Redis Can Do More Than Caching

BYTEBYTEGO
OCT 19, 2023 · PAID

108 5 2 Share

In the last issue, we explored common use cases with Redis. In this issue, we will go deeper and demonstrate how Redis' versatile data structures can power more complex applications like social networks, location-based services, and more.

We will walk through practical examples of building key features like user profiles, relationship graphs, home timelines, and nearby searches using Redis' native data types - Hashes, Sets, Sorted Sets, Streams, and Bitmaps.

Understanding these advanced use cases will provide you with a solid foundation to leverage Redis for your own systems and products. You will gain insight into how Redis enables real-time experiences beyond simple caching.

Social Media

Redis' flexible data structures are well-suited for building social graph databases, which power the core functions of Twitter-like social media applications. Relational databases can struggle with the complex relationships and unstructured data of user-generated content.

Redis provides high performance reads and writes to support features expected of social apps, allowing a small team to launch and iterate quickly. While Redis may not scale to the volumes of major social networks, it can power the first versions of an app through significant user growth.

Redis enables implementing common social media features like:

- User Profiles
- User Relationships (friends, followers)
- Posts
- User Interactions (likes, dislikes, comments, etc)
- Home Timeline

Let's explore how Redis supports these capabilities.

User Profiles

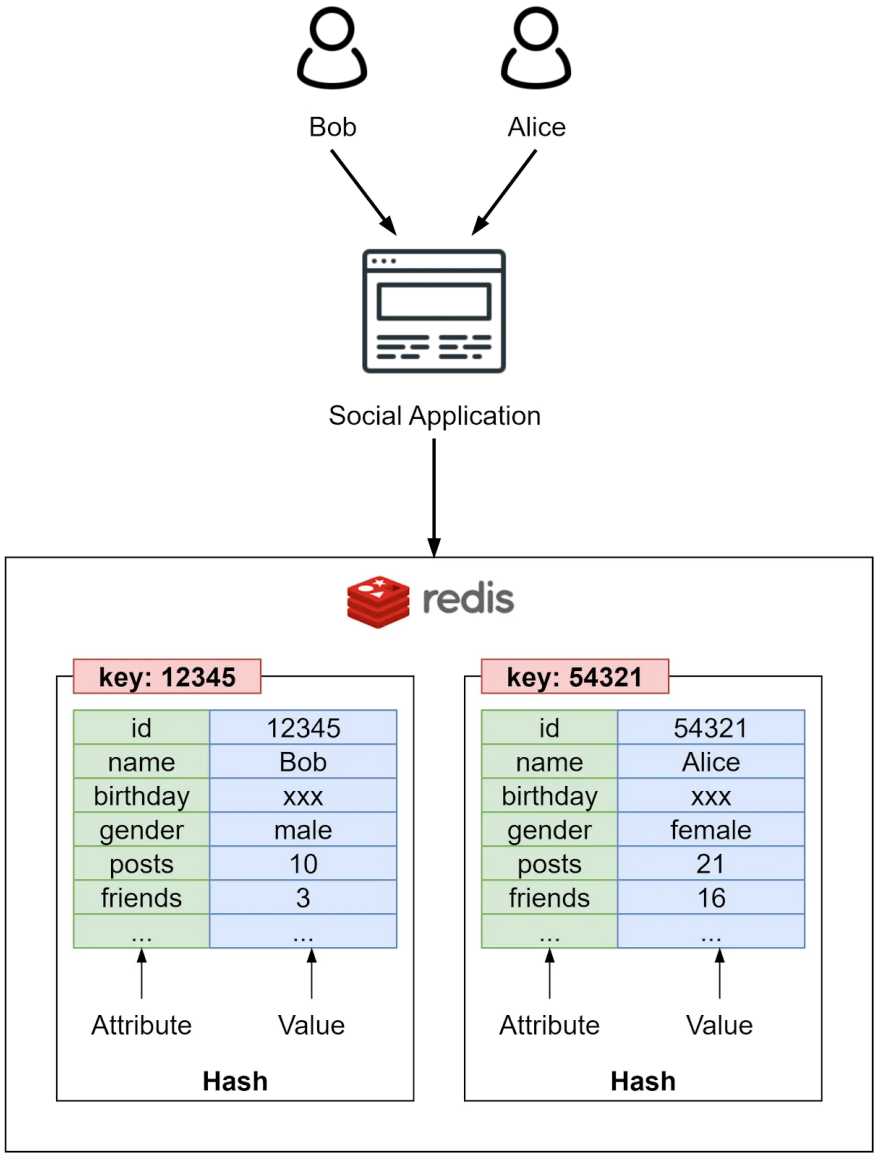
In social applications, a user profile stores identity attributes like name, location, interests, as well as preferences. We can represent each user profile as a Redis Hash, where the key is the user ID and the hash fields contain the profile properties.

