

DS4300 - Large-Scale Storage and Retrieval

Redis and other Key-Value Stores



Key features of key-value databases

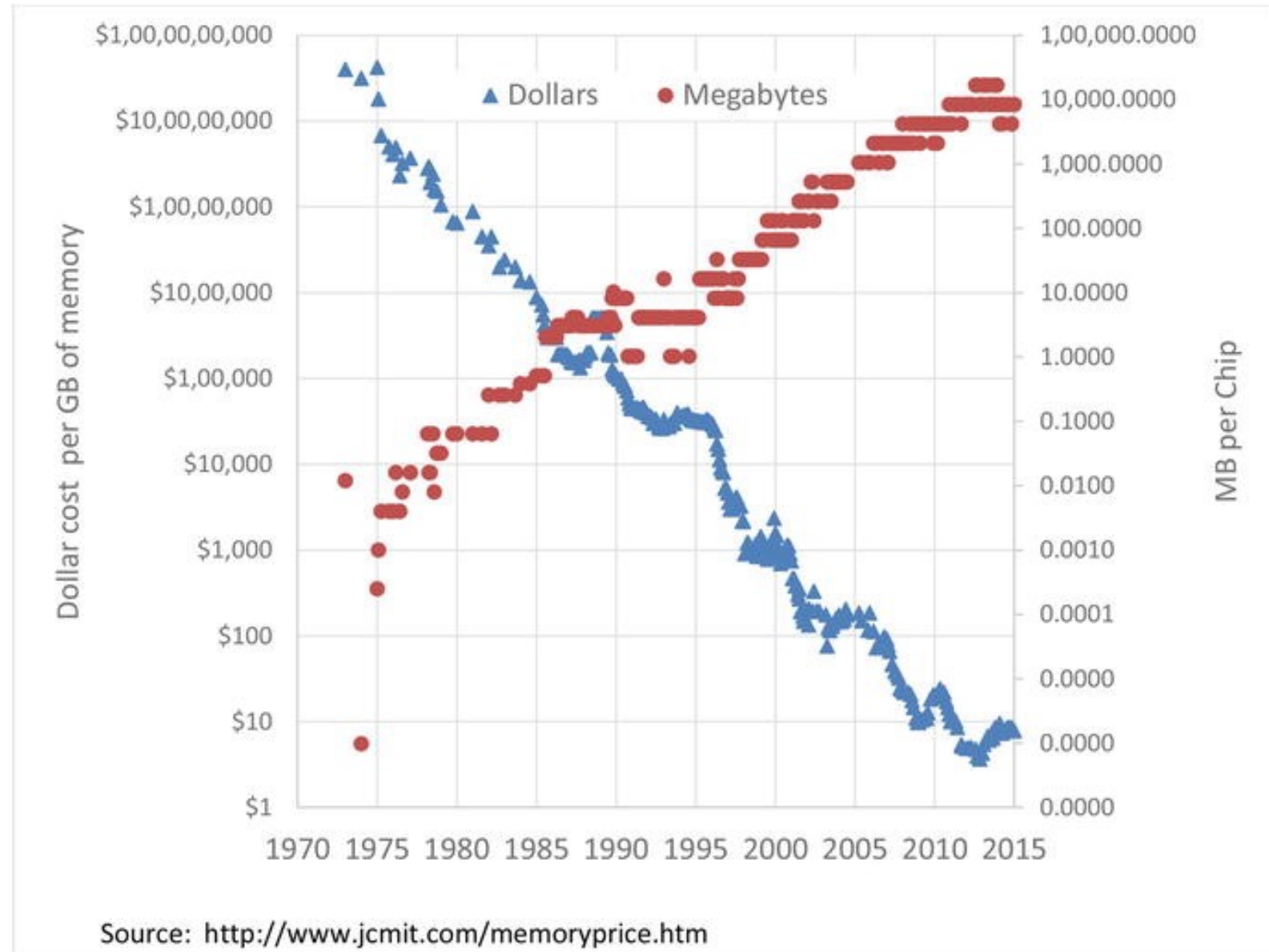
Key-Value Stores are designed around:

- Simplicity
- Speed
- Scalability



In-memory databases

- Cache-less architecture
- Alternative persistence models such:
 - as data replication (in a cluster)
 - Snapshots
 - Transaction logs (Append only files)



Key-Value Store Features

Consistency: Consistency guaranteed only on a single key.
Distributed stores are ***eventually consistent***.
Resolution of update conflicts varies.

Transactions: varies

Querying: All key-value stores query by **key** and usually only by key. Some KV stores support searching within values (Riak).

Data: blob, text, JSON, XML, etc.

Scaling: Sharding by key value



Key Value Use Cases

- **Storing Session Information:** Everything about the current session can be stored by a single PUT and retrieved with a single GET – making it very fast.
- **User Profiles, Preferences:** User information obtained with a single GET operation: language, timezone, product preferences, etc.
- **Shopping Cart Data:** Shopping carts are tied to users and need to be available across browsers, machines, and sessions.
- **Caching Layer:** In front of a disk-based database.



What is Redis?



- Redis (Remote Directory Server) is an open source **in-memory data structure store**
- A NoSQL Key-Value Store
- Uses include:
 - Database
 - Cache
 - Message Broker
- Unlike MongoDB, Redis is memory-based and thus super fast.



Real time analytics



High speed data ingest

Session storage



Application job management



High speed transactions



Message queues



In-app social functionality

Geo Search and Caching



Redis: distinguishing features

- Developed in 2009 by Salvatore Sanfilippo
- Fast! Supports 100,000+ SET operations per second
- **RE**mote **D**ictionary **S**ervice
- A rich collection of commands (> 100)
- Standard objects are Strings or String Collections (Lists, Sorted Lists, Hash Maps, etc.)
- Written in C++
- Clustering via Master-Slave
- Doesn't handle complex data well – only primary key lookups are supported (no secondary indexing method).



Redis persistence

Redis is primarily designed for in-memory caching but supports disk-based persistence mechanisms as well:

The **Snapshot**: Stores copies of the entire Redis system on request or at server shutdown.

