Overview

- History
- Users
- Logical Data Model
  - Atomic Operators
  - Transactions
- Programming Language APIs
- System Architecture
  - Physical Data Structures
  - Data Persistence
  - Replication, Consistency, Availability
- Benchmarks
History

- Early 2009 - Salvatore Sanfilippo, an Italian developer, started the Redis project
- He was working on a real-time web analytics solution and found that MySQL could not provide the necessary performance.
- June 2009 - Redis was deployed in production for the LLOOGG real-time web analytics website
- March 2010 - VMWare hired Sanfilippo to work full-time on Redis (remains BSD licensed)
- Subsequently, VMWare hired Pieter Noordhuis, a major Redis contributor, to assist on the project.
Other Users

- Engine Yard
- GitHub
- Guardian.co.uk
- Digg
- Disqus
- Mercado Libre
- Blizzard Entertainment
- Studi VZ
- Stack Overflow
- Bump Technologies
Without Redis

ORDER BY RAND() accounted for 90% of DB load

MySQL

With Redis

SRANDMEMBER() reduces cost of generating a random document ID

MySQL

Redis
Logical Data Model

Data Model
- **Key**
  - Printable ASCII

- **Value**
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets
Logical Data Model

Data Model

- **Key**
  - Printable ASCII

- **Value**
  - Primitives
    - **Strings**
    - Hashes
    - Lists
    - Sets
    - Sorted Sets
Logical Data Model

**Data Model**
- **Key**
  - Printable ASCII

- **Value**
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets
Logical Data Model

Data Model

- **Key**
  - Printable ASCII

- **Value**
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets
Logical Data Model

**Data Model**

- **Key**
  - Printable ASCII

- **Value**
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - **Sets**
    - Sorted Sets
Logical Data Model

**Data Model**
- **Key**
  - Printable ASCII
- **Value**
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets

![Diagram of Redis Key and Sorted Set]

Redis Key

Value: Redis Sorted Set

- Score 100
  - Value 2
- Score 300
  - Value 1
- Score 50
  - Value 3
- Score 300
  - Value 4
Shopping Cart Example

Relational Model

carts

<table>
<thead>
<tr>
<th>CartID</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>james</td>
</tr>
<tr>
<td>2</td>
<td>chris</td>
</tr>
<tr>
<td>3</td>
<td>james</td>
</tr>
</tbody>
</table>

cart_lines

<table>
<thead>
<tr>
<th>Cart</th>
<th>Product</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>372</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>160</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>201</td>
<td>7</td>
</tr>
</tbody>
</table>

UPDATE cart_lines
SET    Qty = Qty + 2
WHERE  Cart=1 AND Product=28
Shopping Cart Example

Relational Model

carts

<table>
<thead>
<tr>
<th>CartID</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>james</td>
</tr>
<tr>
<td>2</td>
<td>chris</td>
</tr>
<tr>
<td>3</td>
<td>james</td>
</tr>
</tbody>
</table>

cart_lines

<table>
<thead>
<tr>
<th>Cart</th>
<th>Product</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>372</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>160</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>201</td>
<td>7</td>
</tr>
</tbody>
</table>

UPDATE cart_lines
SET    Qty = Qty + 2
WHERE  Cart=1 AND Product=28

Redis Model

set carts_james ( 1 3 )
set carts_chris ( 2 )
hash cart_1 {
    user : "james"
    product_28 : 1
    product_372 : 2
}
hash cart_2 {
    user : "chris"
    product_15 : 1
    product_160 : 5
    product_201 : 7
}

HINCRBY cart_1 product_28 2
Atomic Operators - KV Store

**Strings - O(1)**
- GET key
- SET key value
- EXISTS key
- DEL key

**Hashes - O(1)**
- HGET key field
- HSET key field value
- HEXISTS key field

**SETNX key value**
  - Set if not exists

**GETSET key value**
  - Get old value, set new

**Hashes - O(N)**
- HMGET key f1 [f2 ...]
  - Get fields of a hash

**KKEYS key | HVALS key**
  - All keys/values of hash
Atomic Operators - Sets

Sets - \( O(1) \)
- SADD, SREM, SCARD
- SPOP key
  - Return random member of the set

