The 6 Most Impactful Ways Redis is Used in Production Systems

BYTEBYTEGO
OCT 12, 2023 · PAID

Redis is often referred to as a Swiss Army knife - it's an incredibly versatile in-memory database that can help solve many different problems.

Let's say your online game is experiencing slow response times from your database due to rapidly increasing users. Or your e-commerce site needs to quickly display real-time product inventory for flash sales. Or your web analytics need to track page views at massive scale.

Redis to the rescue! Its lightning fast speeds and flexible data structures make it a go-to tool for many challenging use cases.

In this issue, we'll explore 6 popular Redis use cases including caching, session management, leaderboards, queues, analytics, and more. You'll discover techniques to supercharge your apps.

But first, let's do a quick refresher on Redis data structures we covered in our previous issue:

Basic Data Structures	Strings	Used for cache, counter, distributed locks, sessions	GET, SET, MGET, APPEND, SUBSTR, STRLEN
	Lists	Used for message queues	LPUSH, LPOP, LLEN, LMOVE, LTRIM
	Sets	Used for intersections, unions etc	SADD, SREM, SCARD, SISMEMBER, SINTER
	Hashes	Used for caches	HGET, HSET, HMGET, HINCRBY
	Sorted Sets (ZSet)	Used for ranking	ZADD, ZRANGE, ZREVRANGE, ZRANGEBYSCORE, ZREMRANGEBYSCORE, ZRANK
Added After Redis 2.2	Streams	Append only log for event sourcing, sensor monitoring etc	
	Geospatial	Store and query coordinates	
	Bitmaps	User daily login status	
Added Later - Probabilistic Data Structures for **Big Data**	HyperLogLog	Cardinality for big data	
	Bloom Filter	Check presence of an element in a set	
	Cuckoo Filter	Check presence of an element in a set	
	t-digest	Estimate the percentile of a data stream	
	Top-k	Find the most frequent items in a data stream	
	Count-Min Sketch	Estimate the frequency of an element in a data stream	

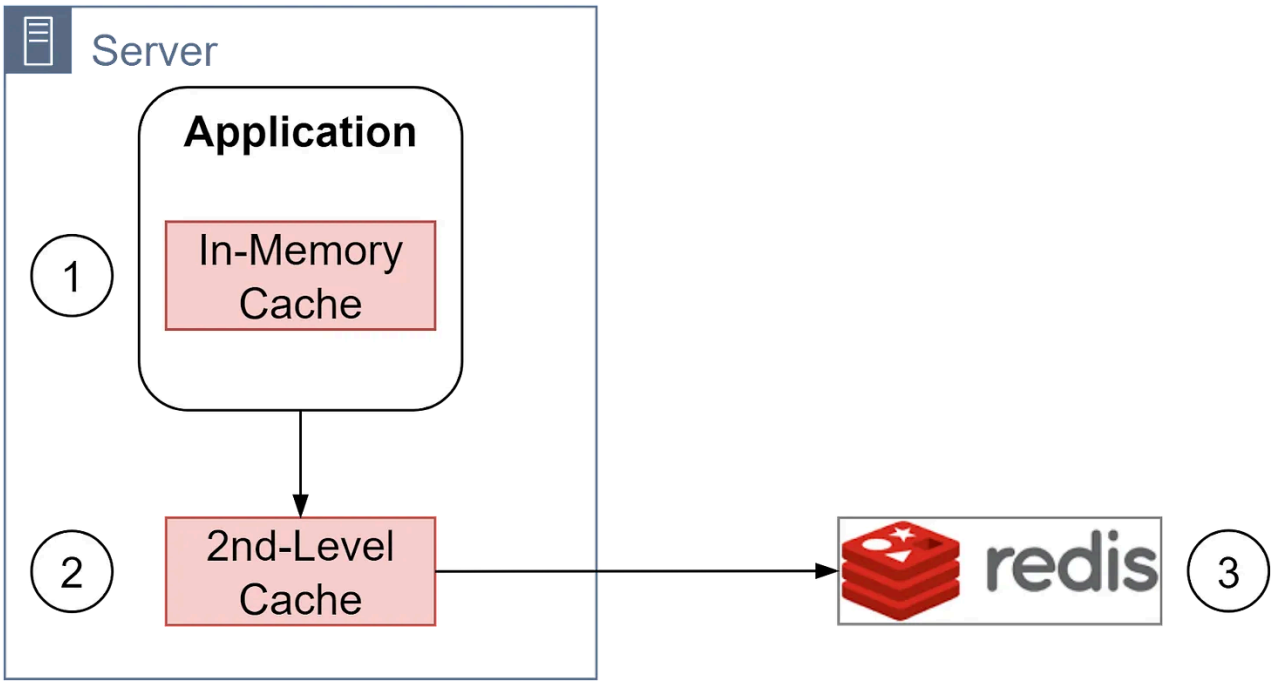
Now what makes Redis so fast? Unlike disk-based databases, Redis stores all data in memory and uses a single-threaded design. This allows Redis to achieve extremely high throughput and lightning fast speeds for reads and writes. The performance combined with versatile data structures like the ones above enables Redis to power many data-intensive real-time applications.