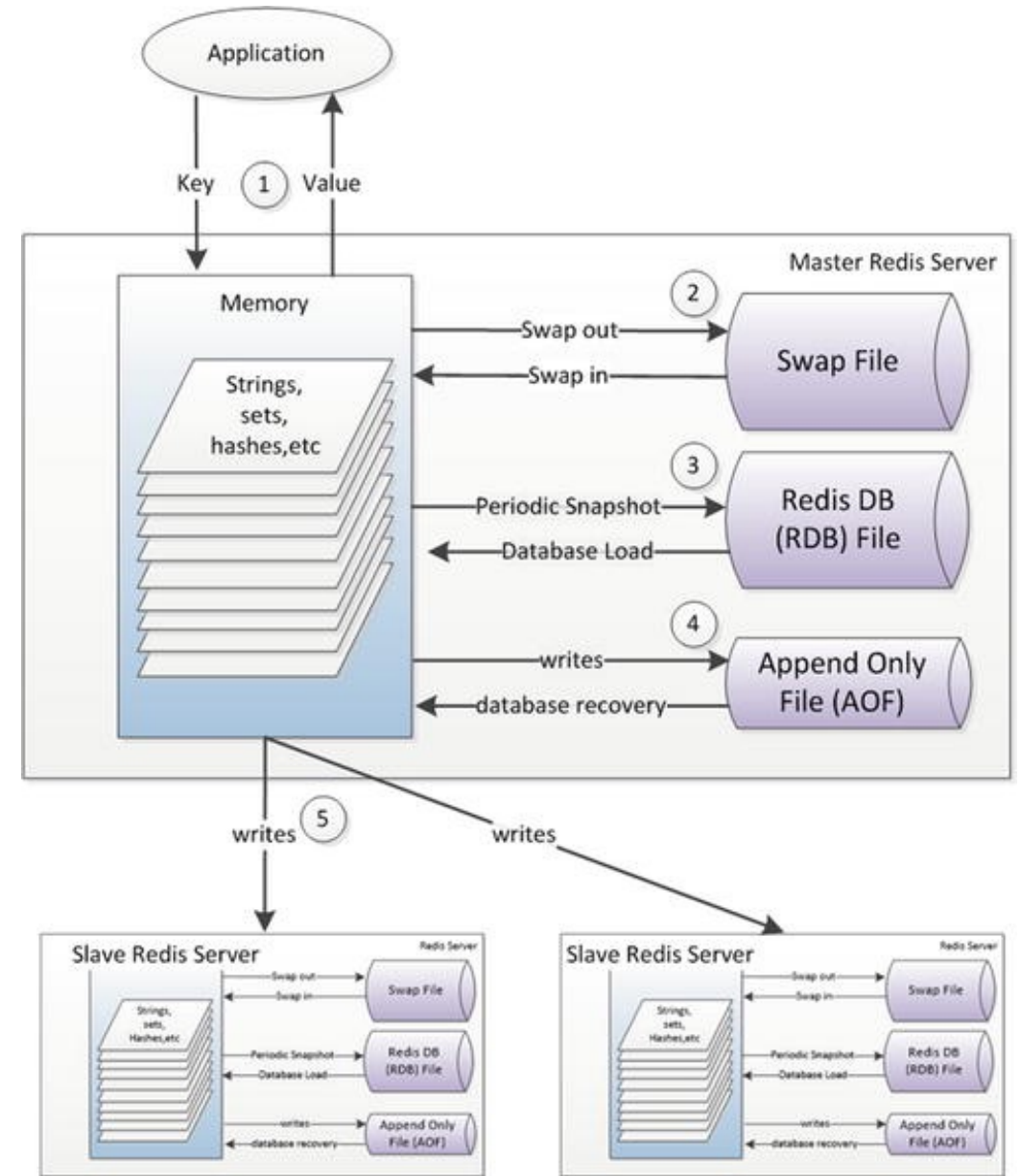
The **Append Only File (AOF)**: A journal of changes that can be used to "roll forward" from a snapshot in case of failure.

- After every write
- Every second
- OS flush intervals

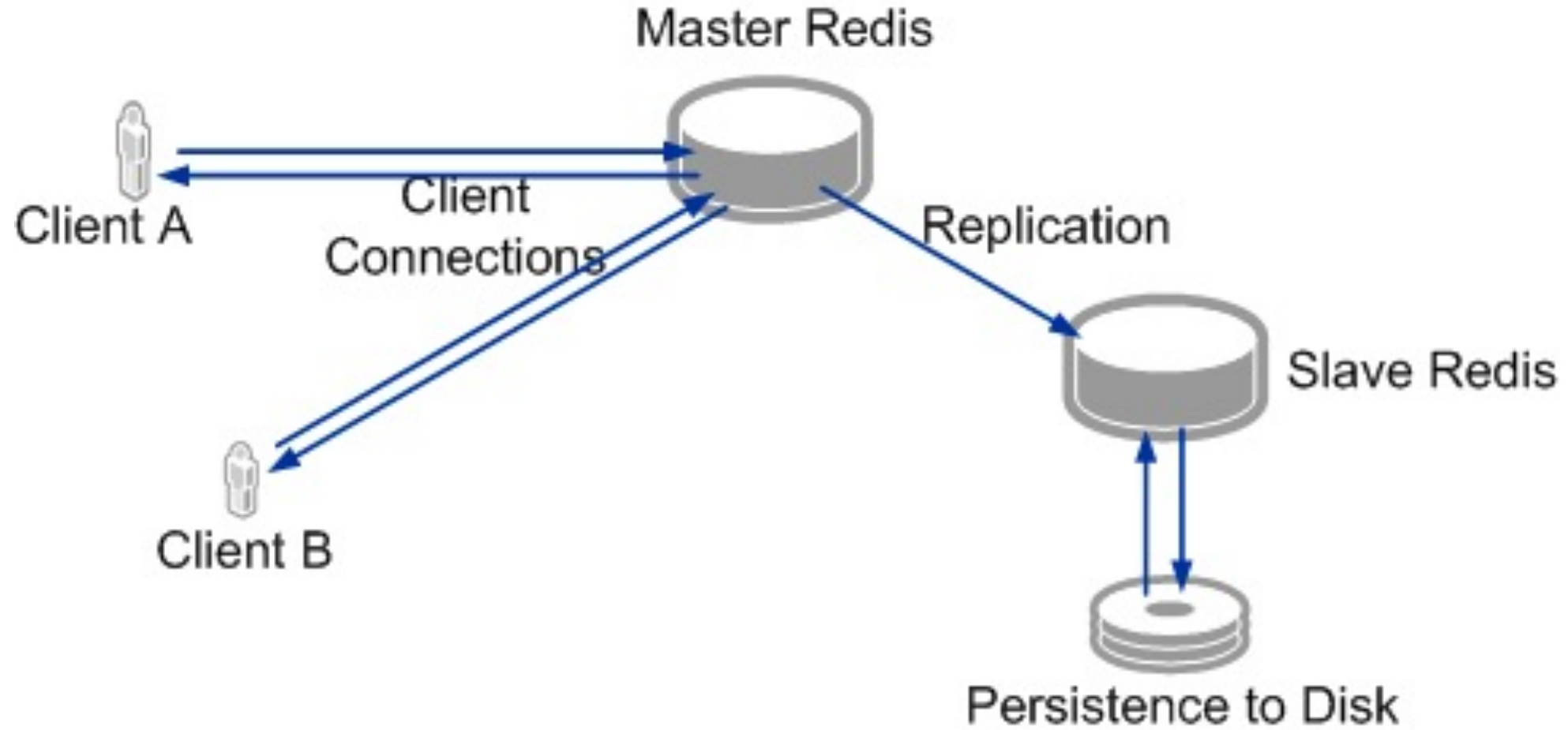


Master-Slave replication

- Performance critical and some data loss is acceptable.
- Replicas do backups, Masters service requests



