

# Hello!



**I am Esmaeil Kazemi**

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

**You can find me at @eskazemi**



# Redis

Redis stands for Remote Dictionary Server





eskazemi

# Map Redis



1- Features

2- application

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

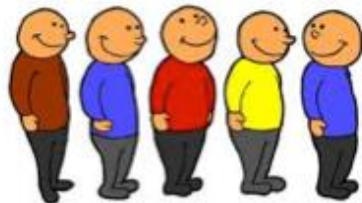
14- Redis vs Hazelcast

15- Redis vs RDBMS



eskazemi

# Redis Message Queue



redis

eskaz





## Redis Queue

### چگونگی صف بندی پیام ها در Redis

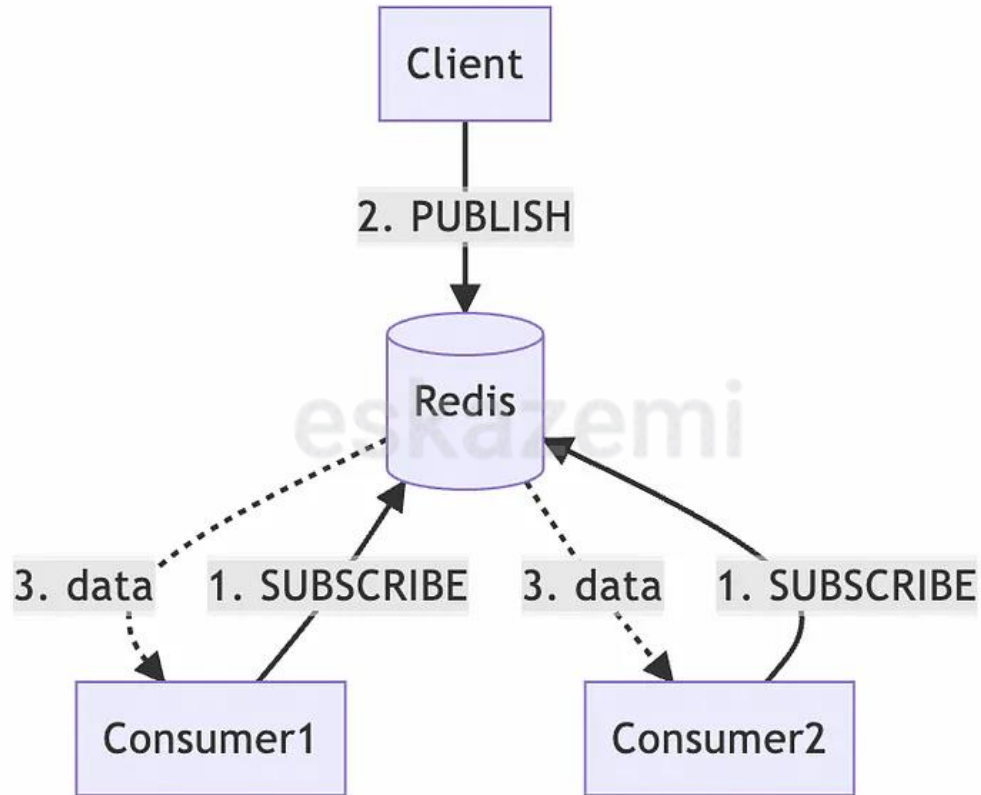
3 راه برای انجام این کار وجود دارد :

Pub/Sub  
List  
Stream

یکی پس از دیگری معرفی می کنیم و سپس یک خلاصه جامع ارائه می دهیم .

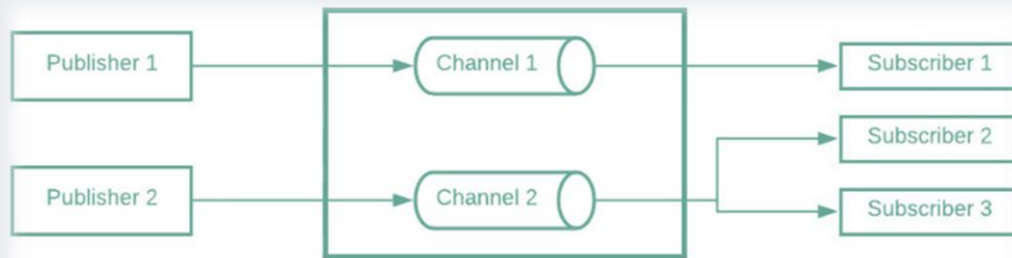


## Pub/Sub





## Pub/Sub



eskazemi Redis Pub/Sub

- ✓ Pub/sub یک پترن پیغام رسانی است که در آن subscriber یا مصرف کننده (consumer) یک موضوع را به عنوان کلید انتخاب می کند (SUBSCRIBE a topic) و سپس پس از انتشار پیام ها توسط فرستنده (publisher) برای همان موضوع در یک کانال (channel) با متصل شدن به کانال (channel) می تواند پیغام ها را خوانده و پردازش کند.
- ✓ subscriber می تواند به **چندین کانال** (channel) وصل شود و محدودیتی وجود ندارد.
- ✓ **چندین** publisher می توانند متصل شوند به **یک** channel
- ✓ در این سیستم فرستنده و گیرنده کاملاً از هم جدا (decouple) هستند.
- ✓ Subscriber **پیام** هایی را دریافت می کند که به **محض وصل شدن** به channel از آن **لحظه** وارد channel می شود پیام های قبل از متصل شدن را نمی بیند.
- ✓ زمانی که publisher پیامی می فرسته به channel اگر subscriber متصل به channel نباشد **پیام ها از بین می رود**.



