**Redis Vs MemCached**

Redis is more powerful, more popular, and better supported than memcached. 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.

**Memcached vs Redis: Direct Comparison**

Both tools are powerful, fast, in-memory data stores that are useful as a cache. Both can help speed up your application by caching database results, HTML fragments, or anything else that might be expensive to generate.

**Points to Consider**

When used for the same thing, here is how they compare using the original question's "Points to Consider":

* **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. I recommend redis, but not because memcached is slow. It's not.
* **Memory usage**: Redis is better.
  + memcached: You specify the cache size and as you insert items the daemon quickly grows to a little more than this size. There is never really 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 you. Redis will never use more than it has to and will give you back memory it is no longer using.
  + I stored 100,000 ~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.
* **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.
* **Scaling**: Both give you tons of headroom before you need more than a single instance as a cache. Redis includes tools to help you go beyond that while memcached does not.

**memcached**

Memcached is a simple volatile cache server. It allows you to store key/value pairs where the value is limited to being a string up to 1MB.

It's good at this, but that's 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.

**redis**

Redis can do the same jobs as memcached can, and can do them better.

Redis can [act as a cache](https://redis.io/topics/lru-cache) as well. It can store key/value pairs too. In redis they can even be up to 512MB.

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 do that as well. In fact, that's the default.

It's super fast too, often limited by network or memory bandwidth.

If one instance of redis/memcached isn't enough performance for your workload, redis is the clear choice. Redis includes [cluster support](https://redis.io/topics/cluster-tutorial) and comes with high availability tools ([redis-sentinel](https://redis.io/topics/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.

**The Redis Superset**

Redis is more than a cache. It is an in-memory data structure server. Below you will find a quick overview of things Redis can do beyond being a simple key/value cache like memcached. *Most* of redis' features are things memcached cannot do.

**Documentation**

Redis is better documented than memcached. While this can be subjective, it seems to be more and more true all the time.

[redis.io](https://redis.io/) is a fantastic easily navigated resource. It lets you [try redis in the browser](https://try.redis.io/) and even gives you live interactive examples with each command in the docs.

There are now 2x as many stackoverflow results for redis as memcached. 2x as many Google results. More readily accessible examples in more languages. More active development. More active client development. These measurements might not mean much individually, but in combination they paint a clear picture that support and documentation for redis is greater and much more up-to-date.

[**Persistence**](https://redis.io/topics/persistence)

By default redis persists your data to disk using a mechanism called snapshotting. If you have enough RAM available it's able to write all of your 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, don't worry, redis has your back there too with 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.

**Many Data Types**

Memcached is limited to strings, but Redis is a data structure server that can serve up many different data types. It also provides the commands you need to make the most of those data types.

**Strings (**[**commands**](https://redis.io/commands#string)**)**

Simple text or binary values that can be up to 512MB in size. This is the only data type redis and memcached share, though memcached strings are limited to 1MB.

Redis gives you 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.

Strings are useful for all sorts of use cases, which is why memcached is fairly useful with this data type alone.

**Hashes (**[**commands**](https://redis.io/commands#hash)**)**

Hashes are sort of like a key value store within a key value store. They map between string fields and string values. Field->value maps using a hash are slightly more space efficient than key->value maps using regular strings.

Hashes are useful as a namespace, or when you want to logically group many keys. With a hash you can grab all the members efficiently, expire all the members together, delete all the members together, etc. Great for any use case where you have several key/value pairs that need to grouped.

One example use of a hash is for storing user profiles between applications. A redis hash stored with the user ID as the key will allow you to store as many bits of data about a user as needed while keeping them stored under a single key. The advantage of using a hash instead of serializing the profile into a string is that you can have different applications read/write different fields within the user profile without having to worry about one app overriding changes made by others (which can happen if you serialize stale data).

**Lists (**[**commands**](https://redis.io/commands#list)**)**

Redis lists are ordered collections of strings. They are optimized for inserting, reading, or removing values from the top or bottom (aka: left or right) of the list.

Redis provides many [commands](https://redis.io/commands#list) for leveraging lists, including commands to push/pop items, push/pop between lists, truncate lists, perform range queries, etc.

Lists make great durable, atomic, queues. These work great for job queues, logs, buffers, and many other use cases.

**Sets (**[**commands**](https://redis.io/commands#set)**)**

Sets are unordered collections of unique values. They are optimized to let you quickly check if a value is in the set, quickly add/remove values, and to measure overlap with other sets.

These are great for things like access control lists, unique visitor trackers, and many other things. Most programming languages have something similar (usually called a Set). This is like that, only distributed.

Redis provides several [commands](https://redis.io/commands#set) to manage sets. Obvious ones like adding, removing, and checking the set are present. So are less obvious commands like popping/reading a random item and commands for performing unions and intersections with other sets.

