REmote **DI**ctionary **S**erver

Chris Keith
James Tavares

Overview

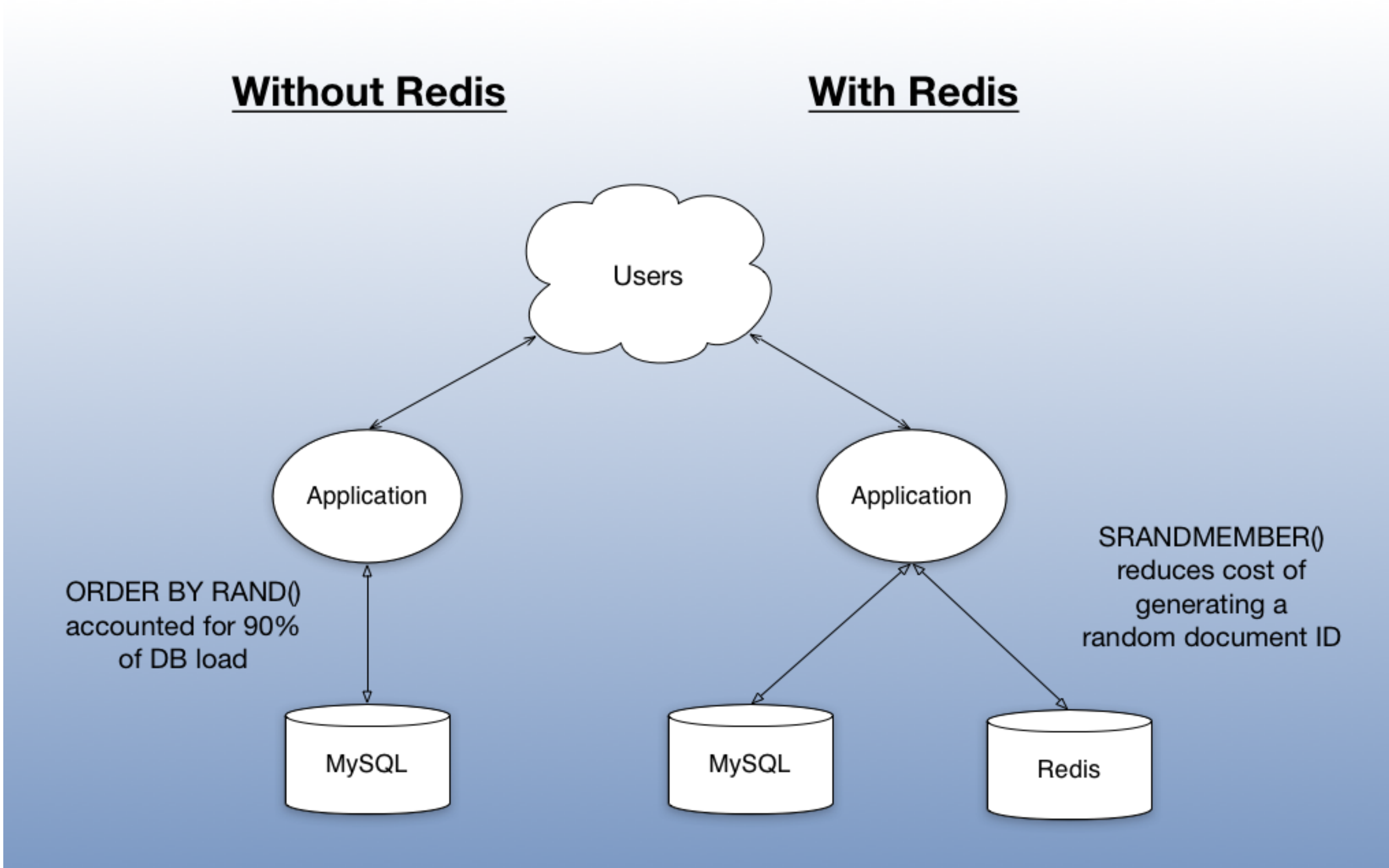
- History
- Users
- Logical Data Model
  - Atomic Operators
  - Transactions
- Programming Language APIs
- System Architecture
  - Physical Data Structures
  - Data Persistence
  - Replication, Consistency, Availability
- Benchmarks

