Pipelining and Transactions in Redis

# Pipelining

Redis pipelining is a technique for improving performance by issuing multiple commands at once without waiting for the response to each individual command. Pipelining is supported by most Redis clients.

## Request/Response Protocols and Round-Trip Time (RTT)

Redis is a TCP server using the client-server model and what is called a Request/Response protocol.

This means that usually a request is accomplished with the following steps:

* The client sends a query to the server, and reads from the socket, **usually** in a blocking way, for the server response.
* The server processes the command and sends the response back to the client.

So, for instance a four commands sequence is something like this:

Client: INCR X

Server: 1

Client: INCR X

Server: 2

Client: INCR X

Server: 3

Client: INCR X

Server: 4

Clients and Servers are connected via a network link. Such a link can be very fast (a loopback interface) or very slow (a connection established over the Internet with many hops between the two hosts). Whatever the network latency is, it takes time for the packets to travel from the client to the server, and back from the server to the client to carry the reply.

This time is called RTT (Round Trip Time). It's easy to see how this can affect performance when a client needs to perform many requests in a row (for instance adding many elements to the same list, or populating a database with many keys). For instance, if the RTT time is 250 milliseconds (in the case of a very slow link over the Internet), even if the server is able to process 100k requests per second, we'll be able to process at max four requests per second.

If the interface used is a loopback interface, the RTT is much shorter, typically sub-millisecond, but even this will add up to a lot if you need to perform many writes in a row.

Fortunately, there is a way to improve this use case.

## Redis Pipelining

A Request/Response server can be implemented so that it is able to process new requests even if the client hasn't already read the old responses. This way it is possible to send *multiple commands* to the server without waiting for the replies at all, and finally read the replies in a single step.

This is called pipelining, and is a technique widely in use for many decades. For instance, many POP3 protocol implementations already support this feature, dramatically speeding up the process of downloading new emails from the server.

Here’s an example of pipelining in redis using the raw netcat utility:

$ (printf "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379

+PONG

+PONG

+PONG

This time we don't pay the cost of RTT for every call, but just once for the three commands.

To be explicit, with pipelining the order of operations of our very first example will be the following:

*Client:* INCR X

*Client:* INCR X

*Client:* INCR X

*Client:* INCR X

*Server:* 1

*Server:* 2

*Server:* 3

*Server:* 4

***IMPORTANT NOTE****: While the client sends commands using pipelining, the server will be forced to queue the replies, using memory. So, if you need to send a lot of commands with pipelining, it is better to send them as batches each containing a reasonable number, for instance 10k commands, read the replies, and then send another 10k commands again, and so forth. The speed will be nearly the same, but the additional memory used will be at most the amount needed to queue the replies for these 10k commands.*

## It’s not Just a Matter of RTT

Pipelining is not just a way to reduce the latency cost associated with the round-trip time, it actually greatly improves the number of operations you can perform per second in a given Redis server.

**This is because without using pipelining, serving each command is very cheap from the point of view of accessing the data structures and producing the reply, but it is very costly from the point of view of doing the socket I/O. This involves calling the read() and write() syscall, that means going from user land to kernel land. The context switch is a huge speed penalty.** When pipelining is used, many commands are usually read with a single read() system call, and multiple replies are delivered with a single write() system call.

Consequently, the number of total queries performed per second initially increases almost linearly with longer pipelines, **and eventually reaches 10 times the baseline obtained without pipelining,** (There is a graph that I did not understand).

## Pipelining vs Scripting

Using Redis scripting, a number of use cases for pipelining can be addressed more efficiently **using scripts that perform a lot of the work needed at the server side**. A big advantage of scripting is that it is able to both read and write data with minimal latency, making operations like read, compute, write very fast (pipelining can't help in this scenario since the client needs the reply of the read command before it can call the write command).

Sometimes the application may also want to send EVAL or EVALSHA commands in a pipeline. This is entirely possible and Redis explicitly supports it with the SCRIPT LOAD command (it guarantees that EVALSHA can be called without the risk of failing).

# Transactions

Redis Transactions allow the **execution of a group of commands in a single step**, they are centered around the commands [MULTI](https://redis.io/docs/latest/commands/multi/), [EXEC](https://redis.io/docs/latest/commands/exec/), [DISCARD](https://redis.io/docs/latest/commands/discard/) and [WATCH](https://redis.io/docs/latest/commands/watch/).

Redis Transactions make two+1 important guarantees:

* All the commands in a transaction are serialized and executed sequentially. A **request sent by another client will never be served in the middle of the execution of a Redis Transaction.** This guarantees that the commands are executed as a single isolated operation.
* The [EXEC](https://redis.io/docs/latest/commands/exec/) command triggers the execution of all the commands in the transaction, so if a client loses the connection to the server in the context of a transaction before calling the [EXEC](https://redis.io/docs/latest/commands/exec/) command none of the operations are performed, instead if the [EXEC](https://redis.io/docs/latest/commands/exec/) command is called, all the operations are performed.

