Redis for Real-Time Personalization

A PSEUDO CODE APPROACH TO IMPLEMENTING PERSONALIZATION WITH REDIS

redislabs
home of redis

Contents

redislabs
home of redis

The process of delivering a high value personalized experience at scale has become crucial to the success of so many applications and user experiences.

Redis and the Enterprise

For a certain class of developers, those who are involved in creating the highest performance applications on the Internet in a variety of domains such as ad-tech, gaming, ecommerce, fin-tech, and many others, Redis needs no introduction. It is well known as a high-scale data repository that has grown into a deep and wide platform to support the most challenging types of applications in existence.

In the enterprise, Redis is still emerging as the platform of choice to support solving the most challenging problems. Adoption continues to grow because Redis has new and better solutions to problems that have traditionally been solved in other ways.

Part of the mission of CITO Research is to find important technologies for the Early Adopters in the enterprise who are seeking to implement high value use cases.

CITO Research has worked to create a new form of explanation, a pseudo code walk through of a real-time personalization application, an example that will bring the value of Redis for enterprise use cases clearly into focus.

If you are wondering how can Redis help your business applications or if you are a Redis expert and advocate who wants to promote wider use of Redis, we hope this content is useful for you.

The Personalization App Implemented in Redis

The process of delivering a high value personalized experience at scale has become crucial to the success of so many applications and user experiences. But the amount of data required from a wide variety of sources, the need to combine recent and historical data layers into a unified whole, and the process of extracting signals from the increasingly larger amounts of data that arrives in high velocity all represent key challenges. In addition, the number of ways to process the data using various forms of analytics, predictive models, machine learning, and AI to adapt to a customer's desires, along with ways we can offer personalization, have all grown dramatically. At the same time, customers expect instant and up-to-date responses.

Being able to create applications and elegantly orchestrate components to address these requirements in a way that delivers high performance is the challenge of real-time personalization.

These requirements represent a massive undertaking for application developers.

Explaining the power of a platform isn't easy. Most of the relevant vendors make the same claims. We feel it is crucial to show in a concrete way how Redis Enterprise technology allows applications to deliver a powerful and speedy form of real-time personalization.

We believe that this document will provide the needed level of explanation. In this pseudo code description of how Redis can be used for real-time personalization, you will find:

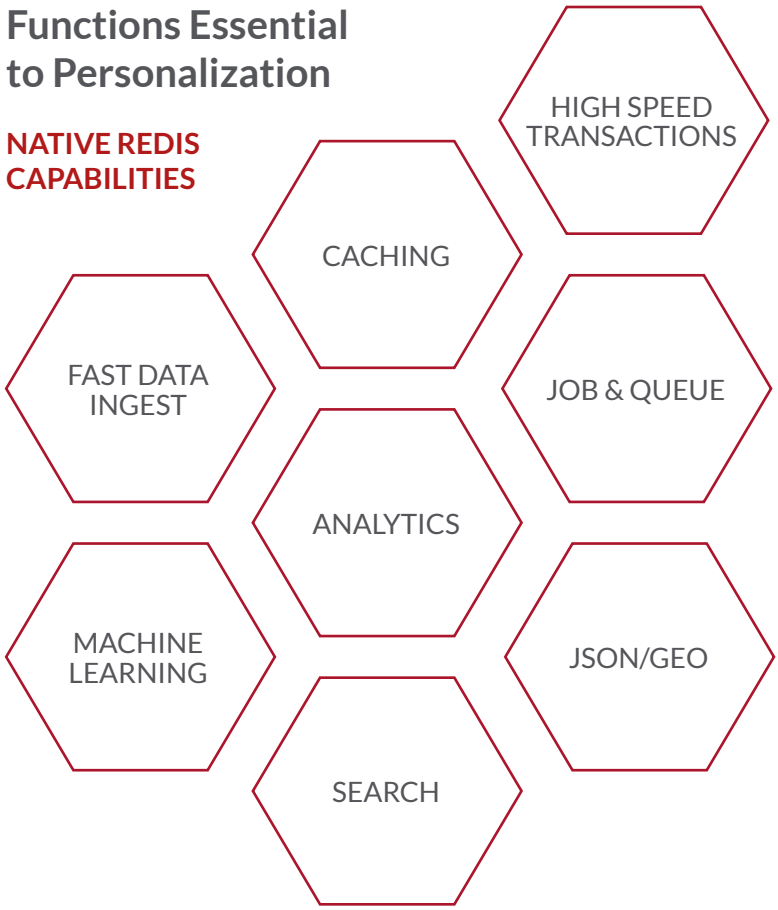
- A description of a generic architecture for real-time personalization that explains the flow of data and application functionality, and highlights important Redis calls, methods, and data structures.
- An annotated set of pseudo code that illustrates how elegantly and compactly real-time personalization can be written in Redis.
- Commentary on the aspects of the Redis enterprise (Redis[e]) platform crucial to supporting real-time personalization.