For example, we can store user Bob's profile in a hash like:

```
HMSET user:bob name Bob location "New York" interests "photography,
hiking"
```

Compared to a relational model, Redis Hash provides flexibility to easily add new profile properties later without modifying the database schema. We just need to define how to retrieve and when adding more attributes to the user profile, because we don't need to go through database schema change.

In our application code, we would define how to retrieve and display the profile objects from these hashes. For example, we may only show name and location, or optionally include interests if present.



User Relationships

One of the major functions of a social application is establishing connections between users, like friend relationships or following others to receive their updates. Modeling these connections efficiently in a relational database can be challenging due to the complex graph-like structure of social networks.

Redis provides a more natural way to represent user relationships using its built-in Set data structure. The diagram below shows a comparison of modeling user relationships in a relational database versus using Redis Sets.

In the relational model, we use join tables to represent connections between users. Answering questions about relationships can involve complex SQL queries:

1. Retrieve all the people that Bob follows or all of Bob's friends
2. Retrieve Alice's friends of friends

3.

For example, to retrieve all of Bob's friends, we would need to query the join table like:

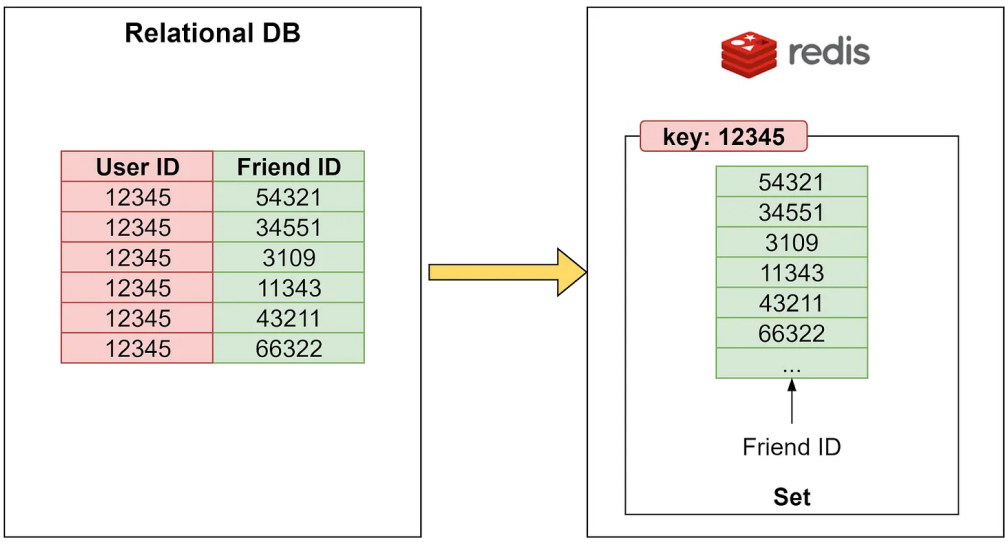
```
SELECT friend_id
FROM relationship_table
WHERE user_id = 12345
```

In Redis, we can store Bob's friend ids directly in a Set with his user id as the key. Retrieving Bob's friends is as simple as returning the members of the ZSet.

```
SMEMBERS {Bob's key}
```

Checking if Alice is in Bob's extended network of friends is also easier with Redis Sets. We can take the intersection of their Sets:

```
SINTER {Bob's key} {Alice's key}
```



By avoiding complex join queries, Redis Sets provide faster reads and writes for managing unordered social connections. The Set data structure maps naturally to representing simple relationships in a social graph.