## Pub/Sub

✓ این ویژگی تقریباً همزمان با Redis متولد شد.  
✓ اما Pub / Sub برای اکثر موارد استفاده محبوب نیست.

چرا؟

✓ **بزرگ ترین مشکل** این است که پیام حداکثر یک بار تحویل داده خواهد شد. وقتی پیامی منتشر می شود، اگر مصرف کننده در حال حاضر آن را دریافت نکند، پیام **ناپدید** می شود. به علاوه، Redis روی پیام ها پافشاری نمی کند. اگر Redis خاموش باشد تمام پیام ها از بین می روند.



## Pub/Sub commands

### - Simple Syndication

- PUBLISH channel message
- SUBSCRIBE channel [channel ...]
- UNSUBSCRIBE [channel [channel ...]]

### - Patterned Syndication

- PSUBSCRIBE pattern [pattern ...]
- PUNSUBSCRIBE [pattern [pattern ...]]

### - Admin

- PUBSUB subcommand [argument [argument ...]]



What type of data can be sent in a message?

Text

Number

Binary



## Pub/Sub command

### PUBLISH

Time complexity:

$$O(N+M)$$

where N is the number of clients subscribed to the receiving channel and M is the total number of subscribed patterns (by any client).

یک پیام به کانال داده شده ارسال می کند.

## Pub/Sub command

### - PUBSUB

- CHANNELS [pattern]
- NUMSUB [channel-1 ... channel-N]
- NUMPAT

برای بررسی کانال هایی که در حال حاضر برای اشتراک در دسترس هستند استفاده می کنیم

## Pub/Sub command

### PUBSUB CHANNELS

#### Time complexity:

$O(N)$  که در آن  $N$  تعداد کانال های فعال و با فرض تطبیق الگوی زمانی ثابت است.

لیست کانال های فعال را بر می گرداند

An **active channel** is a Pub/Sub channel with one or more subscribers (excluding clients subscribed to patterns).

#### Syntax:

```
PUBSUB CHANNELS [pattern]
```

اگر هیچ الگویی مشخص نشود، تمام کانال ها لیست می شوند، در غیر این صورت اگر الگو مشخص شود تنها کانال هایی که با الگوی مشخص شده مطابقت دارند لیست می شوند.

## Pub/Sub command

### PUBSUB CHANNELS

```
client-1:6379> publish ch-2 3.1415
(integer) 1
client-1:6379> publish ch-1 adams
(integer) 1
client-1:6379>

2) "ch-2"
3) (integer) 2
1) "message"
2) "ch-2"
3) "3.1415"
1) "message"
2) "ch-1"
3) "adams"

client-3:6379> pubsub channels *
1) "ch-1"
2) "ch-2"
client-3:6379>
```

شما می توانید ببینید دو کانال فعال یکی ch-1 و دومی ch-2

## Pub/Sub command

### PUBSUB NUMPAT

Time complexity:

$O(1)$

Returns the number of **unique patterns** that are subscribed to by clients (that are performed using the PSUBSCRIBE command).

Syntax:

```
PUBSUB NUMPAT
```

Note that this isn't the **count of clients subscribed** to patterns, but the total number of unique patterns all the clients are subscribed to.

## Pub/Sub command

### PUBSUB NUMSUB

Time complexity:

$O(N)$ : که در آن  $N$  تعداد کانال های درخواستی است.

تعداد مشترکین (exclusive of clients subscribed to patterns) را برای کانال های مشخص شده بر می گرداند.

Syntax:

```
PUBSUB NUMSUB [channel [channel ...]]
```

توجه داشته باشید که فراخوانی این دستور بدون کانال معتبر است. در این حالت فقط یک لیست خالی بر می گرداند.



## Pub/Sub command

### PSUBSCRIBE

Time complexity:

$O(N)$  :  $N$  is the number of patterns to subscribe to.

**Subscribes** the client to the given patterns.

Syntax:

```
PSUBSCRIBE pattern [pattern ...]
```

## Pub/Sub

- PSUBSCRIBE pattern [pattern ...]

- ?
- \*
- [...]
- ^

### Supported glob-style patterns

- `h?llo` subscribes to `hello`, `hallo` and `hxlllo`
- `h*llo` subscribes to `hllo` and `heeeello`
- `h[ae]llo` subscribes to `hello` and `hallo`, but not `hillo`
- `h[^e]llo` subscribes to `hallo`, `hbllo`, ... but not `hello`

## Pub/Sub command

## PSUBSCRIBE

```
client-1:6379> publish ch-1 adams
(integer) 1
client-1:6379> publish ch-1 douglas
(integer) 2
client-1:6379>

2) "ch-2"
3) "3.1415"
1) "message"
2) "ch-1"
3) "adams"
1) "message"
2) "ch-1"
3) "douglas"

client-3:6379> psubscribe ch-?
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "ch-?"
3) (integer) 1
1) "pmessage"
2) "ch-?"
3) "ch-1"
4) "douglas"
```



