In-Memory Computing SUMMIT EUROPE 2019

Redis and Memcached

Speaker: Vladimir Zivkovic, Manager, IT
June, 2019

Capital One

Problem Scenario

- Web Site users wanting to access data extremely quickly (< 200ms)
- Data being shared between different layers of the stack
- Cache a web page sessions

- Research and test feasibility of using Redis as a solution for storing and retrieving data quickly
- Load data into Redis to test ETL feasibility and Performance
- Goal - get sub-second response for API calls for retrieving data

Why Redis

- In-memory key-value store, with persistence
- Open source
- Written in C
- It can handle up to 2^32 keys, and was tested in practice to handle at least 250 million of keys per instance.” - *http://redis.io/topics/faq*
- Most popular key-value store - *http://db-engines.com/en/ranking*

History

- **RE**mote **DI**ctionary **S**erver
- Released in 2009
- Built in order to scale a website: http://lloogg.com/
  - The web application of lloogg was an ajax app to show the site traffic in real time. Needed a DB handling fast writes, and fast ”get latest N items” operation.

Redis Data types

- Strings
- Lists
- Sets
- Sorted Sets
- Hashes

- Bitmaps
- Hyperlogs
- Geospatial Indexes

Redis protocol

- redis[“key”] = “value”
- Values can be strings, lists or sets
- Push and pop elements (atomic)
- Fetch arbitrary set and array elements
- Sorting
- Data is written to disk asynchronously

Memory Footprint

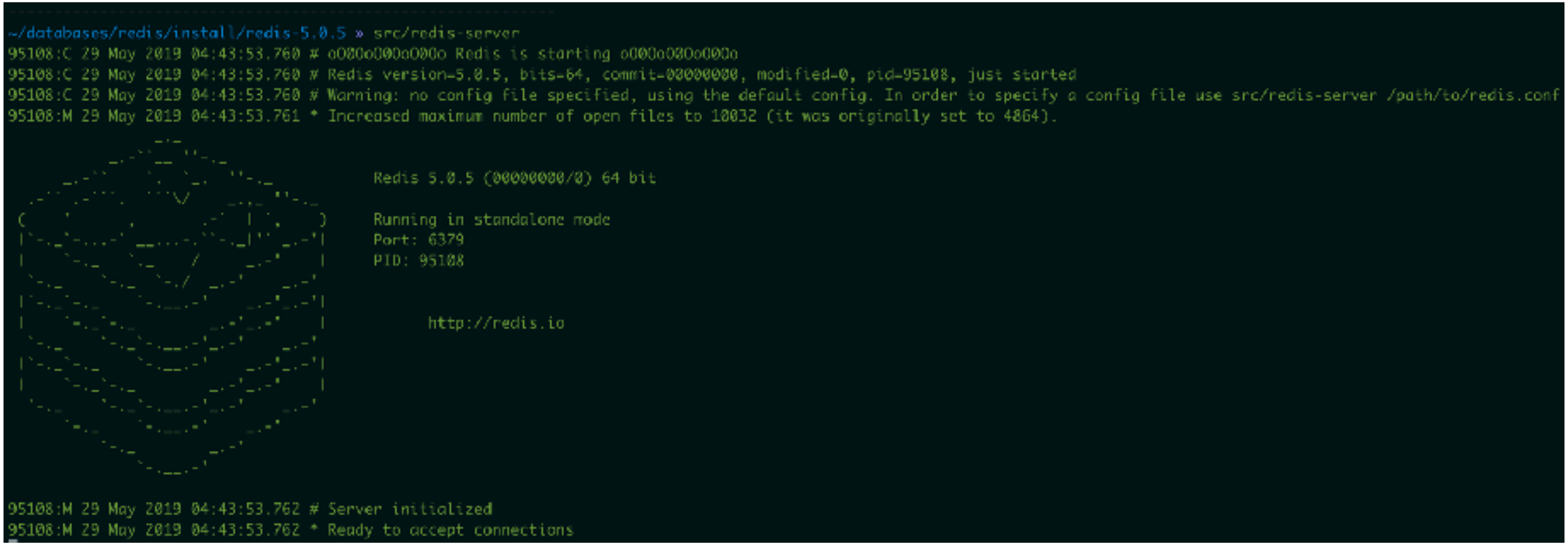
- An empty instance uses ~ 3MB of **memory**.
- For 1 Million small Keys => String Value pairs use ~ 85MB of **memory**.
- 1 Million Keys => Hash value, representing an object with 5 fields, use ~ 160 MB of **memory**.

Installing Redis

```
wget http://download.redis.io/redis-stable.tar.gz
tar xvzf redis-stable.tar.gz
cd redis-stable
make

redis-cli ping
# PONG
```

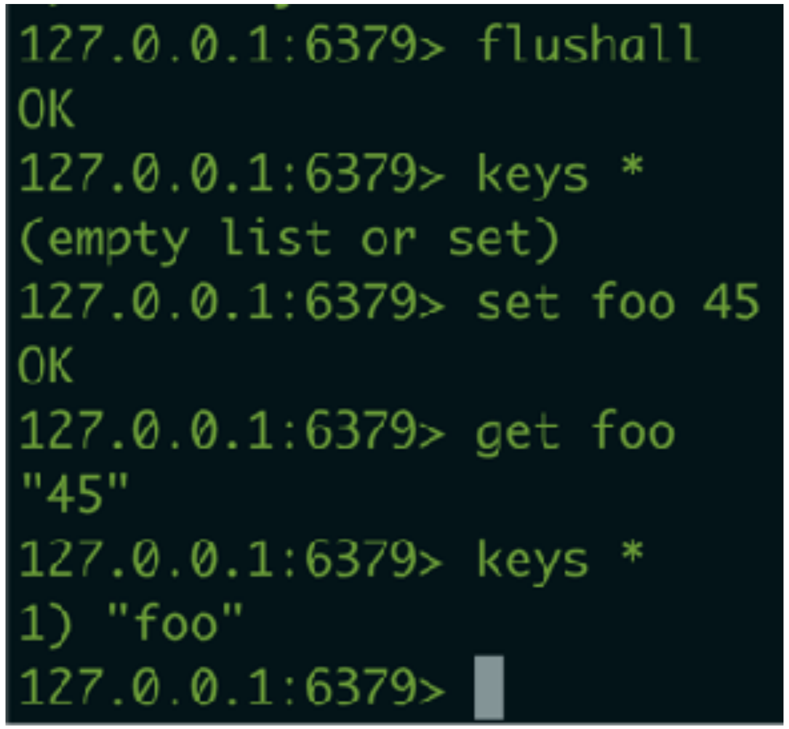
Starting Redis



```
~/databases/redis/install/redis-5.0.5 » src/redis-server
95108:C 29 May 2019 04:43:53.760 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
95108:C 29 May 2019 04:43:53.760 # Redis version=5.0.5, bits=64, commit=00000000, modified=0, pid=95108, just started
95108:C 29 May 2019 04:43:53.760 # Warning: no config file specified, using the default config. In order to specify a config file use src/redis-server /path/to/redis.conf
95108:M 29 May 2019 04:43:53.761 * Increased maximum number of open files to 10032 (it was originally set to 4864).
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 5.0.5 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 95108
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

95108:M 29 May 2019 04:43:53.762 # Server initialized
95108:M 29 May 2019 04:43:53.762 * Ready to accept connections
```

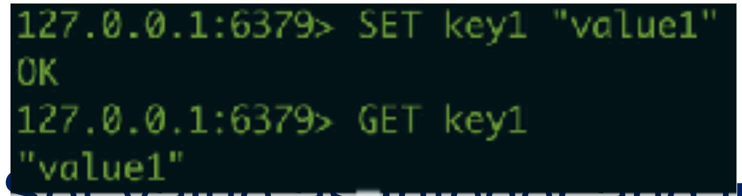
Redis CLI



```
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set foo 45
OK
127.0.0.1:6379> get foo
"45"
127.0.0.1:6379> keys *
1) "foo"
127.0.0.1:6379> 
```

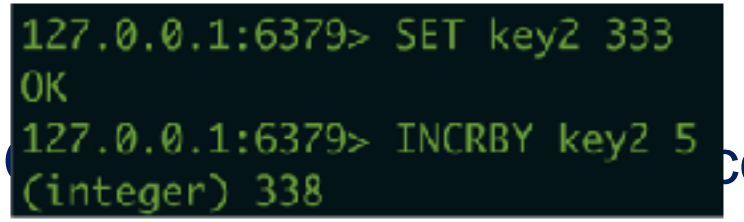
Basic Operations

- Get/Sets – keys are strings, just quote spaces:



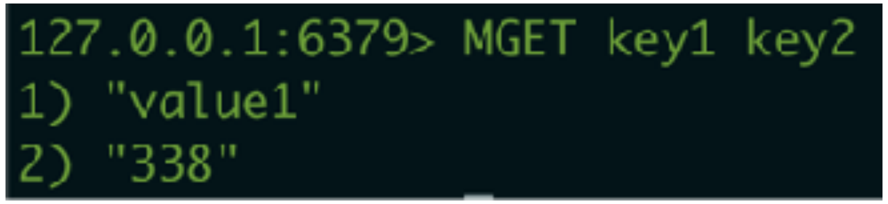
```
127.0.0.1:6379> SET key1 "value1"
OK
127.0.0.1:6379> GET key1
"value1"
```

- Set value as integer and increase it:



```
127.0.0.1:6379> SET key2 333
OK
127.0.0.1:6379> INCRBY key2 5
(integer) 338
```

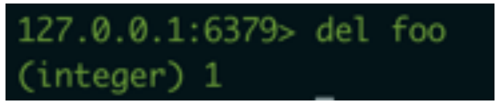
- ce:



```
127.0.0.1:6379> MGET key1 key2
1) "value1"
2) "338"
```

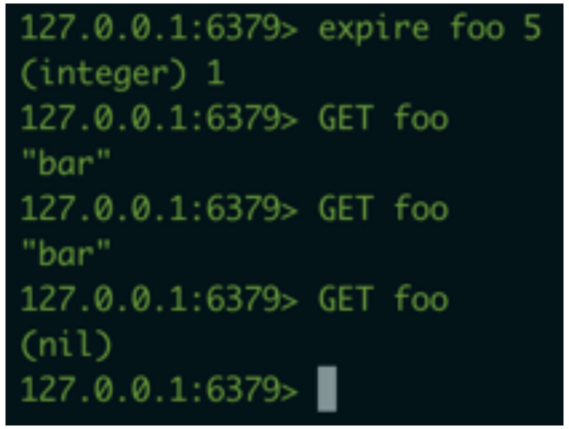
Basic Operations - continued

- Delete key:



```
127.0.0.1:6379> del foo
(integer) 1
```

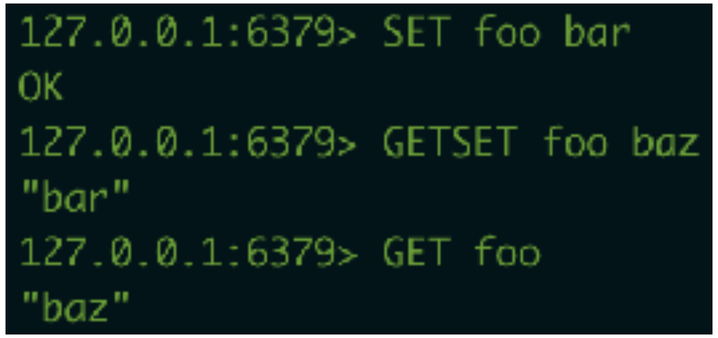
- Keys are lazily expired:



```
127.0.0.1:6379> expire foo 5
(integer) 1
127.0.0.1:6379> GET foo
"bar"
127.0.0.1:6379> GET foo
"bar"
127.0.0.1:6379> GET foo
(nil)
127.0.0.1:6379>
```

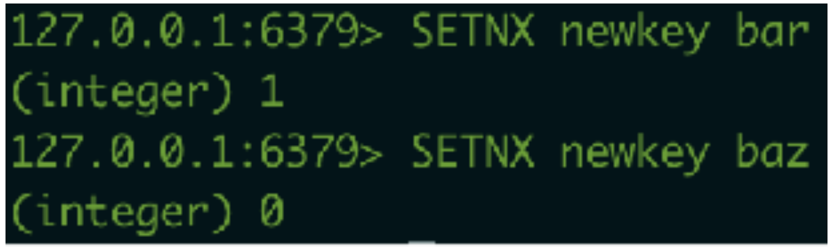
Atomic Operations

- **GETSET** puts a different value inside a key, retrieving the old one:



```
127.0.0.1:6379> SET foo bar
OK
127.0.0.1:6379> GETSET foo baz
"bar"
127.0.0.1:6379> GET foo
"baz"
```

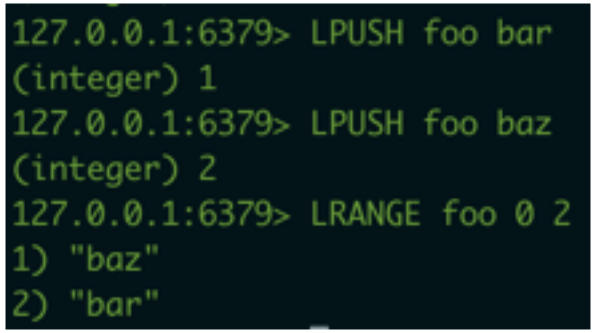
- **SETNX** sets a value only if it does not exist:



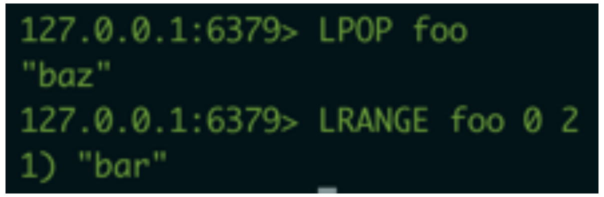
```
127.0.0.1:6379> SETNX newkey bar
(integer) 1
127.0.0.1:6379> SETNX newkey baz
(integer) 0
```

List Operations

- Lists are ordinary linked lists.
- You can push and pop at both sides, extract range, resize.

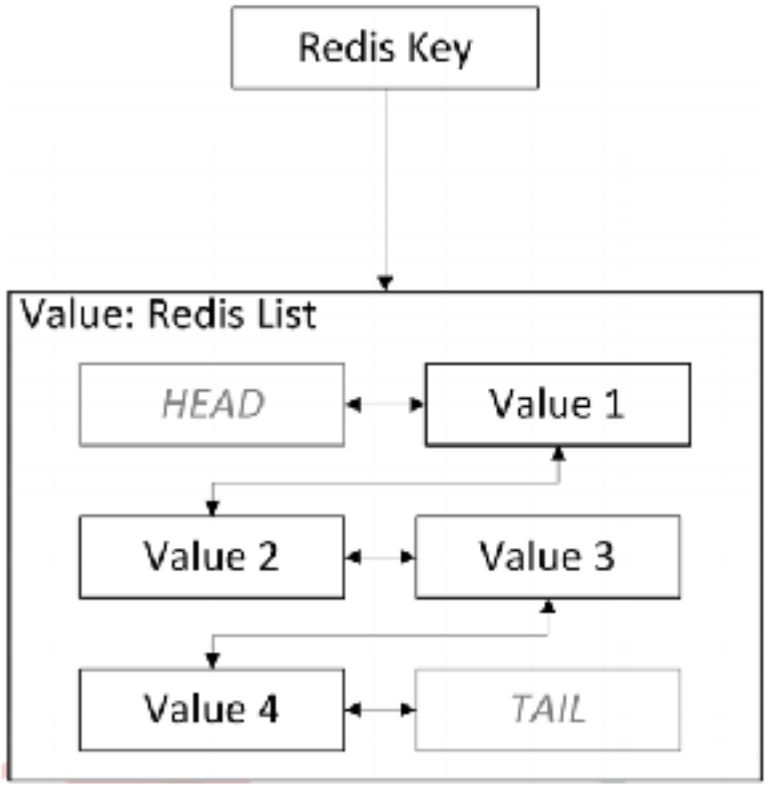


```
127.0.0.1:6379> LPUSH foo bar
(integer) 1
127.0.0.1:6379> LPUSH foo baz
(integer) 2
127.0.0.1:6379> LRANGE foo 0 2
1) "baz"
2) "bar"
```



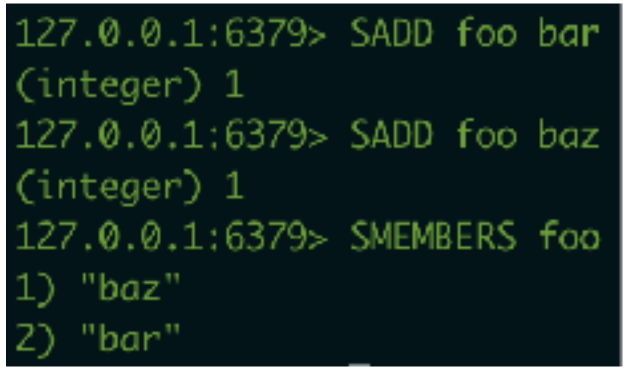
```
127.0.0.1:6379> LPOP foo
"baz"
127.0.0.1:6379> LRANGE foo 0 2
1) "bar"
```

- BLPOP – Blocking POP – wait until a list has elements and pop them.
  - Useful for real-time stuff.

