DATA SHEET

Getting Started with Redis Streams

Redis Streams is a simple, and yet powerful data structure for managing data streams. The data structure is built into Redis, the world's most popular in-memory database that delivers millions of operations per second at sub-millisecond latency with the fewest resources. Redis Streams creates a data channel that connects producers and a variety of consumers with different data needs.

Streams data structure offers:

- A rich choice of options to consumers to read streaming data and data at rest
- Consumer groups to help the consumers to coordinate among themselves while reading the data from the same stream
- Super-fast lookup queries powered by radix trees
- Automatic eviction of data based on the upper limit

Benefits:

- Collect large volumes of data arriving in high velocity (in the order of millions per second)
- Communicate between producers and consumers asynchronously
- Effectively manage consumption of data even when producers and consumers don't operate at the same rate
- Persist data when your consumers are offline or disconnected
- Scale-out the number of consumers
- Implement transaction-like data safety when consumers fail in the midst of consuming data

Redis Enterprise Advantages

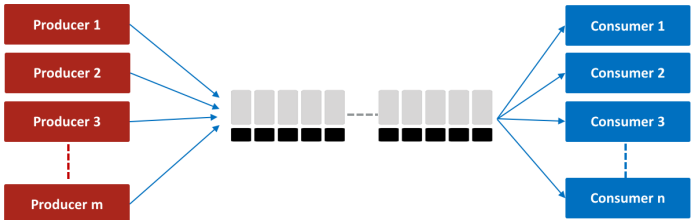
HA, Durability, DR
Tunable features for replication and persistence to maintain high performance when persisting data to disk; Rack-aware, cross-datacenter/ region/cloud in-memory replication with unique WAN compression technology.

Robust Security
SSL-based encrypted communication with clients, administrators and across clusters; certificate and password-based authentication; role-based authorization for administration.

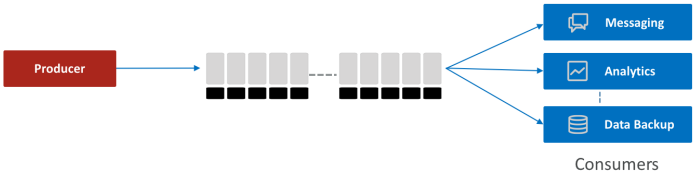
Redis on Flash
Supports Flash memory as a RAM extender; ideal for large dataset sizes; uses tiered memory access to deliver the same sub-millisecond latencies as RAM while reducing costs by 70% or more.

Automation and Support
24x7 enterprise-grade support backed by expertise in managing and scaling 550K+ Redis databases for thousands of customers in production worldwide.

Sample Use Cases:



Messaging: Connect producer and one or more similar consumers



Microservices: Multiple microservices consuming the same data for different purposes

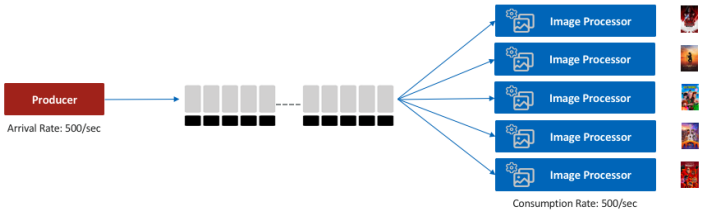
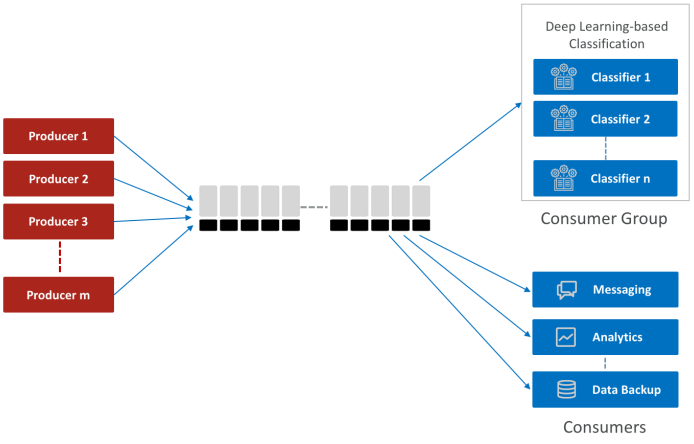