Redis Replication



Redis vs. Memcached

- Memcached is another key-value NoSQL data store that also stores data in memory
- Memcached uses *volatile* cache: Data is not persisted
- Redis persists data to disk in the background (data saved on restart)



Datatypes: Redis is a “Data Structure Server”

Keys

- Usually Strings but can be any binary sequence (e.g., .JPG image)

Values

- Strings
- Lists (Linked Lists)
- Sets (Unique unsorted string elements)
- Sorted Sets – each element has a floating point “score”
- Hashes (String → String)
- Bit Arrays (Bit maps)
- HyperLogLogs: A probabilistic data structure used to estimate set cardinality.
- Geospatial Indexes



Scaling and Partitioning

- Scaling is not the easiest thing to do, but it has gotten better.
- If Redis is used as a cache, scaling up and down is easy.
As a data store, need a fixed key → node store, so the nodes must be fixed
- Partitioning (Sharding): Split data among multiple Redis instances so you have access to total memory of all machines.



Security Model

- Redis is optimized for performance not security
- Designed to be accessed by trusted clients inside a trusted environment



Redis Clients: <http://redis.io/clients>







ActionScript	Bash	C	C#	C++	Clojure
Common Lisp	Crystal	D	Dart	Delphi	Elixir
emacs lisp	Erlang	Fancy	gawk	GNU Prolog	Go
Haskell	Haxe	Io	Java	Julia	Lasso
Lua	Matlab	mruby	Nim	Node.js	Objective-C
OCaml	Pascal	Perl	PHP	PL/SQL	Pure Data
Python	R	Racket	Rebol	Ruby	Rust
Scala	Scheme	Smalltalk	Swift	Tcl	VB
VCL					

Yes!



Clients for Java

(Partial List – Starred clients are recommended)

Jedis	😊	★	🔗		
JRedis	😊		🏠	🔗	
lettuce	😊	★	🏠	🔗	Advanced Redis client for thread-safe sync, async, and reactive usage. Supports Cluster, Sentinel, Pipelining, and codecs.  
redis-protocol				🔗	Up to 2.6 compatible high-performance Java, Java w/Netty & Scala (finagle) client 
RedisClient	😊			🔗	redis client GUI tool
Redisson	😊	★		🔗	distributed and scalable Java data structures on top of Redis server 



Data Structure Server

- No schemas. No column names. But you do need to think about how data will be represented (hashes, sets, etc.)
- Keys → Data structures
- Speed. Speed. Speed.
Use appropriate structures for appropriate cases.



Redis vs. Relational

- In relational DBs you scan tables looking for matching rows or do lookups with an index.
- In redis, you perform direct data retrieval commands.
(There is no query engine.)
- You have complete control over how you represent and store your data.

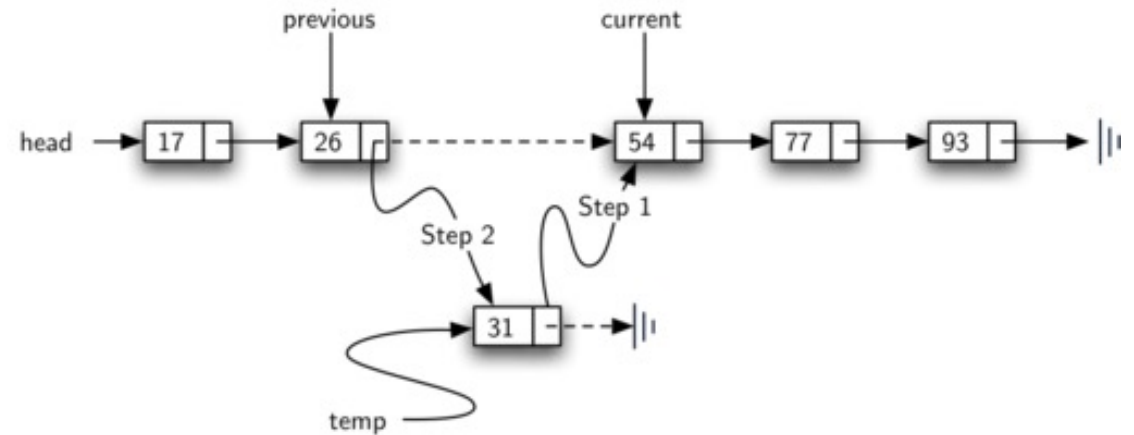


Data Retrieval

In Relational databases, data is stored in fixed tables and you have a standard query language (SQL) to insert, update, or select data.

In Redis, you have to think up-front about how your data will be stored and retrieved.

- Do values need keys?
- Multiple fields to one key
- Count elements under one collection?
- Storing objects
- Increment / Decrement Operations?
- Unique values → Set / Stored Set
- Existence testing?



Data Storage

- Different from SQL
- Only store the data you know you will use – save your memory!
- Targeted at explicit lookups / reads – not *ad hoc* discovery.



Security Model

- Designed to be accessed by trusted clients.
(Avoid opening up to Internet)
- Simple authentication with a password
 - Deny access to main Redis port / Firewall
- Authentication layer, enabled via redis.conf
User must send AUTH command along with the password
- Password is stored in plain text in redis.conf
- No data encryption!



Disabling & Renaming Commands


- Specific commands can be renamed or disabled via redis.conf
- Normal users should not have CONFIG access.

rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52

rename-command CONFIG ""



Windows: Installing Redis: <https://www.memurai.com>



MEMURAI
ULTRA-FAST DATA STORE

Redis for Windows alternative, In-Memory Datastore

Ready for the most demanding production workloads.
Free for development and testing.

DOWNLOAD FOR FREE

LEARN MORE

Current version: 2.0.3 - Redis API 5



Mac / Linux: Installing Redis: <https://redis.io>



redis

Commands

Clients

Documentation

Community

Download

Modules

Support

Try Free

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster. [Learn more →](#)

Try it

Ready for a test drive? Check this [interactive tutorial](#) that will walk you through the most important features of Redis.