Capabilities That Support Real-Time Personalization

Personalization is about knowing your customer so that you are able to provide them with an experience that keeps them engaged with your business. The experience you provide should be tailored to each customer, and doing this requires collecting, analyzing and accessing a multitude of data ranging from a simple click to a customer's location. While the task can be daunting, using a versatile platform like Redis[e] lets you focus on building that real-time experience rather than worrying about whether the website will perform when everyone arrives at scale.

Functions Essential to Personalization

NATIVE REDIS CAPABILITIES



The following sections describe each of these capabilities.

Fast Data Ingest

Native High Performance coupled with PUB/SUB

You can't use yesterday's data for today's users.

Personalization relies on knowing what is happening right now. That means using a current stream of data, and not yesterday's data.

One example of Fast Data Ingest is capturing all user interactions in real-time when someone visits your website. You may have hundreds of thousands of users on your website clicking or scrolling on various pages, as well as spending time reading articles and blog posts, filling in a customer survey, reading or writing reviews, or using interactive chat features with salespeople. You want to be able to capture all of this data, but the amount of data that comes through your website can almost be unmanageable without the right tools. Being able to process everything you collect is a challenge, and Redis can keep up with the volume with very few resources.

In addition to using Redis to handle a very large volume of "writes" data, you can use the PUB/SUB functionality in Redis to trigger different actions or notify processes about the actions a customer is taking. If users get coupons after viewing 10 product videos, for example, you can use PUB/SUB to publish the user's completion of these actions to the coupon service, which subscribes to the action counting service. Once notified, the coupon service knows to serve that user with a coupon, exactly the kind of reaction to user behavior that is the heart of personalization.

TO LEARN MORE

- The NoSQL benchmark - Redis outperforms other NoSQL platforms by up to 8 times https://redislabs.com/docs/nosql-performance-benchmark/
- Redis delivers millions of writes/second with 2 servers and >99% cost savings over other products https://cloudplatform.googleblog.com/2015/04/a-guy-walks-into-a-NoSQL-bar-and-asks-how-many-servers-to-get-1Mil-ops-a-second.html

Caching/Session Store

Intelligent, policy-rich caching and session state management for any data

Personalization relies on understanding as much as possible about a customer. The best systems know what the customer did last year or for many years, as well as last month, last week, yesterday—and what he or she did in the past few minutes.

The amount of data that creates a complete picture of a customer comes from many sources, but must also be available in an instant to support real-time personalization.

Personalization is about understanding as much as you can about a customer so that you can provide that customer with a tailored experience. If you know what that customer did 10 seconds ago, your stream of data is very current. The caching layer serves personalized content with sub-millisecond latency.

There are different ways that the caching layer helps to personalize a website. If you don't know anything about a visitor, at a minimum, you can figure out her location by knowing the link she used to access your site. From this, you know right away whether to display the US or an international version, for example.

Also, as the caching layer collects the most recent items viewed, you can use this information to present offers to a customer after they've visited your site. You know what they're interested in and what you can entice them to purchase.

Any real-time personalization must have a powerful cache that can do two things:

- Assemble information from a variety of heterogeneous sources into a data structure that represents a rich picture of the user.
- Deliver that information to the personalization routines in milliseconds.