Posts

In social apps, users create posts to share ideas, feelings, and status updates. Modeling this user-generated content can also be challenging in relational databases.

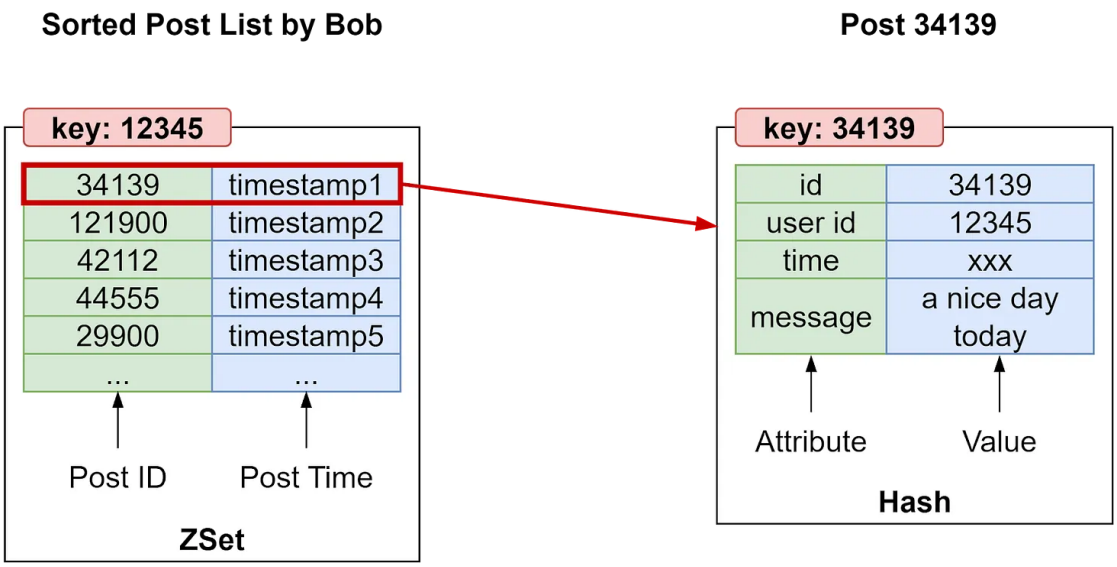
We can leverage Redis more efficiently here as well. For each user, we can store post_ids in a Sorted Set ordered by timestamp. The key can be the user id, and each new post_id is added as a member to the Set.

The post content itself is stored separately in Hashes, with the post_id as the hash key. Each Hash contains attributes like:

- user_id
- timestamp
- message
- etc

The diagram below shows how they work together. When a user creates a new post, we generate a new post_id, create a Hash to represent the post content, and add the post_id to the user's Sorted Set of posts.

This provides a natural way to model posting timelines - new post_ids are added to the tail of the Set, and we can page through posts ordered chronologically using ZRANGE on post_ids.



User Interactions

In social apps, users interact by liking, commenting, sharing posts, etc. Recording these events allows us to analyze usage patterns and feed data to recommendation systems.