First up is caching.

Cache

A cache is an essential part of a system. It provides **a shortcut** to access hot data and improves performance. A typical cache architecture has three layers:

1. Application Cache: This sits inside the application's memory and is usually a hashmap holding frequently accessed data like user profiles. The cache size is small and data is lost when the app restarts.
2. Second Level Cache: This is often an in-process or out-of-process cache like EhCache. It requires configuring an eviction policy like LRU, LFU, or TTL based eviction for automatic cache invalidation. The cache is local to each server.
3. Distributed Cache: This is usually Redis, deployed on separate servers from the application servers. Redis supports different eviction policies to control what data stays in the cache. The cache can be sharded across multiple servers for horizontal scalability. The cache is shared across multiple apps. Redis offers persistence, replication for high availability, and a rich set of data structures.



Redis lets you cache different data types like strings for user's full names, hashes for user profiles, etc. However, the database remains the complete source of truth and holds the full set of data, while Redis caches the hot subsets.

Based on the Pareto principle, around 20% of data tends to make up 80% of accesses. So caching the hottest 20% of data in Redis can improve performance for a majority of requests. This 80/20 rule of thumb can guide what data is cached versus stored solely in the database.

The cache hierarchy allows managing different data sizes/access patterns efficiently. The first level cache holds a small volume of very hot data. The second level cache holds more data, still frequently accessed. The distributed Redis cache can hold large datasets by sharding across servers.

Using Redis as a cache improves performance but introduces complexity around cache coherence. There can be multiple copies of data, so read/write strategies need careful design. Typically one data source is designated as the "source of truth" and writes go there first. The application can implement lazy loading and write-through patterns when using Redis as a cache to keep it updated. Cache aside and read aside are other application-level caching patterns that Redis readily supports.

Caching is a classic time vs space tradeoff - we duplicate data across the system to gain speed. Interested readers can check our previous issues on caching best practices.

Based on the Pareto principle, 20% of the data in the system is mostly accessed, so this should be good guidance for the caching strategy.

Session Store

The Session Store is a critical component for web applications to maintain state across requests. Popular solutions like Redis provide a fast, scalable session store by keeping session data in-memory.

The server uses Redis to store session data and associate it with each user. It assigns every client a unique session ID that is sent on each request to retrieve the correct session. Storing sessions in Redis instead of locally on each app server removes the need for "sticky sessions" when load balancing.

Session data in Redis is serialized as JSON or similar format. This enables structured data to be stored like user profiles, recent actions, shopping carts, and CSRF tokens.

Sessions should expire after a period of inactivity. This practice improves security and frees up stale resources. The expiration time can be configured based on app needs.

The diagram below shows a typical Redis session flow:

Steps 1 and 2 - A user login request is sent to the User Service.