Download it

[Redis 6.2.3 is the latest stable version.](#) Interested in release candidates or unstable versions? [Check the downloads page.](#)

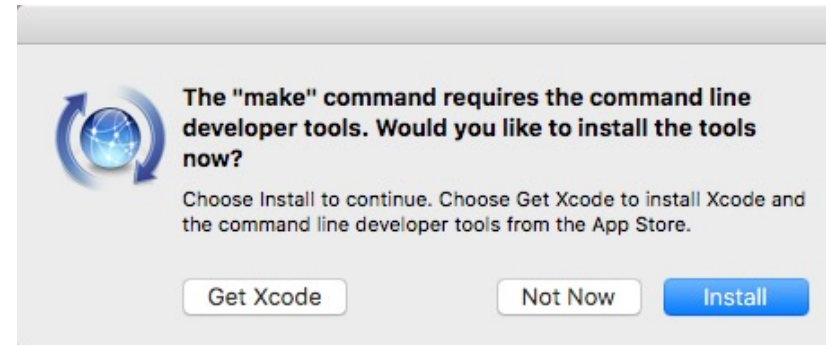
Quick links

Follow day-to-day Redis on [Twitter](#) and [GitHub](#). Get help or help others by subscribing to [our mailing list](#), we are 5,000 and counting!



Make redis (Mac & Linux)

```
[510 uranus:~ ] cd $APPS/redis  
[511 uranus:redis ] cp ~/Downloads/redis-6.2.3.tar.gz .  
[512 uranus:redis ] tar xzf redis-6.2.3.tar.gz  
[513 uranus:redis ] cd redis-6.2.3  
[514 uranus:redis-6.2.3 ] make
```



In my .bash_profile:

```
export REDIS_HOME=$APPS/redis/redis-6.2.3  
export PATH=$REDIS_HOME/src:$PATH
```



Installing Redis with homebrew (Mac)

1. Install homebrew <https://brew.sh/>
2. `$ brew install redis`



Running Redis

To run Redis with the default configuration just type:

```
$ cd $REDIS_HOME/src  
$ ./redis-server
```

If you want to provide your `redis.conf`, you have to run it using an additional parameter (the path of the configuration file):

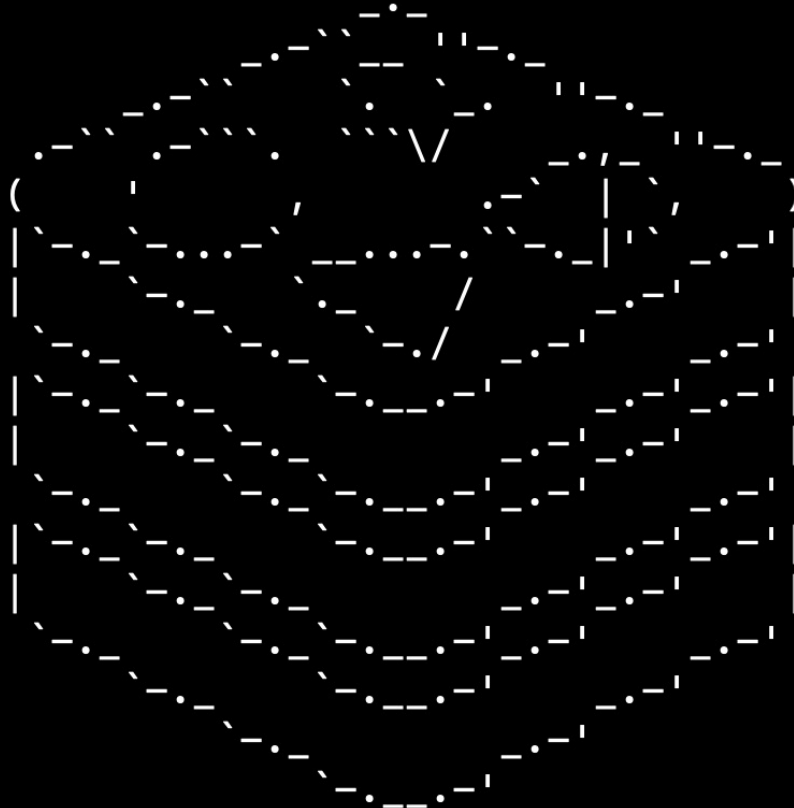
```
$ ./redis-server /path/to/redis.conf
```

It is possible to alter the Redis configuration by passing parameters directly as options using the command line. Examples:

```
$ ./redis-server --port 9999 --slaveof 127.0.0.1 6379  
$ ./redis-server /etc/redis/6379.conf --loglevel debug
```



Running Redis Server: \$ redis-server



```
Redis 6.2.3 (22f17221/1) 64 bit

Running in standalone mode
Port: 6379
PID: 34616

https://redis.io

34616:M 17 May 2021 09:46:20.301 # Server initialized
34616:M 17 May 2021 09:46:20.301 * Loading RDB produced by version 6.2.3
34616:M 17 May 2021 09:46:20.301 * RDB age 3 seconds
34616:M 17 May 2021 09:46:20.301 * RDB memory usage when created 0.98 Mb
34616:M 17 May 2021 09:46:20.301 * DB loaded from disk: 0.000 seconds
34616:M 17 May 2021 09:46:20.301 * Ready to accept connections
```



Running Redis client: redis-cli

```
[525 09:41:21 rachlin:~ ] redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> get foo
"bar"
127.0.0.1:6379> incr mycounter
(integer) 1
127.0.0.1:6379> incr mycounter
(integer) 2
127.0.0.1:6379>
```



DEMO

Redis CLI Basics



Redis Command Help: <https://redis.io/commands>

Filter by group: ☒ All or search for:

- Cluster
- Connection
- Geo
- Hashes
- HyperLogLog
- Keys**
- Lists
- Pub/Sub
- Scripting
- Server
- Sets
- Sorted Sets
- Strings
- Transactions

APPEND key value
Append a value to a key

BGREWRITEAOF
Asynchronously rewrite the append-only file

BGSAVE
Asynchronously save the dataset to disk

BITCOUNT key [start end]
Count set bits in a string

BITFIELD key [GET type offset]...
Perform arbitrary bitfield integer operations on strings



Redis Help Example: Definitions, Examples, Use-Cases

INCR key

Available since 1.0.0.

Time complexity: $O(1)$

Increments the number stored at key by one. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer. This operation is limited to 64 bit signed integers.

Note: this is a string operation because Redis does not have a dedicated integer type. The string stored at the key is interpreted as a base-10 **64 bit signed integer** to execute the operation.

Redis stores integers in their integer representation, so for string values that actually hold an integer, there is no overhead for storing the string representation of the integer.

Return value

Integer reply: the value of key after the increment

Examples

```
redis> SET mykey "10"
"OK"
redis> INCR mykey
(integer) 11
redis> GET mykey
"11"
redis>
```

Pattern: Counter

The counter pattern is the most obvious thing you can do with Redis atomic increment operations. The idea is simply send an **INCR** command to Redis every time an operation occurs. For instance in a web application we may want to know how many page views this user did every day of the year.

To do so the web application may simply increment a key every time the user performs a page view, creating the key name concatenating the User ID and a string representing the current date.