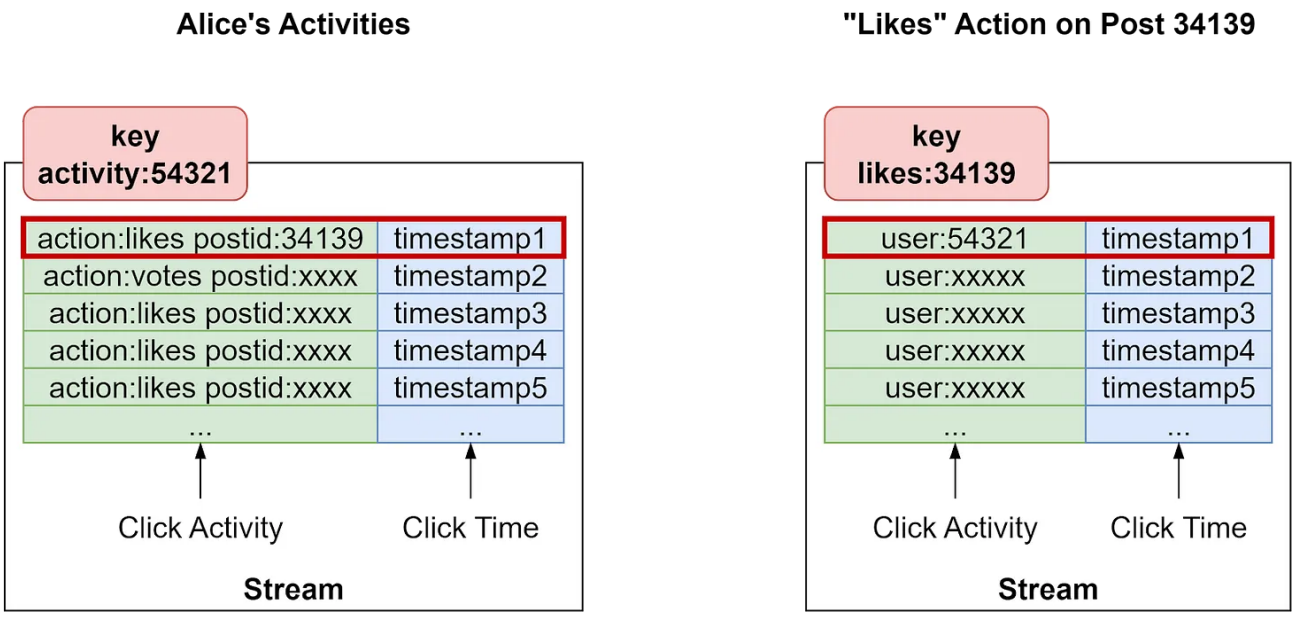
Modeling user interactions in relational databases can require complex schemas across multiple tables. Redis provides a more natural way to represent event streams with Redis Streams.

We can create two sets of Streams - one for user activity and one for post engagement.

When user 54321 (Alice) likes post 34139, we add it to the Stream for Alice's Activities and the Stream for the Likes on Post 34139:

```
XADD activity:54321 * action likes postid 34139
XADD likes:34139 * user 54321
```

The diagram below shows how the two streams are modified.