**Sorted Sets (**[**commands**](https://redis.io/commands#sorted_set)**)**

Sorted Sets are also collections of unique values. These ones, as the name implies, are ordered. They are ordered by a score, then lexicographically.

This data type is optimized for quick lookups by score. Getting the highest, lowest, or any range of values in between is extremely fast.

If you add users to a sorted set along with their high score, you have yourself a perfect leader-board. As new high scores come in, just add them to the set again with their high score and it will re-order your leader-board. Also great for keeping track of the last time users visited and who is active in your application.

Storing values with the same score causes them to be ordered lexicographically (think alphabetically). This can be useful for things like auto-complete features.

Many of the sorted set [commands](https://redis.io/commands#sorted_set) are similar to commands for sets, sometimes with an additional score parameter. Also included are commands for managing scores and querying by score.

**Geo**

Redis has several [commands](https://redis.io/commands#geo) for storing, retrieving, and measuring geographic data. This includes radius queries and measuring distances between points.

Technically geographic data in redis is stored within sorted sets, so this isn't a truly separate data type. It is more of an extension on top of sorted sets.

**Bitmap and HyperLogLog**

Like geo, these aren't completely separate data types. These are commands that allow you to treat string data as if it's either a bitmap or a hyperloglog.

Bitmaps are what the bit-level operators I referenced under Strings are for. This data type was the basic building block for reddit's recent collaborative art project: [r/Place](https://redditblog.com/2017/04/13/how-we-built-rplace/).

HyperLogLog allows you to use a constant extremely small amount of space to count almost unlimited unique values with shocking accuracy. Using only ~16KB you could efficiently count the number of unique visitors to your site, even if that number is in the millions.

**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](https://redis.io/topics/transactions) that use "optimistic locking" ([WATCH](https://redis.io/commands/watch)/[MULTI](https://redis.io/commands/multi)/[EXEC](https://redis.io/commands/exec)).

**Pipelining**

Redis provides a feature called '[pipelining](https://redis.io/topics/pipelining)'. If you have many redis commands you want to execute you can use pipelining to send them to redis all-at-once instead of one-at-a-time.

Normally when you execute a command to either redis or memcached, each command is a separate request/response cycle. With pipelining, redis can buffer several commands and execute them all at once, responding with all of the responses to all of your commands in a single reply.

This can allow you to achieve even greater throughput on bulk importing or other actions that involve lots of commands.

**Pub/Sub**

Redis has [commands](https://redis.io/commands#pubsub) dedicated to [pub/sub functionality](https://redis.io/topics/pubsub), allowing redis to act as a high speed message broadcaster. This allows a single client to publish messages to many other clients connected to a channel.

Redis does pub/sub as well as almost any tool. Dedicated message brokers like [RabbitMQ](https://www.rabbitmq.com/) may have advantages in certain areas, but the fact that the same server can also give you persistent durable queues and other data structures your pub/sub workloads likely need, Redis will often prove to be the best and most simple tool for the job.

**Lua Scripting**

You can kind of think of [lua scripts](https://redis.io/commands/eval) like redis's own SQL or stored procedures. It's both more and less than that, but the analogy mostly works.

Maybe you have complex calculations you want redis to perform. Maybe you can't afford to have your transactions roll back and need guarantees every step of a complex process will happen atomically. These problems and many more can be solved with lua scripting.

The entire script is executed atomically, so if you can fit your logic into a lua script you can often avoid messing with optimistic locking transactions.

**Scaling**

As mentioned above, redis includes built in support for clustering and is bundled with its own high availability tool called redis-sentinel.

**Conclusion**

Without hesitation I would recommend redis over memcached for any new projects, or existing projects that don't already use memcached.

The above may sound like I don't like memcached. On the contrary: it is a powerful, simple, stable, mature, and hardened tool. There are even some use cases where it's a little faster than redis. I love memcached. I just don't think it makes much sense for future development.

Redis does everything memcached does, often better. Any performance advantage for memcached is minor and workload specific. There are also workloads for which redis will be faster, and many more workloads that redis can do which memcached simply can't. The tiny performance differences seem minor in the face of the giant gulf in functionality and the fact that both tools are so fast and efficient they may very well be the last piece of your infrastructure you'll ever have to worry about scaling.

There is only one scenario where memcached makes more sense: where memcached is already in use as a cache. If you are already caching with memcached then keep using it, if it meets your needs. It is likely not worth the effort to move to redis and if you are going to use redis just for caching it may not offer enough benefit to be worth your time. If memcached isn't meeting your needs, then you should probably move to redis. This is true whether you need to scale beyond memcached or you need additional functionality.