

# Hello!



**I am Esmaeil Kazemi**

**I'm interested in learning how are you?**

**You can find me at @eskazemi**





# NOSQL

VS

# SQL



# Redis

Redis stands for Remote Dictionary Server





eskazemi

# Map

# Redis



1- Features

2- use

3- data type

4- Message  
Queue

5- Transactions

6- Pipelining

7- Lua Scripts

8- Persistence

9- Benchmarks

10- configuration

11- ACLs

12- Redis Cluster

13- Redis vs Memcached

14- Redis vs Hazelcast



eskazemi



redis

**Transactions**

**Pipelining**

**Lua Scripts**

## Transactions

همان طور که می دانیم **transaction** به مجموعه ای از دستورات که قراره پشت سرهم اجرا شوند گفته می شود که این تراکنش دو ویژگی دارد :

- **Isolated** : تمامی دستورات که در یک transaction قرار می گیرند به صورت default قرنطینه نیستند یا به عبارتی isolate نمی باشند اگر شما از یکی از کلید ها با استفاده از دستور **WATCH** قرنطینه بکنید در این حالت اگر کسی از خارج از اون **transaction** اقدام به تغییر اطلاعات کند تمام دستورات متوقف می شود (abort) شده و اجرا نمی شوند.

- **Atomic** : یعنی تمام دستورات یا باید کامل اجرا شوند یا هیچ کدام نباید اجرا شوند.

transaction شروع می شود با `MULTI <- command` و شما باید لیستی از `commands` هایی که باید اجرا بشوند در تراکنش بهش پاس بدهید و بعد از اتمام آنها با `EXEC <- command` تراکنش را اجرا کنید.

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

## Transactions

مثال

```
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

دستورات مربوط به آن Transaction و توضیحات آنها

دستورها	توضیحات
1 DICARD	Discards all commands issued after MULTI
2 EXEC	اجرا کردن همه دستور ها بعد از دستور MULTI نوشته شده است
3 MULTI	شروع یک بلوک transaction را نشان می دهد
4 UNWATCH	Forgets about all watched keys
5 WATCH key [key ...]	کلید ها با استفاده از دستور watch قرنطینه شوند در این حالت اگر کسی از خارج از اون transaction اقدام به تغییر اطلاعات کند تمام دستورات متوقف می شود



## Transactions

## محدودیت های Transaction

با استفاده از **transaction** ها ما توانایی استفاده از مقادیر میانی برای دستورهای بعدی را نداریم. ما تنها لیست کامل پاسخ ها را در پایان دریافت می کنیم.

اگر خطایی در یکی از دستورها رخ دهد، ۲ احتمال وجود دارد:

1. اگر خطای نحوی باشد (مانند اشتباه در تعداد آرگومان ها)، هنگام صف بندی دستورها تشخیص داده می شود و تراکنش حتی اجرا نخواهد شد.
2. اگر یک خطای معنایی باشد (مانند یک عملیات بر روی نوع داده اشتباه)، تنها در حین اجرای تراکنش تشخیص داده می شود و خطا در داخل لیست پاسخ ها، به عنوان پاسخ برای دستور خاص بازگردانده خواهد شد. اما دستورهای بعدی در صف به صورت عادی اجرا خواهند شد و تراکنش قطع نخواهد شد. این بدان معنی است که Redis مکانیزم rollback مانند RDBMS سستی ندارند.



چه زمانی باید استفاده کنید از transaction ها؟

شما باید استفاده کنید از transaction ها اگر :

eskazemi

- نیاز به اجرای atomic دستور ها دارید
- برای نوشتن دستورات بعدی به مقادیر میانی نیاز ندارید





# WHAT IS REDIS PIPELINE

## pipeline

کلاینت ها و سرورهای Redis با استفاده از پروتکلی به نام REP مبتنی بر TCP است، با یکدیگر ارتباط برقرار می کنند. در پروتکل مبتنی بر TCP، سرور و کلاینت با یک مدل درخواست / پاسخ ارتباط برقرار می کنند.

کلاینت یک درخواستی به سرور می فرستد، و توسط سرور از طریق سوکت خوانده می شود، سرور که در حال پردازش command (request) می باشد client منتظر پاسخ به روش مسدود کننده است. سرور بعد از پردازش پاسخ را به client بر می گرداند.

هر یک درخواست به Redis ، 250 میلی ثانیه طول می کشد در نهایت در یک ثانیه 4 درخواست توسط Redis پردازش می شود در صورتی که یک سرور Redis توانایی آن را دارد در یک ثانیه 100000 هزار درخواست را پردازش کند. در نتیجه باید از Redis pipeline استفاده کرد pipeline ها در Redis به شما اجازه میدن که چندین درخواست رو داخل یک پکیج یک جا بفرستید و Redis هم چندین پاسخ را داخل یک پکیج برای شما بفرسته

توجه اگر از redis pipeline استفاده نکنیم دیگر نمی توان از حداکثر توان سرور Redis استفاده کرد و سرعت مون کاهش پیدا می کند.

“

یکی از ویژگی های کلیدی Redis پشتیبانی آن از pipeline است که اجازه می دهد چندین دستور Redis در یک شبکه رفت و برگشت به سرور ارسال شود. این می تواند عملکرد برنامه را با کاهش سربار ارتباطات شبکه و پردازش سرور، به میزان زیادی بهبود بخشد.

”

## pipeline