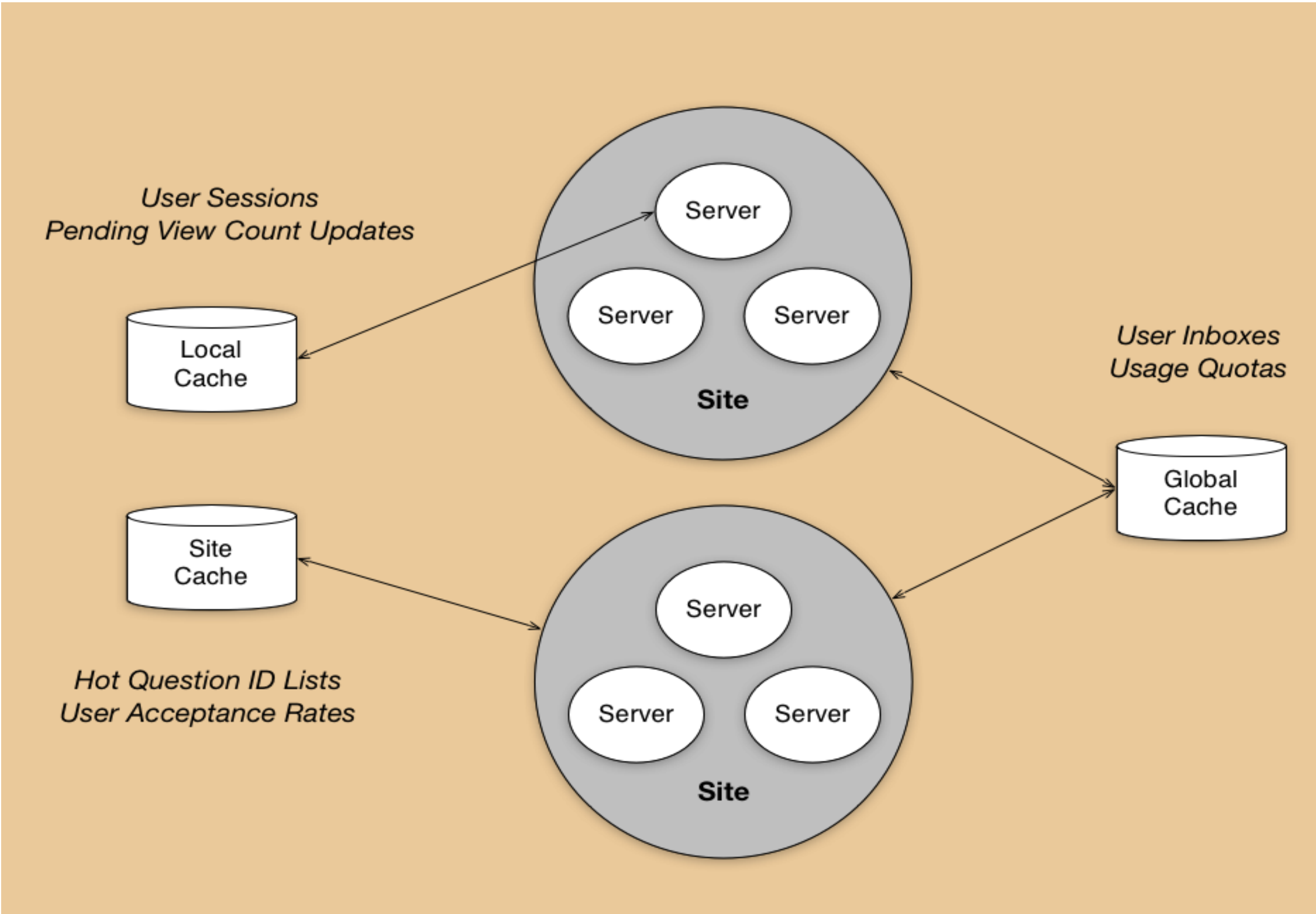
History

- 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

guardian.co.uk




User Sessions
Pending View Count Updates
Local Cache
Server
Server
Server
Site
User Inboxes
Usage Quotas
Global Cache
Site Cache
Server
Server
Server
Site
Hot Question ID Lists
User Acceptance Rates