Being able to handle scale is key because everyone can arrive at your site at once, and being able to access this data instantaneously is integral to personalization. What makes Redis special is its very high speed and very large scale. Redis can handle a variety of data and serve up content of any type, whether that's metadata or images, for example. Redis also has numerous policies that allow you to expire content, like when your session expires or a user doesn't close their browser. The content eviction, algorithms for expiration and notifications for when something expires are all built into Redis, making it a very powerful caching layer.

TO LEARN MORE

- Redis for caching
  https://redislabs.com/solutions/use-cases/redis-for-caching/
- 15 reasons caching is best done with Redis
  https://redislabs.com/docs/15-reasons-caching-is-best-done-with-redis/

High Speed Transactions

ACID with tunable consistency and durability

High speed transactions require ACID controls to ensure that a set of operations are executed as a complete transaction — all or nothing. Including the right controls for data durability and consistency is a key requirement that is important in many scenarios, including payments, shopping cart interactions, order fulfillment, and customer service.

TO LEARN MORE

- Managing transactions in Redis[e]
  https://redislabs.com/docs/managing-transactions-redis/

Analytics

Built-in analytics commands and modules

If you know basic information about your user, you can take them along a more specific path that's guided by business rules.

Personalization is about generating recommendations that are based on a user's profile, demographics and recent actions, along with a few business rules. If you know basic information about your user, you can take them along a more specific path that's guided by business rules.

Recommendation engines can implement collaborative filtering, which is when you lead users in a similar demographic along the same path because they liked the same things, but haven't made those same purchases. By using set intersections to compute similarity scores, you can figure out what that person might be more likely to purchase. To execute these analytics at high speeds and in real-time requires a database that can support these operations.

Redis has built-in commands that can be used to implement real-time analytics such as set intersections, score assignment, statistical estimation, etc., for what data matters most. Estimates based on a probabilistic data structure, such as a hyperloglog, make for a more efficient database since storing and counting each item may require too many computational and storage resources.

TO LEARN MORE

- An Ultra-Fast Recommendations Engine Using Redis and Go
  https://redislabs.com/docs/ultra-fast-recommendations-engine-using-redis-go/
- Redis Hyperloglog: A Deep Dive
  https://redislabs.com/docs/redis-hyperloglog-deep-dive/

Machine Learning

Modules to implement ML and serve ML models 100x faster

Conversion rates are an important metric for websites, and machine learning can work to improve these rates. Machine learning can help to lower the number of errors in presenting the wrong content to a customer so that customers stay engaged and spend more time in your application.

But the algorithms require significant memory, or else the scoring is too slow — the model has to be sized just right to be able to work on the application layer. For models to be accurate and precise, they need to be retrained with recent data as well.

Redis has the capability to store machine learning models in their native format, and update and serve them with minimal computing infrastructure needed to implement these algorithms at scale. The neural network module in Redis is simple feed forward and embedded as a data type — it can train and serve models simultaneously.

Machine learning can help to lower the number of errors in presenting the wrong content to a customer so that customers stay engaged and spend more time in your application.

TO LEARN MORE

- Webinar - Implementing Real-time Machine Learning with Redis-ML
  https://redislabs.com/resources/webinars/past/?post_id=25513
- Real-time Intelligence with Redis-ML and Apache Spark
  https://www.slideshare.net/RedisLabs/redisconf17-realtime-intelligence-with-redisml-and-apache-spark

Job & Queue

PUB/SUB and Lists for job & queue management

Personalization works best when there's seamless coordination of the work that needs to be done and how that work is assigned. Apps need to pre-compute steps to get ahead. That means, if a user connects to his bank, for example, and starts categorizing transactions, the queues get instantiated and the app begins to serve up suggestions for the next transaction.

TO LEARN MORE

- Real-Time Recommendations Using WebSockets and Redis - Ninad Divadkar, Intuit
  https://youtu.be/OqgcTJHSjv8

Search

Library and modules for high performance index and search

When users search for information, you want a specific set of results to pop up, and you want this to happen in an instant. Using indexes within Redis or the RediSearch module, search can be implemented at blazing fast speeds and customized to present users with auto-suggestions, auto-completion of searches and results can be scored so they are most relevant to users.