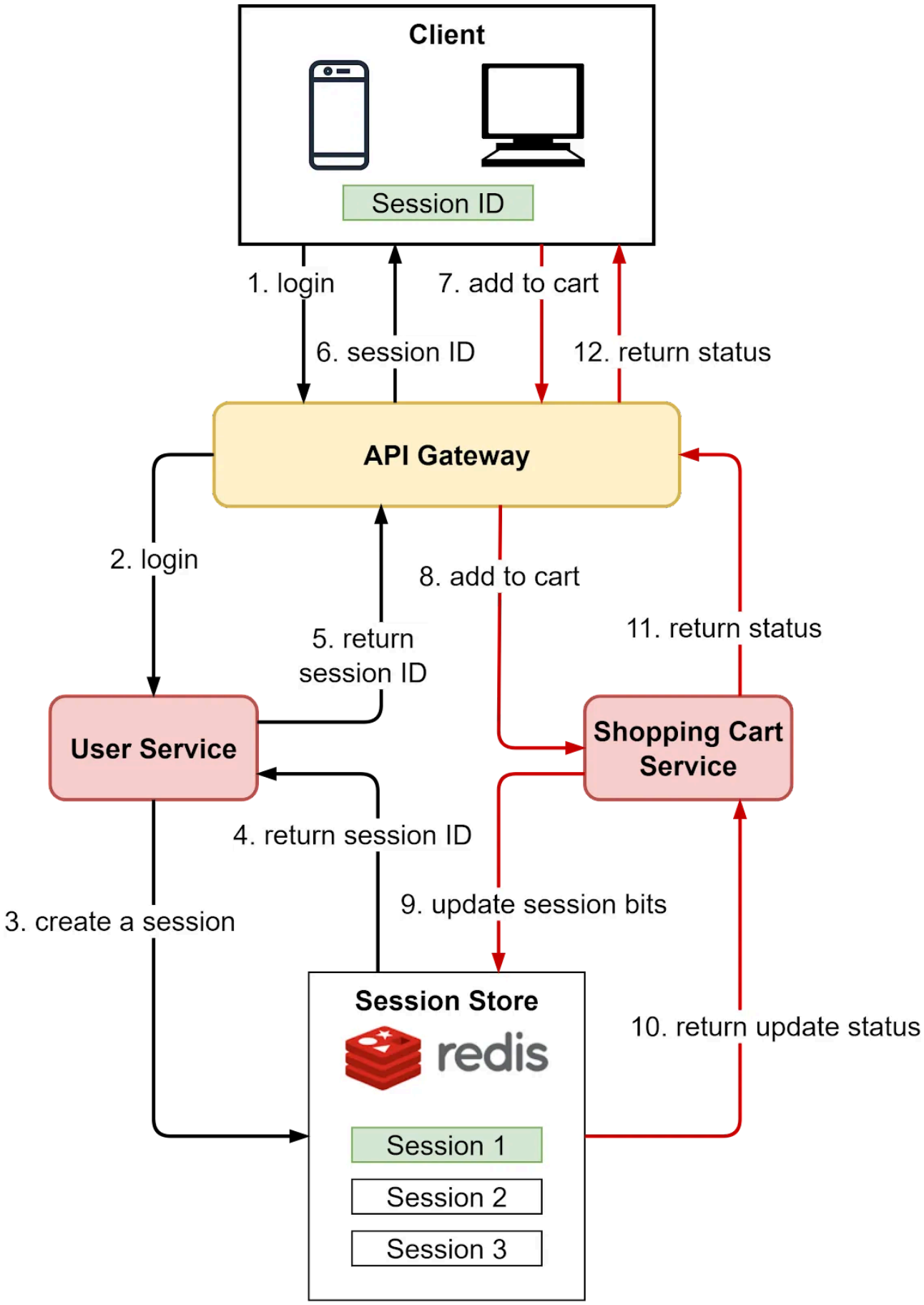
Steps 3 and 4 - The User Service creates a new session in Redis by generating a unique session ID.

Steps 5 and 6 - The User Service sends the session ID back to the client where it is stored locally.

Steps 7 and 8 - The user adds a product to their shopping cart. This sends the request to the Shopping Cart Service.

Steps 9 and 10 - The Shopping Cart Service retrieves the session data from Redis using the session ID. It updates the session object in Redis by adding the new shopping cart items.

Steps 11 and 12 - The Shopping Cart Service returns a success status to the client.



Leaderboard

Leaderboards are a common feature found in many applications today. For example, e-commerce websites often have leaderboards for top sale items, social apps may have leaderboards for step counts, and games need real-time score rankings.

Building highly performant leaderboards is challenging, especially when dealing with millions of users and scores that change rapidly in real-time. Redis' Sorted Set (ZSet) data structure provides an excellent way to implement fast, scalable leaderboards.

The ZSet maintains a collection of members sorted by score. It is implemented internally using a hash table and skip list to allow O(log(N)) time complexity for operations. Some key properties:

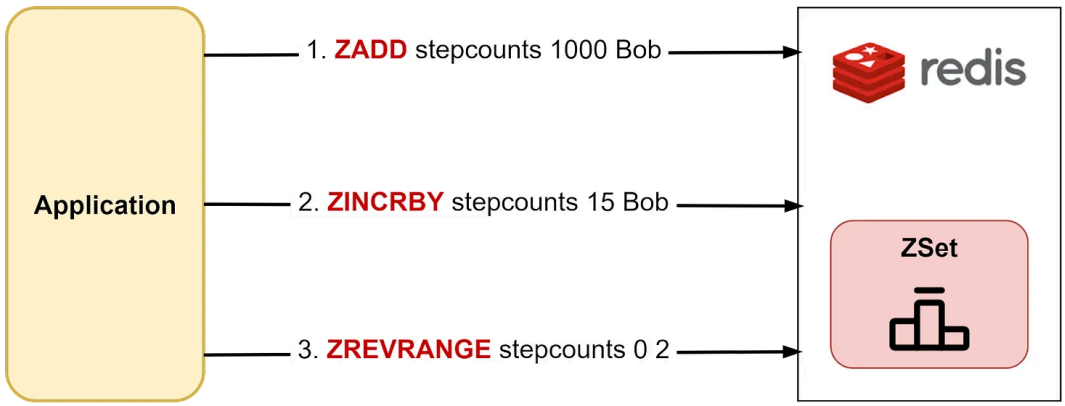
- Members are unique
- Each member has an associated score
- Members are sorted by score
- Supports common ops like add, remove, query range, rank, etc.