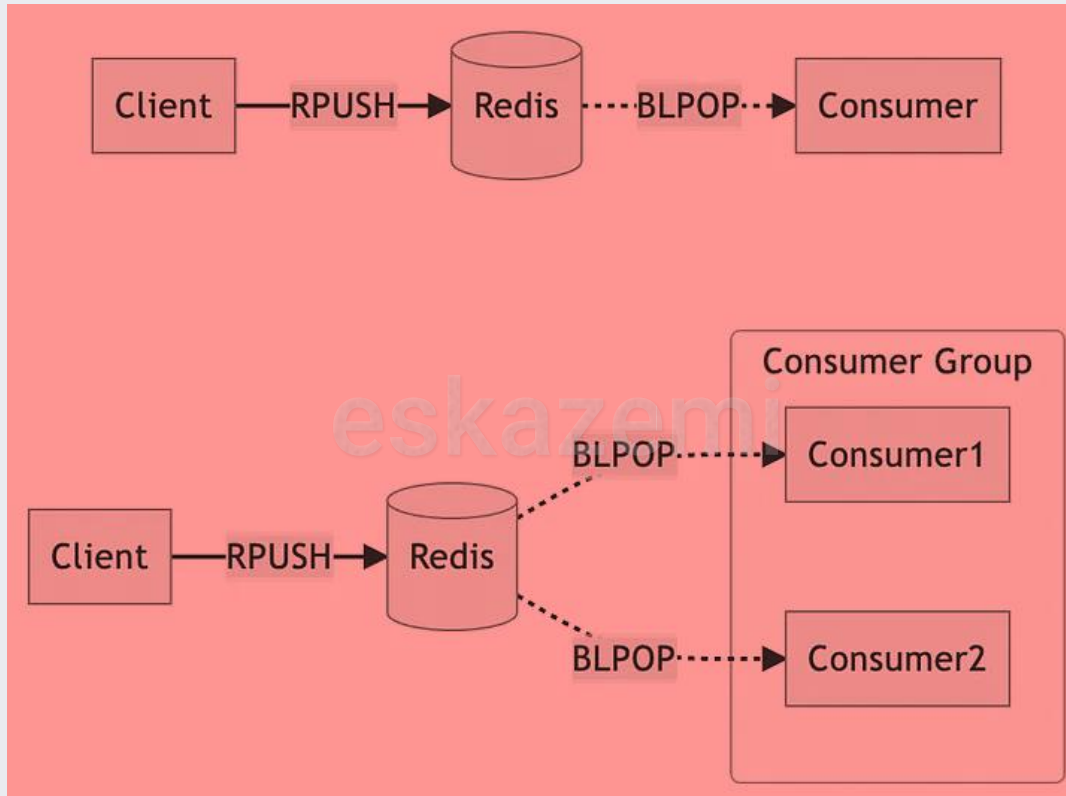
Let's summarize Pub/Sub:

- one-to-one and one-to-many are fine
- at-most-once -> No guaranteed delivery
  - no persistence
  - no consumer group
  - Performance ->
    - 1- Payload size
    - 2- Number of connected subscribers
    - 3- Number of patterns
- Use Cases -> Gaming & Chat





## List





## List

این لیست یک ساختار داده مفید در Redis است و ما می توانیم با استفاده از آن یک صف **FIFO** را به راحتی انجام دهیم. ترفند این است که می توانیم از **BLPOP** برای منتظر ماندن یک پیام در حالت بلاک استفاده کنیم. با این حال، اضافه کردن یک وقفه زمانی توصیه می شود.

باتوجه به شکل، می توانیم ببینیم که اگر مصرف کنندگان متعددی در انتظار یک لیست یکسان هستند، آن ها در حال تبدیل شدن به یک گروه مصرف کننده هستند. بدون پیکربندی هر چیزی، گروه مصرف کننده می تواند خود به خود توسط مصرف کنندگان شکل بگیرد. از سوی دیگر، **یک لیست نمی تواند یک پیام fan-out کند**. اگر یک پیام توسط مصرف کننده (consumer) **BLPOP** باشد، دیگر **نمی توان** این پیام را بازیابی کرد، حتی اگر پیام در آن مصرف کننده گم شده باشد.

با این وجود، لیست Redis می تواند پیام ها را در حافظه نگه دارد. علاوه بر این، اگر **AOF** یا **RDB** را فعال کنید، پیام ها می توانند در دیسک پشتیبان گیری شوند، توجه این رویکرد پایداری داده ها را کاملاً برآورده نمی کند.



## Let's summarize List:

- one-to-one is okay, but **no** one-to-many
  - at-most-once
- persist in-memory, and backup in-disk
  - consumer group works





## Stream

پس از معرفی Pub / Sub و List متوجه می شویم که هیچ کدام از این دو روش خیلی خوب نیستند. آن ها معایب خاص خود را دارند. بنابراین، **Stream** از **Redis نسخه 5** برای حل این مشکلات آمده است.