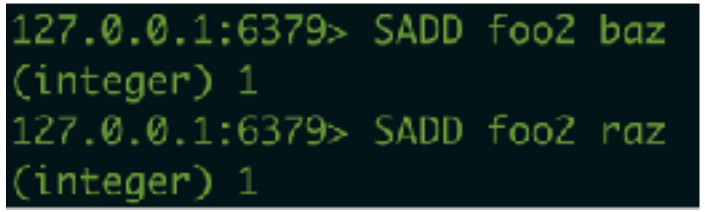


Set Operations

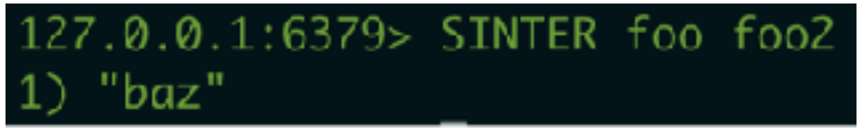
- Sets are sets of unique values with push, pop...
- Sets can be intersected/diffed and union’ed on the server side



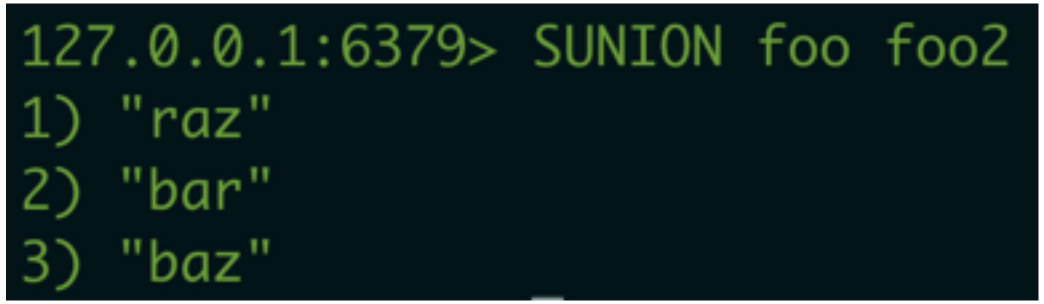
```
127.0.0.1:6379> SADD foo bar
(integer) 1
127.0.0.1:6379> SADD foo baz
(integer) 1
127.0.0.1:6379> SMEMBERS foo
1) "baz"
2) "bar"
```



```
127.0.0.1:6379> SADD foo2 baz
(integer) 1
127.0.0.1:6379> SADD foo2 raz
(integer) 1
```



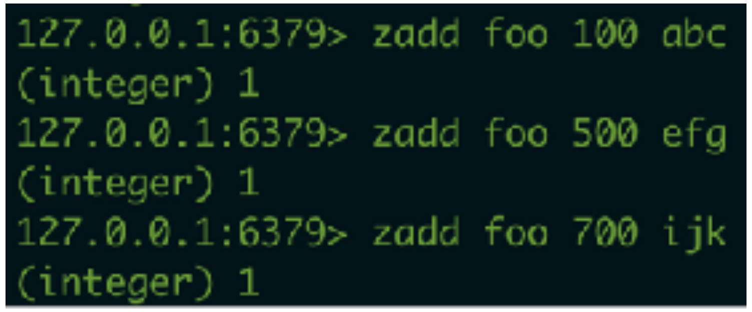
```
127.0.0.1:6379> SINTER foo foo2
1) "baz"
```



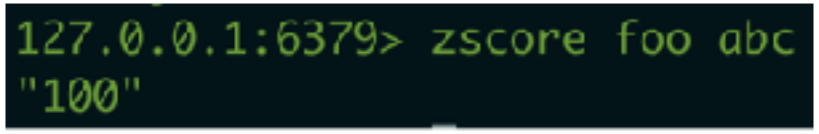
```
127.0.0.1:6379> SUNION foo foo2
1) "raz"
2) "bar"
3) "baz"
```

Sorted Sets

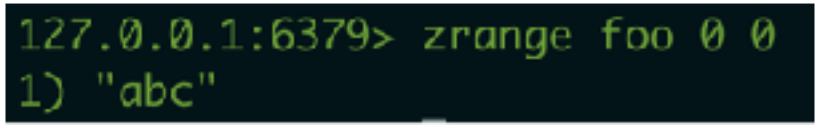
- Same as sets, but with score per element



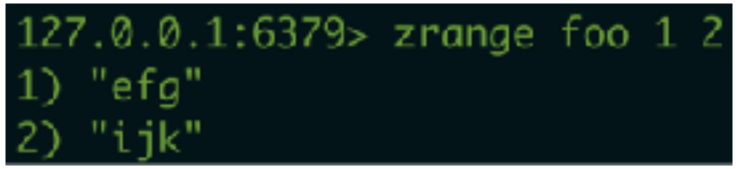
```
127.0.0.1:6379> zadd foo 100 abc
(integer) 1
127.0.0.1:6379> zadd foo 500 efg
(integer) 1
127.0.0.1:6379> zadd foo 700 ijk
(integer) 1
```



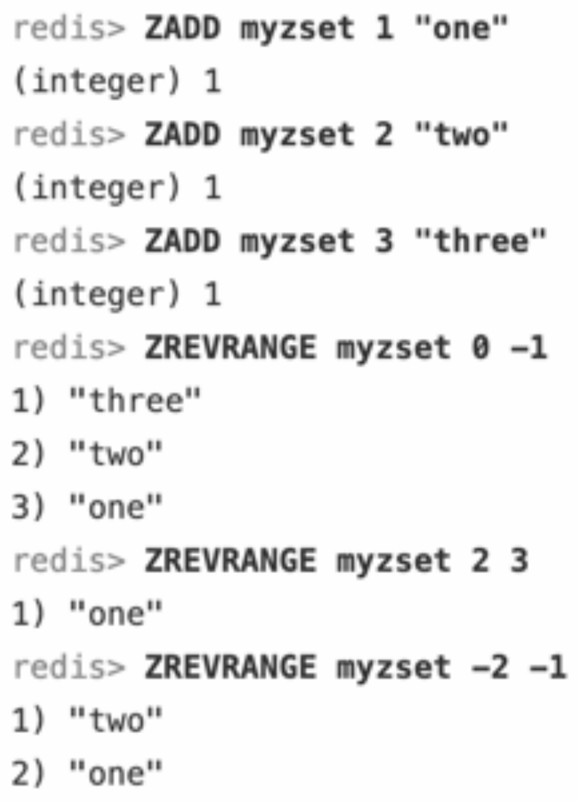
```
127.0.0.1:6379> zscore foo abc
"100"
```



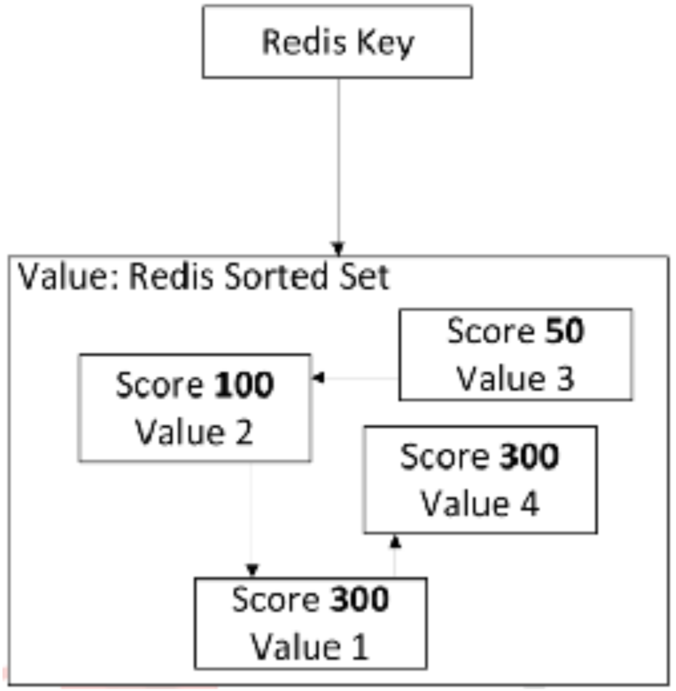
```
127.0.0.1:6379> zrange foo 0 0
1) "abc"
```



```
127.0.0.1:6379> zrange foo 1 2
1) "efg"
2) "ijk"
```



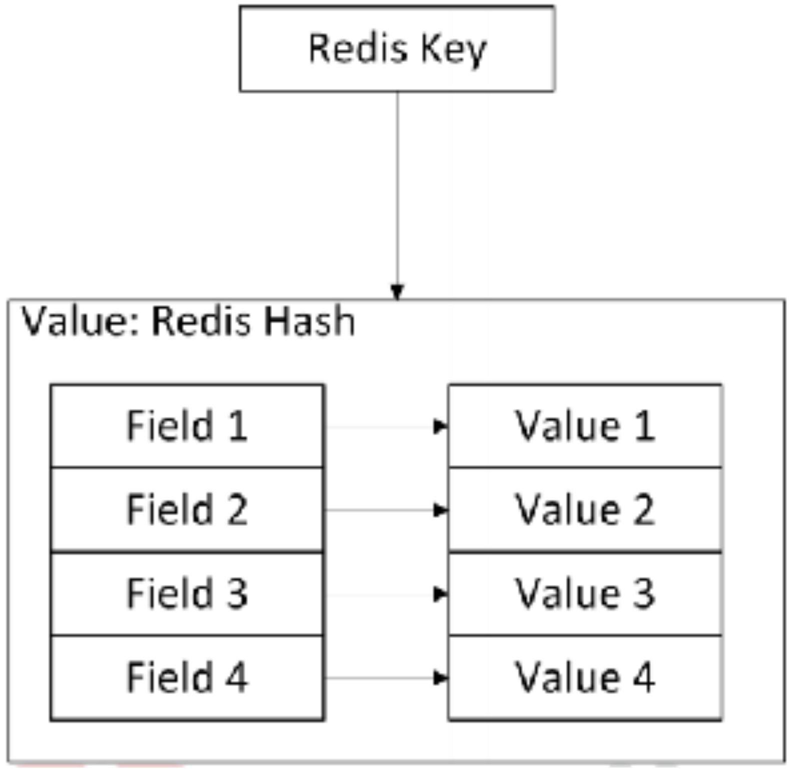
```
redis> ZADD myzset 1 "one"
(integer) 1
redis> ZADD myzset 2 "two"
(integer) 1
redis> ZADD myzset 3 "three"
(integer) 1
redis> ZREVRANGE myzset 0 -1
1) "three"
2) "two"
3) "one"
redis> ZREVRANGE myzset 2 3
1) "one"
redis> ZREVRANGE myzset -2 -1
1) "two"
2) "one"
```



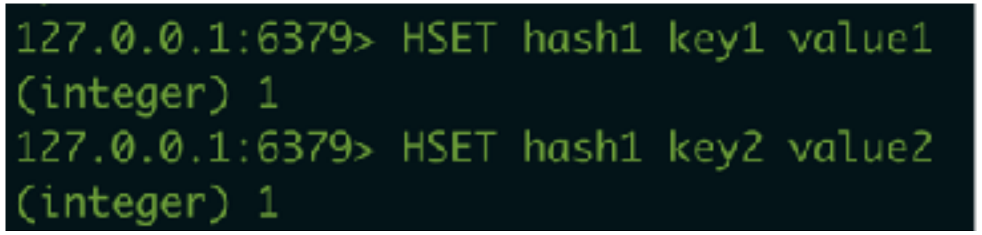
Hashes

- Hash tables as values

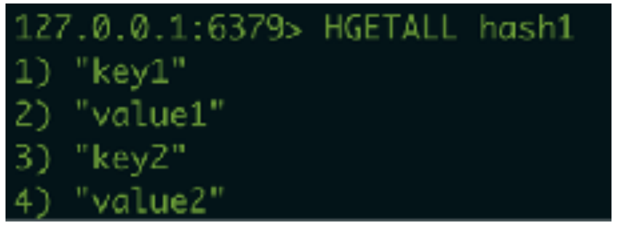
- Like object store with atomic access to object members



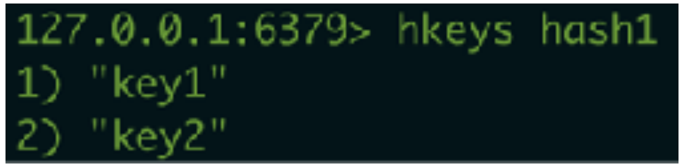
Hashes



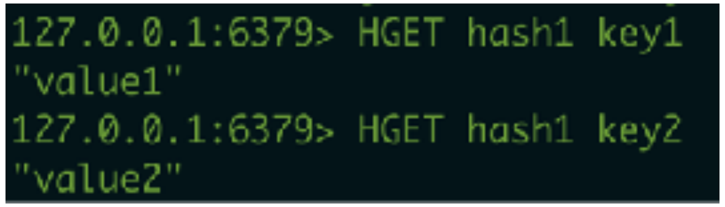
```
127.0.0.1:6379> HSET hash1 key1 value1
(integer) 1
127.0.0.1:6379> HSET hash1 key2 value2
(integer) 1
```



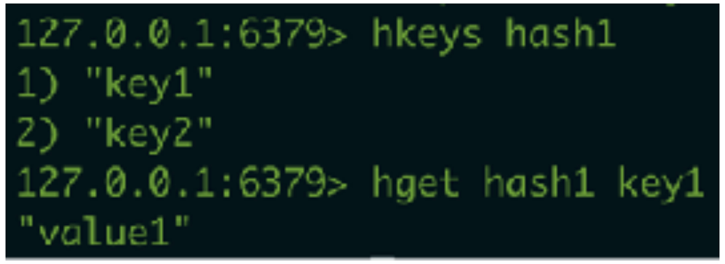
```
127.0.0.1:6379> HGETALL hash1
1) "key1"
2) "value1"
3) "key2"
4) "value2"
```



```
127.0.0.1:6379> hkeys hash1
1) "key1"
2) "key2"
```



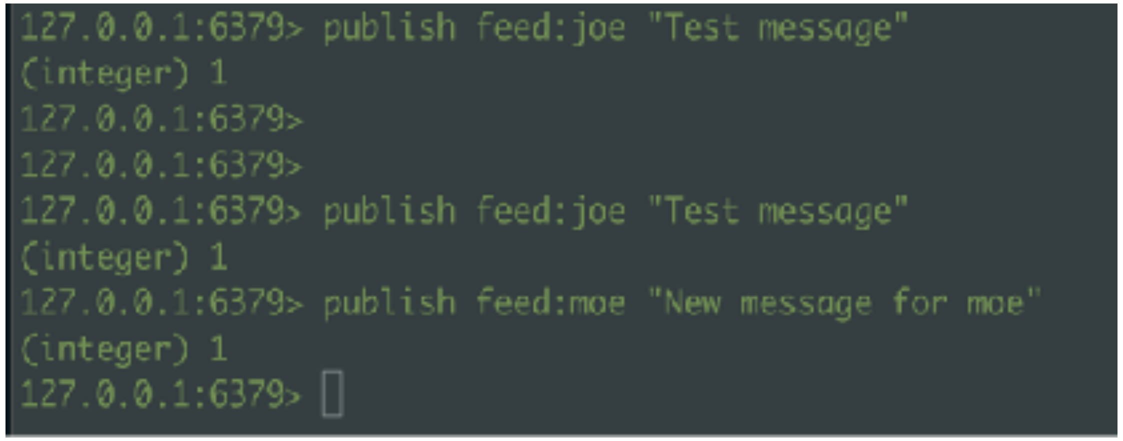
```
127.0.0.1:6379> HGET hash1 key1
"value1"
127.0.0.1:6379> HGET hash1 key2
"value2"
```



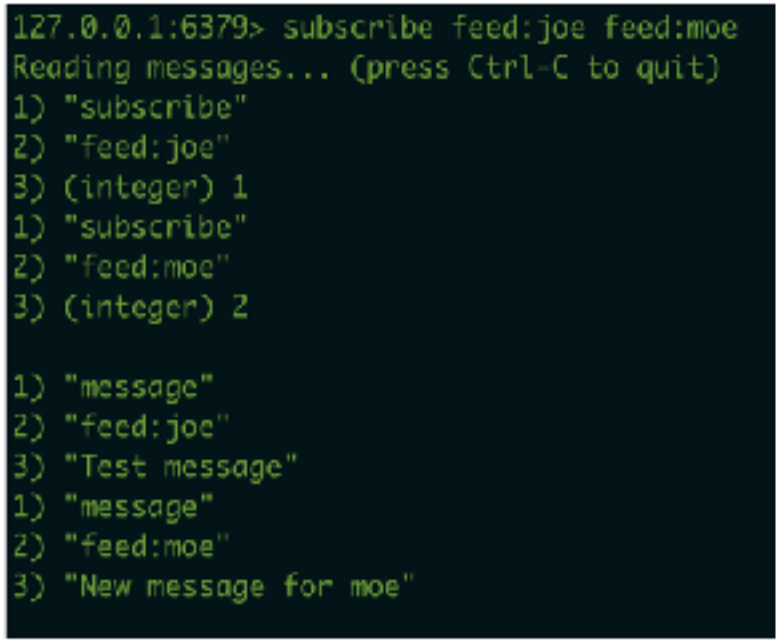
```
127.0.0.1:6379> hkeys hash1
1) "key1"
2) "key2"
127.0.0.1:6379> hget hash1 key1
"value1"
```

Pub/Sub

- Clients can subscribe to channels or patterns and receive notifications when messages are sent to channels.
- Subscribing is O(1), posting messages is O(n)
- Useful for chats, real-time analytics, twitter



```
127.0.0.1:6379> publish feed:joe "Test message"
(integer) 1
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> publish feed:joe "Test message"
(integer) 1
127.0.0.1:6379> publish feed:moe "New message for moe"
(integer) 1
127.0.0.1:6379> 
```



```
127.0.0.1:6379> subscribe feed:joe feed:moe
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "feed:joe"
3) (integer) 1
1) "subscribe"
2) "feed:moe"
3) (integer) 2

1) "message"
2) "feed:joe"
3) "Test message"
1) "message"
2) "feed:moe"
3) "New message for moe"
```