Let's look at how to build a real-time leaderboard for step counts using Redis ZSet commands:

Step 1 - Initialize leaderboard ZSet with ZADD, adding Bob with initial step count of 1000.

Step 2 - Increment Bob's steps by 15 using ZINCRBY on his existing entry.

Step 3 - Get top 3 step counts using ZREVRANGE to return high scores.



For most common leaderboard use cases, Redis ZSet provides an efficient structure that can handle real-time updates with low latency. At extremely high throughput

scenarios with tens of millions of concurrent users, the cost of frequent re-sorting on updates may not scale. In these massive scale cases, more customized solutions optimized for this workload would be required.

Message Queue

Dedicated message queue systems like ActiveMQ or Apache Kafka are common solutions for asynchronous messaging. However, introducing another middleware can add overhead and complexity. Redis provides lightweight pub-sub and queuing capabilities that can fulfill basic messaging needs, without adding another technology to the company's technical stack.

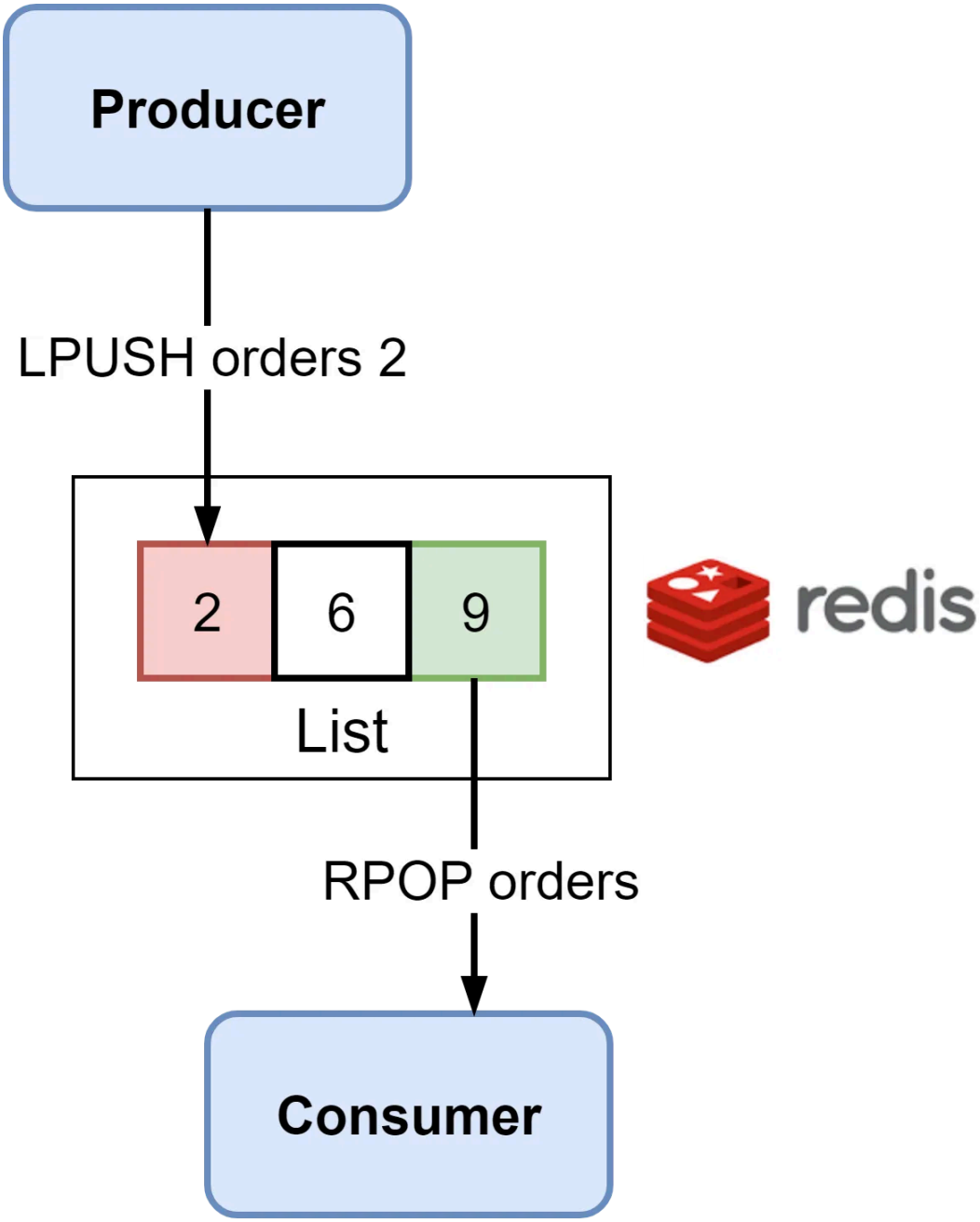
There are 3 approaches to implement message queues in Redis:

1. Use a List
2. Use Streams
3. Use Pub/Sub

Use a List

Redis Lists are simple FIFO (First In First Out) queues. Elements are added to the left side and consumed from the right side. The diagram below shows the details.