همان طور که دیدیم Redis انواع نوع داده ای دارد که می تواند برای رویدادها یا توالی های پیام استفاده شود. **Sorted set** ها **تشنه حافظه** هستند. همچنین انتخاب مناسبی برای **داده های سری زمانی** نیست؛ **زیرا ورودی ها می توانند جابه جا شوند**. لیست ها **fan-out** را ارائه **نمی دهند**، یک پیام به یک client تحویل داده می شود. **ورودی های لیست شناسه های ثابتی ندارند**. برای حالت **one-to-many**، Pub / Sub وجود دارد اما این یک مکانیزم "آتش و فراموشی" است. گاهی اوقات می خواهیم تاریخچه را نگه داریم، **range queries** انجام دهیم، یا پیام ها را پس از اتصال مجدد دوباره بدست آوریم (re-fetch). Pub / Sub **فاقد این ویژگی ها** است.



## Stream

از آنجا که **Stream** بسیار پیچیده تر است، بیایید ابتدا ببینیم **Stream** چه مزایایی به همراه دارد.

- one-to-one and one-to-many are fine
- at-least-once
- persist in-memory, and backup in-disk
- consumer group works

بنابراین **Stream مشکلات** Pub / Sub و List حل کرده و **Redis Stream** با استفاده از **radix trees** و **listpacks** ساخته شده است که آن را کارآمد می سازد در حالی که اجازه دسترسی تصادفی توسط شناسه ها را نیز می دهد.

## Stream



### Use Cases

برای ساخت سیستم چت ، message broker ها ، event sourcing بهتره از Redis Streams استفاده شود.  
هر سیستمی که نیاز به پیاده سازی گزارش یکپارچه دارد، می تواند از Streams استفاده کند.  
برنامه های صف مانند Celery می توانند از stream استفاده کنند



## Stream

**Asynchronous** : Producers and consumers need not be simultaneously connected to the stream. Consumers can subscribe to streams (push) or read periodically (pull).

**Deletion**: While events and logs don't usually have deletion as a feature, Streams supports this efficiently. Deletion allows us to address privacy or regulatory concerns.

**Capped Streams**: Streams can be truncated, keeping only the N most recent messages.

**Counter**: Every pending message has a counter of delivery attempts. We can use this for dead letter queuing.

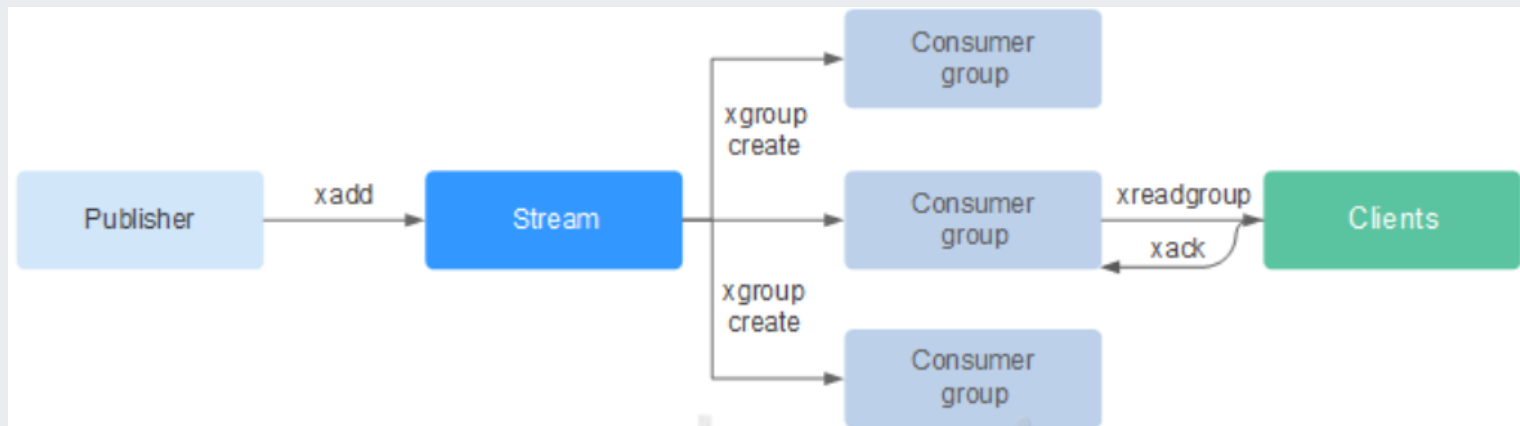
**Scale-Out Options**: Via consumer groups, we can easily scale out. Consumers can share the load of processing a fast-incoming data stream.