TO LEARN MORE

- RediSearch - A High Performance Search Engine as a Search Module
  https://redislabs.com/docs/redisearch-a-high-performance-search-engine-as-a-redis-module/

JSON/Geo/Graph

Intelligent handling of complex datatypes

Using built-in structures to capture data helps understand the user better and to personalize the experience. Geo data, a user's location, can help you provide recommendations based on where that user is. Knowing where that user has shopped or dined, for example, can help you to provide specific recommendations that fit the user's lifestyle. You may also have location and transaction data for shoppers on your site; is the user shopping at home? At work? Or interacting with your online store while in your brick and mortar locations? Where does he or she browse versus purchase?

Redis automatically handles location data with built-in geospatial indices, and Redis modules such as ReJSON or Redis Graph can handle variably structured data too.

TO LEARN MORE

- Redis for Geospatial Data
  https://redislabs.com/docs/redis-for-geospatial-data/
- The Redis Graph module
  http://redismodules.com/modules/redis-graph/
- The ReJSON module
  http://redismodules.com/modules/rejson/

A Single, Unified Platform

The power of Redis[e] for personalization and many other use cases is that all of the functions listed above are part of one unified platform. Once you have your data in Redis[e], you can access it instantly in multiple ways without having to worry about moving it back and forth between separate databases or worrying about whether each service will scale.

Annotated Pseudo Code

To demonstrate the power of Redis[e] in a way that will be easy for non-developers to digest and appreciate, an annotated pseudo code example illustrates the use of Redis[e] for real-time personalization and recommendations in an ecommerce store.

Personalization is a difficult challenge. It requires a full understanding of user interests and user behaviors (purchasing) as well as the behavior of similar users and their purchases to drive recommendations that are served up in real-time.

There are three sections to this app:

	REDIS CAPABILITIES
CAPTURE	
Gather and update all relevant data about the user's purchase history and about items on sale	Ingest millions of items per second
LEARN	
Real-time analytics on all user data, the interests of the user (behavioral and expressed), supplemented by machine learning. Create recommendations per user based on users who bought similar products and who have similar interests.	• Express complex logic in a couple of lines of code • Update transaction history in real-time • Use machine learning techniques
PERSONALIZE	
With all the hard work done and available in real-time, offer the user personalized recommendations	With all data in memory, present user with a personalized experience and compelling recommendations.

In a real-world implementation, each of the sections could be designed as one or more microservices.

Replace hundreds of lines of code with a single command

Redis provides data structures as well as specialized commands that interact with those data structures. Without these commands, you would have to write the logic yourself and you would end up writing hundreds of lines of codes. Commands that demonstrate this capability in the pseudo code that follows include `sismember`, `sadd`, `zadd`, `zrangebyscore`, `zrevrangebyscore`, and others.

In this pseudo code example, we will demonstrate the basic data structures and decision making processes around personalized recommendations. Our example is an online grocery store with an ecommerce application that captures user purchases and items on sale among many other things. The application processes and analyzes the data and makes the following personalization decisions:

1. Recommended products for a customer
2. Segmenting customers based on how frequently they purchase and how many purchases they make
3. Dynamic categorization of customers based on their declared interests and purchase behavior
4. Store other personalized messaging and customizations

Capture

Capture, map, and constantly update data about purchases and items on sale

Personalization applications must know as much as possible about the customer for whom recommendations are being made. The Capture section shows how Redis data structures could be set up to collect that information.

Capture Data Structures

The following data structures are used to capture information about users, transactions, and items on sale. Having all this data updated and available instantly is key to supporting real-time recommendations. Redis[e] handles millions of operations per second at sub-millisecond latency, so we can store and access instantly as much data as we want.

Items on sale
This recommendations engine is designed to know which items are on sale so that users will be motivated to "buy now" to get a special price. The Items on Sale data structure consists of a set of stock-keeping units or SKUs that uniquely identify products that are on sale.

Transactions
Instantly accessible transaction history, with each item in the transaction along with the item count and the price paid per item.

Purchases of each user
A sorted set that shows what the user bought and how many of each item.

Set of items sold, and its purchasers
We mapped the users to what they bought. Now we map the items to the users who bought them. In this way we can find out who is the top purchaser of a particular item.

Transactions by user
How often are users making purchases? The transactions by user data structure maps users to transactions and the timestamps for those transactions (for this pseudo code example, the timestamp is a date).