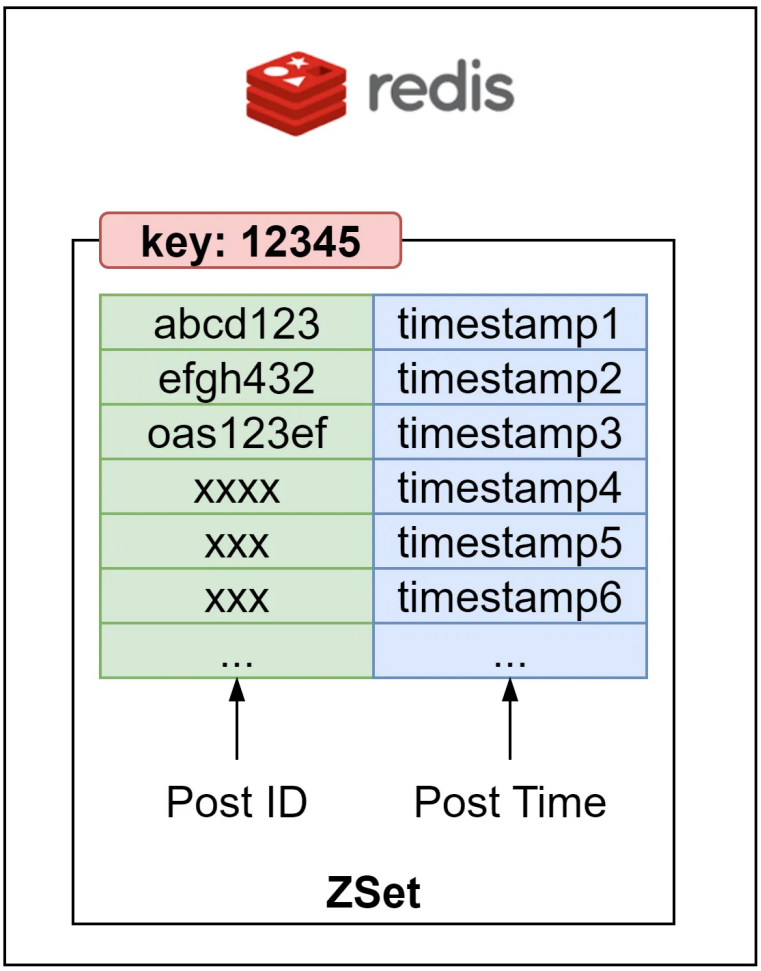


The streams act as persistent logs that can feed downstream processors for analytics. To manage the dynamic nature of streams created on-the-fly, one effective approach is pattern-based polling. Given the predictable naming convention of our streams, such as activity:<userID> and likes:<postID>, consumers can periodically execute SCAN commands combined with match patterns to discover newly created streams. By regularly polling with these patterns, consumers ensure that no streams go unnoticed and start consuming data from them as soon as they are identified. It's worth noting that while pattern-based polling is straightforward and suitable for many use cases, there are several other mechanisms available to handle dynamic stream discovery, depending on the specific requirements and scale of the application.

Personalized Feed

There is normally a home page on a social application for each user. The home page lists post updates, a.k.a. timeline, which is calculated based on recommendation algorithms. This page is the first page when users visit the application, and responsiveness is key to attracting the users' attention.

We can build a personalized feed in Redis using Sorted Sets. The diagram below shows how it works. The key is the user ID, and the value is a ZSet pointing to each post in the timeline. The score in the ZSet represents the order and priority of the posts, where we can sort tosts in descending order.



When a new post is created, we can calculate a relevance score based on factors like post creator, content, user preferences, etc. We add the post ID to the Sorted Set with this score. Higher scored posts will be ranked first in the user's timeline. As new posts come in, we can incrementally update the Sorted Set to maintain a fresh personalized feed.

The Sorted Set data structure provides an efficient way to model constantly updating timelines unique to each user. Fetching posts in timeline order simply requires a ZRANGE command. As scores change, posts automatically re-sort themselves in the Set.

While this Sorted Set modeling is straightforward, it does not scale to networks with millions of users and posts. Every new post requires updating potentially millions of Sorted Sets, one per user. This write amplification limits throughput. For massive scale, alternatives like feed pre-generation and caching may be required. But for most early stage social apps, Sorted Sets provide a simple and effective approach.

In conclusion, we can use Redis to quickly implement a simple social application. Redis' rich data structures support most of the functional requirements out of the box.

Now let's look at another common use case.

Tracking User State with Bitmaps

As users interact with a website or app, we often need to track their state, such as:

- Is the user currently online?
- Has the user signed in today?
- How many active users do we have right now?

It can be inefficient to store each user's state in individual variables. Redis Bitmaps provide a compact way to represent large sets of boolean flags.

For example, we can use the "online:20231019" bitmap, where the date signifies we are tracking online status for 10/19/2023. If user 12345 (Bob) is online, we call SETBIT:

```
SETBIT online:20231019 12345 1
```

To check if Bob is online, we simply use GETBIT:

```
GETBIT online:20231019 12345
```

Bitmaps enable fast boolean logic using operators like AND, OR, NOT, XOR. We can derive aggregate user state by combining multiple bitmaps.

For example, the following operation gets users active on Oct 18 and Oct 19:

```
BITOP AND active_on_both_days online:20231018 online:20231019
```

Bitmaps shine when tracking state for millions of users. Each user occupies just 1 bit, regardless of their actual 0/1 state. Bitmap size depends only on the number of users, not the distribution of bits.

However, high cardinality user IDs can result in large memory overhead. For example, tracking 1 billion users requires at least 128MB per bitmap. It is best to use sequential integer IDs to minimize the potential range in this use case.

Now let's look at a special data structure Redis offers, which is used in location-based service (LBS).

