(K+X\a\x00\x00\x00Amyq\x00X\x04\x00\x00\x00Zq\x01X\x05\x00\x00\x00Aun..."

Key names take up memory too!

Indicate its purpose and uniqueness, but no longer
Best Practices - Clients

Connections
- Connection pooling reduces client and server CPU overhead
- Use exponential back-off for reconnects

Reads or Writes
- Connect to read replicas ('readonly' parameter) for reads
- Use the reader endpoint for replicas (cluster-mode-disabled only)

Performance
- Pipelines greatly increase throughput for bulk inserts
- Use 'CLIENT REPLY {OFF|ON|SKIP}' to control server responses
- Ensure client library supports Redis cluster mode
Q&A
Thank you!