Logical Data Model

Data Model

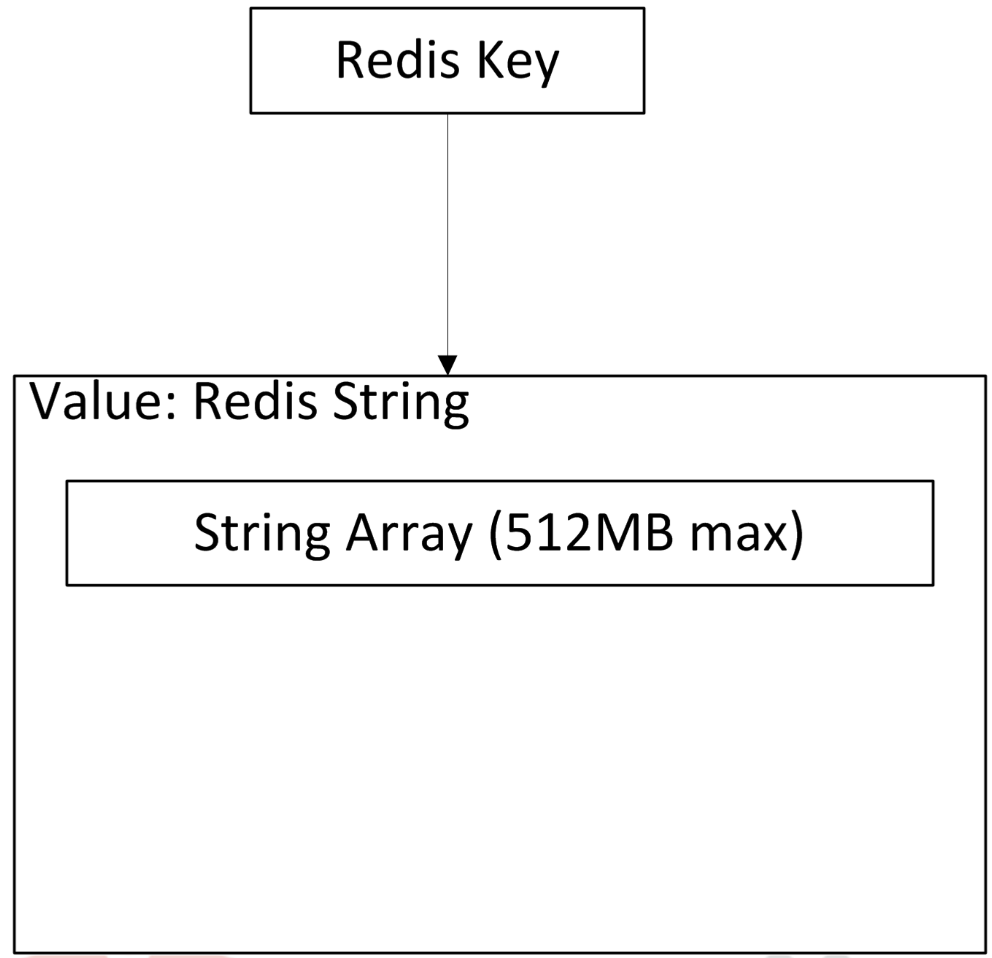
- *Key*
  - Printable ASCII

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

Logical Data Model

Data Model

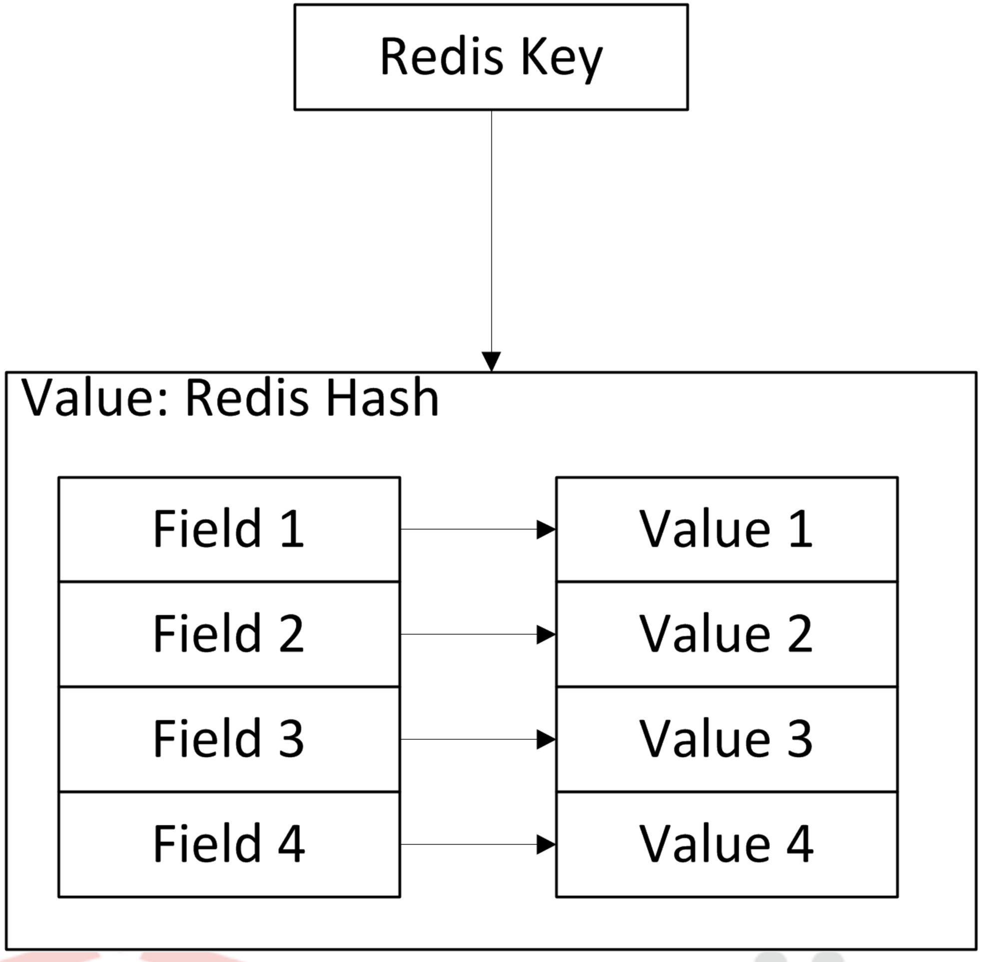
- *Key*
  - Printable ASCII
- *Value*
  - Primitives
    - **Strings**
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets



Logical Data Model

Data Model

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

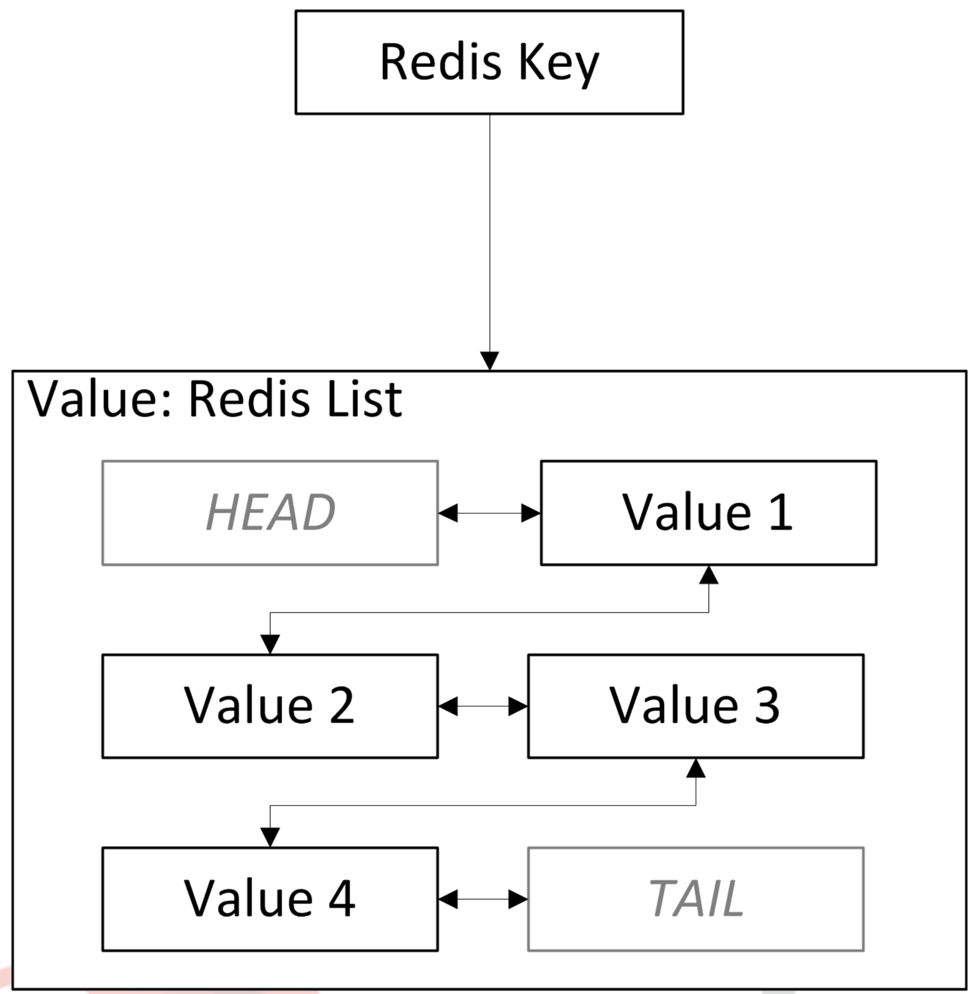


Logical Data Model

Data Model

- *Key*
  - Printable ASCII

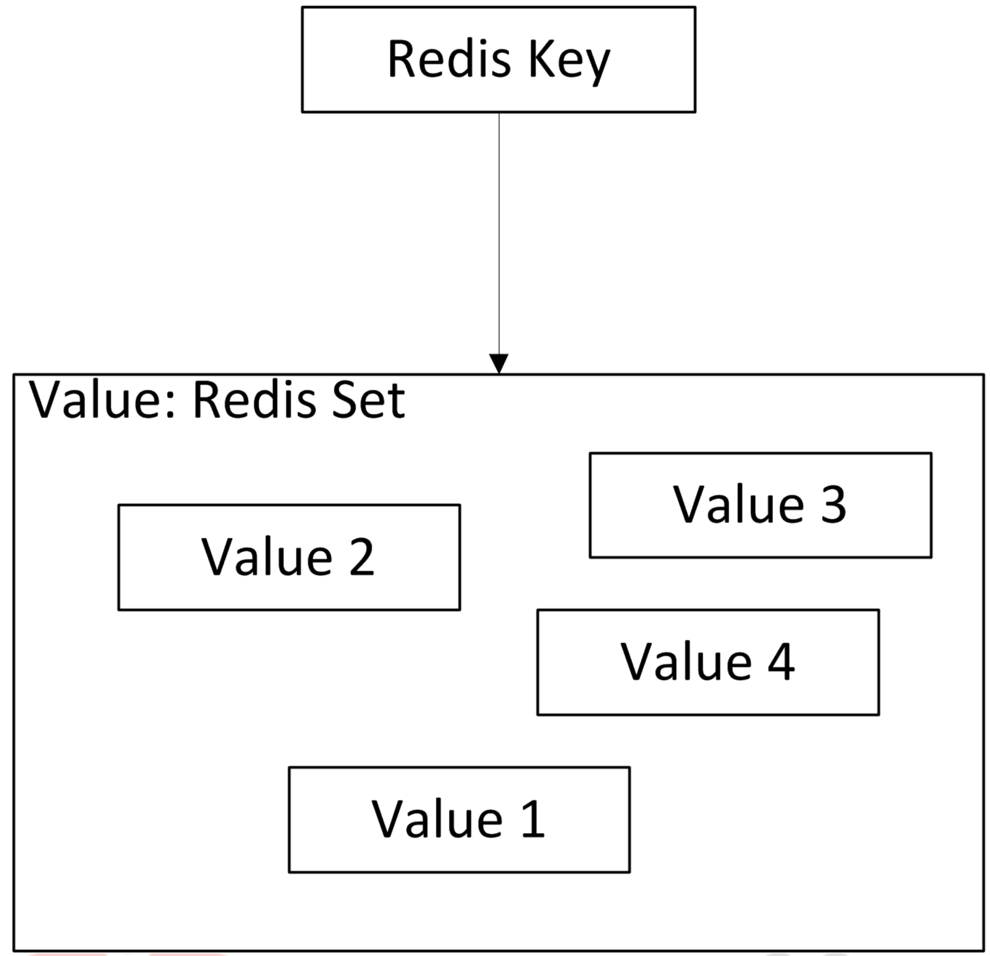
- *Value*
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - **Lists**
    - Sets
    - Sorted Sets