This simple pattern can be extended in many ways:



INCR is an atomic operation

Client A:	Client B:	Value of X
X = GET count		10
X = X + 1	X = GET count	10
SET count X	X = X + 1	11
	SET count X	11

This problem is avoided.



Redis Keys: Guidelines

- Very long keys are not a good idea. For instance a key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons.
- Very short keys are often not a good idea.
"user:1000:followers" better than "u1000flw"
- Try to stick with a schema. For instance "object-type:id" is a good idea, as in "user:1000". Dots or dashes are often used for multi-word fields, as in "comment:1234:reply.to" or "comment:1234:reply-to"



- The maximum allowed key size is 512 MB.

Basic Commands

- SET – Sets a string variable
- GET – Gets a value of a variable
- DEL – Deletes a variable
- EXISTS – Check to see if exists
- INCR – Increments by 1
- INCRBY – Increments by specified amount
- DECR – Decrease by 1
- DECRBY – Decrease by specified amount



MSET

- Sets multiple keys to respective values
- Replaces existing values with new ones

MSET key1 “val1” key2 “val2”



MSETNX

- Sets multiple keys to respective values as long as none of the keys exist
- Will NOT overwrite existing values
- Will NOT perform if even a single key already exists

MSETNX key1 “val1” key2 “val2”



MGET

- Returns values of all specified keys
- Nil is returned if key doesn't hold a value

MGET key1 key2



APPEND

- If key already exists and is a string, the value will be appended at the end of the string
- If the key does NOT exist. It works as SET

APPEND mykey “stringtoappend”



GETRANGE

- Returns the substring of a string value
- Determined by offsets start and end
- Negative offsets can be used to start from the end of the string

GETRANGE mykey 0 -1



RENAME

- Renames a key
- Returns error if key doesn't exist
- If does exist, it is overwritten

RENAME mykey myrenamedkey



RENAMENX

- Renames key to newkey if newkey does not exist
- Returns an error if key does not exist

RENAMENX mykey myrenamedkey



GETSET

- Automatically sets key to value and returns the old value
- Returns an error when key exists
- Can be used with INCR for counting with automatic reset

GETSET mykey “myval”



SETEX

- Set key to hold a string value and timeout after a given amount of seconds

SETEX mykey 10 “hello”

Equivalent to...

SET mykey “hello”

EXPIRE mykey 10



PSETEX

- Same as SETEX except it uses milliseconds instead of seconds

PSETEX mykey 1000 “hello”

PTTL is used to get the remaining time in milliseconds



PERSIST

- Removes existing timeout on a key

PERSIST mykey



SETNX

- Works like SET if the key does NOT exist
- If the key already exists, it will not change

SETNX mykey “hello”



SCAN: Looping over keys

- Iterates the set of keys in the database
- Returns only a small amount per call
- Can be used in production (high performance)
- Takes cursor / position as a parameter



Cursor Based Iterator

- The server returns an updated cursor with every call of the command
- This can be used in the argument of the next call
- Iteration starts when cursor is set to 0
- Terminates when cursor returned from the server is 0

```
redis 127.0.0.1:6379> scan 0
```

```
1) "17"  
2) 1) "key:12"  
   2) "key:8"  
   3) "key:4"  
   4) "key:14"  
   5) "key:16"  
   6) "key:17"  
   7) "key:15"  
   8) "key:10"  
   9) "key:3"  
  10) "key:7"  
  11) "key:1"
```

```
redis 127.0.0.1:6379> scan 17
```

```
1) "0"  
2) 1) "key:5"  
   2) "key:18"  
   3) "key:0"  
   4) "key:2"  
   5) "key:19"  
   6) "key:13"  
   7) "key:6"  
   8) "key:9"  
   9) "key:11"
```

SCAN 0



SCAN Guarantees

- Full iterations will retrieve all elements that were present in the collection from the start to the end
- Never returns any element that was NOT present in the collection from the start to finish



COUNT Option

- COUNT can be defined in a SCAN command to overwrite the default returned per iteration
- The user can specify the amount of work done at every call
- Default COUNT is 10
- COUNT can be changed from one iteration to the next

SCAN COUNT 20



MATCH Option

- Iterate elements that match a pattern specified

SCAN 0 MATCH something

SCAN 0 MATCH k*



Scan With Other Data Types

SSCAN – Used with sets. Returns list of set members

HSCAN – Used with hashes. Returns array of elements with a field and value

ZSCAN – Used with sorted sets. Returns array of elements with associated score



KEYS Pattern

- Returns all keys that match a specific pattern
- Should be avoided in production environments

KEYS *

Supported glob-style patterns:

- `h?llo` matches `hello`, `hallo` and `hxllo`
- `h*llo` matches `hllo` and `heeeello`
- `h[ae]llo` matches `hello` and `hallo`, but not `hillo`
- `h[^e]llo` matches `hallo`, `hblllo`, ... but not `hello`
- `h[a-b]llo` matches `hallo` and `hblllo`

Use `\` to escape special characters if you want to match them verbatim.



RANDOMKEY

- Returns a random key from the database

RANDOMKEY



CONFIG GET

- Used to read the configuration parameters of a running Redis server
- Since Redis 2.6, all configuration parameters are supported
- Takes single argument

CONFIG GET port – Gets the port configuration value

CONFIG GET * - List all supported config params

CONFIG GET *max-*-entries* - hash-max-zipmap-entries, list-max-ziplist-entries,



CONFIG SET

- Used to reconfigure server at runtime without having to do a restart

CONFIG SET configoption “newvalue”



INFO

- Returns information and statistics about a server
- Optional parameter to select specific section of information

**server | clients | memory | persistence | stats | replication | cpu
commandstats | cluster | keyspace | all | default**

INFO

INFO server



CONFIG RESETSTAT

Resets statistics reported using the INFO command

Keyspace hits

Keyspace misses

Number of commands processed

Number of connections received

Number of expired keys

Number of rejected connections

CONFIG RESETSTAT



COMMAND

Returns details about all Redis commands Each top level result contains 6 sub-results...

- Command name
- Command arity – Number of params
- Nested array reply of command flags
- Position of first key in arg list
- Position of last key in arg list
- Step count for location or repeating keys

COMMAND



COMMAND INFO

- Returns details for a specific command
- Same as COMMAND but focus on specific command

COMMAND INFO GET



COMMAND COUNT

- Returns the number of available commands on the server

COMMAND COUNT



CLIENT LIST

Returns info and stats on the clients connected to a server

- ID
- Name
- Flags
- Age
- Address/Port
- Last Command
- Idle Time

CLIENT LIST



CLIENT SETNAME

- Assigns a name to a current client connection
- Displayed in the output of CLIENT LIST

CLIENT SETNAME clientname



CLIENT GETNAME

- Returns the name of the current client connection
- null reply if no name is set

CLIENT GETNAME



CLIENT KILL

- Closes up a given connection
- Can use address/port or ID

CLIENT KILL 127.0.0.1:portnum

CLIENT KILL id



What Is a List?