Capture data structures

NAME	TYPE	STRUCTURE	EXAMPLE
Items on Sale	Set	`skus: <SKU>...`	`skus - 2572857 8275027 2525802`

Transactions	Hash	`transaction:<transaction_id>`
		`item_count:<SKU> <count>`
		`item_price_per_sku:<SKU> price`
			`transaction:2989892`
			`item_count:2572857 1`
			`item_price_per_sku:2572857 199.99`

Purchases of each user	Sorted Set	`purchases_by_user:<userid> : <count> <SKU>`	`purchases_by_user:82498217 1 2572857 5 8275027 12 2525802`

Set of items sold, and their purchasers	Sorted Set	`sales_by_sku:<SKU> <count> <userid>`	`sales_by_sku:2572857`
			`1 82498217`
			`4 2854389`
			`10 2092339`

Transactions by user	Sorted Set	`transactions_by_user:<user_id> - <tx_id> <timestamp>`	`transaction_by_user:82498217`
			`298792 1/23/2017`
			`2989892 3/12/2017`
			`23985205 4/24/2017`

Capture Methods

The Capture methods in this pseudo code example illustrate some simple ways that Redis can track product information and user behavior in preparation for making a recommendation.

Methods used include:

`void addItemOnSale(sku)`
Adds items to the set

`boolean isItemOnSale(sku)`
Says whether an item is available for sale

`void newPurchaseTransaction``(userid, purchaseItems, purchaseItemCount)`
Executes a purchase transaction

Capture items on sale

```
void addItemOnSale(String sku){
   // add the stock keeping unit (sku) to the set using Redis SADD
   // command
   // SADD: Redis command, itemsOnSale: name of the set
   redis.call("sadd", "items_on_sale", sku);
}

boolean isItemOnSale(String sku){

   // returns true if sku is in itemsOnSale set, false otherwise
   return redis.call("sismember","items_on_sale", sku);
}
```

The first two methods are utilities that, respectively, add items to the set of items on sale and check to see whether a given item is on sale.

Capture transaction information

```
void newPurchaseTransaction(String userid, String[] purchaseItem,
String[] purchaseItemCount){

   // Store the transaction data
   // get a unique id for each transaction
   String transactionId = getTransactionId();

   // Create the transaction Hash data structure
   // Iterate through the product list and update the data structure
   redis.call("hset",transactionId, "userid", userid);

for(int i=0; i<purchaseItems.length; i++){

   // Add transaction
   redis.call("hset", transactionId, "item_count:"+ purchaseItem[i],
   purchaseItemCount[i]);

   // Track user purchases
   redis.call("zincrby","purchases_by_user:"+userid,
   purchaseItemCount[i], purchaseItem[i]);

   // Update items sold to each customer
   redis.call("zincrby", "sales_by_sku:"+purchaseItem[i],
   purhaseItemCount[i], userid);
}

   // Update the sorted set that holds all user transactions
   redis.call("zadd", "transaction_by_user:"+userid, currenttime,
   transactionId);

   updateCustomerLeaderBoard(userid);

}
```

`newPurchaseTransaction` is used to capture the data related to a particular transaction. The number of items is captured by SKU, and a leader board is updated.

Everything happens in real-time: We are constantly updating all the data structures as purchases occur so the latest information is always available for recommendations.

Every purchase updates all the data structures. Everything is being captured in real-time. When a user logs in, we know instantly the items he has purchased the most. And if an items is on sale, we also know that immediately.

The Capture section of the code shows how Redis can be used to get all relevant data, update it in real-time, and create data structures that will serve as the basis for real-time recommendations.

Advanced technique: Pipelining

All of the Redis calls in this section could be pipelined for even greater efficiency. Pipelining enables you to send multiple commands to the server at once without waiting for the replies at all. The client can then read all the replies in a single step. Read more about pipelining.

Learn

Real-time Analytics on User Interests and Behavior

In the Learn segment, we seek to gain a full picture of everything users are interested in based on expressed interests at sign up as well as user behavior (products they have looked at or purchased).

Learn Data Structures

In this section, we create a variety of data structures to analyze user interests and behavior, up to and including creating recommendations for users.