**Blocking**: Consumers need not keep polling for new messages.

**At-Least Once Delivery**: This makes the system robust.

**Persistent**: Unlike Pub/Sub, messages are persistent

## commands

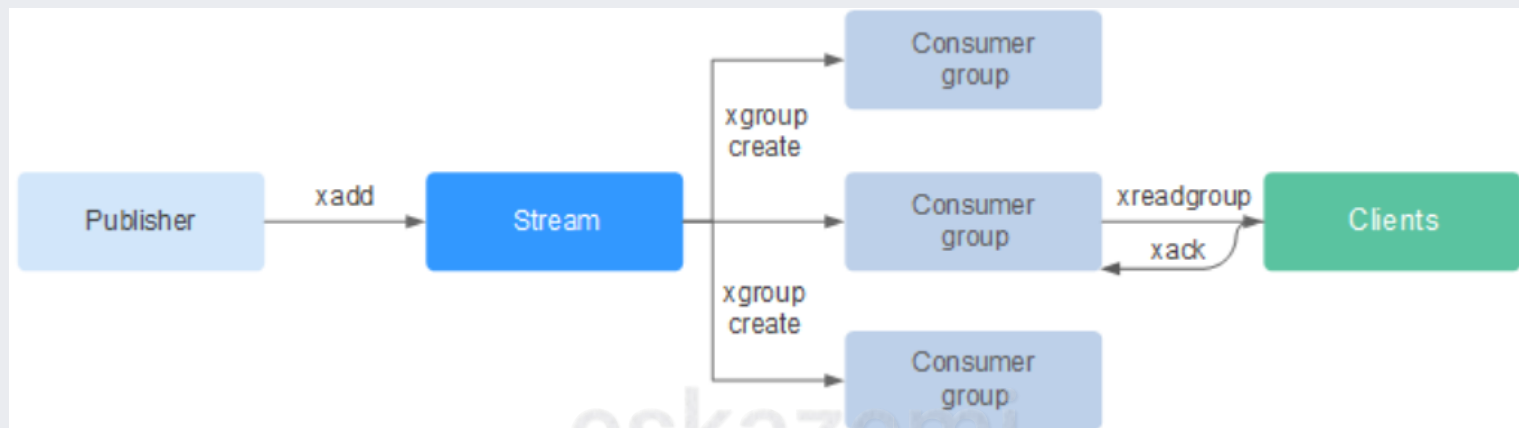


**Adding:** `XADD` is the only command for adding data to a stream. Each entry has a unique ID that enables ordering.

**Reading:** `XREAD` and `XRANGE` read items in the order determined by the IDs. `XREVRANGE` returns items in reverse order. `XREAD` can read from multiple streams and can be called in a blocking manner.

**Deleting:** `XDEL` and `XTRIM` can remove data from the stream.

## commands

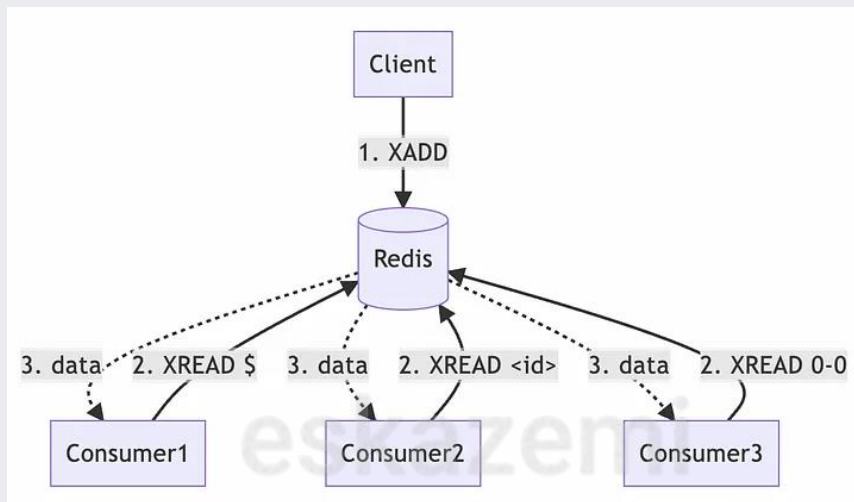


**Grouping:** XGROUP is for managing consumer groups. XREADGROUP is a special version of XREAD with support for consumer groups. XACK, XCLAIM and XPENDING are other commands associated with consumer groups.

**Information:** XINFO shows details of streams and consumer groups. XLEN gives number of entries in a stream.



## Stream



نمودار مانند **Pub / Sub** است، اما **جریان کار** به **List** نزدیک تر است. تولیدکننده (**producer**) می تواند در هر زمانی پیام تولید کند و سپس **XADD** را به **Redis Stream** ارسال کند. شما می توانید **Stream** را به عنوان یک لیست در نظر بگیرید که تمام پیام های ورودی را **حفظ** می کند. مصرف کنندگان همچنین می توانند پیام ها را در هر زمانی از طریق **XREAD** بازیابی کنند. شناسه در دستور **XREAD** نشان دهنده جایی است که می خواهید پیام را از آنجا بخوانید.

**\$:** No matter what messages are in Stream before, only retrieve from now on.

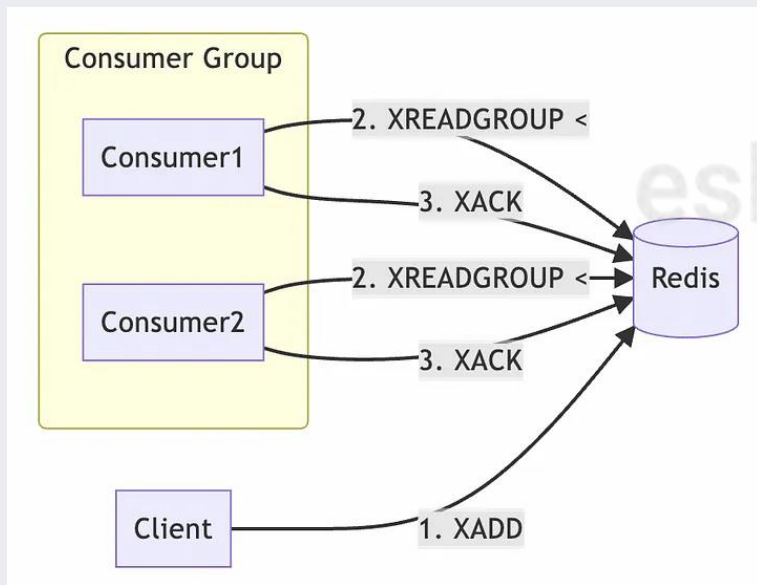
**0-0:** Always read from the head.