Publish / Subscribe

```
>>> r = redis.Redis(...)
>>> p = r.pubsub()
```

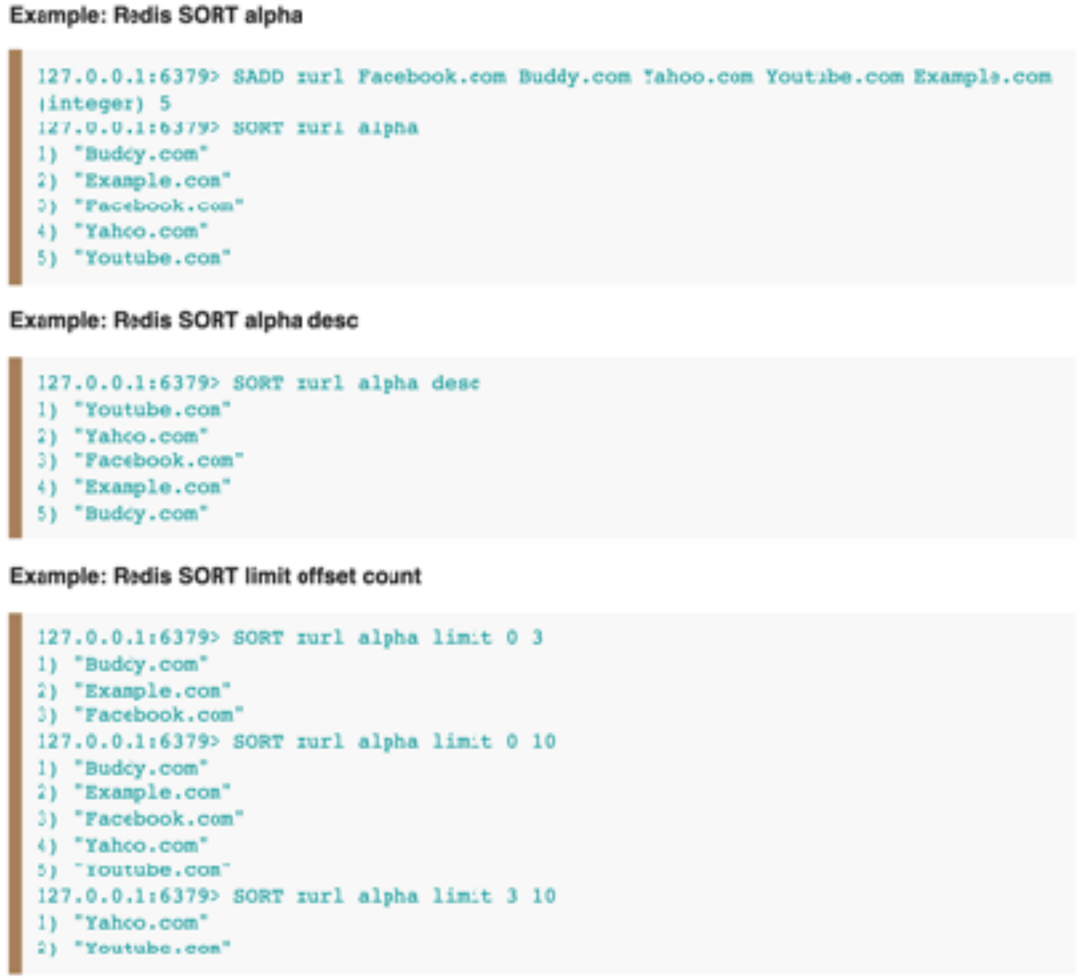
```
>>> p.subscribe('my-first-channel', 'my-second-channel', ...)
```

```
>>> p.get_message()
{'pattern': None, 'type': 'subscribe', 'channel': 'my-second-channel', 'data': 1L}
>>> p.get_message()
{'pattern': None, 'type': 'subscribe', 'channel': 'my-first-channel', 'data': 2L}
>>> p.get_message()
{'pattern': None, 'type': 'psubscribe', 'channel': 'my-*', 'data': 3L}
```

```
# the publish method returns the number matching channel and pattern
# subscriptions. 'my-first-channel' matches both the 'my-first-channel'
# subscription and the 'my-*' pattern subscription, so this message will
# be delivered to 2 channels/patterns
>>> r.publish('my-first-channel', 'some data')
2
>>> p.get_message()
{'channel': 'my-first-channel', 'data': 'some data', 'pattern': None, 'type': 'message'}
>>> p.get_message()
{'channel': 'my-first-channel', 'data': 'some data', 'pattern': 'my-*', 'type': 'pmessag
```

In-Memory Computing SUMMIT EUROPE 2019

Sort Keys



Example: Redis SORT alpha

```
127.0.0.1:6379> SADD zurl Facebook.com Buddy.com Yahoo.com Youtube.com Example.com
(integer) 5
127.0.0.1:6379> SORT zurl alpha
1) "Buddy.com"
2) "Example.com"
3) "Facebook.com"
4) "Yahoo.com"
5) "Youtube.com"
```

Example: Redis SORT alpha desc

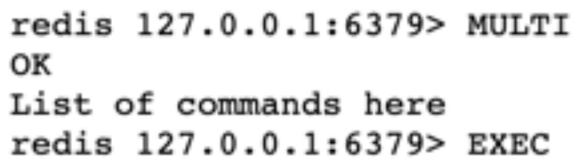
```
127.0.0.1:6379> SORT zurl alpha desc
1) "Youtube.com"
2) "Yahoo.com"
3) "Facebook.com"
4) "Example.com"
5) "Buddy.com"
```

Example: Redis SORT limit offset count

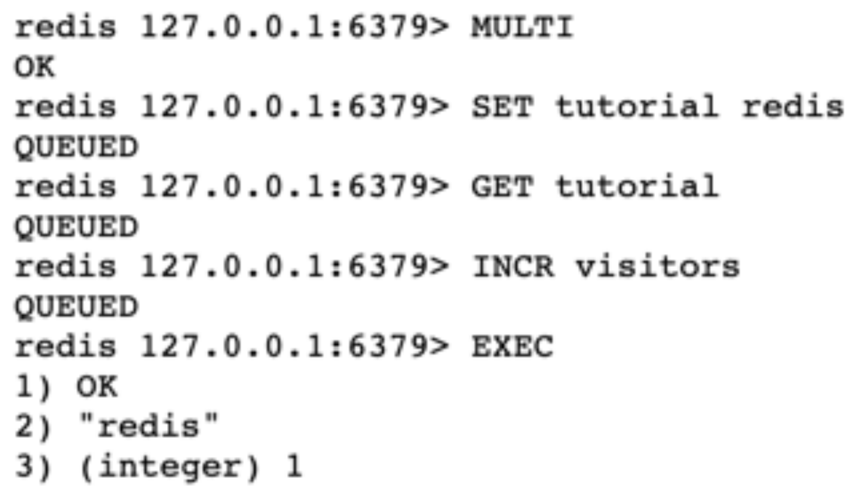
```
127.0.0.1:6379> SORT zurl alpha limit 0 3
1) "Buddy.com"
2) "Example.com"
3) "Facebook.com"
127.0.0.1:6379> SORT zurl alpha limit 0 10
1) "Buddy.com"
2) "Example.com"
3) "Facebook.com"
4) "Yahoo.com"
5) "Youtube.com"
127.0.0.1:6379> SORT zurl alpha limit 3 10
1) "Yahoo.com"
2) "Youtube.com"
```

Transactions

- Redis transaction is initiated by command **MULTI** and then you need to pass a list of commands that should be executed in the transaction, after which the entire transaction is executed by **EXEC** command.



```
redis 127.0.0.1:6379> MULTI
OK
List of commands here
redis 127.0.0.1:6379> EXEC
```



```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET tutorial redis
QUEUED
redis 127.0.0.1:6379> GET tutorial
QUEUED
redis 127.0.0.1:6379> INCR visitors
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) "redis"
3) (integer) 1
```

- Transactions can be discarded with DISCARD.

Integration between Database and Redis

- All front-end data is in RAM, denormalized and optimized for speed.
- Front-end talks only to Redis.
- Usage of Redis set features as keys and scoring vectors.
- All back-end data is on mysql, with a manageable, normalized schema.
- Admin talks only to MySql.
- Sync queue in the middle keeps both ends up to date.
- ORM is used to manage and sync data.
- Automated indexing in Redis generates models from MySql.

Redis Security

- It is designed to be accessed by trusted clients inside trusted environments.

Network security

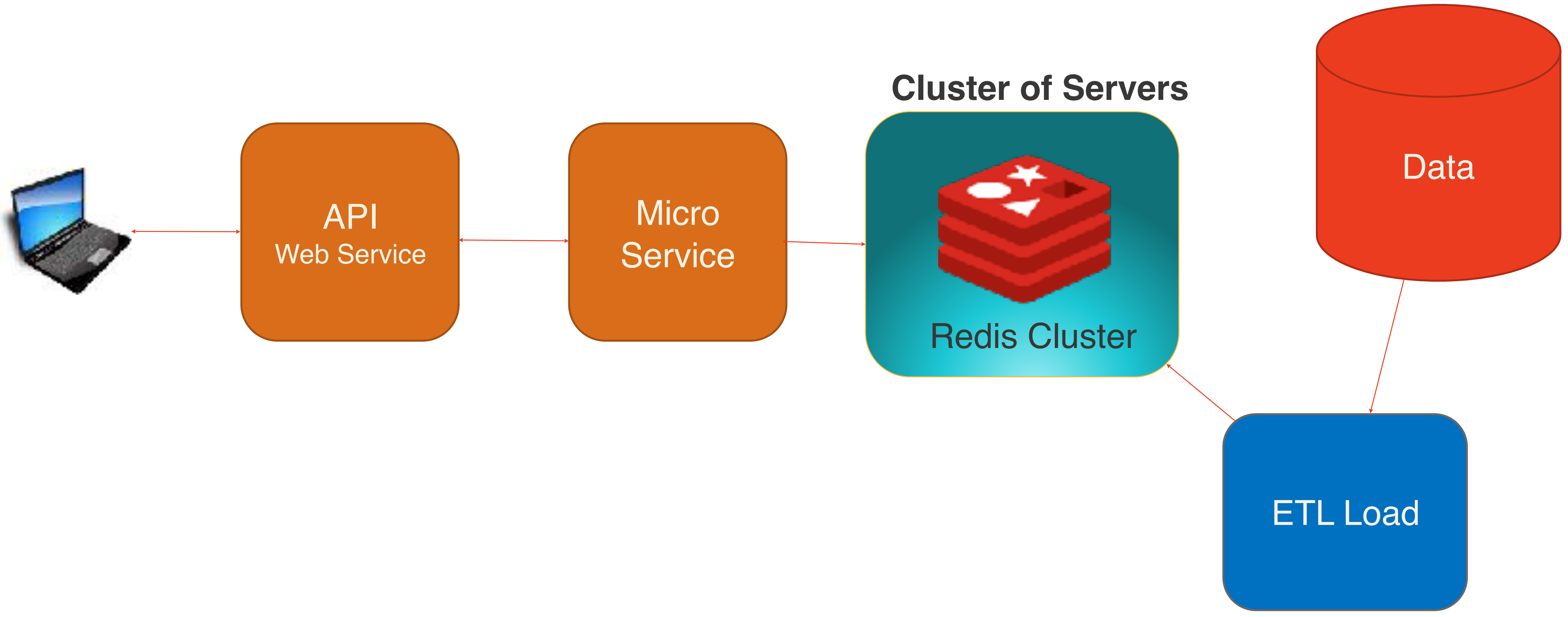
- Access to the Redis port should be denied to everybody but trusted clients in the network, so the servers running Redis should be directly accessible only by the computers implementing the application using Redis.
- Layer of authentication is optionally turned on editing the **redis.conf** file.
- When the authorization layer is enabled, Redis will refuse any query by unauthenticated clients. A client can authenticate itself by sending the **AUTH** command **followed by the password**.

Redis Password

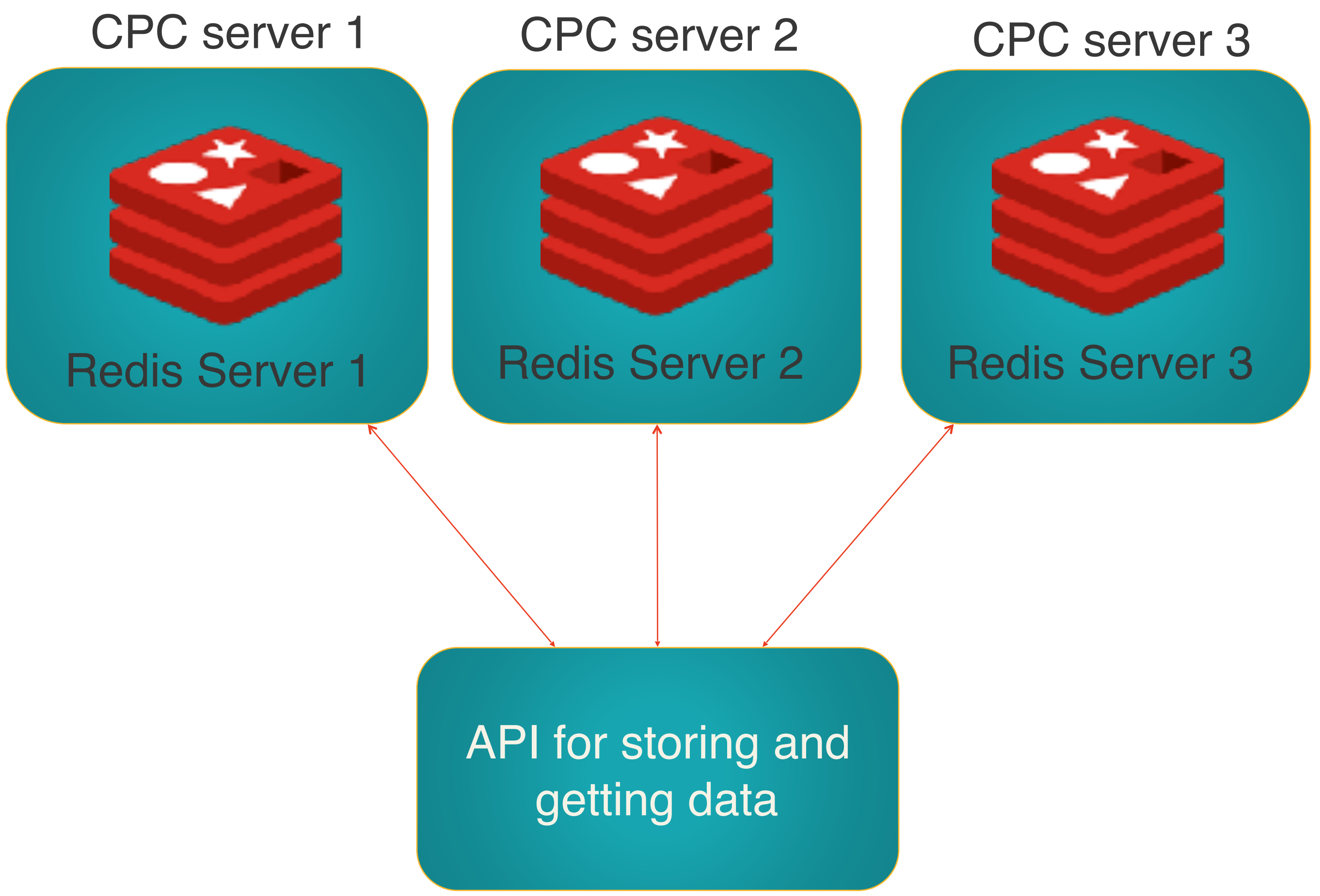
- User – granular setup

```
user newuser somepassword * +#readonly -#slow +zadd
user newuser2 otherpass stats:* +hgetall
user admin strongpass * +#all
```