Purchaser leaderboard
The purchaser leaderboard is a sorted set that answers the question, who is buying the most?

Purchases in last N days by a user
How many purchases has the user made in the last N days?

Recommendations for a user
Recommendations for a given user are captured as a sorted set. This data structure is updated based on two things: the interests selected by the user and their purchases (which demonstrate interests that may not have been indicated).

Global list of all user interest categories
A list of all the categories that users can express interest in when they sign up or on their profile page.

The list of categories an item belongs to
We create a set for each item, with a list of categories that apply to that item.

Interest categories that the user selects while signing up
For each user, we store a set of the interests they selected when they signed up.

Interest categories based on user behavior (purchases)
We store interest categories per user based on their purchasing behavior.

Union of user category sets
We then create a set of all categories a user is interested in, via a union of user-selected categories with behaviorally indicated categories (purchases).

Recommendations for a user by category
We create a sorted set for each combination of category and user to support recommendations by category.

Learn data structures

NAME	TYPE	STRUCTURE	EXAMPLE

Purchaser leaderboard	Sorted Set	`tx_count_leader_board - <score> <userid>`	`tx_count_leader_board`
			`113 2858233`
			`172 1238752`
			`233 8232334`

Purchases in last N days by a user	Sorted Set	`purchases_in_<n days>_by:<userid> - <score> <userid>`	`purchases_in_7_by:2858233`
			`4 3453432`
			`6 3453212`
			`9 435324`

Recommendations for a user	Sorted Set	`reco_items_for_user:<userid> - <score> <itemid>`	`reco_items_for_user:2858233`
			`53 435324`
			`66 3453212`
			`154 3453432`

Global list of all user interest categories	Set	`user_interest_category <set of user interest categories>`	`user_interest_category - "dairy", "produce", "bread", "organic"`
The list of categories an item belongs to	Set	`item_to_categories_map:<sku>: <set of interests>`	`item_to_categories_map:80708 - "milk", "dairy", "organic"`
Interest categories that the user selects while signing up	Set	`categories_user_selected:<userid> <set of categories - a subset of global list>`	`categories_user_selected:9349720 - "organic", "gluten_free"`
Interest categories based on user behavior (purchases)	Set	`categories_by_behavior:<userid> <set of categories - a subset of global list>`	`categories_by_behavior:9349720 - "bread", "promo"`
Union of user category sets	Set	`Union of categories_user_selected:<userid> and categories_by_behavior:<userid> interest_categories:<userid> <set of categories - a subset of global list>`	`interest_categories:9349720 - "bread", "promo", "organic", "gluten_free"`

Recommendations for a user by category	Sorted Set	`reco_items_by_category: <category>:user:<userid> - <items>`	`reco_items_by_category:organic:user:8237292`
			`10 3453432`
			`90 3453212`
			`200 435324`

How fast are we updating those data structures? Redis is an in-memory database. It handles millions of writes per second, enabling us to do millions of updates per second.

Learn Methods

We use the following methods in the Learn section:

`void updateCustomerLeaderBoard(userid)`
Updates purchaser leaderboard

`boolean is AmongTop1000(userid)`
Says whether a customer is among the top 1000 users

`String[] mostPurchasedItems(userid, topN)`
Returns the top n purchased items of a user

`String[] purchasesInLastNDays(userid, lastNDays)`
Returns the items purchased by a user in last n days

`void setRecommendationsByPurchaseHistory(userid)`
Sets recommendations for a user based on their purchase behavior

`void addUserInterestCategory(category)`
Manages a global set of user interest categories

`void setItemToCategoriesMap(item, category)`
Manages the list of categories an item belongs to

`void setUserToCategoriesMap(userid, category)`
Manages the list of categories a user is interested in

`void setDynamicUserBehavior(userid)`
Updates user interest categories based on purchase history

`void setRecommendationsByInterests(userid)`
Uses Redis-ML module to set recommendations per category

```
// Update leaderboard for each transaction
// The customer who performs most transactions is
// the leader
void updateCustomerLeaderBoard(String userid){

   redis.call("zincrby", "tx_count_leader_board", 1,
   userid);

}

// Returns true if the user is among the top 1000 users
// based on the number of transactions
boolean isAmongTop1000(String userid){

   // Find the reverse rank based on the number of
   // transactions
   int score = redis.call("zrevrank",
   "tx_count_leader_board", userid);

   if(score < 1000){
       return true;
   }

   return false;

}
```

This section of code determines the top user based on the number of purchases (rank on the leaderboard). It also assesses whether the user is ranked in the top 1000 of all purchasers (you might call them star users or power users and offer them special deals based on that status).