**<id>:** Start from the specific message id.



## Stream

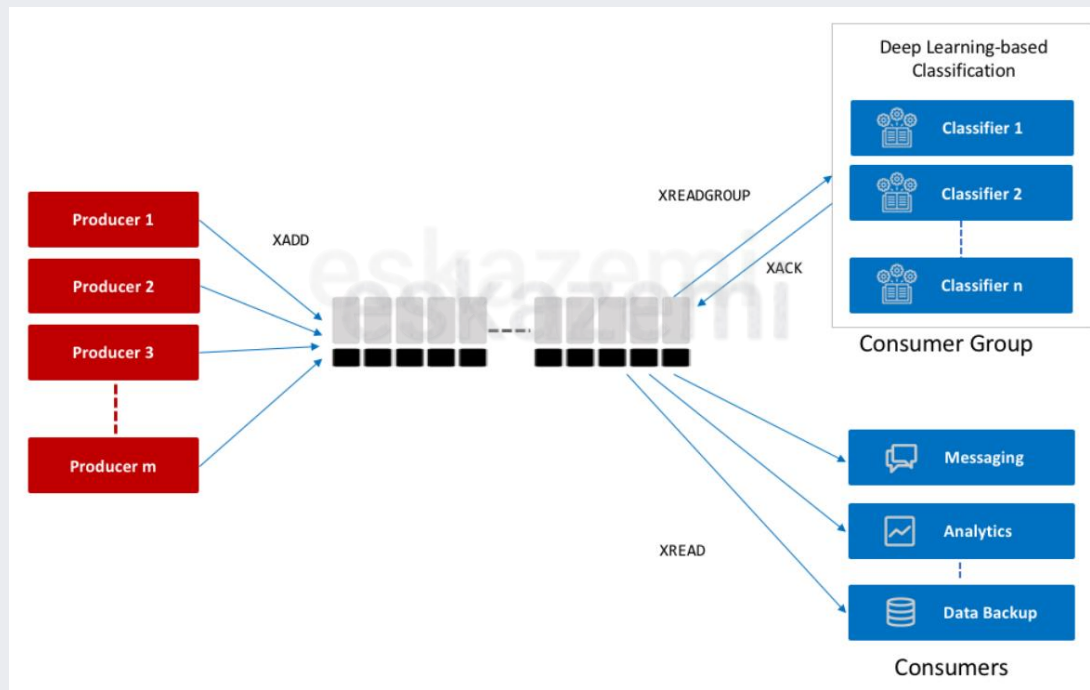
Apart from supporting one-to-one mapping, Stream supports consumer groups as follows:



برای دستیابی به تضمین at-least-once مانند اکثر سیستم های صف بندی، مصرف کننده (consumer) باید stream را پس از پردازش یک پیام با استفاده از XACK تایید کند.

استفاده از شناسه مخصوص < در اینجا برای شروع خواندن از موقعیتی است که هیچ کس در گروه آن را نخوانده است.

## consumer groups in Redis





## consumer groups in Redis

یک گروه مصرف کننده (consumer) به مصرف کنندگان (consumers) آن گروه اجازه می دهد تا وظیفه مصرف پیام ها از یک Stream را به اشتراک بگذارند. بنابراین، **یک پیام** در یک Stream می تواند تنها **توسط یک مصرف کننده (consumer)** در آن گروه مصرف کننده (consumer) مصرف شود. این کار ، بار برداش مصرف کننده برای پردازش تمام پیام ها را کاهش می دهد.

یک consumer همیشه باید خود را با یک نام **منحصر به فرد** معرفی کند.



## consumer groups in Redis

یک **Stream** می تواند **چندین گروه consumer** داشته باشد. هر گروه مصرف کننده (consumer) **ID** آخرین پیام مصرف شده (پردازش شده) را **ردیابی** می کند.

این **شناسه** توسط تمام consumer های گروه به اشتراک گذاشته می شود. هنگامی که یک مصرف کننده (consumer) پیامی را می خواند، شناسه آن به لیست ورودی های در انتظار (**PEL**) اضافه می شود. مصرف کننده باید تصدیق کند که پیام را با استفاده از دستور **XACK** پردازش کرده است. پس از تایید، لیست انتظار به روزرسانی می شود.

consumer دیگر می تواند با استفاده از دستور **XCLAIM** یک پیام در حال انتظار را دریافت کرده و پردازش آن را آغاز کند. این کار به بازیابی از شکست ها کمک می کند. با این حال، اگر قابلیت اطمینان بالا مهم نباشد، consumer می تواند از دستور **NOACK** از زیر مجموعه **XREADGROUP** استفاده کند.

## More details about IDs in Redis Streams

ورودی های یک Stream با استفاده از شناسه ها مرتب می شوند. هر ID دارای دو بخش است که با یک فاصله از هم جدا شده اند: **UNIX millisecond timestamp** و سپس شماره ترتیبی برای تشخیص ورودی های اضافه شده در همان زمان میلی ثانیه. هر بخش یک عدد ۶۴ بیتی است. برای مثال، ۱۵۲۶۹۱۹۰۳۰۴۷۴ - ۵۵ یک شناسه معتبر است.