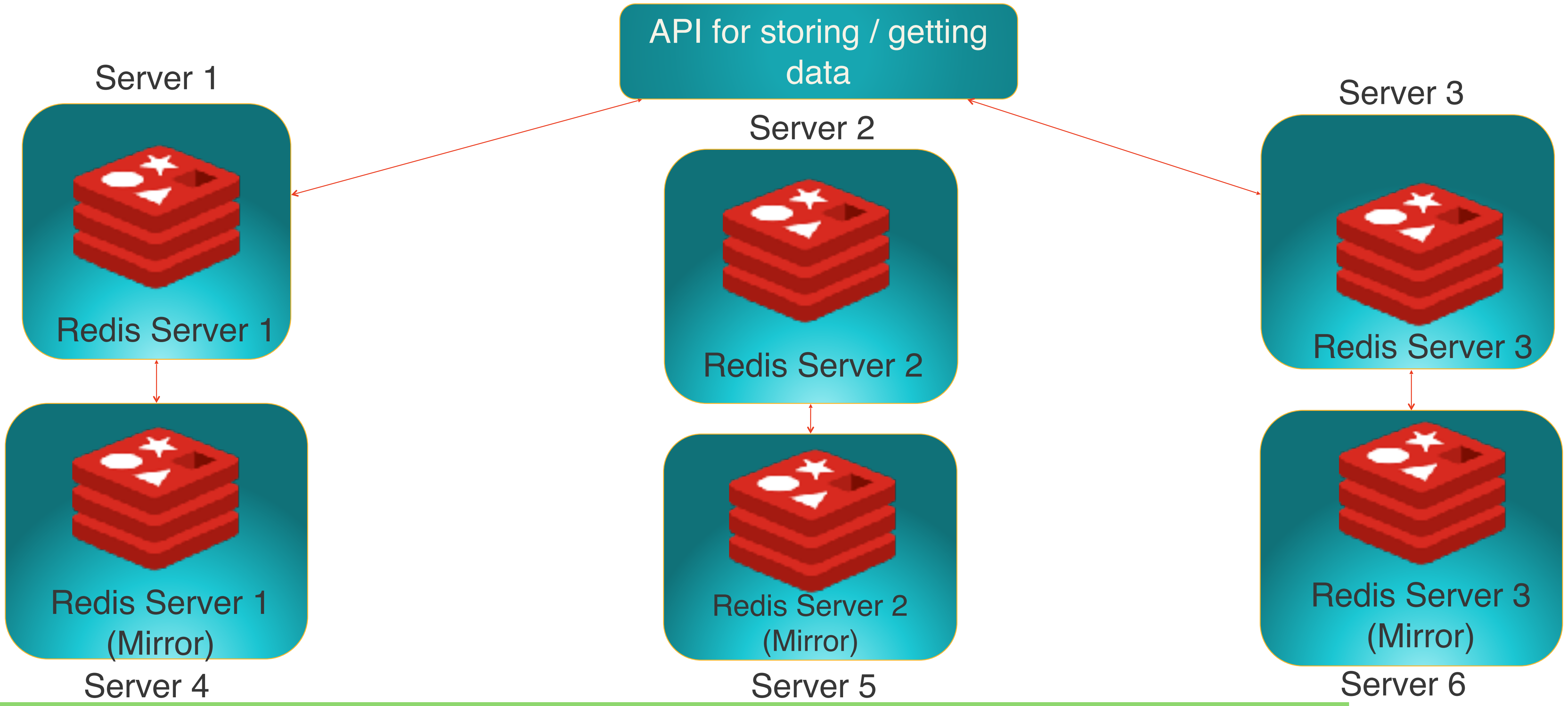
Application Architecture



Redis Cluster – Data Sharding

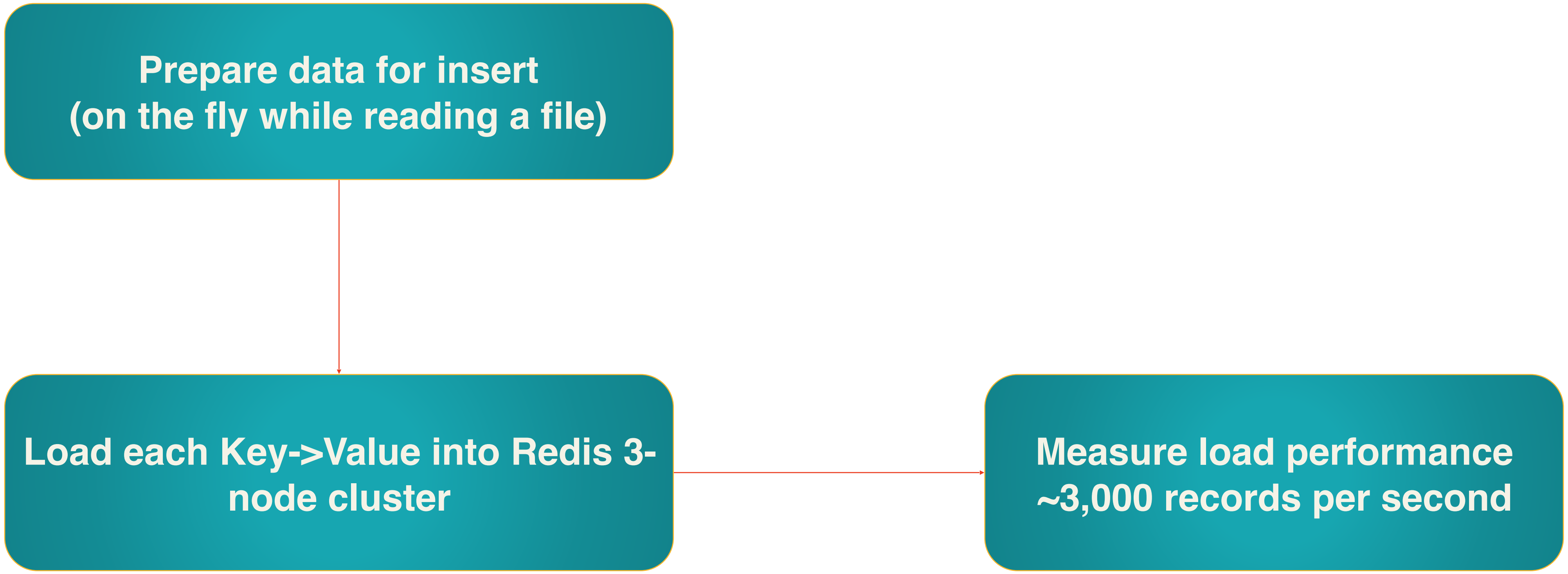


Mirroring Servers for HA



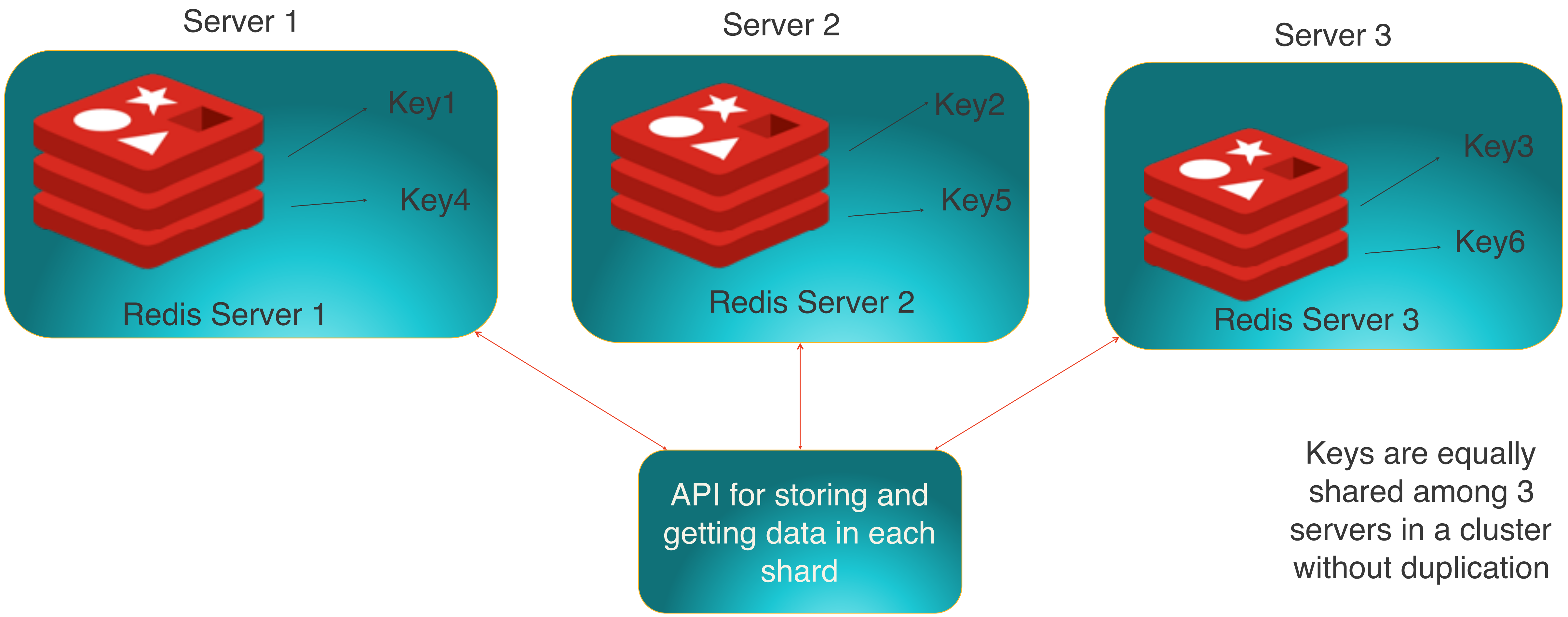
In-Memory Computing SUMMIT EUROPE 2019

Data Load to Redis


Prepare data for insert
(on the fly while reading a file)
Load each Key->Value into Redis 3-node cluster
Measure load performance ~3,000 records per second

Redis - Data Distribution

CPC server 2



Keys are equally shared among 3 servers in a cluster without duplication

RediSQL

RediSQL is the Fast, in-memory, SQL engine.

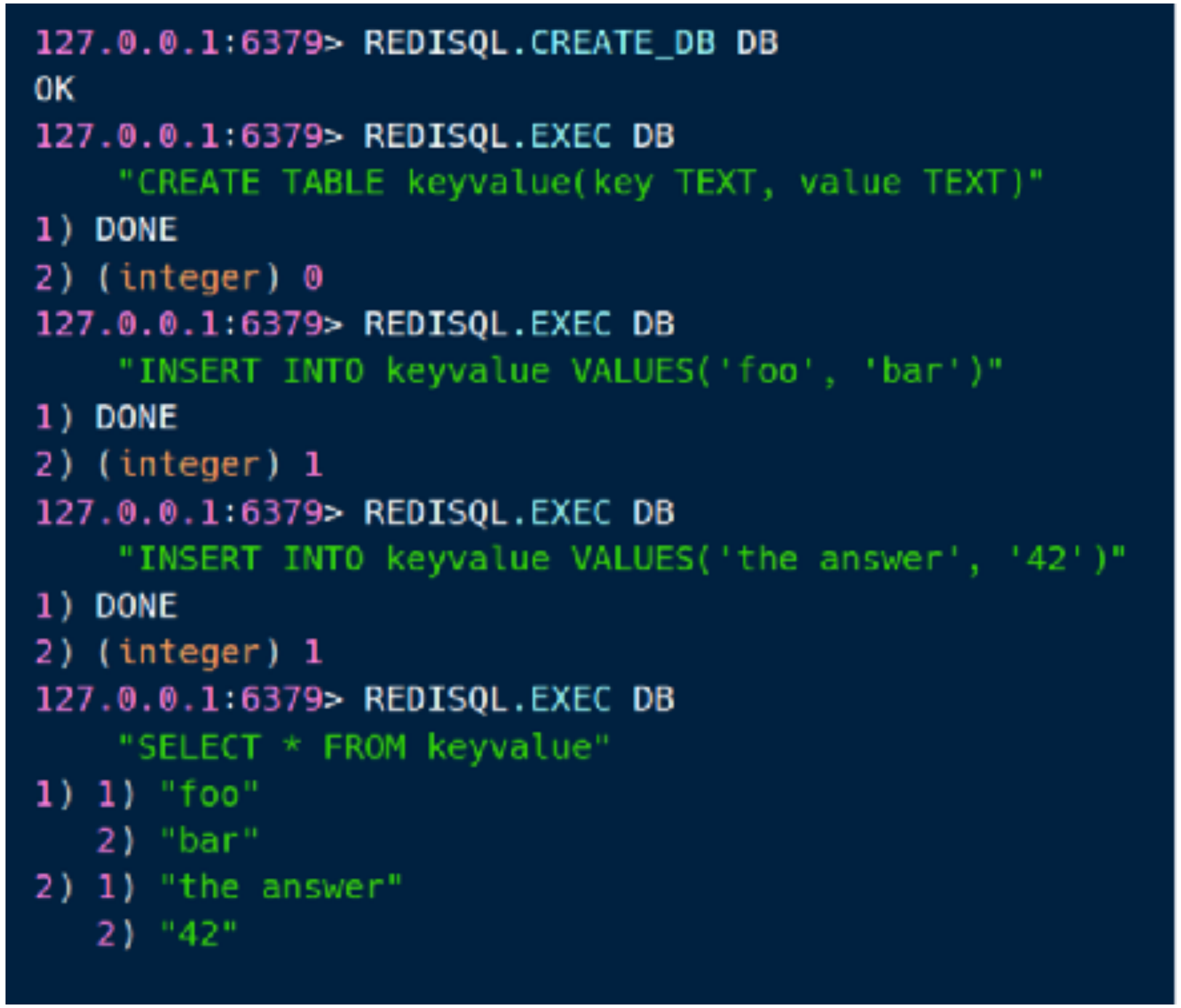
- Fast access and fast queries
- RediSQL works mainly in memory, it can reach up to **130.000 transaction per second.**

https://redisql.com/

RediSQL features

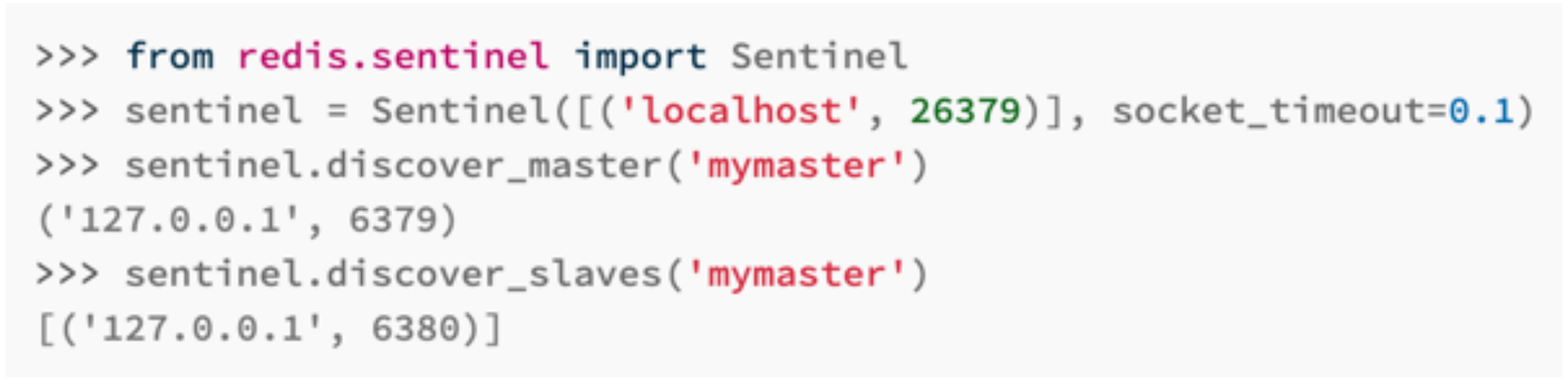
- Complete JSON support
- RediSQL exploits the **JSON1** module of SQLite to bring that capability to easy and quickly manage JSON data inside SQL statements and tables.
- In RediSQL you are able to **manipulate JSON** in every standard way.
- Full **text search** support
- RediSQL fully supports also the FTS{3,4,5} engine from SQLite, giving you a full text engine. You will be able to manage and search for data.

RediSQL

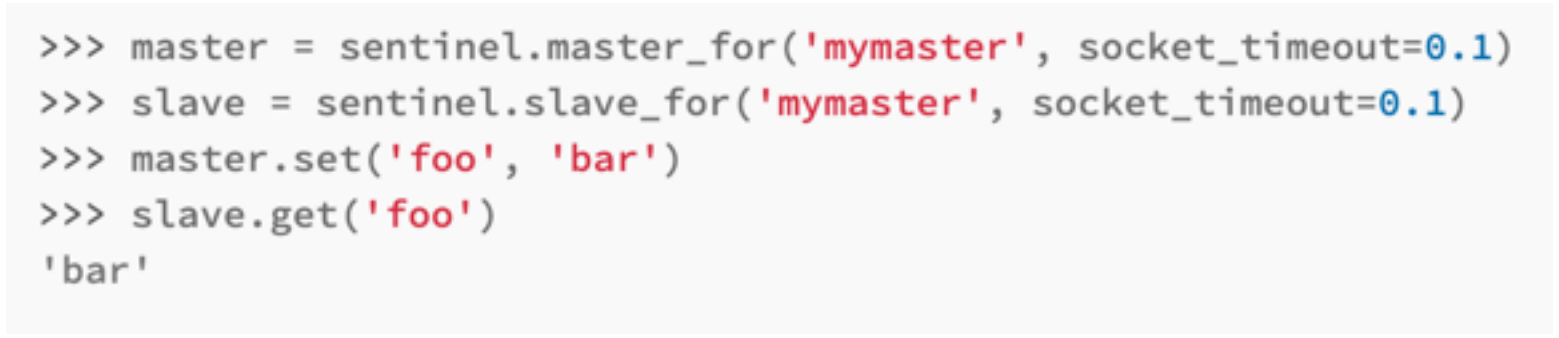


```
127.0.0.1:6379> REDISQL.CREATE_DB DB
OK
127.0.0.1:6379> REDISQL.EXEC DB
    "CREATE TABLE keyvalue(key TEXT, value TEXT)"
1) DONE
2) (integer) 0
127.0.0.1:6379> REDISQL.EXEC DB
    "INSERT INTO keyvalue VALUES('foo', 'bar')"
1) DONE
2) (integer) 1
127.0.0.1:6379> REDISQL.EXEC DB
    "INSERT INTO keyvalue VALUES('the answer', '42')"
1) DONE
2) (integer) 1
127.0.0.1:6379> REDISQL.EXEC DB
    "SELECT * FROM keyvalue"
1) 1) "foo"
   2) "bar"
2) 1) "the answer"
   2) "42"
```

Redis Cluster

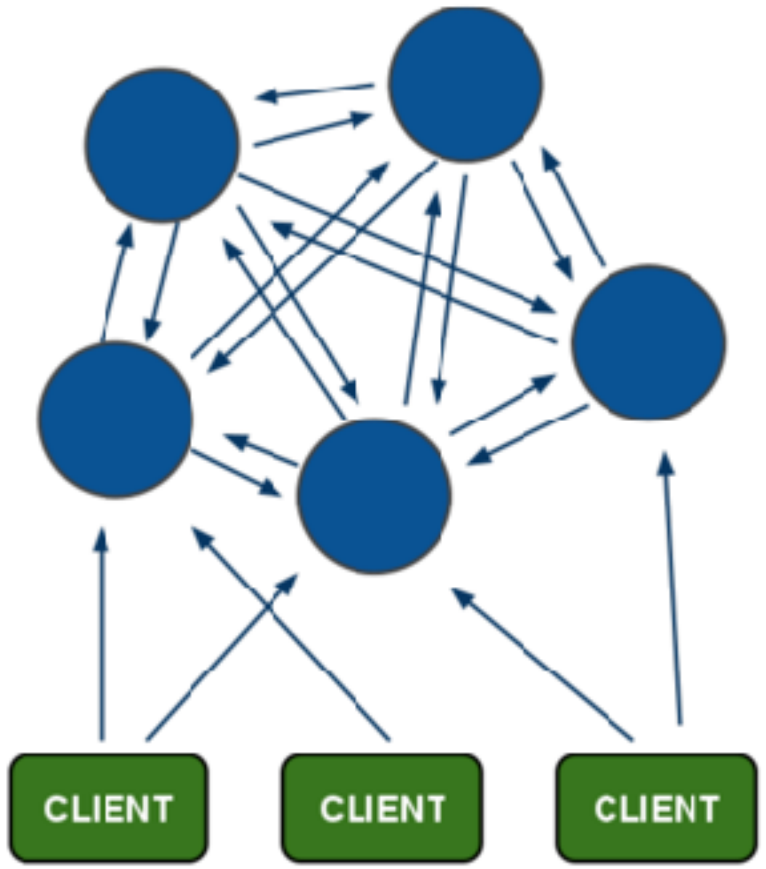


```
>>> from redis.sentinel import Sentinel
>>> sentinel = Sentinel([('localhost', 26379)], socket_timeout=0.1)
>>> sentinel.discover_master('mymaster')
('127.0.0.1', 6379)
>>> sentinel.discover_slaves('mymaster')
[('127.0.0.1', 6380)]
```



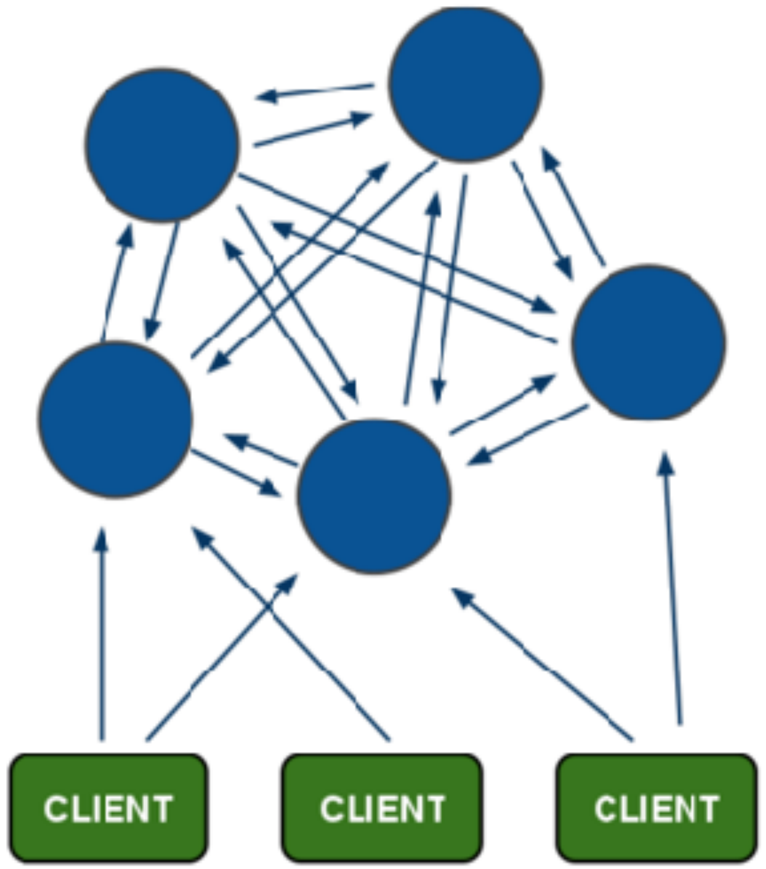
```
>>> master = sentinel.master_for('mymaster', socket_timeout=0.1)
>>> slave = sentinel.slave_for('mymaster', socket_timeout=0.1)
>>> master.set('foo', 'bar')
>>> slave.get('foo')
'bar'
```

Redis Cluster



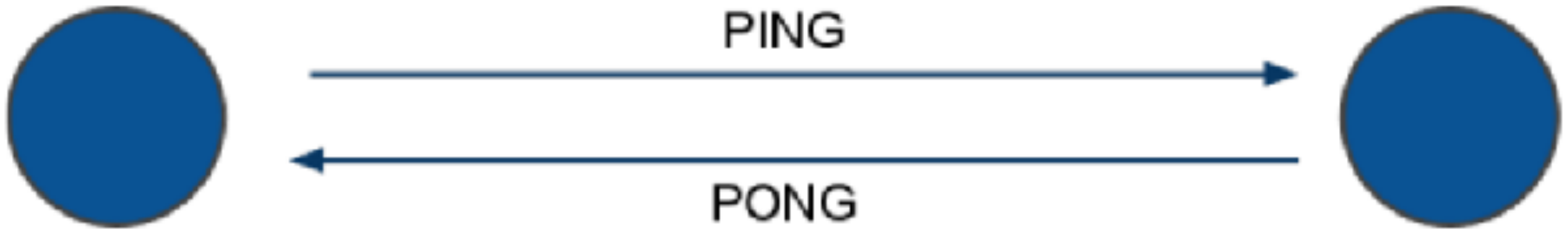
- **Redis Cluster** is an active-passive **cluster** implementation that consists of master and slave nodes.
- The **cluster** uses hash partitioning to split the key space into 16K key slots, with each master responsible for a subset of those slots.
- Each node in a **cluster** requires two TCP ports.