Depending on your ecommerce application, you might rank users by amount spent, number of visits to the site, or other criteria.

You might also use this logic in the opposite way and deliberately target users who are either new or infrequent buyers to give them extra incentive to become customers or more frequent customers right now.

```
// Returns top n items purchased by each user
String[] mostPurchasedItems(String userid, int topN){

   // This command reverse sorts the items purchased by
   // the user
   // based on the number of purchases and returns the
   // topN items
   String[] topSkuIds =
       redis.call("zrevrangebyscore","purchases_by user:"
       +userid, "+inf", "0", "limit", "0", topN);

   return topSkuIds;

}
```

This section of the code returns a list of what users are buying and the top items each user buys.

```
// Returns all purchases done by the user in the last N number of days
String[] purchasesInLastNDays(String userid, double lastNDays){

   // Retrieve all transactions in the last N days
   String[] txIdsLastNDays =
       redis.call("zrevrangebyscore", "transaction_by user:"+userid, "+inf",
       (currenttime - lastNDays));

   // Clear the sorted set if it already exists
   redis.call("del","purchases_in_"+lastNDays+"_by:"+userid);

   // Iterate through each transaction and retrieve items purchased
   for(int i=0; i<txIdsLastNDays; i++){
       String[] txItems = redis.call("hgetall","transaction:"+txIdsLastNDays [i]);

       for(int j=0; j<txItems; j++){
           // Split the string, item:<skuid> to get the 2nd part
           String skuId = txItems[j].split(":")[1];

           // The next element in the array is the count
           j++;
           int count = txItems[j];

           // add the item to the sorted set along with the count purchased
           redis.call("zincrby","purchases_in_"+lastNDays+ "_by:"+userid, count, skuid);
       }
   }

   return redis.call("zrevrangebyscore","purchases in_"+lastNDays+"_by:"+userid, +inf, 0,
   "WITHSCORES");

}
```

The `setRecommendationsByPurchaseHistory` method creates recommendations based on purchase history. It generates a list of items to recommend to the user. In this example we compute it using the following procedure:

1. Get the top items purchased by the user, A. Let the items be 1, 2, 3, 4, 5
2. Get the list of other users who have purchased 1, 2, 3, 4, 5. Let the list of users be K, L, M, N, O, and P
3. Get the top items purchased by K, L, M, N, O, and P. Let the items be 4, 5, 6, 7, 8, 9. Now we can recommend 6, 7, 8, 9 - other items purchased by users who have purchased things this user has purchased.

This example demonstrates how you could apply sorted sets and their functions to build simple recommendations based on purchasing behavior.

```
void setRecommendationsByPurchaseHistory(String userid){

   // get the top 5 purchased items by this user
   // 5 is an arbitrary number
   String[] items = getMostPurchasedItems(userid, 5);

   // Prepare a set of other users who purchased these items.
   // This is done by the UNION operation on sales_by_sku:<item> set
   redis.call("zunionstore","reco_prep_users:"+userid, 5
          "sales_by_sku:"+items[0],
          "sales_by_sku:"+items[1],
          "sales_by_sku:"+items[2],
          "sales_by_sku:"+items[3],
          "sales_by_sku:"+items[4],
          "aggregate", "sum");

   // Now we know people who purchased "X"also purchased "Y"
   String[] usersWithCommonInterest = redis.call("zrange", "reco_prep_users:"+userid, 0, inf);

   // Compute the union of all their purchases to arrive at the recommendations
   for(commonInterestUser in usersWithCommonInterest){
       redis.call("zunionstore","reco_items_for_user:"+userid,
           2, "reco_items_for_user:"+userid, "purchases_by_user:"+commonInterestUser,
           "aggregate","sum");
}

// Now reco_items_for_user:<userid> is a sorted set of items sorted by how many times
// they were purchased by similar users

}
```

```
void addUserInterestCategory(String category){
   redis.call("sadd", "user_interest_category", category);
}

void setItemToCategoriesMap(String item, String category){
   // check if the category exists. If not, add it to the set
   if(!redis.call("sismember", "user_interest_category", category)){
       addUserInterestCategory(category);
}

   // adds category to the set of categories associated with the item
   redis.call("sadd", "item_to_categories:"+item, category);
}
```