Sets - \( O(N) \)
- SDIFF key1 key2
- SUNION key1 key2

Sets - \( O(C) \)
- SINTER key1 key2
Atomic Operators - Sets

**Sets - O(1)**
- SADD, SREM, SCARD
- SPOP key
  - Return random member of the set

**Sets - O(N)**
- SDIFF key1 key2 ...
- SUNION key1 key2 ...

**Sets - O(C*M)**
- SINTER key1 key2 ...

**Sorted Sets - O(1)**
- ZCARD key

**Sorted Sets - O(log(N))**
- ZADD key score member
- ZREM key member
- ZRANK key member

**Sorted Sets - O(log(N)+M))**
- ZRANGE key start stop
- ZRANGEBYSCORE key min max
Transactions

- All commands are serialized and executed sequentially
- Either all commands or no commands are processed
- Keys must be overtly specified in Redis transactions
- Redis commands for transactions:
  - WATCH
  - MULTI
  - DISCARD
  - EXEC
  - UNWATCH
# Programming Language APIs

<table>
<thead>
<tr>
<th>ActionScript</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Lua</td>
</tr>
<tr>
<td>C#</td>
<td>Objective-C</td>
</tr>
<tr>
<td>C++</td>
<td>Perl</td>
</tr>
<tr>
<td>Clojure</td>
<td>PHP</td>
</tr>
<tr>
<td>Common Lisp</td>
<td>Python</td>
</tr>
<tr>
<td>Erlang</td>
<td>Ruby</td>
</tr>
<tr>
<td>Go</td>
<td>Scala</td>
</tr>
<tr>
<td>Haskell</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>haXe</td>
<td>Tcl</td>
</tr>
<tr>
<td>Io</td>
<td></td>
</tr>
</tbody>
</table>
API Examples

```php
/**
 * PHP Example
 */
require_once 'Predis.php';

$redis = new Predis\Client(
    ['db' => 0,
     'host' => 'localhost',
     'port' => 6379,
    ]);

/**
 * SET Benchmarks
 */
$redis->del('some: key');
$start = microtime(TRUE);
for ($i = 0; $i < OPERATIONS; $i++) {
    $redis->set('some: key', $i);
}
$end = microtime(TRUE);
=output('SET', $start, $end, PRECISION);
```

```python
## Python Example

```python
import redis

r = redis.Redis(host='localhost', port=6379, db=0)