Redis Cluster



- All nodes are directly connected with a service channel.
- TCP baseport+4000, example 6379 -> 10379.
- Node to Node protocol is binary, optimized for bandwidth and speed.
- Clients talk to nodes as usually, using ascii protocol, with minor additions.
- Nodes don't proxy queries.

What nodes talk about?



PING: are you ok?
I'm master for XYZ hash slots.
Config is FF89X1JK

Gossip: this are info about other nodes I'm in touch with:

A replies to ping, I think its state is OK.
B is idle, I guess it's having problems but I need some ACK.

PONG: Sure I'm ok! I'm master for XYZ hash slots. Config is FF89X1JK

Gossip: I want to share with you some info about random nodes:

C and D are fine and replied in time.
But B is idle for me as well!
IMHO it's down!.

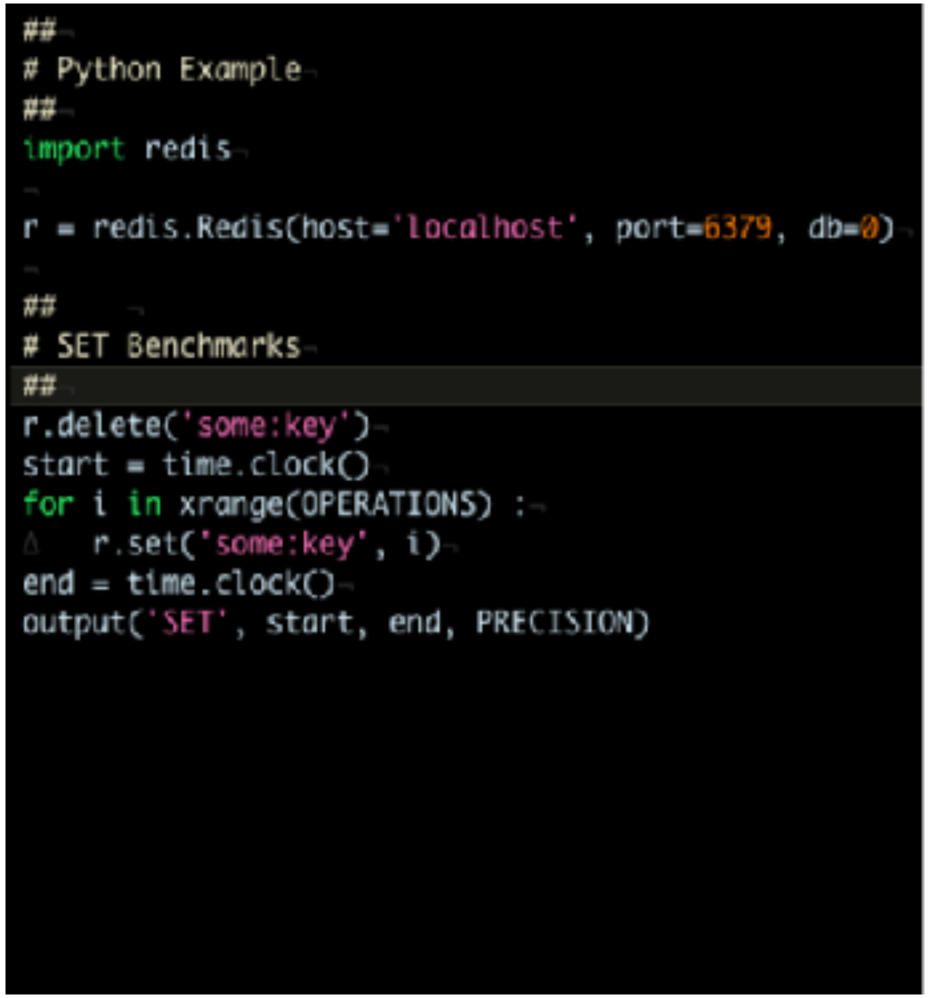
Using Redis with Python

- In order to use Redis with Python you will need a Python Redis client

```
pip install redis

 import redis
 r = redis.Redis (
             host='hostname’,
             port=port,
             password='password’)
 r = redis.Redis(host='localhost', port=6379, db=0)
 r.set('foo', 'bar’)
 r.get('foo')
```

Redis API - Python

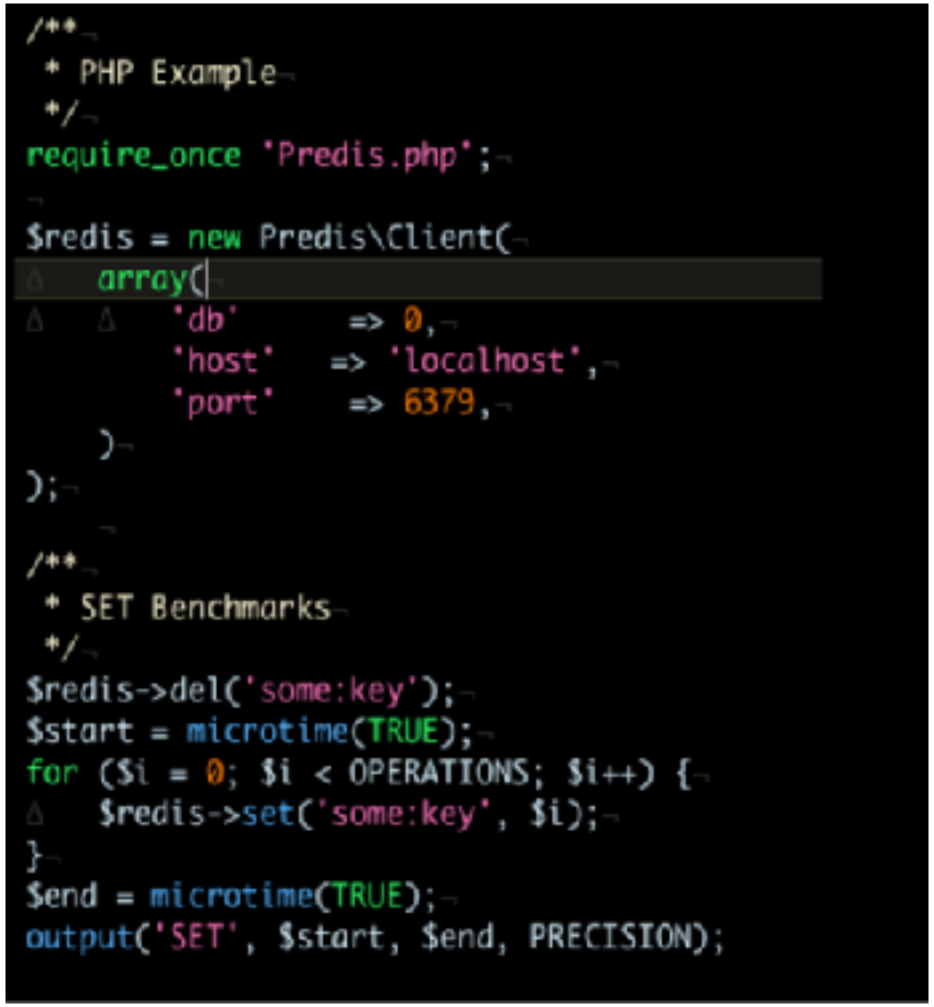


```
##
# Python Example
##
import redis

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)
```

Redis API - 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);
```

Pipelining

- Redis provides a feature called 'pipelining' - send many commands to redis all-at-once instead of one-at-a-time.
- With pipelining, redis can buffer several commands and execute them all at once, responding with a single reply.
- This can allow you to achieve even greater throughput on bulk importing or other actions that involve lots of commands.

Pipelines

```
>>> r = redis.Redis(...)
>>> r.set('bing', 'baz’)

# Use the pipeline() method to create a pipeline instance
>>> pipe = r.pipeline()

# The following SET commands are buffered
>>> pipe.set('foo', 'bar’)
>>> pipe.get('bing’)

# the EXECUTE call sends all buffered commands to the server, returning
# a list of responses, one for each command.
>>> pipe.execute()
[True, 'baz']
```

Running out of memory?

- Redis will either be **killed by the Linux kernel OOM killer, crash with an error, or will start to slow down**.
- With modern operating systems malloc() returning NULL is not common, usually the **server will start swapping** (if some swap space is configured), and Redis performance will start to degrade.
- Redis has built-in protections allowing the user to **set a max limit to memory usage**. If this limit is reached Redis will start to reply with an error to write commands (but will continue to accept read-only commands), or we can configure it to evict keys when the max memory limit is reached.

Redis Threading

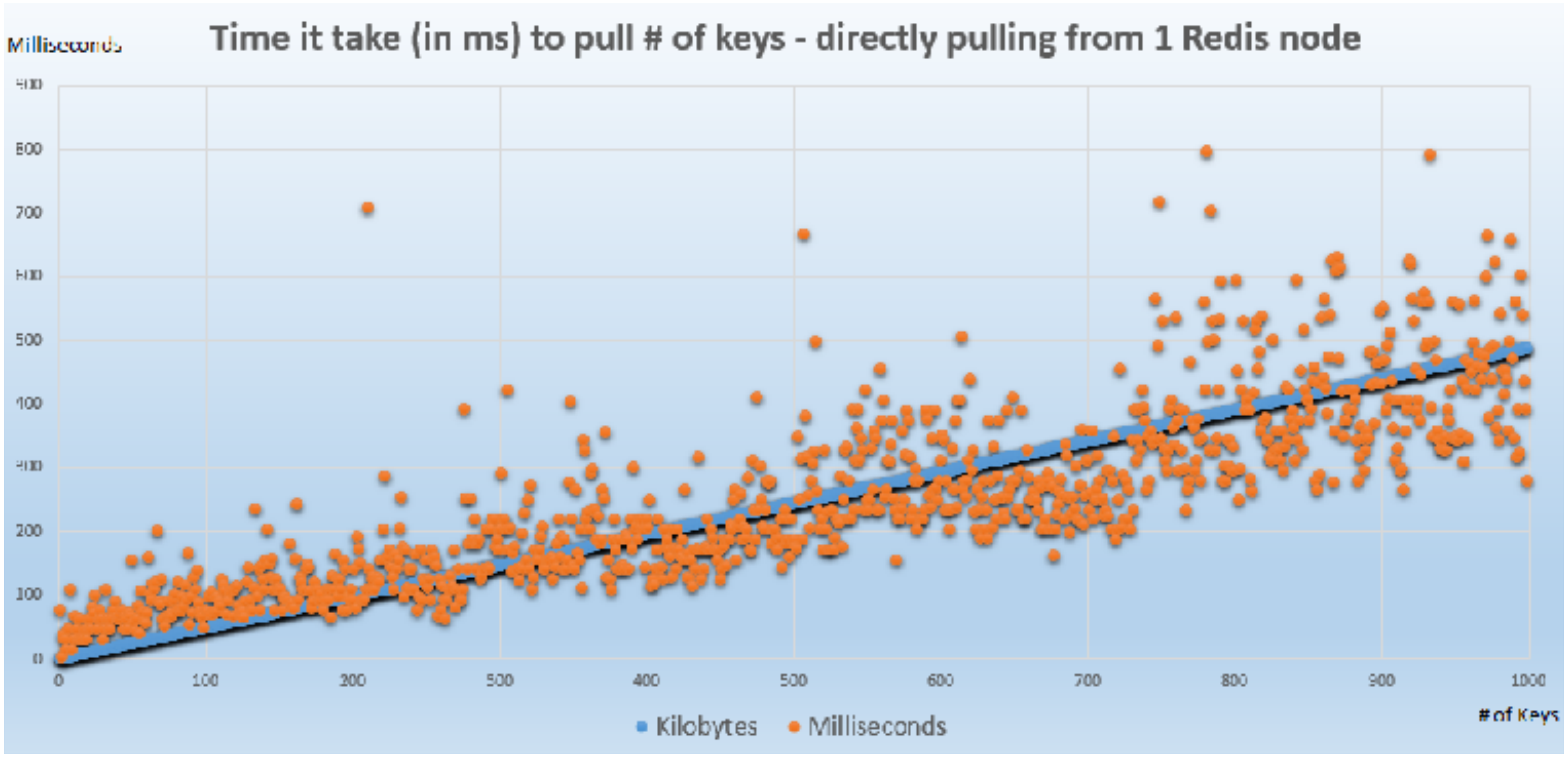
- Redis is single threaded.
- Usually **Redis is either memory or network bound**.
- Using pipelining Redis running on Linux system can deliver even 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU.
- To maximize CPU usage - start multiple instances of Redis in the same box and treat them as different servers.
- With Redis 4.0+ it became more threaded. For now this is limited to deleting objects in the background, and to blocking commands implemented via Redis modules.

Data Persistence

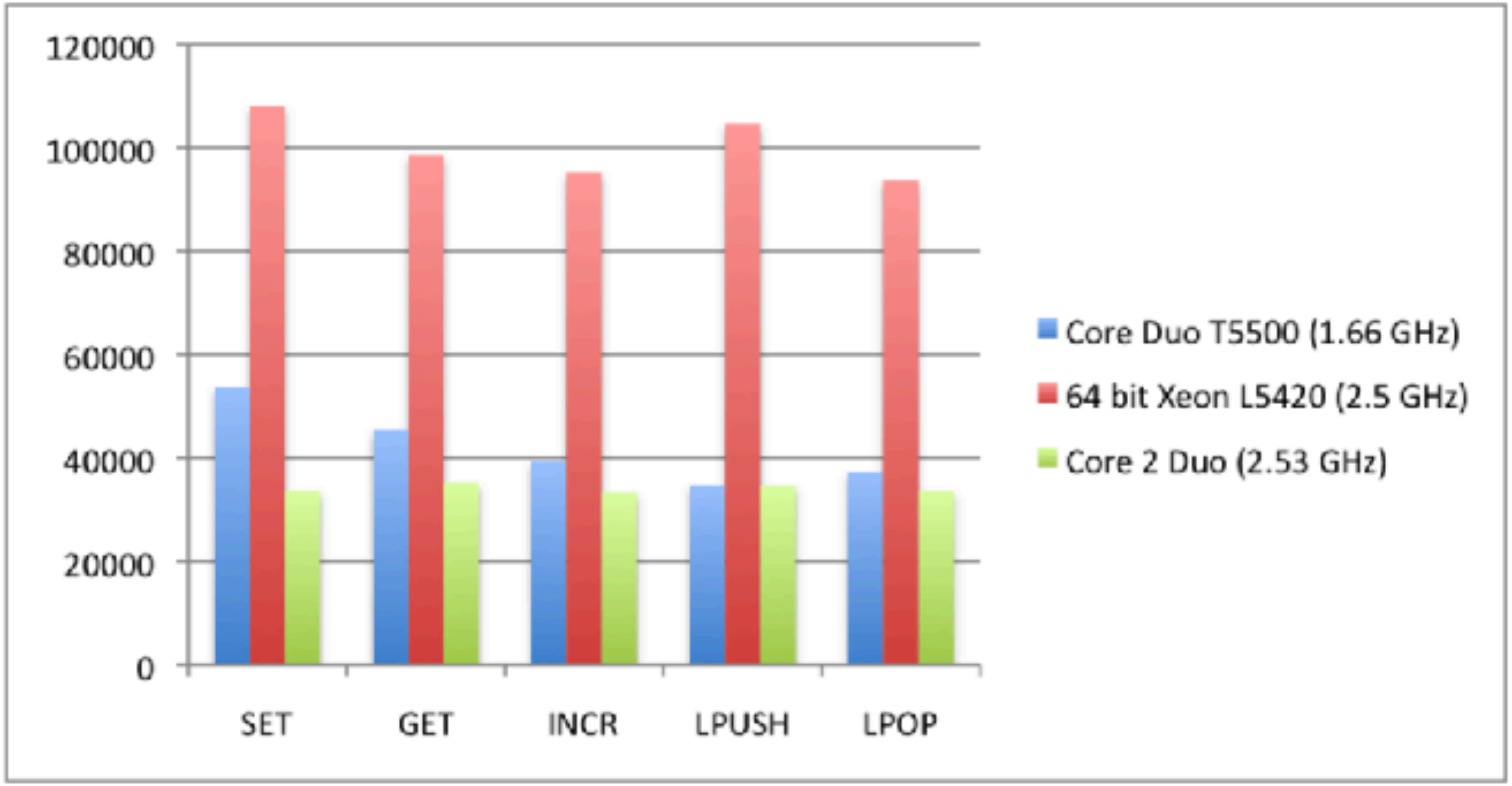
- **Periodic Dump ("Background Save")**
  - fork() with Copy-on-Write, write entire DB to disk
  - When?
    - *After every X seconds and Y changes, or,*
    - *BGSAVE command*

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

Performance Testing
Multiple keys read at once



Benchmark - Hardware



Lessons Learned

- 64-bit instances consume much more RAM
- Use MONITOR to see what is going on
- Master/Slave sync if far from perfect (via manual setup)

Redis Use Cases

- Stock prices
- Analytics
- Real-time data collection
- Real-time communication
- And wherever you used memcached before

Memcached

Memcached

What is Memcached?

- High-performance, distributed memory object caching system. Used in speeding up dynamic web applications by alleviating database load.

When can we us it?

- Anywhere if we have a spare RAM
- Mostly used in wiki, social networking and book marketing sites.

Why should we us it?

- If we have a high-traffic site that is dynamically generated with a high database load that contains mostly read threads when Memcached can help lighten the load on a database.

Memcached

History of Memcached

- Brad Fitzpatrick from Danga Interactive developed Memcached to **enhance speed of livejournal.com**, which was then doing 20M+ dynamic page loads per day.
- Memcached **reduced DB load to almost 0**, yielding faster page load time and better resource utilization.
- **Facebook is the biggest user of memchaced** after live journal. They have > 100 dedicated Memcached servers.

Memcached Installation