```
$(echo -en "PING\r\n SET tutorial redis\r\nGET tutorial\r\nINCR
visitor\r\nINCR visitor\r\nINCR visitor\r\n"; sleep 10) | nc localhost 6379
+PONG
+OK
redis
:1
:2
:3
```

به طور مثال اینجا یه نمونه **pipeline** نشان داده شده است در اینجا چندین **command** رو همزمان اجرا کردیم با استفاده از **pipeline** ، **command** هایی همچون ping ، ایجاد یک کلید با مقدار redis و بعد همه کلید ها رو گرفتیم و ... همان طور که می بینید همه **command** ها یکبار submit میشه و Redis خروجی همه **command** ها در یک مرحله بر گردانده است.

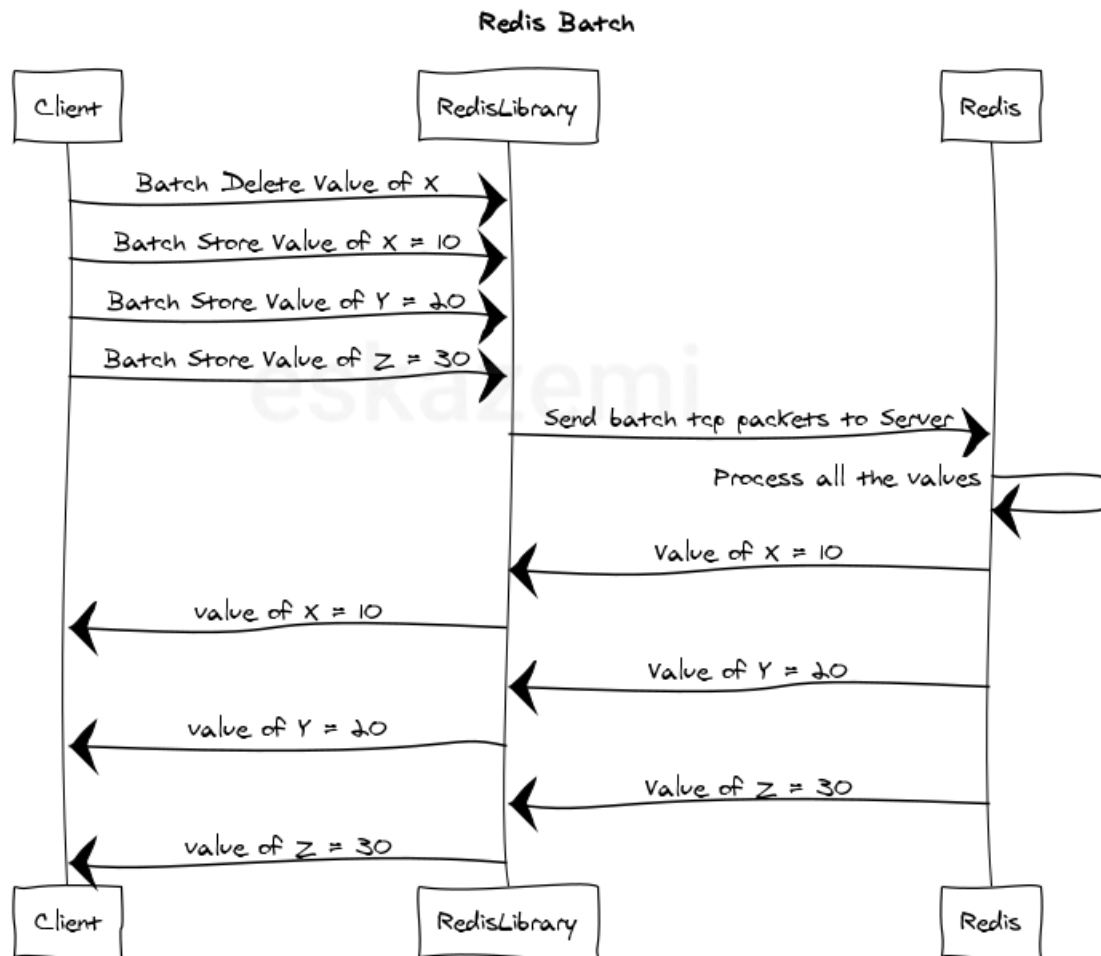
### pipeline