Logical Data Model

Data Model

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

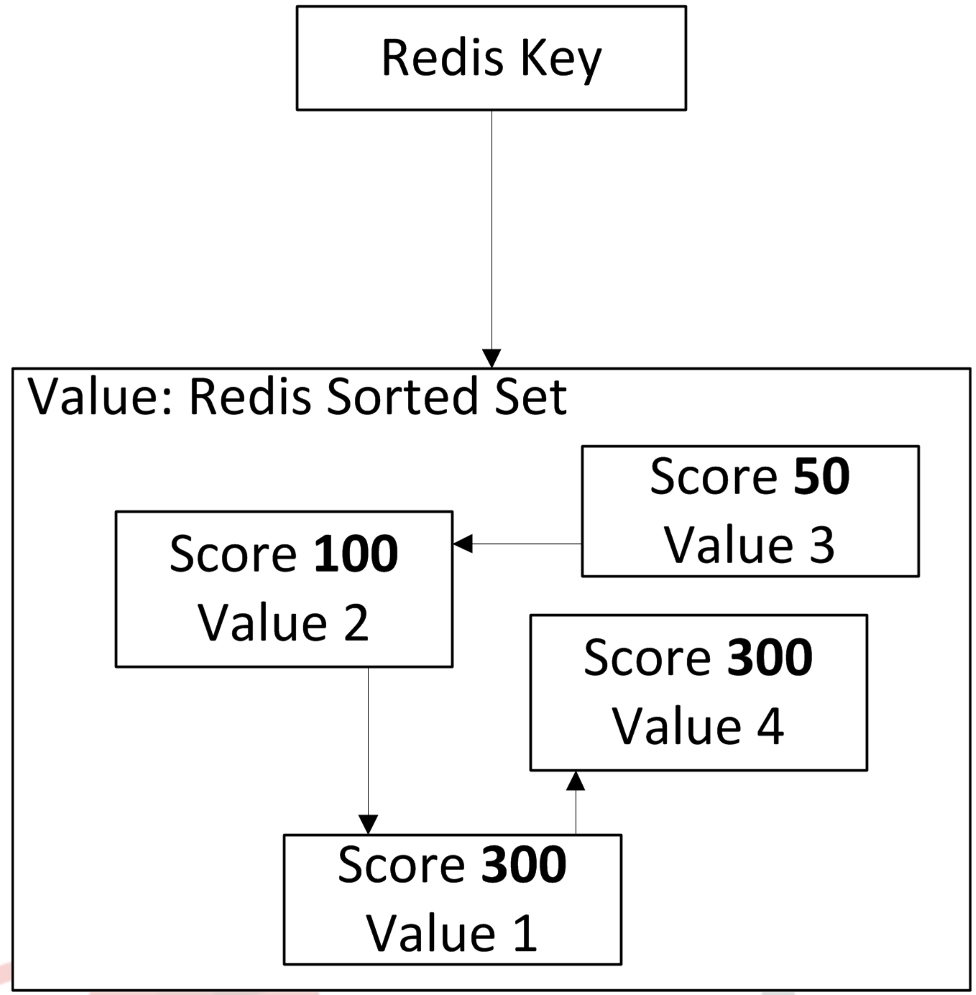


Logical Data Model

Data Model

- *Key*
  - Printable ASCII

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



Shopping Cart Example

Relational Model

carts

CartID	**User**
1	james
2	chris
3	james

cart_lines

Cart	**Product**	**Qty**
1	28	1
1	372	2
2	15	1
2	160	5
2	201	7

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

Shopping Cart Example

Relational Model

carts

CartID	User
1	james
2	chris
3	james

cart_lines

Cart	Product	Qty
1	28	1
1	372	2
2	15	1
2	160	5
2	201	7

```
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

SETNX key value

  - Set if not exists

- GETSET key value
  - Get old value, set new

Hashes *- O(1)*

- HGET key field
- HSET key field value
- HEXISTS key field

HDEL key field

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