- Lists are groups of strings (1 dimensional)
- Sorted by insertion order
- Elements can be pushed on the head or tail of the list
- Often used as producer/consumer queues



Inserting Elements

LPUSH Inserts a new element on the head (left)

RPUSH Inserts a new element on the tail (right)

A new list is created when LPUSH or RPUSH is ran against an empty key

The key is removed from the keyspace if a list operation will empty the list



LPUSH/RPUSH Example

LPUSH mylist a

“a”

LPUSH mylist b

“b”, “a”

RPUSH mylist “c”

“b”, “a”, “c”



LRange

- Returns specified elements of the list
- Offsets are zero-based indexes
- Offsets can be negative indicating offsets starting from the end of the list

`["Eric","Shawn","Jose"]`

LRange friends 0 -1

- 1) Eric
- 2) Shawn
- 3) Jose

LRange friends 1 2

- 1) Shawn
- 2) Jose



LLEN

- LLEN returns the length of the list

LLEN friends

3

If list is empty, 0 is returned



LPOP

Removes and returns the first element of a list

LPOP friends

Eric



RPOP

Removes and returns the last element of a list

RPOP friends

Jose



What Is a Set?

- Unordered collection of strings
- Can add, remove and test for existence
- Do NOT allow repeating members
- Support server side commands to compute sets
starting from existing sets



SADD & SREM

SADD

Adds given values to a set

Values that already exist will be ignored

SREM

Removes values from a set

SADD carmakes “Toyota” // Adds Toyota to carmakes set

SREM carmakes “Honda” // Removes Honda from carmakes set



SISMEMBER

SISMEMBER

Tests if the given value is in the set

Returns 1 if the value is there and 0 if it is not

SISMEMBER carmakes "Toyota"



SMEMBERS

SMEMBERS

Returns a list of all of the members of a set

SMEMBERS carmakes



SCARD

SCARD

Returns the count of members / elements in a set

Returns 0 if key does not exist

SCARD carmakes



SMOVE

SMOVE

Moves member from one set to another

SMOVE people users “John Doe”



SUNION

SUNION

Combines two or more sets and returns a list of members

SUNION carmakes truckmakes



SDIFF

SDIFF

Returns the members of the set resulting from the difference between the first and all successive sets

Keys that do not exist are considered empty sets

SDIFF key1 key2



SRANDMEMBER

SRANDMEMBER

Returns a random member of a set

Optional parameter to return a specified count

SRANDMEMBER carmakes

SRANDMEMBER carmakes 3



SPOP

SPOP

Removes and returns a random member from a specified set

Like SRANDMEMBER, a second parameter is allowed to specify a count of members

SPOP carmakes

SPOP carmakes 3



Sorted Sets

- Since sets are unsorted, they can pose problems for some projects
- Sorted Sets were created to solve that issue
- Every member is associated with a “score”
- Can access data very quickly
- Like sets, elements may only appear once



Score Properties

- Score is required
- Must be a float / number
- Score of 500 = 500.0
- Score is NOT unique, Values are



ZADD & ZREM

ZADD

Adds given values to a sorted set

ZREM

Removes values from a sorted set

ZADD people 1960 “John Doe”

ZREM people “John Doe”



ZRANGE

ZRANGE

Works like LRANGE for lists. Fetches values within a specified range. Ordered lowest to highest by score

ZREVRANGE

Same as ZRANGE except ordered highest to lowest

ZRANGE people 0 -1

ZRANGE people 2 4

ZREVRANGE 0 -1



ZRANGEBYSCORE

ZRANGEBYSCORE

Works like ZRANGE but uses range of score values

ZRANGEBYSCORE people 1950 1990

Gets people with a score between 1950 and 1990



ZRANK

ZRANK

Returns the rank of a member with scores ordered from high to low

Rank is 0-based

ZREVRANK

Gets the rank of a member in the reverse order

ZRANK people “John Doe”

ZREVRANK people “John Doe”



ZCARD & ZCOUNT

ZCARD

Returns the number of members in the sorted set

ZCARD people

ZCOUNT

Returns the number of elements in the sorted set at key with a score between min and max.

ZCOUNT people(1, 3)



ZINCRBY

ZINCRBY

- Increments the score of member in the sorted set
- If member does not exist, it will be added with increment as its score
- The score value should be the string representation of a numeric value, and accepts double precision floating point numbers. It is possible to provide a negative value to decrement the score.

ZINCRBY people 1 “John Doe”



ZSCORE

ZSCORE

Returns the score of a member

If member does not exist nil is returned

ZSCORE people “John Doe”



What Is a Hash?

- Maps between string fields and string values
- Perfect for representing objects (Similar to JSON objects)
- Very compact
- Can hold up to 4 billion key/value pairs



HSET

HSET

Sets a field in the hash

If a key does not exist, a new key holding a hash is created

If field exists in the hash, it is overwritten

1 is returned if field is a new field in the hash and value was set

0 is returned if the field already exists in the hash and the value was updated

HSET user1 name “John”



HMSET

HMSET

Sets multiple fields to their respective values

Overwrites any existing fields in the hash

HMSET user2 name "Jill" email "jill@gmail.com" age "25"