```
import redis

conn = redis.Redis()
pipeline = conn.pipeline(transaction=False)
pipeline.get('foo')
pipeline.get('bar')
pipeline.execute()
```

شروع یک redis pipeline و گرفتن مقدار ها در کتابخانه redis مربوط به زبان پایتون

## نحوه ی عملکرد pipeline با جزئیات

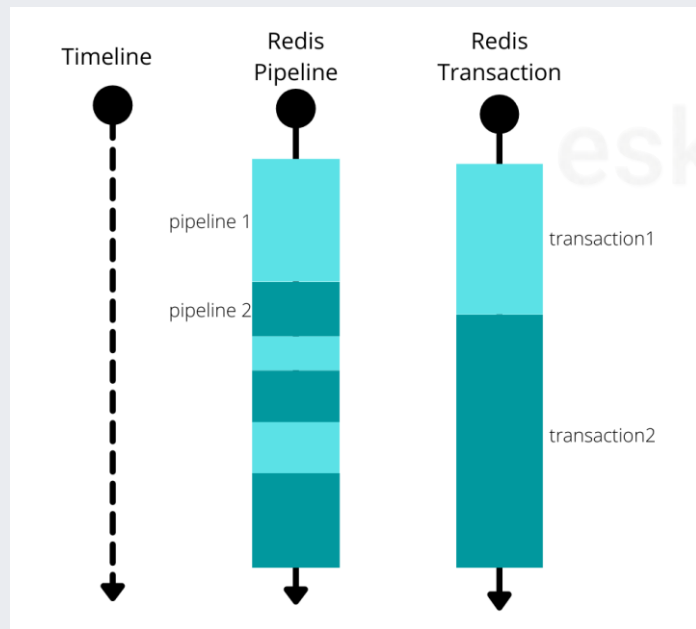




## Transactions vs Pipeline

### pipeline

ما دیده ایم که **pipeline** راهی برای ارسال دسته دسته جمعی دستور ها از سمت کلاینت به سرور است ، همین موضع در مورد **Transaction** نیز صدق می کند، تفاوت آنها در چیه؟



Transaction



atomic

Pipelines



به این معنی که 2 تا transaction نمی توانند همزمان اجرا شوند در حالی که چندین pipeline می توانند اجرا شوند به صورت همزمان توسط Redis-server و پایان یابند.

## انتخاب اندازه مناسب pipeline

### pipeline

هنگام استفاده از **pipeline**، مهم است که اندازه pipeline مناسب را برای تعادل عملکرد و استفاده از حافظه انتخاب کنید. اندازه **pipeline** به تعداد فرمان هایی اشاره دارد که در یک درخواست pipeline به هم متصل شده و به سرور فرستاده می شوند.

**Big size** : اندازه خط لوله بزرگ تر می تواند با کاهش سربار ارتباط شبکه و پردازش سرور منجر به عملکرد بهتر شود. با این حال، اندازه های خط لوله بزرگ تر همچنین به حافظه بیشتری در سمت client برای نگهداری درخواست های در حال انتظار نیاز دارد و می تواند احتمال از دست رفتن داده ها را در صورت شکست یک درخواست افزایش دهد.

**Small size** : اندازه خط لوله کوچک تر می تواند استفاده از حافظه را کاهش دهد و قابلیت اطمینان را بهبود بخشد، اما همچنین می تواند به دلیل افزایش سربار ارتباطات شبکه و پردازش سرور منجر به عملکرد کم تر شود.