زمانی که دستور **XADD** فراخوانی می شود، شناسه ها به صورت خودکار تولید می شوند. با این حال، یک client می تواند ID خود را مشخص کند اما باید یک ID بزرگ تر از تمام ID های دیگر در Stream باشد.

شناسه های ناقص زمانی هستند که بخش دوم حذف می شود. Redis با دستور **XRANGE** بخش دوم را به صورت قابل قبول برای ما پر خواهند کرد. با **XREAD**، بخش دوم همیشه -۰ است.

## More details about IDs in Redis Streams

برخی از شناسه ها خاص هستند:

\$: با XREAD برای بلاک کردن پیام های جدید، نادیده گرفتن پیام های موجود در Stream

+ & -: استفاده شده با XRANGE، برای مشخص کردن حداقل و حداکثر شناسه های ممکن در Stream. به عنوان مثال، دستور زیر هر ورودی در استریم را برمی گرداند:

```
XRANGE my stream + -
```

>: با XREADGROUP، برای دریافت پیام های جدید (که هرگز به client دیگر تحویل داده نمی شوند). اگر این دستور از هر شناسه دیگری استفاده کند، اثر آن برگرداندن ورودی های در انتظار آن client است.



## Stream Consumer Failover

حفظ نام‌های منحصربه‌فرد برای **Consumer** ها در یک سیستم توزیع‌شده می‌تواند چالش‌برانگیز باشد، به‌ویژه در سناریوهایی مانند اجرای مصرف‌کنندگان در کانتینرهای درون Kubernetes (K8s) که در آن scale-out و scale-in می‌توانند به صورت پویا رخ دهند. برای مقابله با این چالش، می‌توانید از ترکیبی از تکنیک‌ها برای اطمینان از پردازش جریانی قابل اعتماد و مقیاس‌پذیر بدون تکیه بر نام **Consumer** ها استفاده کنید:

همان طور که قبلاً توضیح دادیم **Redis Streams** یک ویژگی به نام **Consumer Groups** را ارائه می‌دهد که به چندین **Consumer** اجازه می‌دهد تا با هم کار کنند تا یک جریان را پردازش کنند. هر **Consumer** در یک گروه با یک شناسه منحصر به فرد شناسایی می‌شود که به طور خودکار توسط **Redis** اختصاص داده می‌شود. شناسه مصرف‌کننده به نمونه یا نام خاص مصرف‌کننده وابسته نیست، بلکه به خود گروه مرتبط است. این امکان **مقیاس بندی پویا** و **شکست را بدون نیاز به حفظ نام مصرف‌کنندگان** فراهم می‌کند.



## Stream Consumer Failover

**Dynamic Consumer Registration** : به جای تکیه بر نام های از پیش تعریف شده **Consumer** ، می توانید به صورت پویا مشتریان را در صورت آنلاین شدن در گروه **Consumer** ثبت کنید. هنگامی که یک نمونه مصرف کننده جدید شروع می شود، می تواند یک شناسه مصرف کننده منحصر به فرد ایجاد کند (به عنوان مثال، با استفاده از یک (UUID) و خود را در گروه مصرف کننده ثبت کند. به این ترتیب، گروه **Consumer** بدون نیاز به نامگذاری صریح، فهرستی به روز از **Consumer** های فعال را حفظ می کند.

**Consumer State Tracking** : برای رسیدگی به سناریوهای شکست، هر **Consumer** می تواند وضعیت خود، مانند آخرین شناسه پیام پردازش شده یا مهر زمانی را در مکانیزم ذخیره سازی دائمی (مانند Redis یا پایگاه داده خارجی) ردیابی کند. این به **Consumer** اجازه می دهد تا پردازش را از جایی که متوقف کرده است، از سر بگیرد، حتی اگر با شناسه **Consumer** دیگری دوباره راه اندازی شود.

**Consumer Acknowledgment** : Redis Streams مفهوم تایید پیام را ارائه می دهد. مصرف کنندگان می توانند به صراحت پردازش موفقیت آمیز یک پیام را با تایید شناسه آن تایید کنند. این تضمین می کند که پیام ها حتی در صورت خرابی یا راه اندازی مجدد، پیام ها از بین نمی روند. پیام های تایید نشده برای پردازش مجدد به سایر **Consumer** ها در گروه تحویل داده می شود. با استفاده از این تکنیک ها، می توانید به پردازش جریان قابل اعتماد و مقیاس پذیر در یک سیستم توزیع شده بدون تکیه بر حفظ نام **Consumer** دست یابید.

خوشبختانه، Redis روشی برای دریافت این پیام های در انتظار ارائه می دهند. جریان کار به این صورت است:



همه شناسه های پیام در انتظار را پیدا کنید.  
این شناسه ها را برای انتقال مالکیت مطالبه کنید.