The producer uses the *LPUSH* command to add messages to the end of the List. Consumers use the *RPOP* command to retrieve messages from the front of the List.



Since Lists provide only basic queuing, consumers must poll Redis to check for new messages. This can waste CPU cycles. An alternative is using BRPOP, a blocking pop command, to wait for new messages efficiently.

When messages are consumed, they are removed from the List. To persist messages like Kafka, BRPOPPUSH can be used to add consumed messages to a backup List.

A downside is that slow consumers can cause messages to pile up in memory. Lists don't support consumer groups to parallelize message processing. So Lists work best for simple ephemeral queuing.

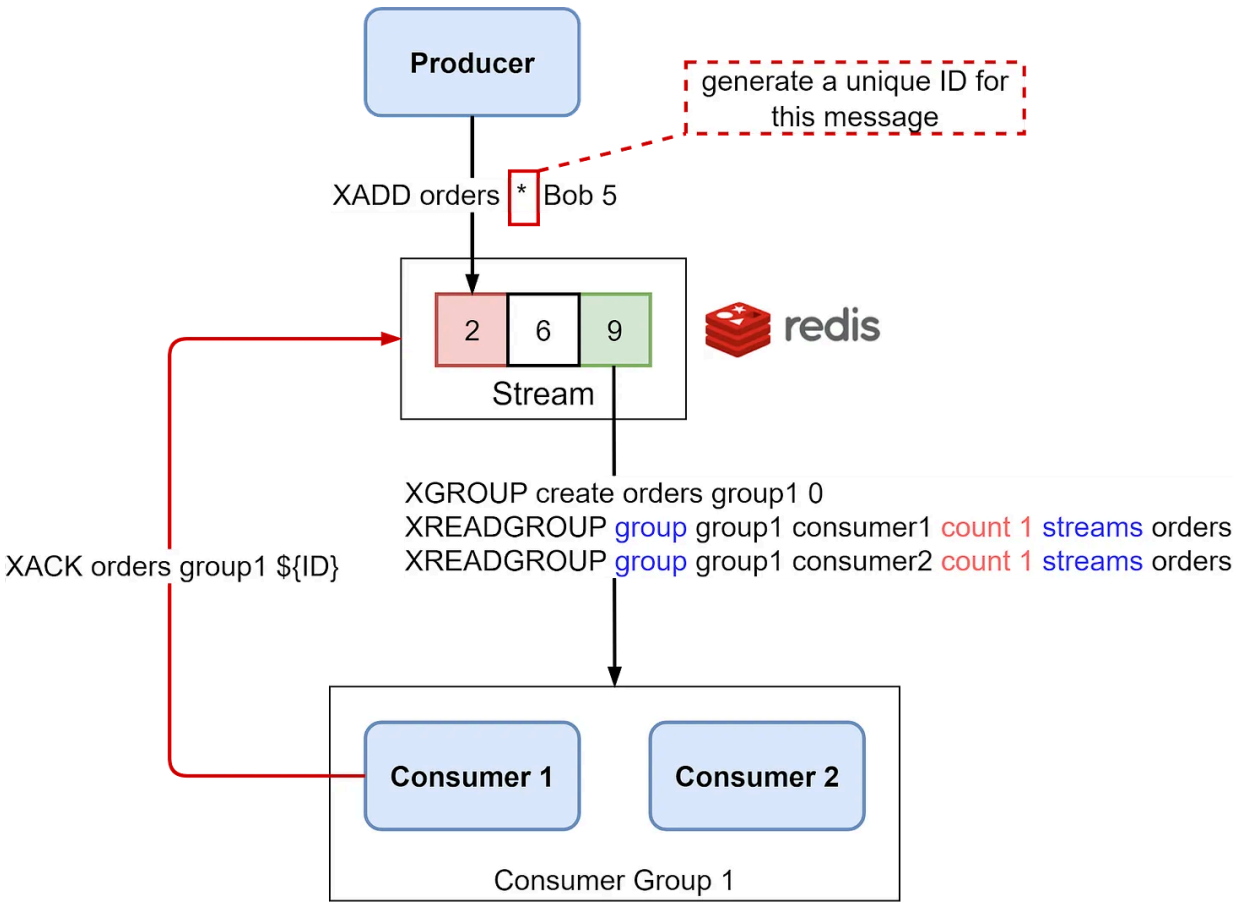
Use Streams

Redis Streams are append-only logs well-suited for event sourcing. Streams add better message persistence, ordering guarantees, and support for consumer groups compared to basic Lists.