## چه زمانی نمی توانیم استفاده کنیم از pipeline

### pipeline

1 - زمانی که نوشتنی به دنبال خواندن باشد امکان پذیر نیست در یک **pipeline** چون که نتایج همه command ها را باهم در انتها می بینیم. به عبارتی دیگه ابتدا مقدار کلیدی را بخوانیم و بخواهیم از مقدار این کلید یک کلید دیگه با همین مقدار ایجاد کنیم.

2 - زمانی که داریم استفاده می کنیم از pipeline و همچنین از **Redis** در حالت خوشه ای (cluster) استفاده می کنیم باید مطمئن شویم که کلیدها باید در یک hash slot باشند و نه فقط در همان گره چرا که در حالت خوشه ای Redis اگر تمام کلیدها به یک slot یک سان نگاشت نشوند، Redis ارور برخواهد گردانید ( CROSSSLOT Keys in request don't hash to the same slot ) با این حال این خطا را می توان با استفاده از hash-tags کنترل کرد. با hash-tags می توانیم کلیدها را مجبور کنیم که روی یک same hash slot نگاشت شوند.

# Redis Lua Scripting



## Lua Scripts

### leaderboard

1.	Andrew	10
2.	Bella	20
3.	Andy	30
4.	Dolly.	33
5.	Cathy.	40
6.	Heather	42
7.	Gilbert.	43
8.	Lilly.	45
9.	Dinesh.	50
10.	Jon snow.	58
11.	Ygritte.	60

فرض بگیرید که ما یک تابلوی امتیاز داریم که در ZSET نگهداری می شود ، اکنون چیزی که دنبال آن هستیم رتبه یک کاربر می خواهیم به علاوه همسایگان آن کاربر در تابلوی امتیازات . در ظاهر بیان مسئله بسیار آسان به نظر می رسد ، بیایید تابلوی امتیازات خود را تجسم کنیم و ببینیم چگونه می توان این مسئله را حل کرد.

## Lua Scripts

راه حل ساده به نظر می رسد اجرا ی دو دستور پشت سرهم است!

1- گرفتن رتبه کاربر با استفاده از دستور زیر و برگرداندن رتبه کاربر (رتبه 5) `zrank leaderboard Heather`

2- برای بدست آوردن همسایگان کاربر می توان 3 کاربر بالایی و پایینی را با دستور بدست آورد

`zrange leaderboard 2 8`

## Lua Scripts

اجرای دو دستور در Redis client