**This part needs to be explored more:**

When using the [append-only file](https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/#append-only-file) Redis makes sure to use a single write(2) syscall to write the transaction on disk(Does it fsync to disk regardless of the configuration for AOF?). However, if the Redis server crashes or is killed by the system administrator in some hard way it is possible that only a partial number of operations are registered. Redis will detect this condition at restart, and will exit with an error. Using the redis-check-aof tool it is possible to fix the append only file that will remove the partial transaction so that the server can start again.

* Starting with version 2.2, Redis allows for an extra guarantee to the above two, in the form of optimistic locking in a way very similar to a check-and-set (CAS) operation. This is documented [later](https://redis.io/docs/latest/develop/using-commands/transactions/#cas) in this document.

## Usage

A Redis Transaction is entered using the [MULTI](https://redis.io/docs/latest/commands/multi/) command. The command always replies with OK. At this point the user can issue multiple commands. Instead of executing these commands, Redis will queue them. All the commands are executed once [EXEC](https://redis.io/docs/latest/commands/exec/) is called.

Calling [DISCARD](https://redis.io/docs/latest/commands/discard/) instead will flush the transaction queue and will exit the transaction.

The following example increments keys foo and bar atomically.

> MULTI

OK

> INCR foo

QUEUED

> INCR bar

QUEUED

> EXEC

1) (integer) 1

2) (integer) 1

As is clear from the session above, [EXEC](https://redis.io/docs/latest/commands/exec/) returns an array of replies, where every element is the reply of a single command in the transaction, **in the same order the commands were issued.**

When a Redis connection is in the context of a [MULTI](https://redis.io/docs/latest/commands/multi/) request, all commands will reply with the string QUEUED (sent as a Status Reply from the point of view of the Redis protocol). A queued command is simply scheduled for execution when [EXEC](https://redis.io/docs/latest/commands/exec/) is called.

## Errors inside a Transaction

During a transaction it is possible to encounter two kind of command errors:

* A command may fail to be queued, so there may be an error before [EXEC](https://redis.io/docs/latest/commands/exec/) is called. For instance, the command may be syntactically wrong (wrong number of arguments, wrong command name, ...), or there may be some critical condition like an out of memory condition (if the server is configured to have a memory limit using the maxmemory directive).
* A command may fail *after* [EXEC](https://redis.io/docs/latest/commands/exec/) is called, for instance since we performed an operation against a key with the wrong value (like calling a list operation against a string value).

Starting with Redis 2.6.5, the server will detect an error during the accumulation of commands. It will then refuse to execute the transaction returning an error during [EXEC](https://redis.io/docs/latest/commands/exec/), discarding the transaction.

Errors happening *after* [EXEC](https://redis.io/docs/latest/commands/exec/) instead are not handled in a special way: **all the other commands will be executed even if some command fails during the transaction.**

This is more clear on the protocol level. In the following example one command will fail when executed even if the syntax is right:

MULTI

+OK

SET a abc

+QUEUED

LPOP a

+QUEUED

EXEC

\*2

+OK

-WRONGTYPE Operation against a key holding the wrong kind of value

[EXEC](https://redis.io/docs/latest/commands/exec/) returned two-element [bulk string reply](https://redis.io/docs/latest/develop/reference/protocol-spec/#bulk-string-reply) where one is an OK code and the other an error reply. It's up to the client library to find a sensible way to provide the error to the user.

It's important to note that **even when a command fails, all the other commands in the queue are processed** – Redis will not stop the processing of commands.

Another example, again using the wire protocol with telnet, shows how syntax errors are reported ASAP instead:

MULTI

+OK

INCR a b c

-ERR wrong number of arguments for 'incr' command

**This time due to the syntax error the bad**[**INCR**](https://redis.io/docs/latest/commands/incr/)**command is not queued at all.**

**I add: Not all the syntax errors are reported asap, as far as I checked,** wrong number of arguments **will result in an immediate error and when running exec you’ll get:**

EXECABORT Transaction discarded because of previous errors.

When this happens, no command will be executed. But I had another syntax error which was not of wrong number of arguments type, and the commands before and after it were processed anyways so be careful.

## What about Rollbacks?

Redis does not support rollbacks of transactions since supporting rollbacks would have a significant impact on the simplicity and performance of Redis.

## Discarding the Command Queue

DISCARD can be used in order to abort a transaction. In this case, no commands are executed and the state of the connection is restored to normal.

> SET foo 1

OK

> MULTI

OK

> INCR foo

QUEUED

> DISCARD

OK

> GET foo

"1"

## Redis Scripting and Transactions

[redis scripts](https://redis.io/docs/latest/commands/eval/)(using eval for example) are transactional. Everything you can do with a Redis Transaction, you can also do with a script, and usually the script will be both simpler and faster. (We don’t dig deeper than this at this time)

## I add: Pipelining and Transactions

I saw an official video from Redis university that said most redis clients implement pipelines transactional.