The diagram below shows how it works.

Producers use XADD with a * parameter to append new messages to a Stream. The * tells Redis to automatically generate a unique ID for each message. If * is not provided, the user must specify their own unique ID to avoid duplicates. These IDs enable tracking message order, preventing duplicates, and acknowledging delivery after processing.

XGROUP creates a consumer group. XREADGROUP allows scaled-out consumption by multiple consumers in a group reading from the same Stream in parallel. For example, we can configure “Consumer 1” and “Consumer 2” in “Consumer Group 1”share the message load.



Consumed messages are temporarily staged in a pending entries list. Once acknowledged by the consumer via XACK, the message is removed from the list to

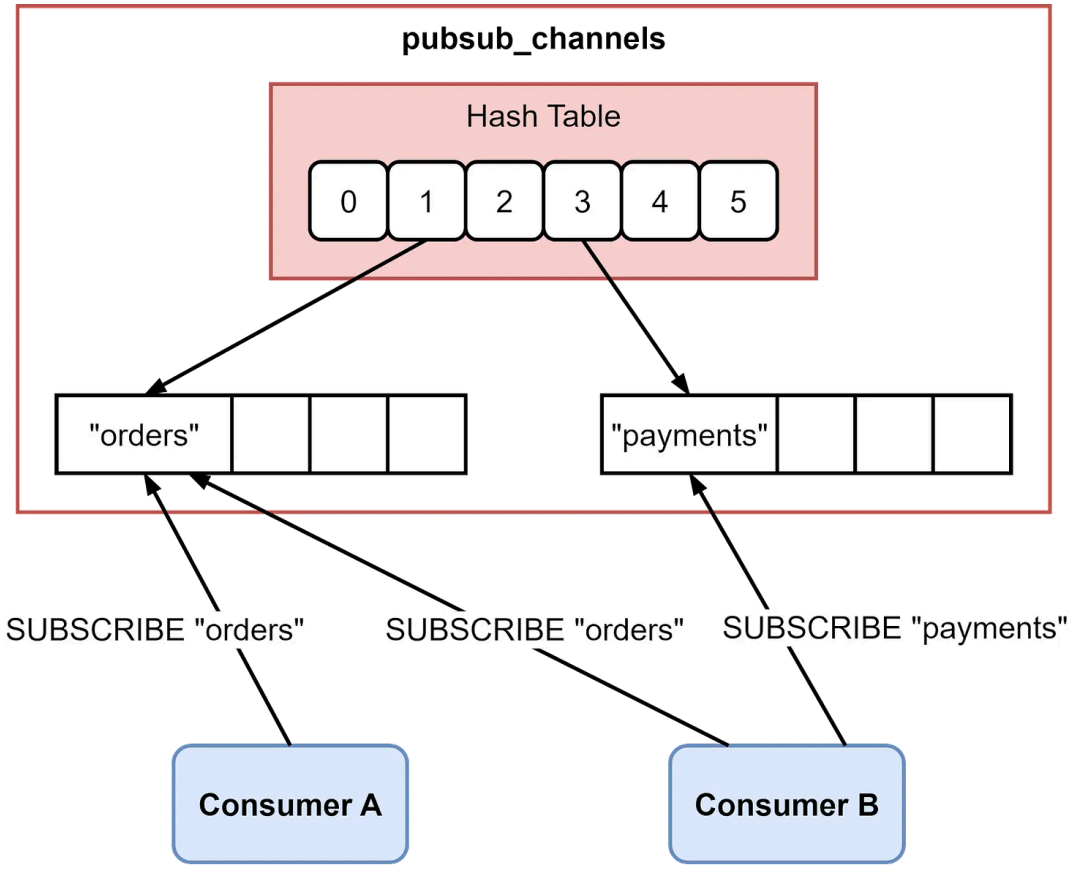
avoid redelivery.

With better delivery guarantees and throughput, Streams are a more robust queuing option compared to Lists.

Use Pub/Sub

Redis provides Pub/Sub messaging capabilities through the SUBSCRIBE, UNSUBSCRIBE, and PUBLISH commands. Clients can subscribe to channels using SUBSCRIBE, unregister with UNSUBSCRIBE, and send messages via PUBLISH similar to a topic-based system like Kafka.

The consumers can subscribe to one or more channels. If there are messages published to these channels, Redis pushes them to all the subscribed consumers.



Redis Pub/Sub can be used to build simple chat applications. Clients subscribe to user/chat channels and publish messages between themselves.

However, there are limitations compared to a robust brokered messaging system like Kafka:

- No persistence guarantees - Messages are ephemeral.
- No delivery assurances - Messages can be lost.
- No ordering guarantees - Messages may be received out of order.