```
127.0.0.1:6379> zrange leaderboard 0 -1
1) "Andrew"
2) "Bella"
3) "Andy"
4) "Dolly"
5) "Cathy"
6) "Heather"
7) "Gilbert"
8) "Dinesh"
9) "Lilly"
10) "Ygritte"
11) "Jon snow"
127.0.0.1:6379> zrank leaderboard Heather
(integer) 5
127.0.0.1:6379> zrange leaderboard 2 8
1) "Andy"
2) "Dolly"
3) "Cathy"
4) "Heather"
5) "Gilbert"
6) "Dinesh"
7) "Lilly"
127.0.0.1:6379> |
```

“

گرچه در ظاهر راه حل خوبی است ، اما باگ است، زیرا تابلوی امتیازات می تواند می تواند بین صدا زدن zrank و zrange **تغییر** کند.

راه حلی که می تواند به ذهن خطور کند استفاده از **Redis pipeline** است ، با این حال ، pipeline را نمی بتوان استفاده کرد چون خروجی zrank به عنوان ورودی zrange استفاده کرد، چون که نمی توان **هر دو را در یک pipeline** استفاده کرد .

”



## Lua Scripts

یکی از راه حل هایی که در واقع جواب می دهد، استفاده از نوعی قفل قبل از به دست آوردن رتبه کاربر و سپس استفاده از `zrange` و سپس برداشتن قفل است. و قبل از نوشتن در تابلوی امتیازات باید قفل را چک کنیم، و اگر قفل وجود دارد، دوباره امتحان کنیم تا قفل برداشته شود. روش قفل کردن این است که مطمئن شوید هیچ نوشتن بین `zrank` و `zrang` اتفاق نمی افتد. در حالی که این روش جواب می دهد، ما به وضوح پیچیدگی کل عملیات را افزایش داده ایم.

while reading from redis.

```
take_lock()
  get_rank()
  get_neighbours()
remove_lock()
```

while writing to redis.

```
locked = get_lock();
if not locked:
  write_to_leaderboard()
else:
  keep_retrying()
```

## Lua Scripts

یک راه عالی برای حل این مسائل استفاده از **Lua Scripts** است. Redis به ما اجازه می دهد **Lua Scripts** را روی سرور آپلود و اجرا کنیم و از آنجا که اسکریپت ها روی سرور اجرا می شوند، خواندن و نوشتن داده از طریق اسکریپت ها بسیار کارآمد است.

Redis اجرای atomic ← **Lua Scripts** ← تضمین می کند.

در حین اجرای اسکریپت، تمام فعالیت های سرور در طول زمان اجرای آن مسدود می شوند.

```
> EVAL "return 'Hello, world!'" 0 "Hello, world!"
```

بیایید با یک مثال ساده شروع کنیم :

با نوشتن یک Lue Scripts ساده مشکل را حل می کنیم.

## Lua Scripts

```
local rank = redis.call('zrank', KEYS[1], ARGV[1]); local min =  
math.max(rank - ARGV[2], 0);  
local max = rank + ARGV[2];  
local ldb = redis.call('zrange', KEYS[1], min, max); return {rank+1, ldb};
```

بیایید خط به خط اسکریپت را بشکنیم:

در خط اول ما گرفتیم رتبه کاربر که KEYS[1] نام تابلوی امتیازات است و ARGV[1] هست username برای اینکه اطلاعات همسایگان کاربر را بدست آوریم.

خطوط 2 و 3 فقط برای بدست آوردن کران پایین و بالایی برای محدوده cmd هستند. در ARGV[2]، ما در حال دریافت offset برای تابلوی امتیازات هستیم. برای کمترین مقدار یک کردیم که زیر 0 نرود.

در خط چهارم ما در حال گرفتن leaderboard هستیم

در خط پنجم یک آرایه با رتبه کاربر و همسایگان مقداردهی اولیه کرده و آن را بر گدانده ایم

## Lua Scripts

```
127.0.0.1:6379> eval "local rank = redis.call('zrank', KEYS[1], ARGV[1]); local min = math.max(rank - ARGV[2], 0); local max = rank + ARGV[2]; local ldb = redis.call('zrange', KEYS[1], min, max); return {rank, ldb};" 1 leaderboard Heather 3
```