```
wget http://memcached.org/latest
tar -zxvf memcached-1.x.x.tar.gz
cd memcached-1.x.x
./configure && make && make test && sudo make install
```

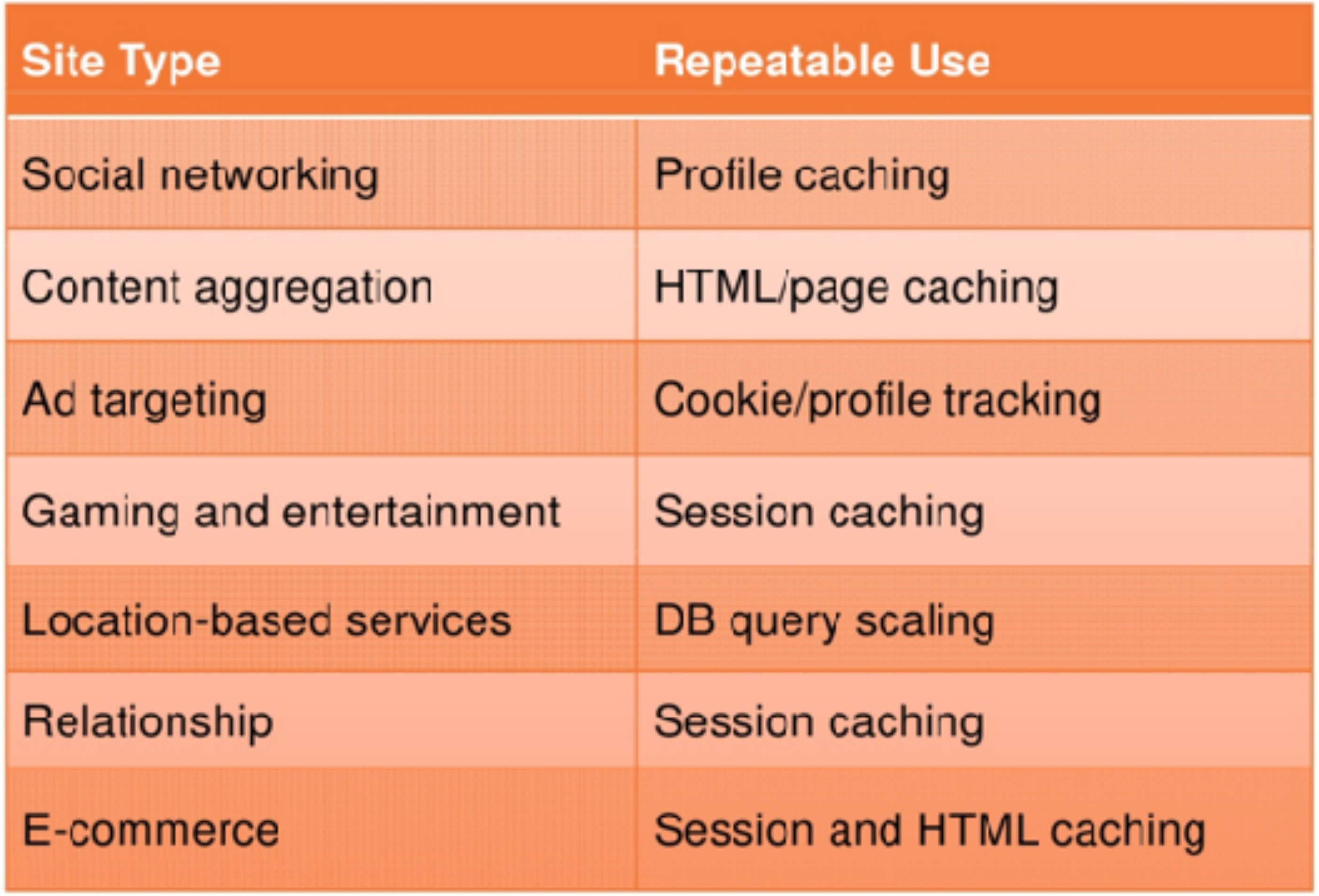
Memcached

- **Limits**
  - Key size = (250 bytes)
  - 32bit/64bit (maximum size of process)
- **LRU**
  - Least recently accessed items are cycled out
  - One LRU exists per “slab class”
  - LRU “evictions” need not be common
- **It has Threads**

What Memcached is NOT

- 🚫 A persistent data store
- 🚫 A database
- 🚫 Application-specific
- 🚫 A large object cache
- 🚫 Fault-tolerant or highly available

Memcached - Use Cases

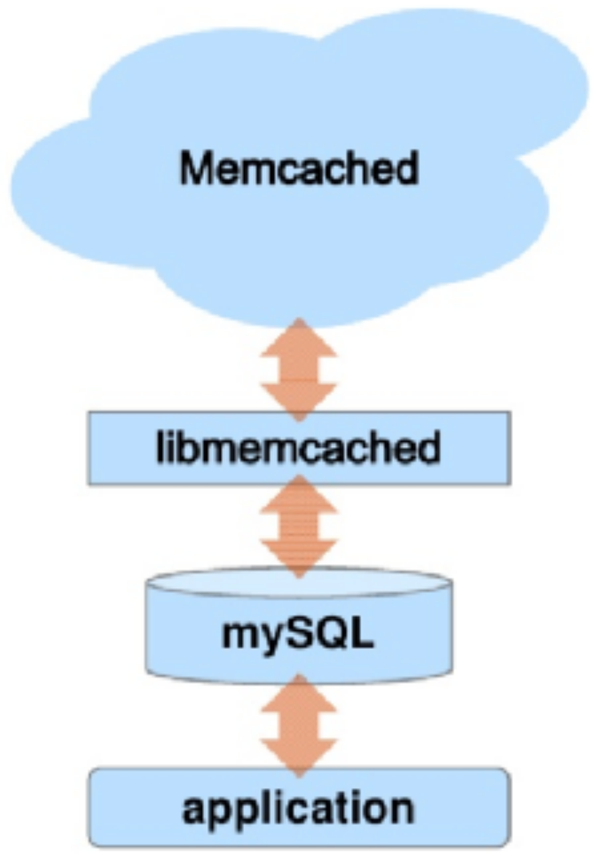


Site Type	Repeatable Use
Social networking	Profile caching
Content aggregation	HTML/page caching
Ad targeting	Cookie/profile tracking
Gaming and entertainment	Session caching
Location-based services	DB query scaling
Relationship	Session caching
E-commerce	Session and HTML caching

In-Memory Computing SUMMIT EUROPE 2019

Memcached - Integration with Database

- Suite of functions that work with Memcached and MySQL
- Leverage power of SQL engine
- Combine tasks
- Open source



Use of Memcached

- Homepage data (often, shared expensive)
- Great for summaries
  - Overview
  - Pages where it is not that big a problem if data is a little bit out of date (e.g. search results)
- Good for quick and dirty optimizations

When NOT to use Memcached

- When you have very large objects
- When have keys larger than 250 characters
- When running in un-secure environment
- When persistence is needed, or a database

Redis and Memcached Comparison

Memory Usage

Redis is in general better.

Memcached:

- Specify the cache size and as you insert items the daemon quickly grows to a little more than this size.
- There is not a good a way to reclaim any of that space, short of restarting memcached. All your keys could be expired, you could flush the database, and it would still use the full chunk of RAM you configured it with.

Redis:

- Setting a max size is up to us. Redis will never use more than it has to and will give you back memory it is no longer using.
- Example of storing 100K ~2KB strings (~200MB) of random sentences into both. Memcached RAM usage grew to ~225MB. Redis RAM usage grew to ~228MB. After flushing both, redis dropped to ~29MB and memcached stayed at ~225MB. They are similarly efficient in how they store data, but only one is capable of reclaiming it.

Redis VS Memcached

- Memcached is a simple volatile cache server.
- It is good at this, but that is all it does. You can access those values by their key at extremely high speed, often saturating available network or even memory bandwidth.
- When you restart memcached your data is gone. This is fine for a cache. You shouldn't store anything important there.
- If you need high performance or high availability there are 3rd party tools, products, and services available.

Disk I/O and Read/Write

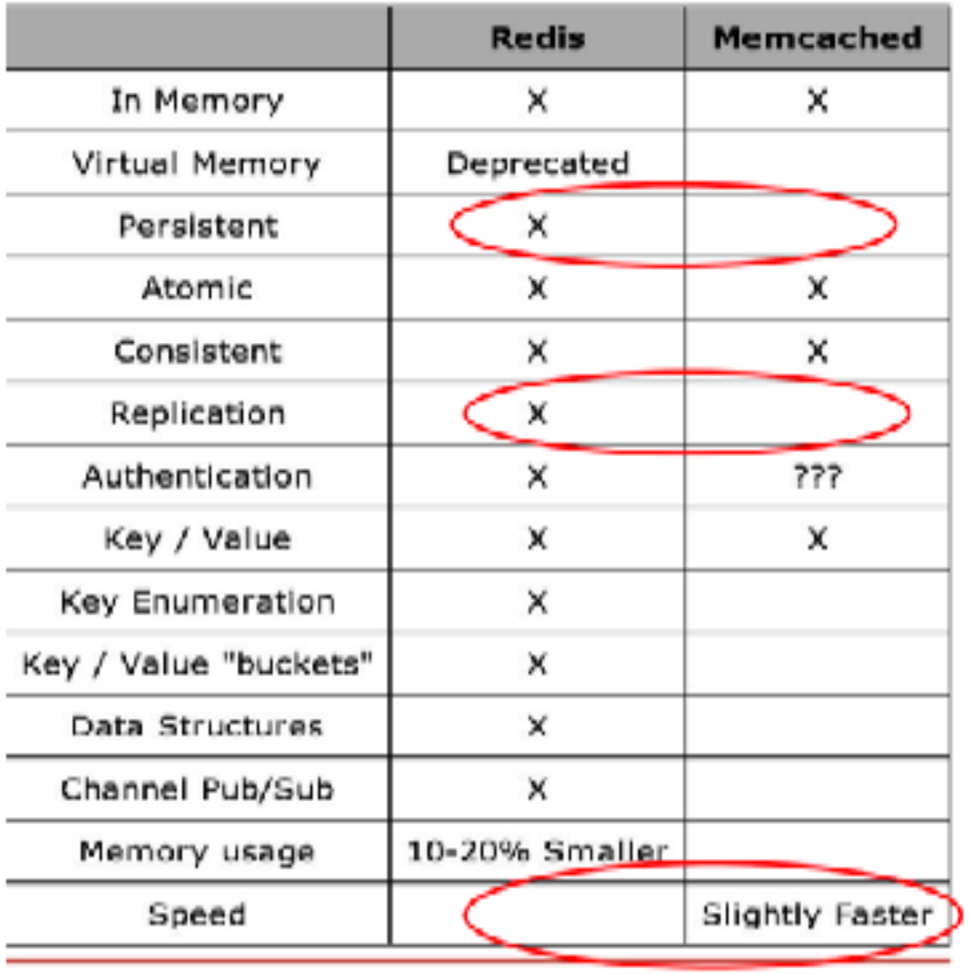
Disk I/O dumping:

- ➢ A clear win for Redis since it does this by default and has very configurable persistence.
- ➢ Memcached has no mechanisms for dumping to disk without 3rd party tools.

Read/write speed:

- ➢ Both are extremely fast. Benchmarks vary by workload, versions, and many other factors but generally show redis to be as fast or almost as fast as memcached.

Redis VS Memcached



	Redis	Memcached
In Memory	X	X
Virtual Memory	Deprecated	
Persistent	X	
Atomic	X	X
Consistent	X	X
Replication	X	
Authentication	X	???
Key / Value	X	X
Key Enumeration	X	
Key / Value "buckets"	X	
Data Structures	X	
Channel Pub/Sub	X	
Memory usage	10-20% Smaller	
Speed		Slightly Faster

Redis VS Memcached

- Redis can do the same jobs as memcached can, and better.
- Redis can act as a cache as well. It can store key/value pairs too. In Redis Value can be up to 512MB.
- Memcached had default maximum object size is 1MB. In version 1.4.2 and later, you can change the maximum size of an object using the `-I` command line option.
- In Redis you can turn off persistence and it will happily lose your data on restart too. If you want your cache to survive restarts it lets you can do that as well (default).

Redis VS Memcached

- If one instance of redis/memcached isn't enough performance for your workload, redis is the clear choice.
- Redis includes cluster support and comes with high availability tools (redis-sentinel) right "in the box". Over the past few years redis has also emerged as the clear leader in 3rd party tooling.
- Companies like Redis Labs, Amazon, and others offer many useful redis tools and services. The ecosystem around redis is much larger. The number of large scale deployments is now likely greater than for memcached.

Persistence