Redis Pub/Sub supports basic fire-and-forget messaging but lacks reliability mechanisms like persistence, delivery assurance, and ordering guarantees.

Website Analytics

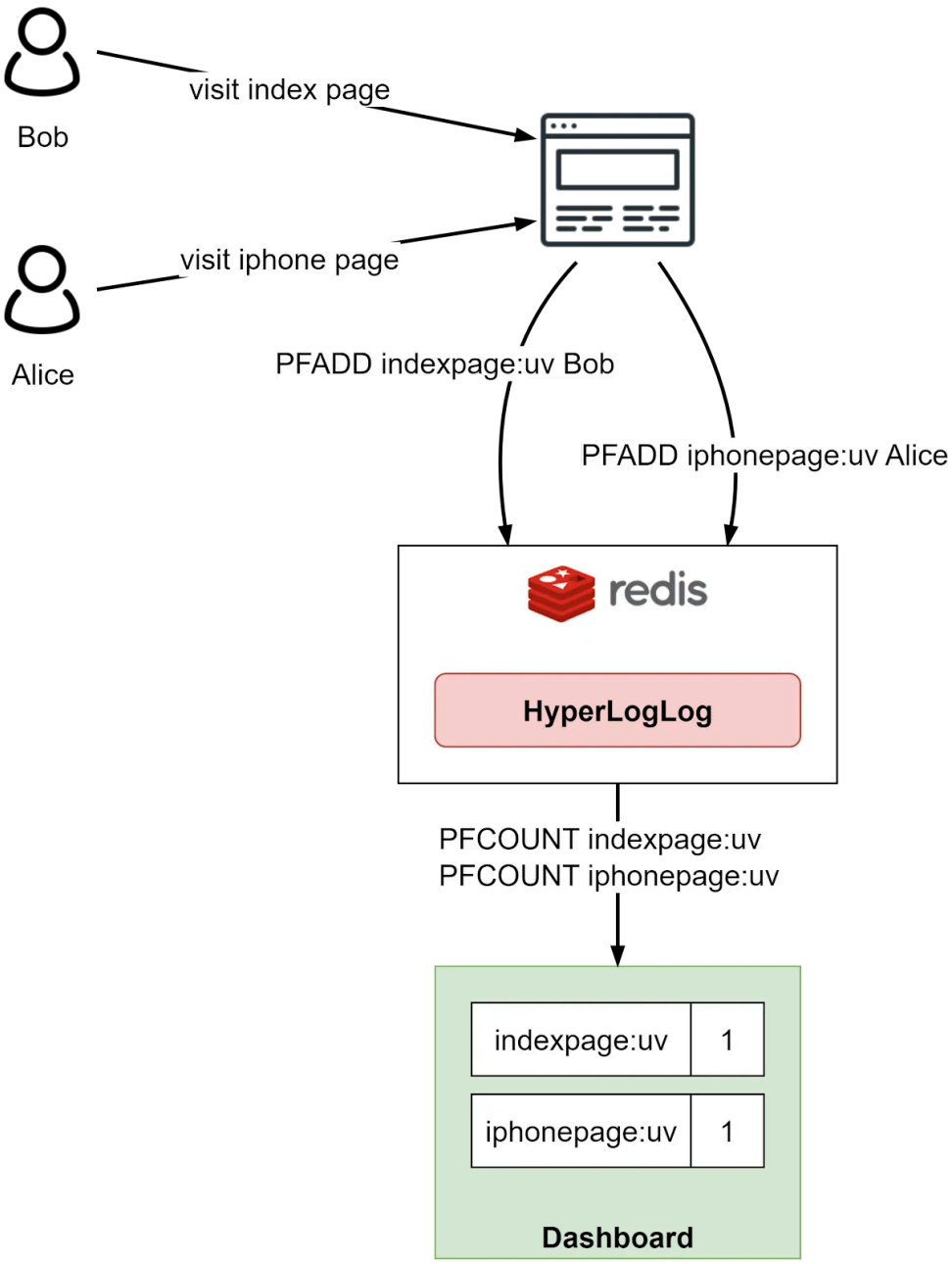
In internet applications, we often need to perform analytics on large amounts of data. For example, we may need to calculate the page views (PV) for certain web pages, or the number of unique visitors (UV) to those pages.

Counting PV and UV becomes challenging when dealing with massive user volumes - tens or hundreds of millions of visitors. Using a simple HashSet to store each user ID does not scale, as the memory footprint grows linearly with unique users.

Instead, we can leverage Redis' probabilistic data structures like HyperLogLog to efficiently estimate cardinality for PV and UV metrics.

The HyperLogLog data structure provides an approximate cardinality count using constant memory, irrespective of the number of elements added. By tuning the memory footprint, we can control the standard error - allocating more memory improves accuracy. But even at low memory, HyperLogLog can estimate massive cardinality with reasonable error. This allows tracking page views and unique visitors at web scale, while keeping memory consumption fixed.