Location-Based Services

In ride-hailing applications or Yelp-like applications, we often need to find nearby taxis or nearby restaurants. Finding nearby friends is also an interesting feature for social apps. Redis provides a Geospatial data structure since version 3.2 to enable location-based services.

Let's look at an example of building a "nearby restaurants" feature. First, we load all the restaurants and their latitude/longitude coordinates into Redis:

```
GEOADD restaurants:open -110.331245 58.069327 pizza
GEOADD restaurants:open -110.331245 58.134221 steak
GEOADD restaurants:open -109.533987 55.039818 chinese
```

Behind the scenes, Redis stores these locations in a sorted set using a space-efficient structure called Geohash, which indexes map coordinates into strings for fast lookups.

To find nearby restaurants, we simply query by latitude/longitude and a radius. For example:

```
GEOSEARCH restaurants:open FROMLONLAT -110.331240 58.069310 BYRADIUS 1 km
WITHDIST
```

This returns the restaurants within 1 kilometer of the indicated lat/lng. The Geospatial data structure makes it easy to perform these location-based searches without complex application code.

Real-World Use Cases

Redis provides versatile data structures that enable many industry applications. Let's look at a few examples across fraud detection, gaming, and e-commerce.

Fraud Detection

BioCatch, an Israeli digital identity company, uses Redis as a system configuration database to capture behavioral data during active user sessions. This builds user behavior profiles in real time. The fraud detection service can then respond and make a decision within 40 milliseconds before substantial damages occur.

Gaming

In gaming, responsiveness is critical for good player experiences. As covered previously, Redis enables fast leaderboards, message queues, and caching. Scopely, maker of *The Walking Dead: Road to Survival* mobile game, uses Redis for many of these capabilities. Leaderboards track player rankings, queues manages background workload, and caching speeds up data access.

eCommerce

In eCommerce, shopping cart abandonment rate is a key website metric. To improve user experience while browsing products, placing orders, and paying, product inventory can be cached in Redis for fast access. Gap Inc. uses Redis to cache its inventory, achieving excellent performance even during seasonal peaks like Black Fridays.

If you are interested in the industry use cases, read more on this post.

Expanding Redis with Libraries

The raw Redis data structures provide powerful building blocks for many use cases. However, additional libraries can make development even more productive.

For example, the Java redisson library provides distributed data structures on top of Redis. This includes useful types like:

- SetMultimap - Key-value map with Set values

- ScoredSortedSet - Sorted set with floating point scores
- ConcurrentMap - Thread-safe concurrent map
- ListMultimap - Key-value map with List values
- BlockingQueue - Queue with blocking ops

These compose seamlessly with Redis while adding higher-level abstractions. The library handles serialization, network calls, and other details.

Developers can focus on their domain logic rather than lower-level distributed programming. Building custom types like aggregated leaderboards or relationship graphs becomes much simpler.

Other languages like Python, Go, and Node.js have similar libraries that extend Redis in productive ways. Search for "redis client library" plus your language to find options.

When evaluating libraries, look for ones that are actively maintained and have wide adoption. They should make working with Redis easier without sacrificing performance or control.

Summary

Redis is an incredibly versatile in-memory data store that goes far beyond basic caching. Its rich data structures and high performance enable real-time experiences critical for modern applications.

In this issue, we explored several advanced Redis use cases like building social graphs, tracking user state, location-based searches, and more. While other databases may also work, Redis provides intuitive data models and speed at scale.

Of course, every technology has tradeoffs. In-memory systems like Redis introduce persistence and consistency challenges. There is no one-size-fits-all solution. Systems should be designed based on business requirements and carefully weighed against pros and cons.

Hopefully this issue has provided a solid overview of Redis' capabilities and how it can be applied. Redis likely has a role to play in most modern application architectures, for caching, queues, pub/sub, and other peripherals. Its versatility and simplicity make Redis a powerful tool for any developer.

Thank you for following along on this Redis journey. We hope you walk away with new ideas to make your systems and products better using Redis.

108 Likes · 2 Restacks

5 Comments

Write a comment...

5 more comments...

© 2023 ByteByteGo · Privacy · Terms · Collection notice
Substack is the home for great writing