```
1) (integer) 5
```

```
2) 1) "Andy"
```

```
2) "Dolly"
```

```
3) "Cathy"
```

```
4) "Heather"
```

```
5) "Gilbert"
```

```
6) "Dinesh"
```

```
7) "Lilly"
```

```
127.0.0.1:6379>
```

```
127.0.0.1:6379> █
```

eskazemi

## Lua Scripts

در حالی که موارد بالا کار می کند، ارسال هر بار این اسکریپت بزرگ از طرف client کارآمد نیست

خوشبختانه Redis داری API هایی است که با آن می توانیم اسکریپت ها روی سرور ذخیره کنیم و به جای اینکه هر بار از سمت client این اسکریپت های بزرگ را دریافت کنیم از آنها استفاده کنیم

لطفا توجه داشته باشید که حافظه پنهان اسکریپت Redis همیشه **ناپایدار** است و پایدار نیست. **حافظه پنهان** ممکن است هنگام راه اندازی مجدد سرور، در طول شکست زمانی که یک کپی (replica) نقش اصلی را بر عهده می گیرد، یا به صراحت توسط SCRIPT FLUSH این بدان معناست که اسکریپت های حافظه پنهان زودگذر هستند و متحویات حافظه پنهان (کش) می توانند در هر زمانی از بین برود.

## Lua Scripts

دستور زیر برای ذخیره اسکریپت که بر می گرداند یک کلید SHA1 که اختصاصی برای اسکریپت است و بعد از آن از دستور دوم می توان برای فراخوانی اسکریپت استفاده کرد.

```
> SCRIPT LOAD <script>
```

```
> EVALSHA <sha1>
```

بیا یاد نگاهی بیندازیم به اجرای دو دستور:

## Lua Scripts

```
127.0.0.1:6379> script load "local rank = redis.call('zrank', KEYS[1], ARGV[1]
); local min = math.max(rank - ARGV[2], 0); local max = rank + ARGV[2]; local
ldb = redis.call('zrange', KEYS[1], min, max); return {rank, ldb};"
"ce79631a2e3d6313be47a8cd3160a8b8165e4bf1"
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> evalsha "ce79631a2e3d6313be47a8cd3160a8b8165e4bf1" 1 leaderboa
rd Heather 3
1) (integer) 5
2) 1) "Andy"
   2) "Dolly"
   3) "Cathy"
   4) "Heather"
   5) "Gilbert"
   6) "Dinesh"
   7) "Lilly"
127.0.0.1:6379> █
```

## Lua Scripts

## جمع بندی:

- ✓ از آنجایی که اسکریپت ها در سرور اجرا می شوند ، خواندن و نوشتن داده ها از طریق اسکریپت ها بسیار کارآمد است. محلی بودن داده ها تاخیر کلی را کاهش می دهد و منابع شبکه را ذخیره می کند.
- ✓ **Lua Scripts** را می توان استفاده کنیم در جاهایی که به atomicity نیاز است و از pipeline و Transaction نمی توان استفاده کرد، چرا که Redis ، atomicity بودن **Lua Scripts** را تضمین می کند.
- ✓ دستور EVAL برای اجرای اسکریپت ها استفاده می شود اما ارسال اسکریپت های طولانی کارآمد نیست، در این صورت می توانیم اسکریپت را با استفاده از `SCRIPT LOAD <script>` روی سرور ذخیره کنیم و سپس از دستور `EVALSHA <SHA>` برای اجرای اسکریپت استفاده کنیم.
- ✓ حافظه پنهان اسکریپت Redis همیشه فرار است و ثبات ندارد.





How does Redis Lua scripting improve performance?

Redis Lua scripting improves performance by reducing the number of round-trips between the client and the server, as well as by allowing for complex operations to be executed on the server-side.



# Thanks!



**Any questions?**

**You can find me at:**

- @eskazemi
- m.esmaeilkazemi@gmail.com



eskazemi