بنابراین، گردش کار تکمیل شده در یک consumer bootstrap به صورت زیر است:

```
> XPENDING StreamName GroupName
```

```
> XCLAIM StreamName GroupName <ConsumerName in uuid> <min-idle-time> <ID-1> <ID-2> ... <ID-N>
```

min-idle-time یک روش بسیار مفید است. با استفاده از min-idle-time، می توانیم از چندین مصرف کننده که پیام های یک سان را در یک زمان مطالبه می کنند، جلوگیری کنیم. مصرف کننده اول برخی پیام ها را مطالبه می کند، بنابراین چنین پیام هایی دیگر بلااستفاده نخواهند بود. از این رو، دیگر مصرف کنندگان نمی توانند دوباره این پیام ها را مطالبه کنند.



## Redis Stream Persistence

Redis تضمین نمی کند که داده ها به هیچ وجه از دست نروند، حتی اگر سخت گیرانه ترین تنظیمات روشن شود. اگر از Redis به عنوان message queue استفاده کنیم، باید اقدامات بیشتری برای اطمینان از پایداری انجام دهیم. رایج ترین راه، **event-sourcing** است. قبل از انتشار یک پیام، ما این پیام را در ذخیره سازی پایدار مانند **MySQL** می نویسیم. مصرف کنندگان ما می توانند به طور کلی کار کنند. با این حال، اگر خطایی رخ دهد، ما همچنان می توانیم از پیام های پایدار در **MySQL** برای بازیابی کار خود استفاده کنیم.

علاوه بر این، اگر stream پیام های بیشتر و بیشتری دریافت کند، استفاده از حافظه رم یک فاجعه خواهد بود. اگر به دفترچه راهنمای Redis نگاه کنیم، می توانیم دستور **XDEL** را پیدا کنیم. با این حال، **XDEL** پیام ها را حذف نمی کند، بلکه تنها آن پیام ها را به عنوان استفاده نشده علامت گذاری می کند، و پیام ها هنوز وجود دارند.





## Redis Stream Persistence

چگونه می توانیم از نشت حافظه در رم جلوگیری کنیم؟ ما می توانیم از **MAXLEN** استفاده کنیم در حالی که **XADD** فراخوانی می کنیم . خط فرمان عبارت است از:

```
> XADD StreamName MAXLEN 1000 * foo bar
```

اما یک نکته وجود دارد که باید بدانید، **MAXLEN** روی عملکرد **Redis** بسیار تاثیر می گذارد. این برنامه برای مدتی فرآیند اصلی را مسدود می کند و در طول این مدت هیچ دستوری نمی تواند اجرا شود. اگر پیام های ورودی زیادی وجود داشته باشد و مقدار پیام های صف بندی شده به حداکثر برسد، Stream برای **حفظ** **MAXLEN** بسیار شلوغ خواهد بود.



## Redis Stream Persistence

یک رویکرد جایگزین می تواند اتخاذ شود. به جای رفع محدودیت هارد، می توانیم به Redis این حق را بدهیم که در زمان آزاد خود، طول مناسبی را انتخاب کنند. از این رو، دستور به صورت زیر خواهد بود:

```
> XADD StreamName MAXLEN ~ 1000 * foo bar
```

علامت ~ به این معنی است که حداکثر طول آن حدود ۱۰۰۰، ممکن است ۹۰۰ یا حتی ۱۳۰۰ باشد. Redis زمان خوبی را برای کنار گذاشتن اندازه مناسب برای آن انتخاب خواهند کرد.

## Stream Vs List Vs Pub/Sub

ویژگی	Stream	List	Pub/Sub
Complexity of seeking items	High ( $O(\log(N))$ )	low ( $O(N)$ )	low
Offset	Supported. Each item has a unique ID. The ID is not changed as other items are added or evicted.	List: Not supported. If an item evicted, the latest item cannot be located.	-
Data persistence	AOF and RDB files	AOF and RDB files	Pub/Sub: Not supported.
Consumer Group	supported	-	Pub/Sub: Not supported.

Comparing Redis Stream data type with other types. Source: Huawei Cloud 2020

## Stream Vs List Vs Pub/Sub

ویژگی	Stream	List	Pub/Sub
Acknowledgement	Supported	Pub/Sub: Not supported.	
Delivery	At-most-once	At-most-once	At-least-once
Eviction	Streams are memory efficient by blocking to evict the data that is too old and using a redix tree and listpack.	-List: Not supported. If an item evicted, the latest item cannot be located.	-
Scale-out	support	support	-
Fan-out	support	-	support



## Stream Vs List Vs Pub/Sub

**list** برخلاف دیگر فرمان های بلاک کننده Redis ، وقفه های زمانی اش بر حسب **ثانیه** است ، فرمان های XREAD و XREADGROUP در Stream وقفه های زمانی در حد **میلی ثانیه** . تفاوت دیگر این است که هنگام بلاک کردن در عملیات Pub/Sub و list ، اولین کلاینت در زمان رسیدن داده های جدید سرویس دهی خواهد شد. با دستور XREAD Stream ، هر client که **Stream** را مسدود کند، داده های جدید را دریافت خواهد کرد.





از دیدگاه من، این سه رویکرد مزایا و معایب خود را دارند با توجه به نیاز یکی از موارد باید انتخاب شود:

**Pub/Sub:** Best-effort notification.

**List:** Tolerate message queues with some data loss.

**Stream:** Loose streaming process.



# Thanks!



**Any questions?**

**You can find me at:**

- **@eskazemi**
- **m.esmaeilkazemi@gmail.com**



**eskazemi**