- By default redis persists data to disk using a mechanism called snapshotting. If you have enough RAM available it's able to write all data to disk with almost no performance degradation. It's almost free!

- In snapshot mode there is a chance that a sudden crash could result in a small amount of lost data. If you absolutely need to make sure no data is ever lost, redis has **AOF (Append Only File)** mode. In this persistence mode data can be synced to disk as it is written. **This can reduce maximum write throughput to however fast your disk can write**, but should still be quite fast.

- There are many configuration options to fine tune persistence if you need, but the defaults are very sensible. These options make it easy to setup redis as a safe, redundant place to store data. It is a *real* database.

Transactions and Atomicity

- Commands in redis are atomic, meaning you can be sure that as soon as you write a value to redis that value is visible to all clients connected to redis.

- There is no wait for that value to propagate. Technically memcached is atomic as well, but with redis adding all this functionality beyond memcached it is worth noting and somewhat impressive that all these additional data types and features are also atomic.

- While not quite the same as transactions in relational databases, redis also has transactions that use "optimistic locking" (WATCH/MULTI/EXEC).

Redis VS Memcached - Conclusion

- Memcached is limited to strings
- Redis is more powerful, more popular, and better supported than memcached. It has more tools for leveraging this datatype by offering commands for bitwise operations, bit-level manipulation, floating point increment/decrement support, range queries, and multi-key operations. Memcached doesn't support any of that.
- Memcached can only do a small fraction of the things Redis can do.
- Redis is better even where their features overlap.
- For anything new, use Redis.

Redis on AWS

Redis on AWS – Replication and Persistence

- Now supports Redis 5.0.3 - latest GA version of open-source Redis.
- Redis has a primary-replica architecture and **supports asynchronous replication** where data can be replicated to multiple replica servers.
- This provides **improved read performance** (as requests can be split among the servers) and faster recovery when the primary server experiences an outage.
- For persistence, Redis supports point-in-time backups (copying the Redis data set to disk).

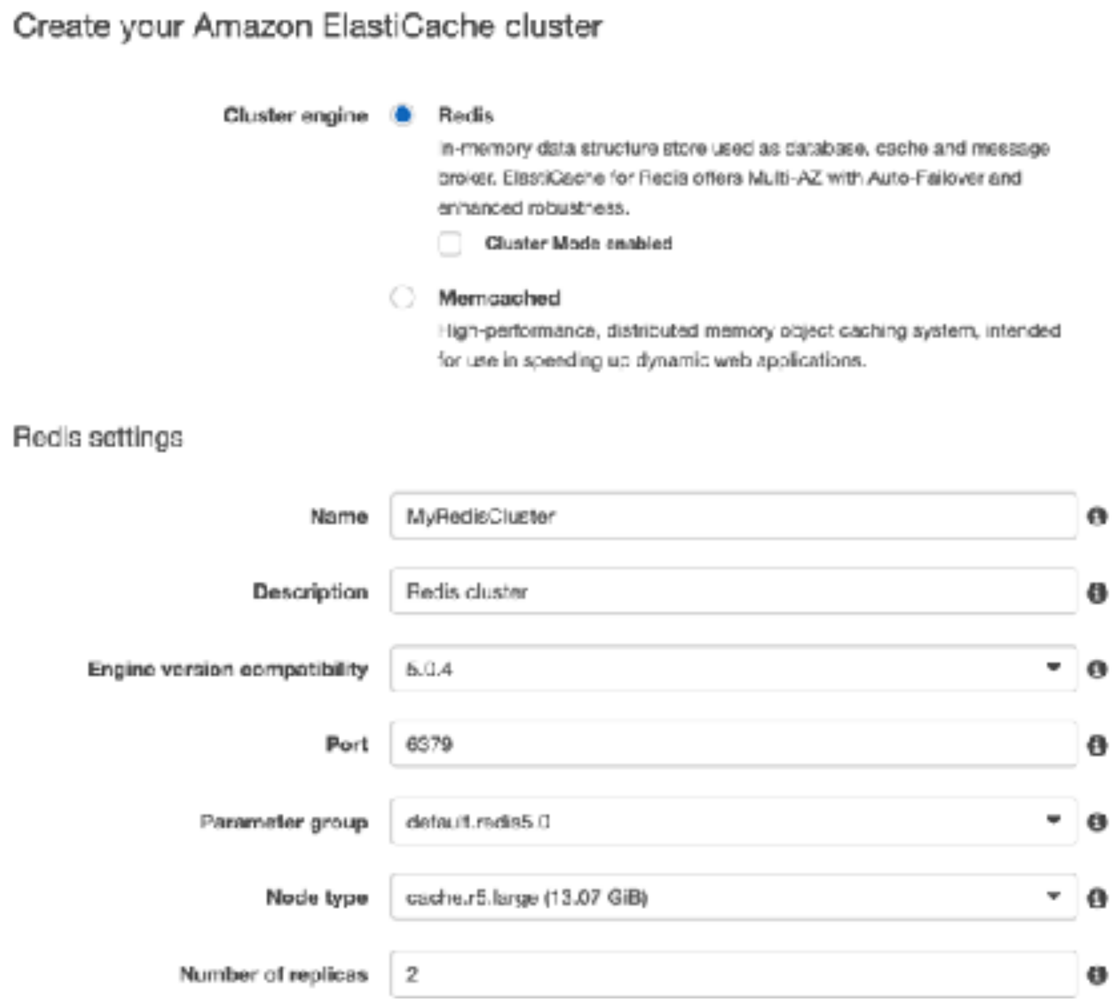
Redis on AWS – Replication and Persistence

- HA and scalable
- Redis offers a **primary-replica architecture or a clustered topology**. This allows you to build highly available solutions providing consistent performance and reliability.
- When we need to adjust a cluster size, various options to scale up and scale in or out are also available. This allows for a cluster to grow with demands.

Redis on AWS - ElastiCache

- AWS ElastiCache is a **fully managed** service for Redis.
- No need to perform management tasks such as hardware provisioning, software patching, setup, configuration, monitoring, failure recovery, and backups.
- Continuously **monitors** clusters to keep Redis **up and running**
- It provides detailed monitoring metrics associated with nodes, to **diagnose and react to issues quickly**.
- ElastiCache adds **automatic write throttling**, intelligent **swap memory management**, and **failover** enhancements to improve upon the availability and manageability of open source Redis.

Redis on AWS – Creating a New Cluster



Create your Amazon ElastiCache cluster

Cluster engine

[✓ Redis]
In-memory data structure store used as database, cache and message broker. ElastiCache for Redis offers Multi-AZ with Auto-Failover and enhanced robustness.

☐ Cluster Mode enabled

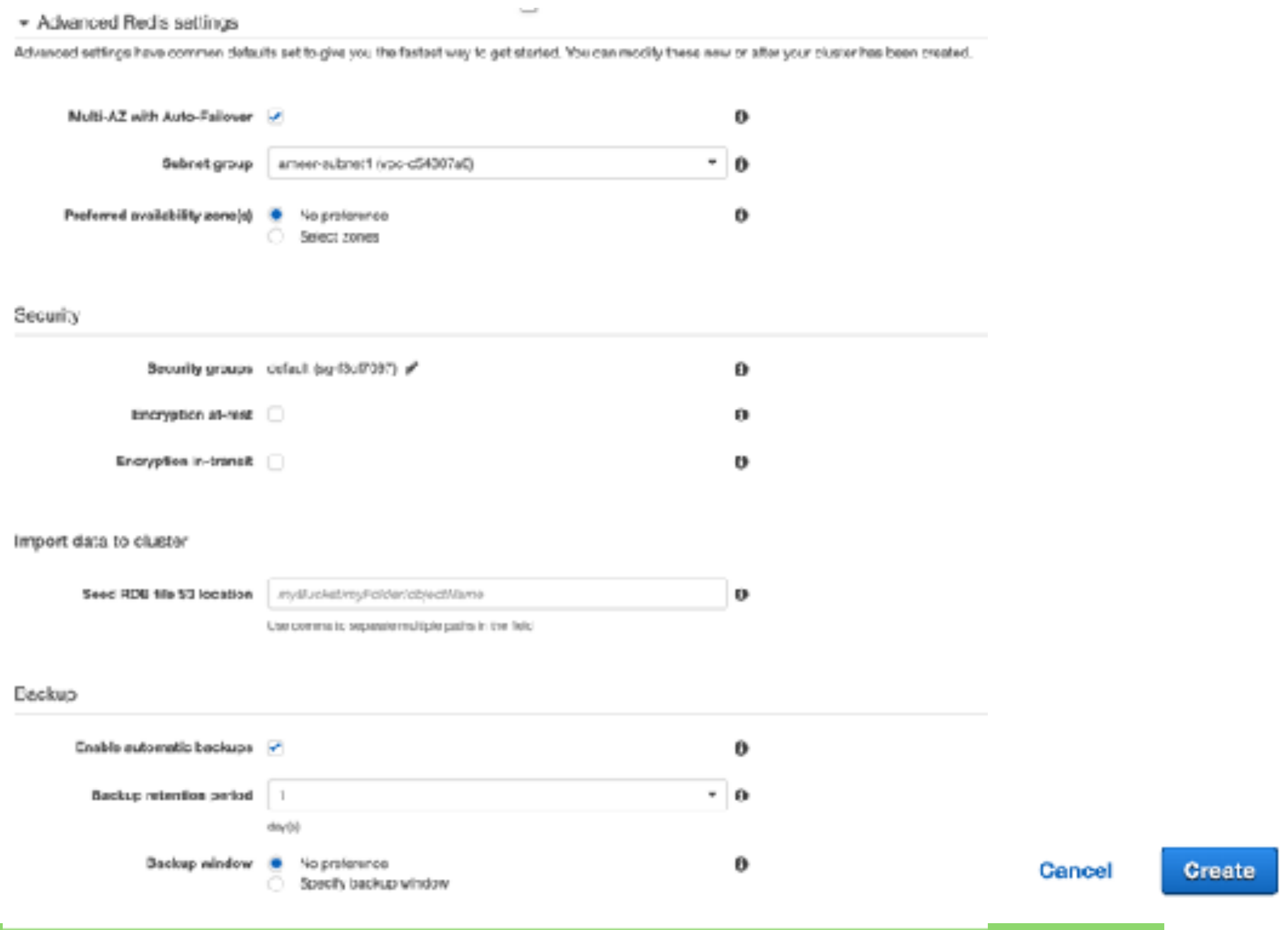
Memcached
High-performance, distributed memory object caching system, intended for use in speeding up dynamic web applications.

Redis settings

Name	MyRedisCluster
Description	Redis cluster
Engine version compatibility	5.0.4
Port	6379
Parameter group	default.redis5.0
Node type	cache.r5.large (13.07 GiB)
Number of replicas	2

In-Memory Computing SUMMIT EUROPE 2019

Redis on AWS – Creating a Cluster



▾ Advanced Redis settings

Advanced settings have common defaults set to give you the fastest way to get started. You can modify these now or after your cluster has been created.

Multi-AZ with Auto-Failover	✓
Subnet group	ameer-subnet1 (vpc-d54307a0)
Preferred availability zone(s)	● No preference ○ Select zones

Security

Security groups	default (sg-f3cf7097)
Encryption at-rest	☐
Encryption in-transit	☐

Import data to cluster

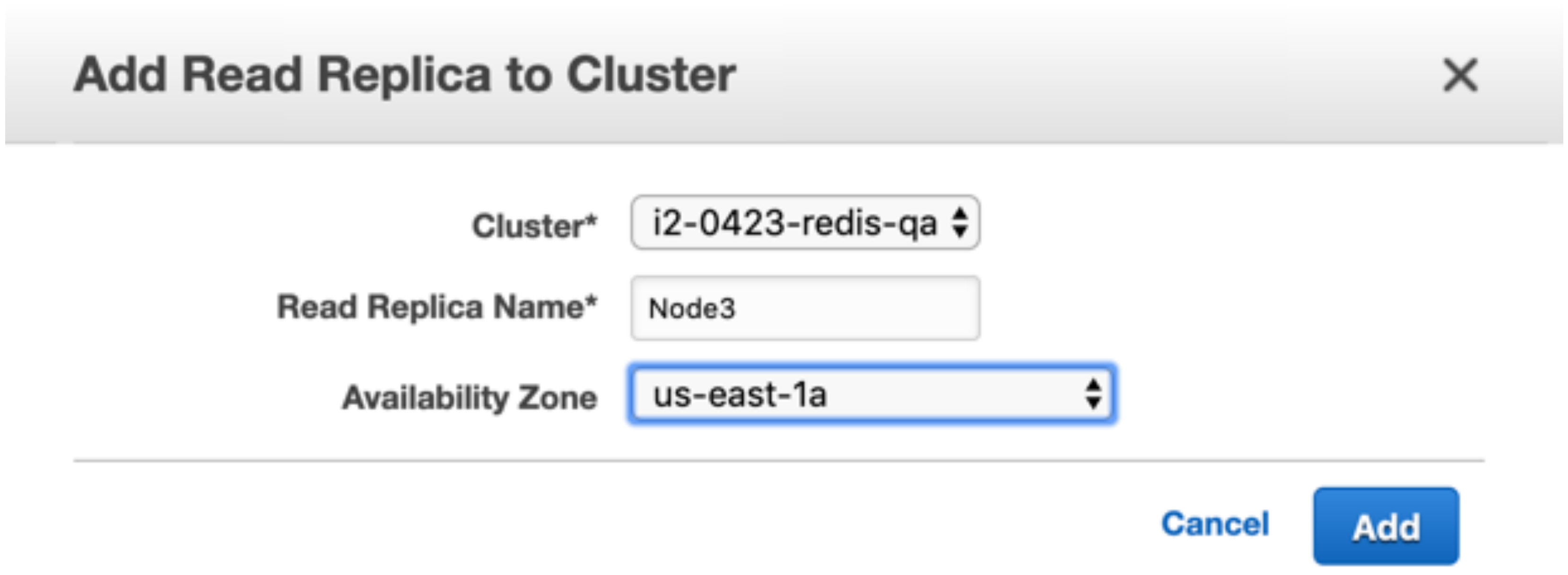
Seed RDB file S3 location	myBucket/myFolder/objectName

Use comma to separate multiple paths in the field

Backup