HGET

HGET

Gets a value associated with a field in a hash

Returns value or nil if the field is not present

HGET user1 name



HMGET

HMGET

Returns values associated with multiple fields in a hash

For every field that does not exist, a nil value is returned

HMGET user1 name age



HGETALL

HGETALL

Gets all fields and values in a hash

Returns every field name followed by its value

HGETALL users



HDEL

HDEL

Removes the specified fields from the hash

Specified fields that do not exist are ignored

Returns the number of fields that were removed from the hash

HDEL user2 age



HEXISTS

HEXISTS

Check for an existing field in a hash

Returns 1 if the hash contains the field and 0 if it does not

HEXISTS name user3



HINCRBY

HINCRBY

Increments the number sorted in the hash

If key does not exist, a new key holding a hash is created

If field does not exist the value is set to 0 before operation is performed

HINCRBY user3 age 1



HKEYS

HKEYS

Returns all field names in the hash

HKEYS user1



HLEN

HLEN

Returns the number of fields contained in the hash

Returns 0 if key does not exist

HLEN user1



HVALS

HVALS

Returns all values in a hash

HVALS user1



HSTRLEN

HSTRLEN

Returns the string length of the value associated with the field in the hash

If the key or field do not exist, 0 is returned

HSTRLEN user1 name



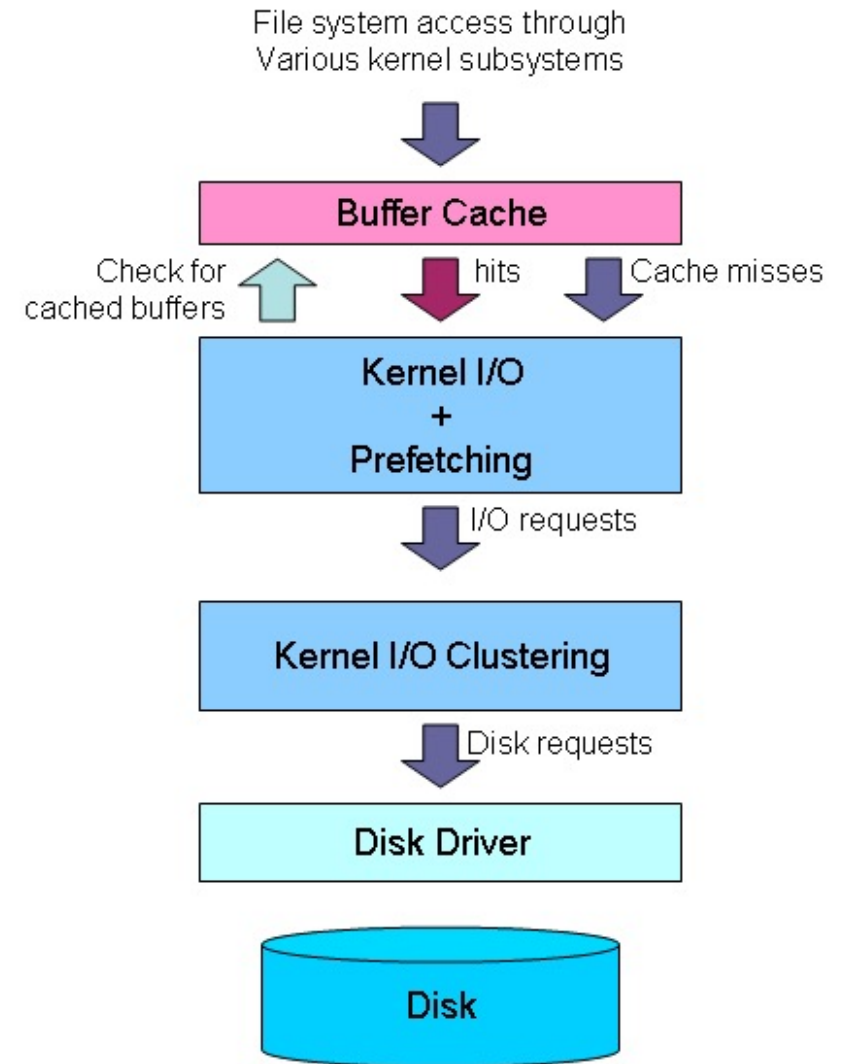
Data Persistence Features

- Datasets are all stored in memory
- Datasets can be saved to disk
- Redis fork – Creating child processes which are an exact copy of the parent
- Copy-On-Write Snapshot



Persistence Process

1. Client sends write command to database (Client memory)
2. Database receives the write (Server memory)
3. Database calls system call that writes data on disk (Kernel buffer)
4. The OS transfers the write buffer to the disk controller (Disk cache)
5. Disk controller writes to physical media (Physical Disk)



Pools

Multiple Redis servers running on the same machine using different ports

- Efficient memory usage
- More CPUs used (one server per CPU)
- Better fine-tuning



Replication

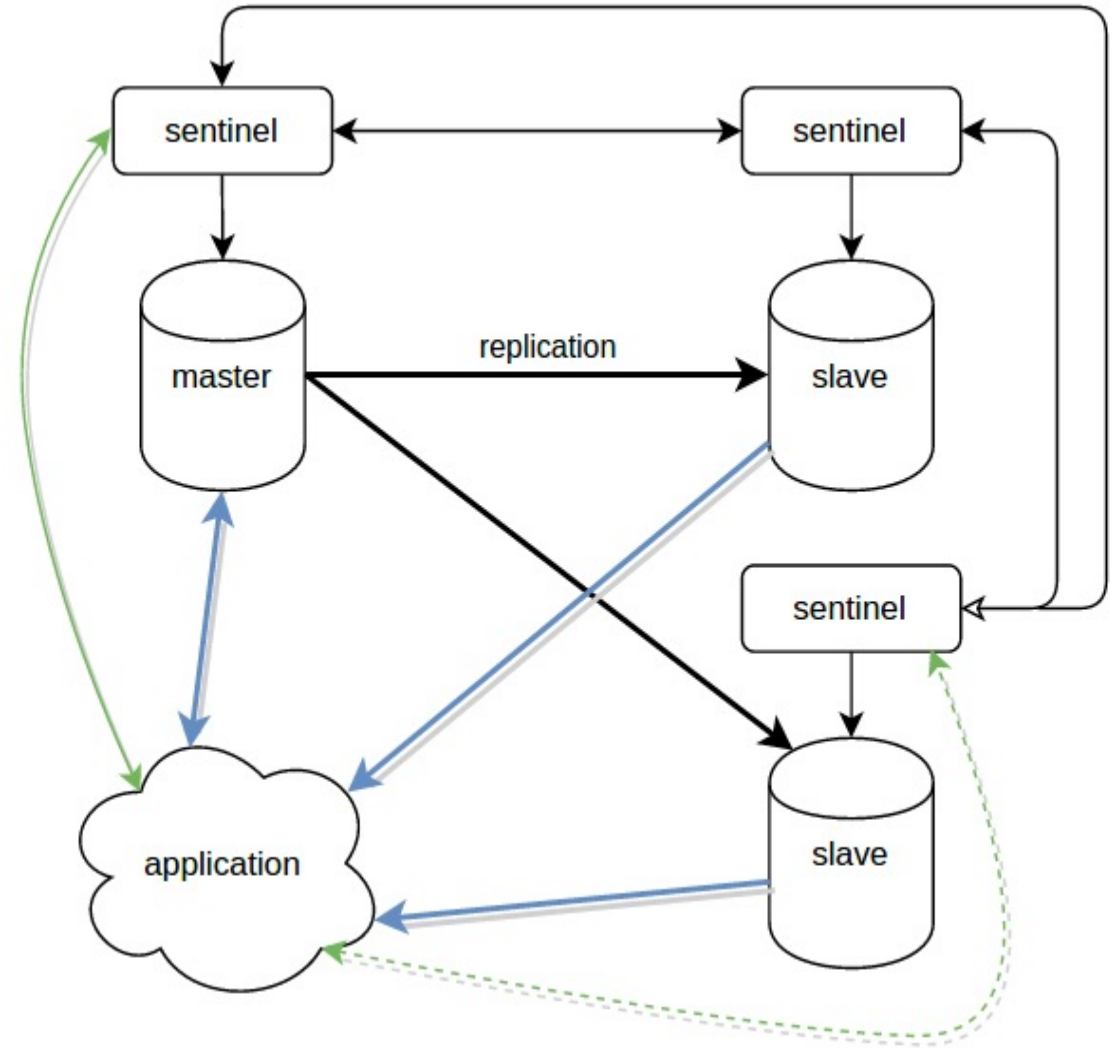
Simple master-slave replication allows slave Redis servers to be copies of master servers