Here's how it works:



To track page views:

- When a user visits a page, we PFADD their user ID to a HyperLogLog key mapped to that page.
- To get the estimated page views, we run PFCOUNT on the HyperLogLog key.

This allows tracking page views at massive scale while consuming minimal memory. A similar approach can be used to deduplicate and count unique visitors across pages.

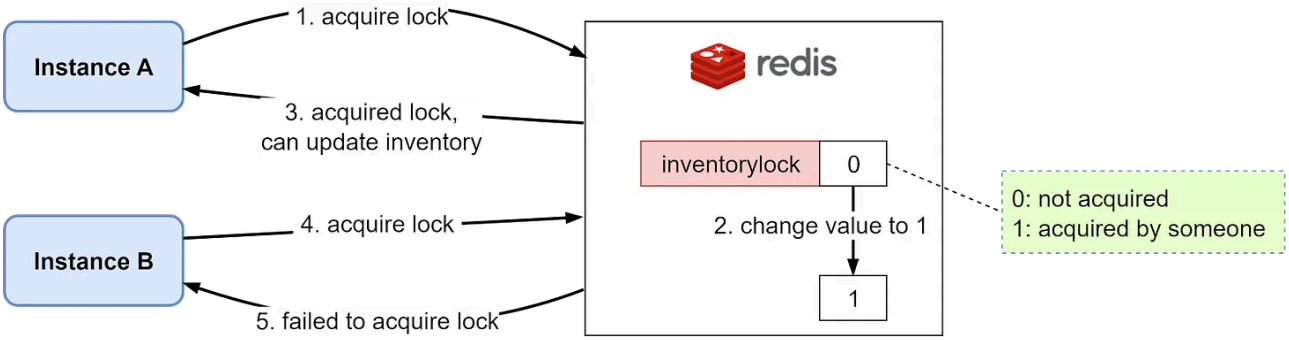
The tradeoff is we lose exact accuracy. But for many real-world use cases, a rough estimate within 1-2% error is sufficient. HyperLogLog provides a great way to enable

PV/UV analytics at internet-scale user volumes.

Flash Sale

In a flash sale, the limited quantity of each product leads to a large number of concurrent write requests to update the product inventory status. We can use Redis' key-value store to implement a distributed lock to coordinate these concurrent requests: only the instance that acquires the lock can proceed to update the inventory and must release the lock when done.

The diagram below illustrates how this works:



Suppose instance A wants to update the inventory, it tries to acquire the lock by setting the value of the *inventorylock* key to 1 if it's currently 0, indicating the lock is available. Since Redis executes all commands sequentially in a single thread, the requests from multiple instances are executed in the order they are received.

When instance B then wants to update the inventory, the lock has already been set to 1 by instance A, so B's request fails.

We can use the *SETNX* command to atomically acquire the lock if it's available.

```
SETNX inventorylock 1
// Update the inventory
DEL inventorylock
```

There are two potential issues with this simple approach:

1. The lock may never be released if instance A crashes before deleting the key. This prevents any other instance from acquiring the lock.
2. The lock held by Instance A could be released unintentionally by Instance B or another instance.

To address these, we can modify the approach:

1. Add an expiration timeout so the lock key automatically deletes after 10 seconds
2. Set a unique instance identifier value for the lock key. Only the lock owner instance can delete the key:

```
SET inventorylock instanceA NX PX 10000
// Update the inventory
```

This ensures only instance A can release the lock within 10 seconds. If the lock expires, any instance can retry acquiring it.

However, while this Redis-based approach works well for many use cases, it has some limitations at extremely high throughput scenarios:

- At massive scale with millions of concurrent requests per second across many servers, the cost of repeatedly acquiring and releasing locks in Redis can become a bottleneck.
- There is no fairness or ordering guarantee between contending clients - a lock request from one client can be starved indefinitely if the lock is continuously held by other clients.

So in high throughput systems requiring distributed locking, Redis may not be robust enough compared to dedicated lock services, but those are more complicated to operate.

Summary

In this issue, we discussed various Redis use cases including caching, shared sessions, leaderboards, message queue, analytics and flash sale.

Redis provides high performance and flexible data structures like strings, hashes, lists, sets, sorted sets, bitmaps, and hyperloglogs. This makes Redis a versatile tool to power many different kinds of real-time applications.

We hope this issue provided a useful overview of popular ways Redis is used in production across many applications. We'll cover more use cases including search, social media, etc. Stay tuned!

167 Likes · 10 Restacks

1 Comment

Write a comment...

kwan meng keong kwan's Substack Oct 18, 2023

how to get the previous post on redis ?

LIKE REPLY SHARE