```
WATCH key:a key:b
val = GET key:a
val = val + 1
MULTI
SET key:a val
SET key:b 'foo'
SET key:c 'bar'
EXEC
```

Programming Language APIs

ActionScript
C
C#
C++
Clojure
Common Lisp
Erlang
Go
Haskell
haXe
Io
Java
Lua
Objective-C
Perl
PHP
Python
Ruby
Scala
Smalltalk
Tcl

API Examples

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

$redis = new Predis\Client(
    array(
        '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
##
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*

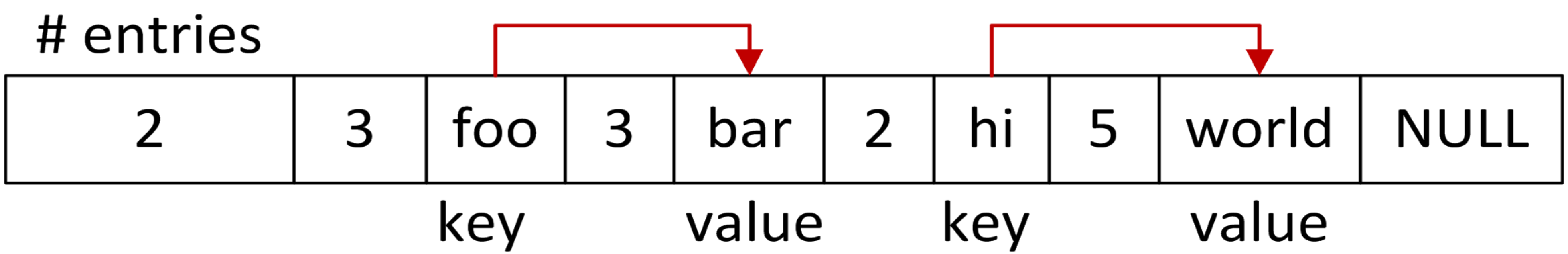
- Sharding: application's job!

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(mem_size) add/delete



  - > 512 entries
    - Hash Table

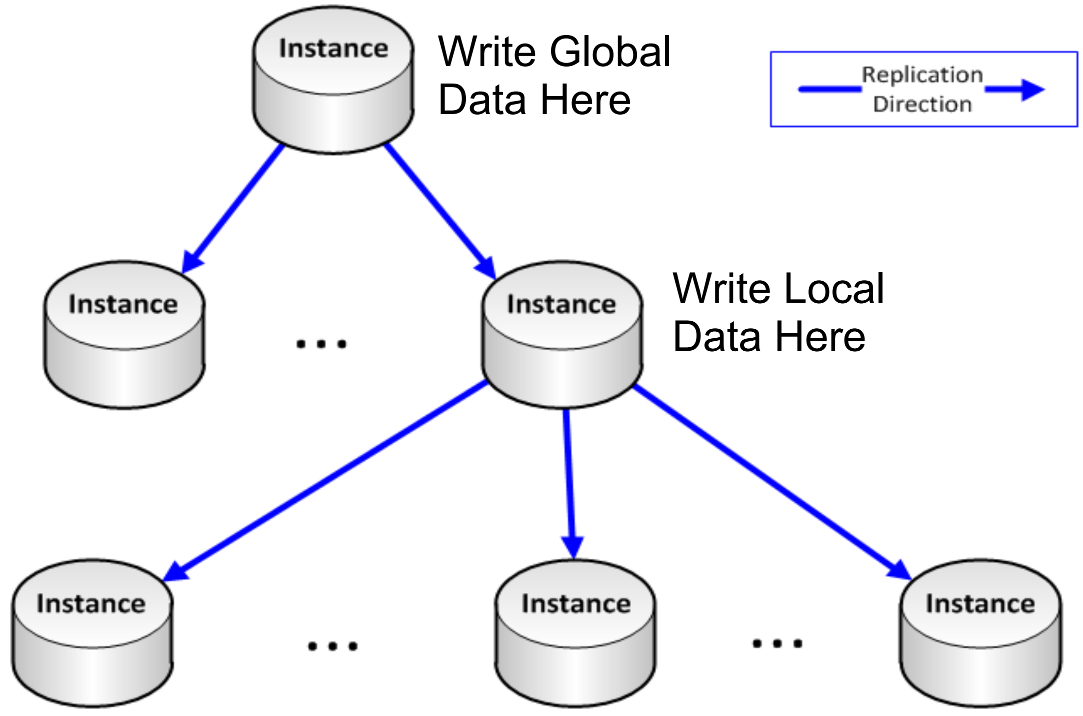
Data Structure Internals

- **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 => Values
  - Hash Table: Key => Score

- **List Collection**
  - ≤ 512 entries: "ziplist"
    - O(mem_size) inserts/deletes
  - > 512 entries: Doubly Linked List
    - O(1) left/right push/pop