- Asynchronous replication
- Multiple slaves
- Connections from other slaves
- Non blocking on slave side
- Scalability & data redundancy
- Slave read-only



Replication Process

1. Master starts saving in the background and starts buffering all new commands that will modify the dataset
2. After background saving, the master transfers the database file to the slave
3. The slave saves the files to the disk and loads it into memory
4. The master sends all buffered commands to the slave



Persistence Options

- RDB – Point-in-time snapshots
- AOF – Write operation logging
- Disabled
- Both RDB & AOF



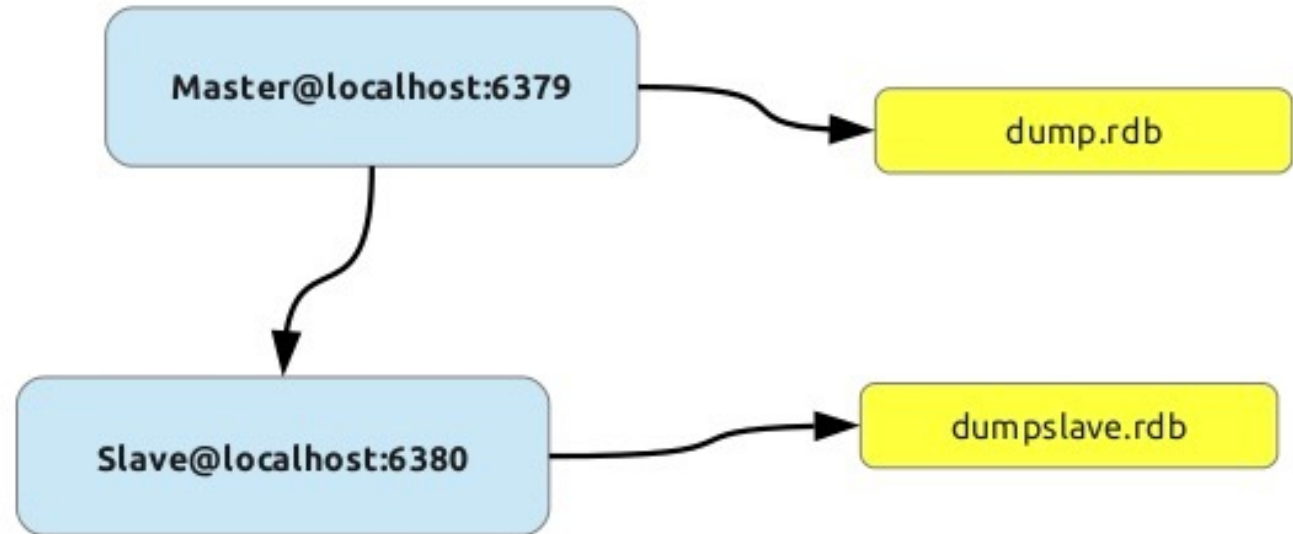
RDB – Redis Database File

- Simplest persistence mode
- Enabled by default
- Single-file point-in-time representation
- Uses snapshots



Master / Slave DB replication:

```
$> redis-server --port 6380 --slaveof localhost 6379  
--dbfilename dumpslave.rdb
```



RDB Advantages

- Easy to use
- Very compact
- Perfect for backup & recovery
- Maximizes Redis performance
- Allows faster restarts with big datasets compared to AOF



Snapshotting

- Controlled by the user
- Can be modified at runtime
- Snapshots are produced as .rdb files
- **SAVE & BGSAVE** Commands



SAVE & BGSAVE

SAVE

- Performs a synchronous save of the dataset producing a point-in-time snapshot of all data inside of the Redis instance in the form of an .rdb file
- Should not be called in production environments as it will block all other clients. Instead, use BGSAVE

SAVE 60

Dump dataset to disk every 60 seconds

BGSAVE

Saves the DB in the background and the parent continues to serve clients



RDB Disadvantages

- Limited to save points
- Not good if you want to minimize the chance of data loss if Redis stops working
- Needs to fork() often which can be time consuming and can wear on CPU performance



AOF – Append Only File

- Main persistence option
- Every operation gets logged (more reliable)
- Log is the same format used by clients
- Can be piped to another instance
- Dataset can be reconstructed



AOF Rewrite

- Used when AOF file gets too big
- Rewrite database from scratch
- Directly access data in memory
- No need for disk access
- Once written, the temp file is synched on to disk



fsync Policies

- No fsync – Done by OS. Usually every 30s or so
- fsync every second (default)
- fsync at every query (slow)



AOF Advantages

- Much more durable
- Single file with no corruption
- Automatically rewrites in the background if it gets too big
- Easy to understand log / instructions



AOF Disadvantages

- Takes longer to load in memory on server restart
- Usually bigger than the equivalent RDB files (2-3x)
- Can be slower than RDB depending on the fsync policy
- More bugs ?



Redis Lists

- Lists in redis are implemented as Linked Lists
- $O(1)$ for push/pop operations but $O(n)$ for index access (Use sorted sets instead)

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```



Common Use Cases for Lists

Lists are useful for a number of tasks, two very representative use cases are the following:

- Remember the **latest updates** posted by users into a social network.
- Communication between processes, using a **consumer-producer pattern** where the producer pushes items into a list, and a consumer (usually a *worker*) consumes those items and executed actions. Redis has special list commands to make this use case both more reliable and efficient.



Blocking Operations on Lists

For interprocess communication
Avoid polling

See: <https://redis.io/topics/data-types-intro>



Bitmap Operations



HyperLogLogs

Notes from <https://redis.io/topics/data-types-intro>

A HyperLogLog is a probabilistic data structure used in order to count unique things (technically this is referred to estimating the cardinality of a set). Usually counting unique items requires using an amount of memory proportional to the number of items you want to count, because you need to remember the elements you have already seen in the past in order to avoid counting them multiple times. However there is a set of algorithms that trade memory for precision: you end with an estimated measure with a standard error, which in the case of the Redis implementation is less than 1%. The magic of this algorithm is that you no longer need to use an amount of memory proportional to the number of items counted, and instead can use a constant amount of memory! 12k bytes in the worst case, or a lot less if your HyperLogLog (We'll just call them HLL from now) has seen very few elements.



Publish / Subscribe

- PSUBSCRIBE
- PUBLISH
- PUBSUB
- PUNSUBSCRIBE
- SUBSCRIBE
- UNSUBSCRIBE