## Optimistic Locking with Using Check-and-Set

WATCH is used to provide a check-and-set (CAS) behavior to Redis transactions. Watched keys are monitored in order to detect changes against them. If at least one watched key is modified before the EXEC command, the whole transaction aborts, **and EXEC returns a Null reply to notify that the transaction failed.**

For example, imagine we have the need to atomically increment the value of a key by 1 (let's suppose Redis doesn't have INCR). The first try may be the following:

val = GET mykey

val = val + 1

SET mykey $val

This will work reliably only if we have a single client performing the operation in a given time. If multiple clients try to increment the key at about the same time there will be a race condition. For instance, client A and B will read the old value, for instance, 10. The value will be incremented to 11 by both the clients, and finally [SET](https://redis.io/docs/latest/commands/set/) as the value of the key. So, the final value will be 11 instead of 12.

Thanks to [WATCH](https://redis.io/docs/latest/commands/watch/) we are able to model the problem very well:

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

Using the above code, if there are race conditions and another client modifies the result of val in the time between our call to [WATCH](https://redis.io/docs/latest/commands/watch/) and our call to [EXEC](https://redis.io/docs/latest/commands/exec/), the transaction will fail.

We just have to repeat the operation hoping this time we'll not get a new race.

This form of locking is called *optimistic locking*. In many use cases, multiple clients will be accessing different keys, so collisions are unlikely – usually there's no need to repeat the operation.

### A Deeper Look at the Watch Command

So what is [WATCH](https://redis.io/docs/latest/commands/watch/) really about? It is **a command that will make the**[**EXEC**](https://redis.io/docs/latest/commands/exec/)**conditional**: we are asking Redis to perform the transaction only if none of the [WATCH](https://redis.io/docs/latest/commands/watch/)ed keys were modified. This includes modifications made by the client(even the very same connection that is WATCHING the key), like write commands, and by Redis itself, like expiration or eviction. If keys were modified between when they were [WATCH](https://redis.io/docs/latest/commands/watch/)ed and when the [EXEC](https://redis.io/docs/latest/commands/exec/) was received, the entire transaction will be aborted instead.

**NOTE**

* In Redis versions before 6.0.9, an expired key would not cause a transaction to be aborted. [More on this](https://github.com/redis/redis/pull/7920)
* **Commands within a transaction won't trigger the**[**WATCH**](https://redis.io/docs/latest/commands/watch/)**condition since they are only queued until the**[**EXEC**](https://redis.io/docs/latest/commands/exec/)**is sent.**

[WATCH](https://redis.io/docs/latest/commands/watch/) can be called multiple times. Simply **all the**[**WATCH**](https://redis.io/docs/latest/commands/watch/)**calls will have the effects to** **watch for changes starting from the call, up to the moment**[**EXEC**](https://redis.io/docs/latest/commands/exec/)**is called**.

You can also send any number of keys to a single [WATCH](https://redis.io/docs/latest/commands/watch/) call.

**When**[**EXEC**](https://redis.io/docs/latest/commands/exec/)**is called, all keys are**[**UNWATCH**](https://redis.io/docs/latest/commands/unwatch/)**ed, regardless of whether the transaction was aborted or not. Also, when a client connection is closed, everything gets**[**UNWATCH**](https://redis.io/docs/latest/commands/unwatch/)**ed.**

It is also possible to use the [UNWATCH](https://redis.io/docs/latest/commands/unwatch/) command (without arguments) in order to flush all the watched keys. Sometimes this is useful as we optimistically lock a few keys, since possibly we need to perform a transaction to alter those keys, but after reading the current content of the keys we don't want to proceed. When this happens, we just call [UNWATCH](https://redis.io/docs/latest/commands/unwatch/) so that the connection can already be used freely for new transactions.

### Using WATCH to implement ZPOP

A good example to illustrate how [WATCH](https://redis.io/docs/latest/commands/watch/) can be used to create new atomic operations otherwise not supported by Redis is to implement ZPOP ([ZPOPMIN](https://redis.io/docs/latest/commands/zpopmin/), [ZPOPMAX](https://redis.io/docs/latest/commands/zpopmax/) and their blocking variants have only been added in version 5.0), that is a command that pops the element with the lower score from a sorted set in an atomic way. This is the simplest implementation:

WATCH zset

element = ZRANGE zset 0 0

MULTI

ZREM zset element

EXEC

**if**[**EXEC**](https://redis.io/docs/latest/commands/exec/)**fails (i.e. returns a**[**Null reply**](https://redis.io/docs/latest/develop/reference/protocol-spec/#nil-reply)**) we just repeat the operation.**

**(What’s the difference between using the WATCH command and just doing the entire operation in a transaction??)**