Enable automatic backups	✓
Backup retention period	1 day(s)
Backup window	● No preference ○ Specify backup window

Redis on AWS – Adding a Node



Add Read Replica to Cluster

Cluster*	i2-0423-redis-qa
Read Replica Name*	Node3
Availability Zone	us-east-1a

Redis on AWS - Read Replicas

- When to consider using a Redis read replica?
  - ➢ Scaling beyond the compute or I/O capacity of a single primary node for read-heavy workloads.
  - ➢ Data protection scenarios; in the unlikely event or primary node failure or that the Availability Zone in which your primary node resides becomes unavailable, you can promote a read replica in a different Availability Zone to become the new primary.

- In the event of a failover, any associated and available read replicas should automatically resume replication once failover has completed (acquiring updates from the newly promoted read replica).
- For read replicas, you should be aware of the potential for lag between a read replica and its primary cache node, or "inconsistency".

Redis on AWS – ElastiCache Failover

- What happens during failover and how long does it take?
  - ➢ ***ElastiCache flips the DNS record*** *for a cache node to point at the read replica, which is in turn promoted to become the new primary.*
  - ➢ *Start-to-finish, failover typically completes within sixty seconds.*

- Read replica may only be provisioned in the same Region as primary cache node.

Redis on AWS - Backup and Restore

- Backup and Restore is a feature that allows to create snapshots of ElastiCache for Redis clusters.
- ElastiCache stores the snapshots, allowing users to use them to restore Redis clusters.
- A snapshot is a copy of entire Redis cluster at a specific moment.

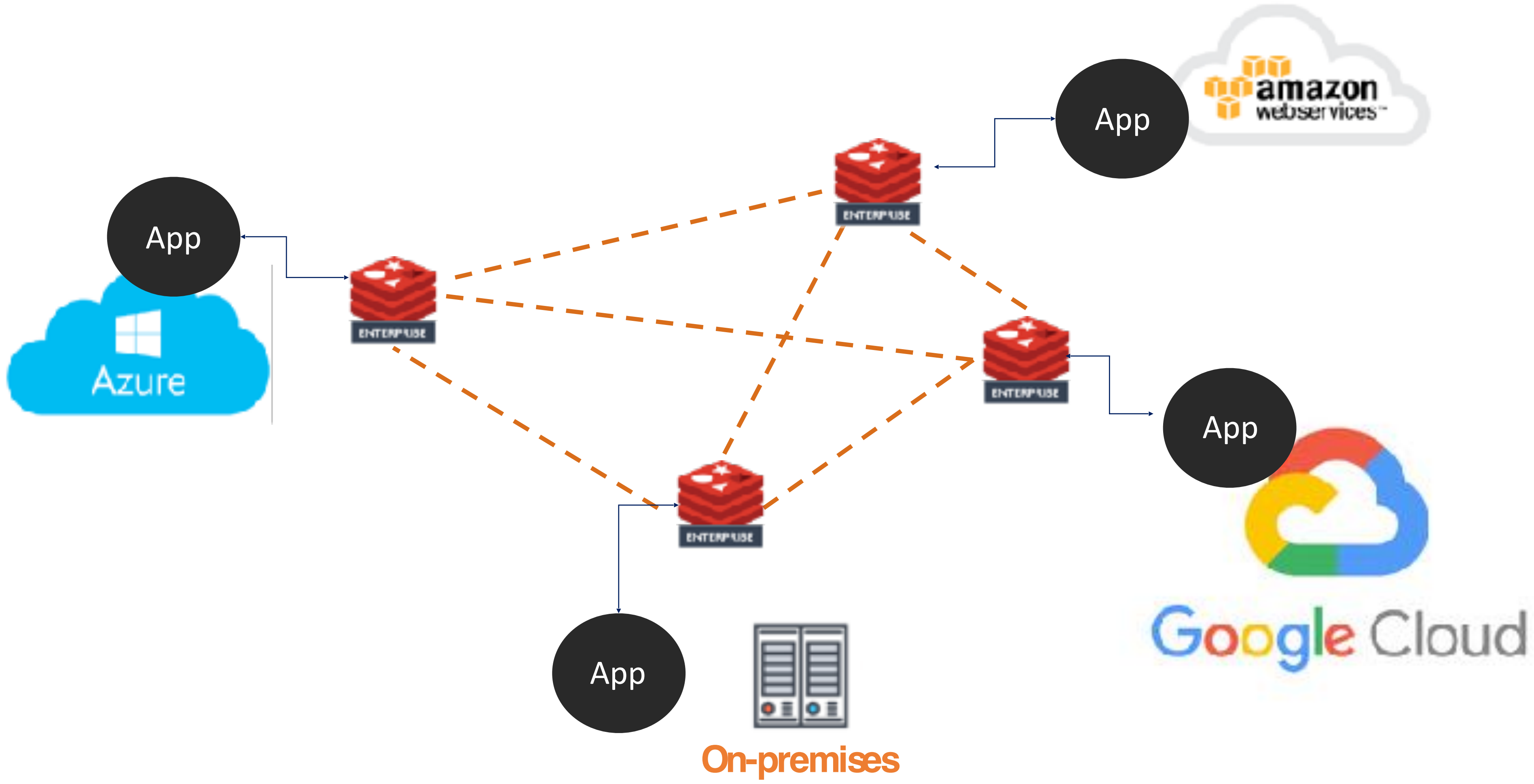
Redis on AWS – Encryption

- Encryption **in-transit** feature enables to encrypt all communications between clients and Redis server as well as between the Redis servers (primary and read replica nodes).

- Encryption at-rest allows for encryption of data during backups and restore - data backed up and restored on disk and via Amazon S3 is encrypted.

Multi-Cloud and Hybrid Cloud-OnPrem Support

Active-Active or Active-Passive

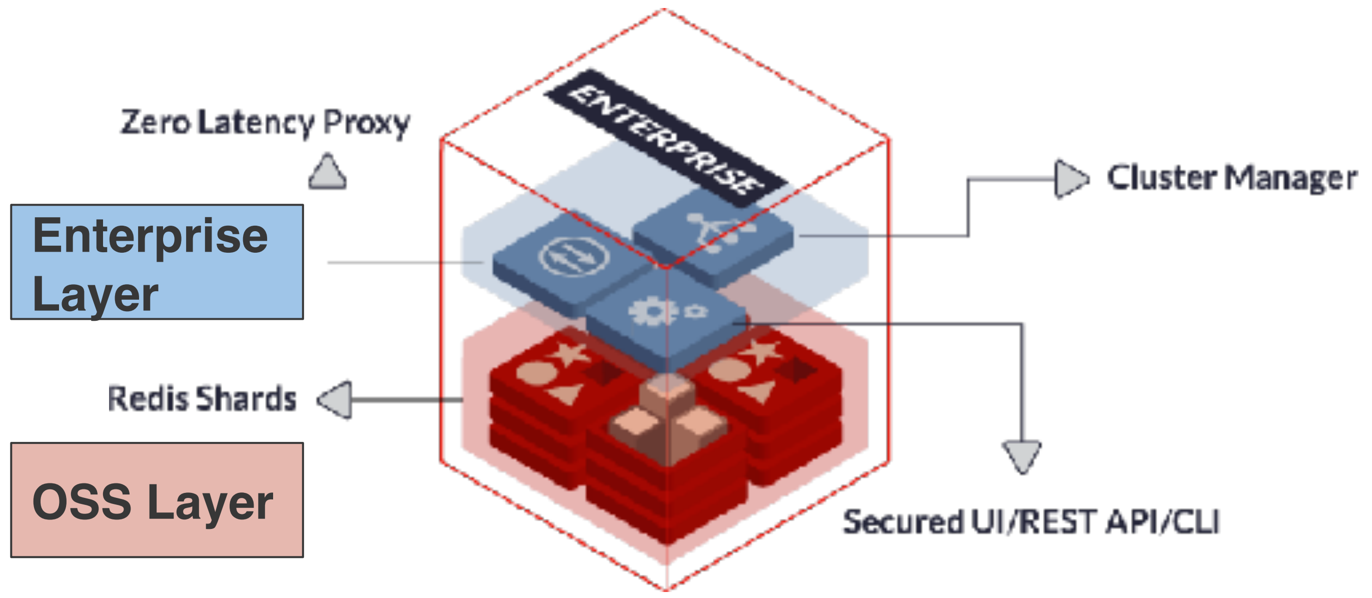


Covers Transactional, Operational and Real-Time Analytical Workloads

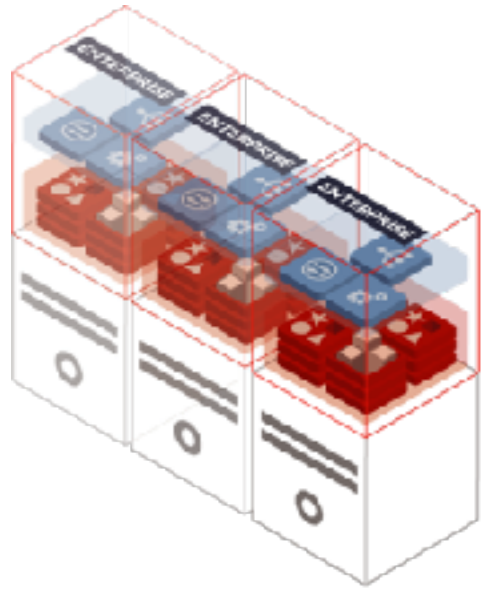
TRANSACTIONAL	ANALYTICS	OPERATIONAL
✓ Authorization	✓ Counting	✓ Accelerated Reporting
✓ Authentication	✓ Leaderboards	✓ Real-time Attribution
✓ Price Management	✓ Page Ranking	✓ Search
✓ Advertising Bids	✓ Recommendation Engine	✓ Order History
✓ Messaging	✓ Time-series Analysis	✓ Inventory Tracking
✓ Location-based Processing	✓ Session Analysis	
✓ User Session Management	✓ Secondary Index	

Redis Enterprise Technology

Redis Enterprise Node

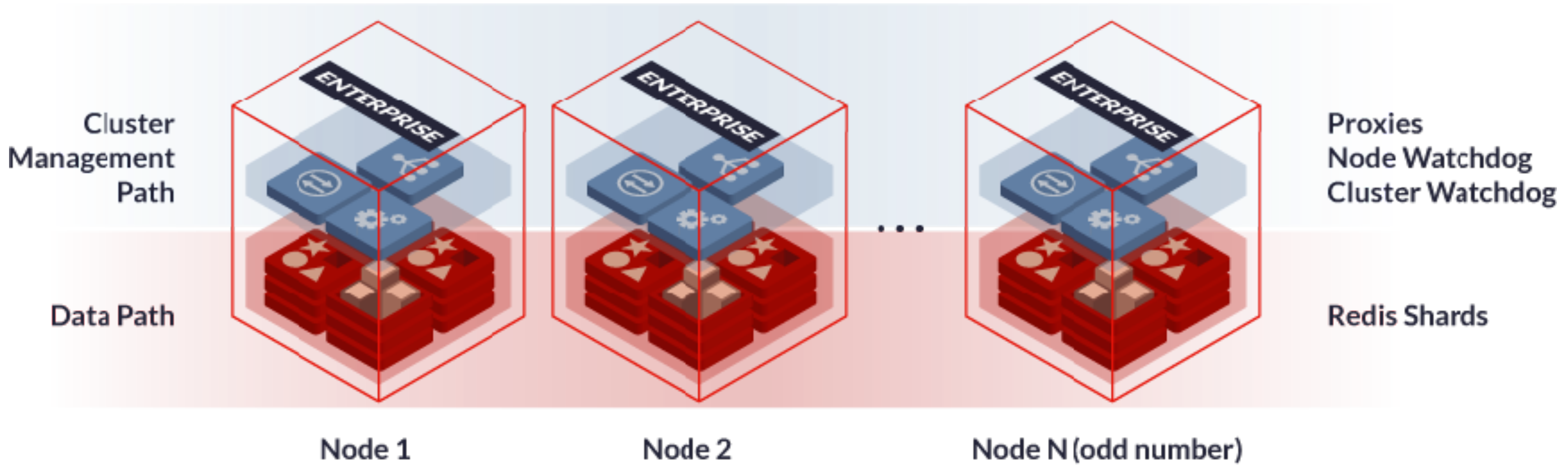


Redis Enterprise Cluster



- Shared nothing cluster architecture
- Fully compatible with open source commands & data structures

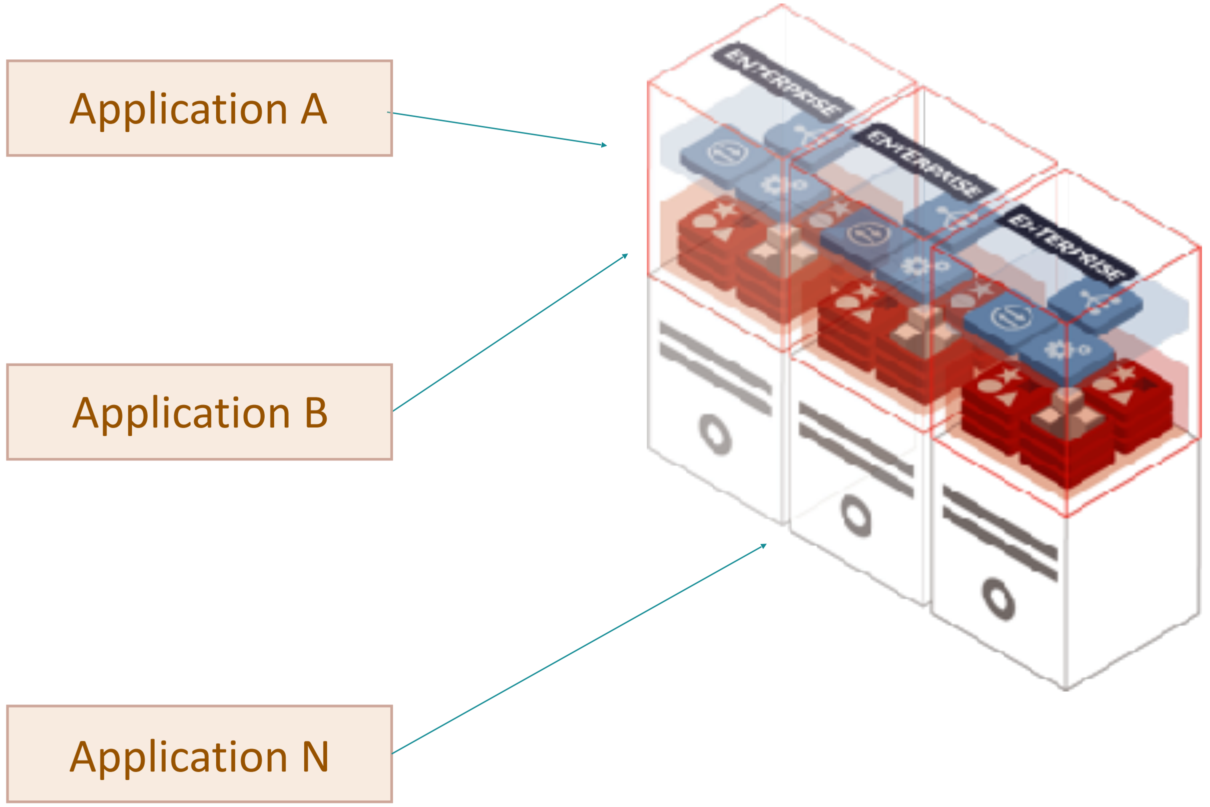
Redis Enterprise: Shared Nothing Symmetric Architecture



Unique multi-tenant container - like architecture enables running hundreds of databases over a single, average cloud instance without performance degradation and with maximum security provisions.

Redis Enterprise : Multi-Tenancy Maximizes Resource Utilization

200+ applications or shards on a single 4vcore cloud instance



- Shard isolation/protection
- Noisy-neighbor cancellation
- Minimizing CPU consumption of inactive applications

True Linear Scalability
Cluster Throughput (@ 1 msec Latency)

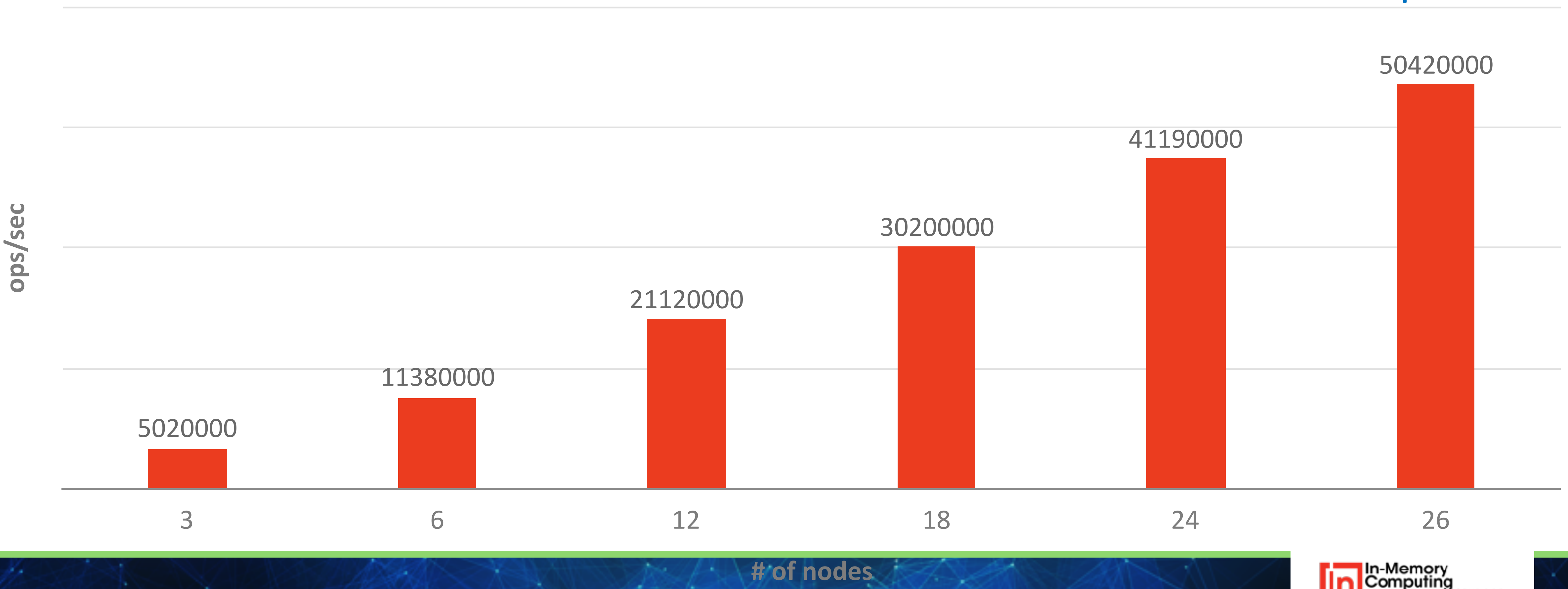
20M ops/sec

30M ops/sec

50M ops/sec

1.92M – per node

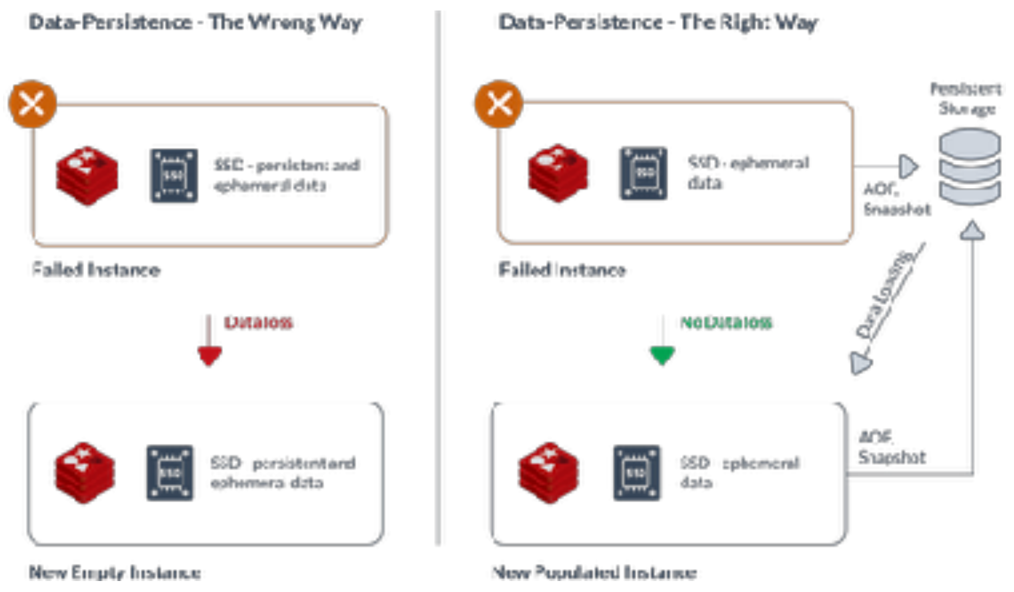
97.65K – per shard

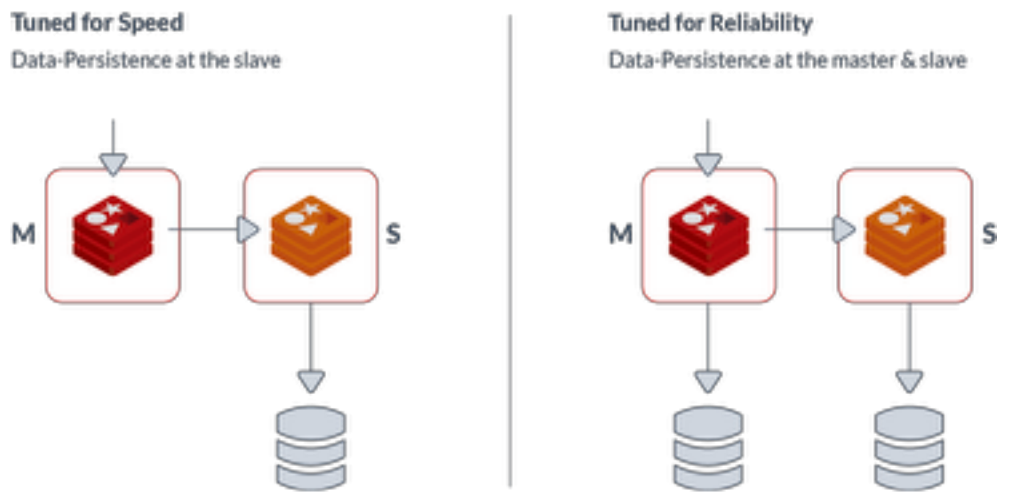


Redis Enterprise

Reduced Infrastructure	Up to 70% reduced Infrastructure Costs
Programmer Productivity	Programmer only has to worry about the connection to the ONE end point
Operational Maintenance	Automatic cluster and scale management

Durability At Memory Speeds

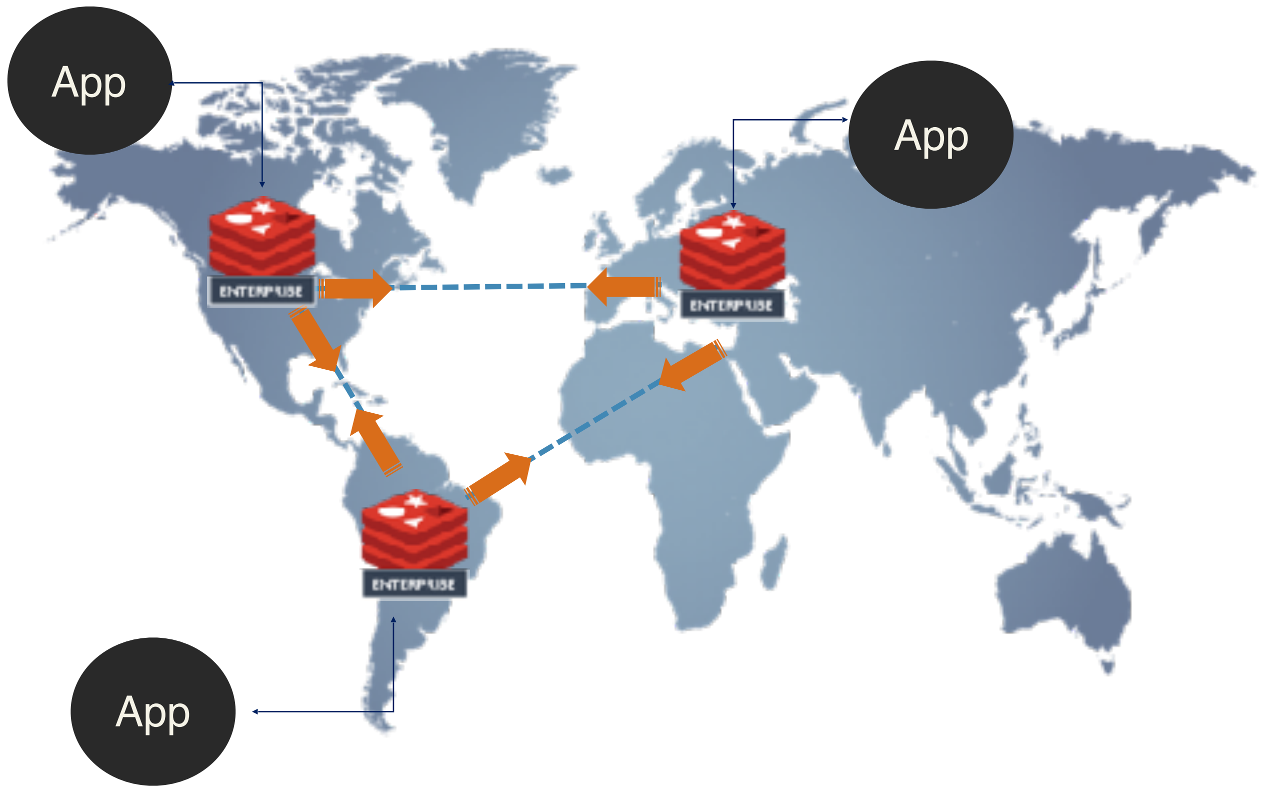



Tuned for Speed
Data-Persistence at the slave
M
S
Tuned for Reliability
Data-Persistence at the master & slave
M
S

- Multiple data persistence options (AOF, Snapshot)
- Every node in the cluster is connected to NAS making the cluster immune to data loss

- Enabling data persistence only at the slave-level for speed
- Delivering master and slave to be attached to storage for reliability

Active-Active Geo Distribution (CRDT-Based)



- Proven technology backed by deep academic research
  - Local latencies guaranteed with consensus free protocol
  - Built-in conflict resolution
  - Strong eventual consistency
- Multiple enhancements to make CRDTs fully Redis compatible (CRDB)

In-Memory Computing SUMMIT EUROPE 2019