Image Processing: Scaling out consumers to catch up with the production rate



Artificial Intelligence: Many producers and consumers with varied data needs

Quick Reference Guide

Adding data to a stream

1. THE DEFAULT METHOD FOR ADDING DATA

```
XADD [stream name] * [key] [data]
```

Example:

```
XADD mystream * name Anna
```

2. ADDING DATA WITH USER-MANAGED IDS FOR EACH ENTRY

```
XADD [stream name] [id] [key] [data]
```

Example:

```
XADD mystream 10000000 name Anna
```

3. ADDING DATA WITH A MAXIMUM LIMIT

```
XADD [stream name] MAXLEN [length] * [key] [data]
```

Example:

```
XADD mystream MAXLEN 1000000 * name Anna
```

4. ADDING DATA WITH AN APPROXIMATE MAXIMUM LIMIT

```
XADD [stream name] MAXLEN [length] * [key] [data]
```

Example:

```
XADD mystream MAXLEN ~ 1000000 * name Anna
```

Consuming data from the stream via consumer groups

1. CREATE A CONSUMER GROUP

```
XGROUP CREATE [stream name] [group name] $ [MKSTREAM]
```

MKSTREAM is optional. It creates the stream if it doesn't already exist.

Example:

```
XGROUP CREATE mystream mygroup $ MKSTREAM
```

2. READ FROM A CONSUMER GROUP

```
XREADGROUP GROUP [group name] COUNT [n] [consumer name]
STREAMS [stream name] >
```

The special character ">" at the end tells Redis Stream to fetch only data entries that are not delivered to any other consumers.

Example:

```
XREADGROUP GROUP mygroup COUNT 2 Alice STREAMS mystream
>
```

3. ACKNOWLEDGE AFTER READING

```
XACK [stream name] [group name] [id]
```

Example:

```
XACK mystream mygroup 1526569411111-0
```

4. FIND THE MESSAGES IN THE PENDING LIST – CONSUMED BUT NOT ACKNOWLEDGE

```
XPENDING [stream name] [group name] - + [count]
[consumer name]
```

Example:

```
XPENDING mystream mygroup - + 10 Bob
```

5. CLAIM PENDING MESSAGES FROM ANOTHER CONSUMER

```
XCLAIM [stream name] [group name] [consumer name] [min
pending time] [id]
```

Example:

```
XCLAIM mystream mygroup Alice 0 1526569411113-0
```

Consuming data from the stream

1. READ EVERYTHING FROM THE BEGINNING OF THE STREAM

Situation: The stream already has the data you need to process, and you want to process it all from the beginning.

```
XREAD COUNT [n] STREAMS [stream name] 0
```

Example:

```
XREAD COUNT 100 STREAMS mystream 0
```

2. READ EVERYTHING FROM A POINT IN THE STREAM

Situation: The stream already has the data you need to process, and you want to process it all from a point.

```
XREAD COUNT [n] STREAMS [stream name] [id]
```

Example:

```
XREAD COUNT 100 STREAMS mystream 1518951481323-1
```

3. CONSUME DATA ASYNCHRONOUSLY (VIA A BLOCKING CALL)

Situation: Your consumer consumes and processes data faster than the rate at which it is added to the stream.

```
XREAD BLOCK [milliseconds or 0] STREAMS [stream name]
[last id + 1]
```

Example:

```
XREAD BLOCK 60000 STREAMS mystream 1518951123456-1
```

This blocks the call indefinitely:

```
XREAD BLOCK 0 STREAMS mystream 1518951123456-1
```

4. READ ONLY NEW DATA AS IT ARRIVES

Situation: You are interested in processing only the new set of data starting from this point in time.

```
XREAD BLOCK [milliseconds or 0] STREAMS [stream name] $
```

Example:

```
XREAD BLOCK 60000 STREAMS mystream $
```

5. ITERATE THE STREAM TO READ PAST DATA

Situation: Your data stream already has enough data, and you want to query it to analyze data collected so far.

```
XRANGE [stream name] [start id] [end id]
```

Example:

```
XRANGE mystream 1518951123450-0 1518951123460-0
```

With count:

```
XRANGE mystream 1518951123450-0 1518951123460-0 COUNT
10
```

With no lower or upper bound:

```
XRANGE mystream - + COUNT 10
```

Reverse of XRANGE is XREVRANGE

Example:

```
XRANGE mystream + - COUNT 10
```