## SET Benchmarks

r.delete('some: key')
start = time.clock()
for i in xrange(OPERATIONS):
    r.set('some: key', i)
end = time.clock()
output('SET', start, end, PRECISION)
```
System Architecture

- **Redis Instance**
  - Main memory database
  - Single-threaded event loop (no locks!)

- **Virtual Memory**
  - Evicts "values" rather than "pages"
  - Smarter than OS with complex data structures
  - *May use threads*

- Sharding: application's job!
Data Persistence

- **Periodic Dump ("Background Save")**
  - fork() with Copy-on-Write, write entire DB to disk
  - When?
    - After every X seconds and Y changes, or,
    - BGSAVE command

- **Append Only File**
  - On every write, append change to log file
  - Flexible fsync() schedule:
    - Always, Every second, or, Never
  - Must compact with BGREWRITEAOF
Data Structure Internals

- **Key-Value Store ("hash table")**
  - Incremental, auto-resize on powers of two
  - Collisions handled by chaining

- **Hash Collection**
  - ≤ 512 entries
    - "zipmap" -- $O(N)$ lookups, $O(\text{mem}_\text{size})$ add/delete

<table>
<thead>
<tr>
<th># entries</th>
<th>2</th>
<th>3</th>
<th>foo</th>
<th>3</th>
<th>bar</th>
<th>2</th>
<th>hi</th>
<th>5</th>
<th>world</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>key</td>
<td>value</td>
<td>key</td>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- > 512 entries
  - Hash Table
Data Structure Internals

- **Set Collection**
  - \(< 512\) integers: "intset" -- \(O(N)\) lookups
  - everything else: Hash Table

- **Sorted Set Collection -- \(O(\log N)\) insets/deletes**
  - Indexable Skip List: Scores+Key \(\Rightarrow\) Values
  - Hash Table: Key \(\Rightarrow\) Score

- **List Collection**
  - \(< 512\) entries: "ziplist"
    - \(O(\text{mem\_size})\) inserts/deletes
  - \(> 512\) entries: Doubly Linked List
    - \(O(1)\) left/right push/pop
Replication Topology (Tree)

Slave Roles:
- Offload save-to-disk
- Offload reads (load balancing up to client)
- Data redundancy

- Master is largely "unaware" of slaves
  - No quorums (only master need accept the write)
- Selection of master left to client!
  - All nodes accept "writes"
  - All nodes are *master of their own slaves*
  - Writes propagated downstream ONLY (asynchronously)
Redis & CAP Theorem

- **C & A**
  - Writes: single master
  - Reads: any node
    - Eventually consistent, no read-your-own-writes

- **C & P**
  - On failure: inhibit writes
  - Consequence: decreased availability

- **A & P**
  - On failure: elect new master
  - Consequence: inconsistent data, no easy reconciliation

- "Redis Cluster" is in development but not currently available
Benchmarks - Client Libraries

![Bar Chart](image.png)

- SET
- GET
- INCR
- LPUH
- LPOP

Legend:
- C
- PHP
- Python
Benchmarks - Hardware

![Bar Chart]

- **Core Duo T5500 (1.66 GHz)**
- **64 bit Xeon L5420 (2.5 GHz)**
- **Core 2 Duo (2.53 GHz)**
Questions?
Additional Slides
Motivation

- Imagine
  - lots of data
  - stored in main memory as:
    - hash maps, lists, sets, and sorted sets
    - $O(1)$ -- GET, PUT, PUSH, and POP operations
    - $O(\log(N))$ -- sorted operations
- Imagine 100k requests per second per machine

- Imagine Redis!

- Our Goal:
  - Give you an overview of Redis externals & internals
Replication Process

● Chronology
  o SLAVE: Connects to master, sends "SYNC" command
  o MASTER: Begins "background save" of DB to disk
  o MASTER: Begins recording all new writes
  o MASTER: Transfers DB snapshot to slave
  o SLAVE: Saves snapshot to disk
  o SLAVE: Loads snapshot into RAM
  o MASTER: Sends recorded write commands
  o SLAVE: Applies recorded write commands
  o SLAVE: Goes live, begins accepting requests
Used in "crowd-sourcing" application for reviewing documents related to MP's (members of Parliament) expense reports

Major challenge was providing a random document to a review

Initial implementation used SQL "ORDER BY RAND()" command to choose an new document for a reviewer

RAND() statement account for 90% of DB load

Redis implementation leveraged SRANDMEMBER() command to generate a random element (document id) from a set

Redis was also used to manage account registration process for document reviewers
Uses Redis as a three level site-caching solution
- "Local-Cache" for 1 server/site pair
  - user sessions/pending view count updates
- "Site-Cache" for all servers in a site
  - Hot question id lists/user acceptance rates
- "Global-Cache" is shared among all sites and servers
  - Inbox/usage quotas
- Cache typically includes approximately 120,000 keys
  - Most expire within minutes
  - Number is expected to grow as confidence is gained
- Peak load is a few hundred reads/writes per second
- CPU Usage on dedicated Redis machine is reported to be 0%
- Memory usage on dedicated Redis machine is < 1/2 GB
Schema

*Informal, free-form "Namespaces"

**Example Keys:**
- `user:1000:pwd`
  - User 1000's password
- `user.next.id`
  - The next user ID to be assigned
# Redis and the CAP Theorem

Achieving the ideals of the CAP Theorem depends greatly on how an instance of Redis is configured. A clustered version of Redis is in development but not currently available.

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Availability</th>
<th>Partition Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single node instance of Redis would provide the highest levels of consistency. Writes propagate down the replication tree. Consistent writes must be written directly to the master node. Consistent reads depend on the speed of the synchronization process.</td>
<td>Adding more nodes increases availability for reads and writes. However, adding more nodes greatly increases the complexity of maintaining consistent data due to the &quot;down-hill&quot; propagation of write operations.</td>
<td>Tolerating network partitions is a major weakness of a Redis system. Logic for detecting failures and promoting slave nodes to master's and reconfiguring the replication tree must be handled by the application developer.</td>
</tr>
</tbody>
</table>