Replication Topology (Tree)



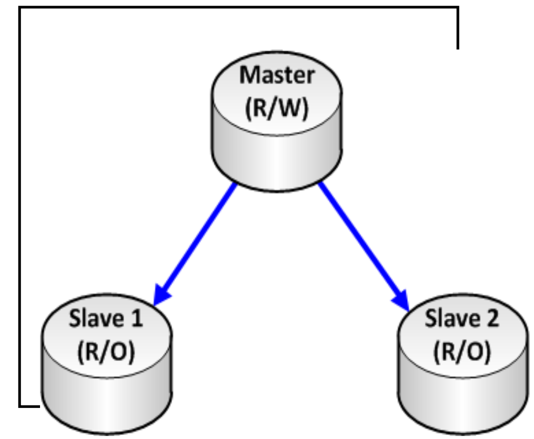
Slave Roles:

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

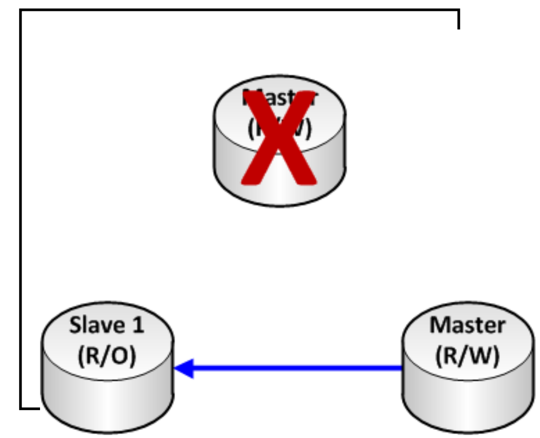
- Master is largely "unaware" of slaves
  - No quorums (only master need accept the write)
- Selection of master left to client!
  - All nodes accept "writes"
  - All nodes are *master of their own slaves*
  - Writes propagated downstream ONLY (asynchronously)

Redis & CAP Theorem

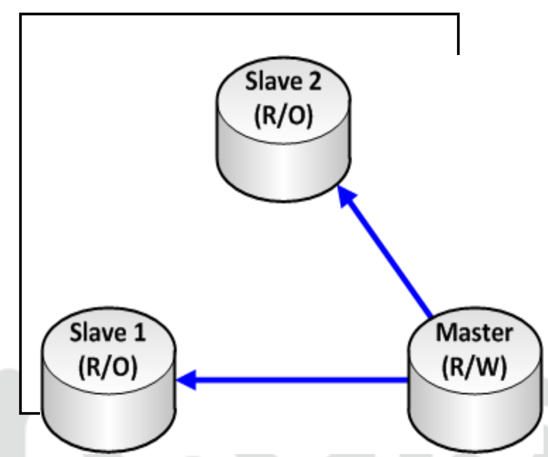
- **C & A**
  - Writes: single master
  - Reads: any node
    - Eventually consistent,
      no read-your-own-writes



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

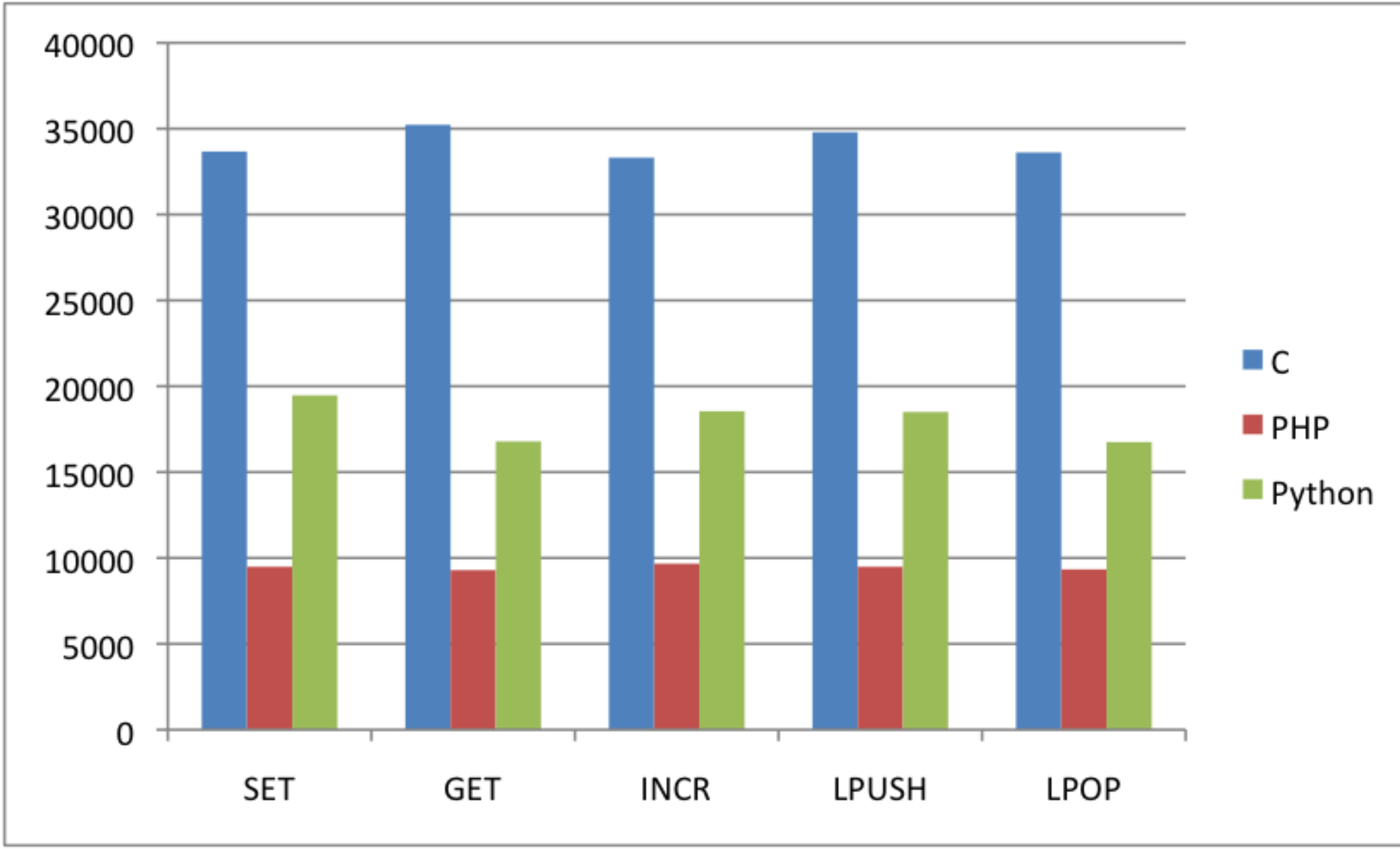


- **A & P**
  - On failure: elect new master
  - Consequence: inconsistent data,
    no easy reconciliation

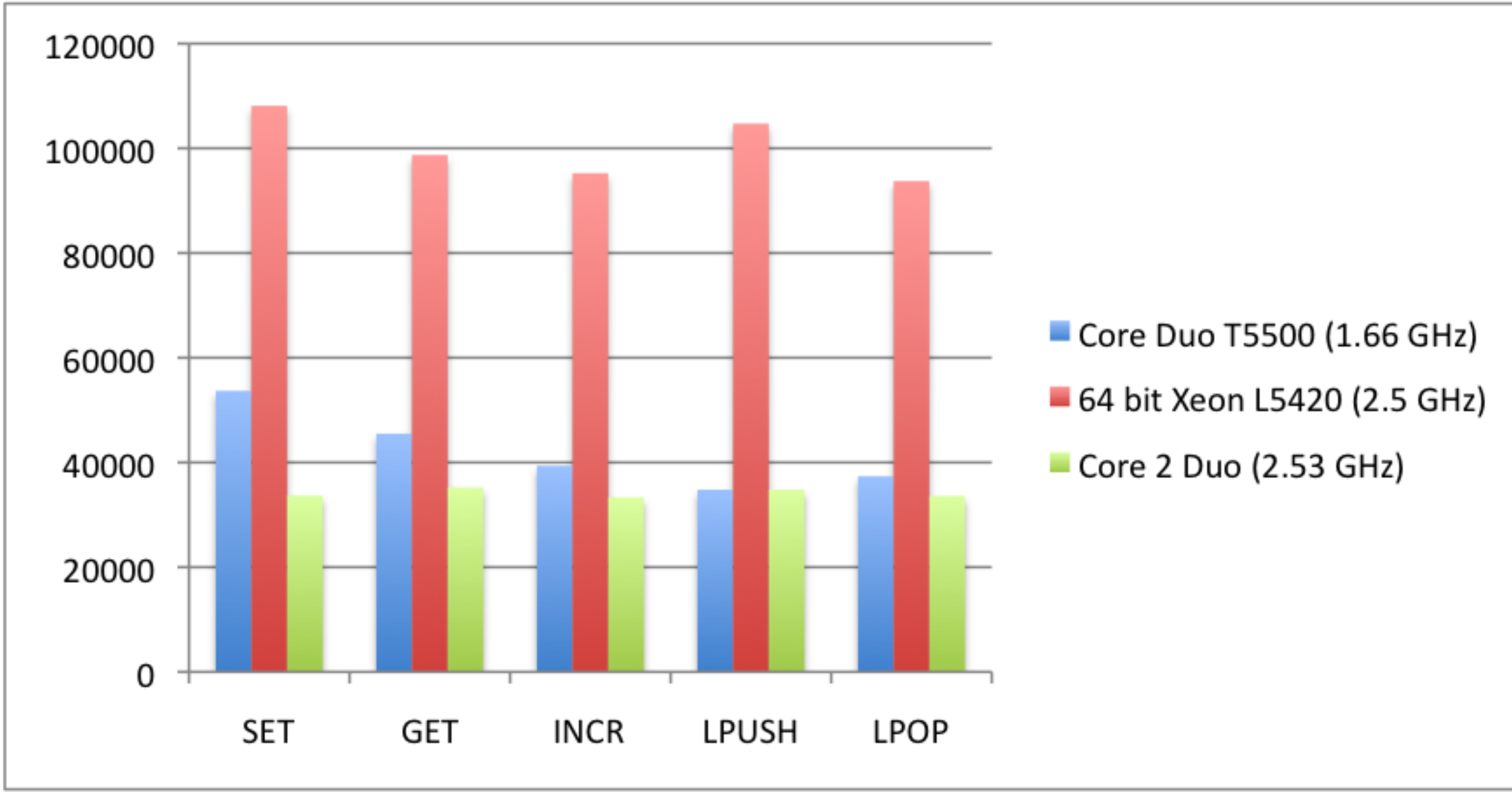


- "Redis Cluster" is in development
  but not currently available

Benchmarks - Client Libraries



Benchmarks - Hardware



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
  - SLAVE: Connects to master, sends "SYNC" command
  - MASTER: Begins "background save" of DB to disk
  - MASTER: Begins recording all new writes
  - MASTER: Transfers DB snapshot to slave
  - SLAVE: Saves snapshot to disk
  - SLAVE: Loads snapshot into RAM
  - MASTER: Sends recorded write commands
  - SLAVE: Applies recorded write commands
  - SLAVE: Goes live, begins accepting requests

guardian.co.uk

- 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

redis

stackoverflow

- 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

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.

Consistency

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.

Availability

Adding more nodes increases availability for reads and writes. However, adding more nodes greatly increases the complexity of maintaining consistent data due to the "down-hill" propagation of write operations.

Partition Tolerance

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.