These methods, `addUserInterestCategory` and `setItemToCategoriesMap`, assign categories to users that can be used in recommendations.

```
// Allow the users to select their area of interest.
// Call this method to maintain user -> interests mapping
void setUserToCategoriesMap(String userid, String category){
   // check if the category exists. If not, add it to the set
   if(!redis.call("sismember", "userInterestCategory", category)){
       addUserInterestCategory(category);
   }

   // Maintain user interest
   redis.call("sadd", "categories_user_selected:"+userid, category);
}
```

`setUserToCategoriesMap` allows users to select their area of interest.

Even though users select their interest categories, sometimes their behavior shows that their real interests are different from their stated interests. The `setDynamicUserBehavior` method evaluates a user's purchase patterns and determines their areas of interest based on the items they purchase. Here is how we determine it:

1. Pull the top n purchased items of a user. Say, 1, 2, 3, 4, 5.
2. Get the areas of interests associated with those products. Say m, n, o, p, q, r.
3. Store that information per user to drive recommendations.

```
void setDynamicUserBehavior(String userid){

   String[] topPurchasedItems = mostPurchasedItems(userid, 10); // 10 is an arbitrary number

   for(item in topPurchasedItems){
       // categories_by_behavior:<userid> <- categories_by_behavior:<userid> UNION
       // item_to_categories:<item>
       redis.call("sunionstore", "categories_by_behavior:"+userid, "categories_by_behavior:"
       +userid, "item_to_categories:"+item);
   }
}
```

Enrich Recommendations with Machine Learning Models

Redis makes it easy to write code that leverages results from machine learning models. Each category has a machine learning model that recommends items based on attributes of a particular user. This section of the code shows how this machine learning model is being leveraged to create recommendations.

One of the very powerful features of Redis is its modules, which provide specialized functionality that can be easily added and used. This section of the code uses redis-ml, the machine learning module. Modules are continually being added; other modules include Redis Graph, RediSearch, and ReJSON. These modules provide a powerful way to enrich the way you use Redis.

Note that with machine learning, while the model is created by a data scientist, any developer can use that model in their code without extensive expertise. This creates a write-once, use-many scenario where many developers can leverage the work of data scientists who create, train, and tune machine learning models.

```
// Uses Redis machine learning module to recommend based on interests.
// The method assumes user interest categories are already established
void setRecommendationsByInterests(String userid){

   // Sample data: "age:31", "sex:male", "food_1:pizza", "food_2:sriracha"
   String[] featureVector = redis.call("hget", "userid:features");

   // Merge user interests: combine user selected categories and dynamic purchase behavior
   redis.call("sunionstore", "interest_categories:"+userid,
       "categories_user_selected:"+userid, "categories_by_behavior:"+userid);

   Category[] userInterestCategories = redis.call("smembers", "interest_categories:"+userid);

   // For each category we have a machine learning model that will recommend the most suitable items
   // according to the users feature vector. The models are trained on Spark and stored on Redis-ML.
   for(category in userInterestCategories){
       // Get all items of this category
       String[] items = redis.call("smembers", "item_to_categories:"+category);
       //for each category get a score from the random forest classifier
       for(item in items){
           category.itemScores[item] = RedisRandomForestClassify(forestId = "category:item",
           featureVector)
       }

       // sort the classification results and get the top results to render recommendations
       results[category] = category.itemScores.sort()[0:n_items]

       // add recommended items for this user under each category
       redis.call("sadd","reco_items_by_category:"+category+":user:"+userid, results[category]);
 }

}
```

The `setRecommendationsByInterests` method uses a machine learning model already trained to set specific recommendations for each user.

Personalize

Real-time Recommendations

Because of the work done so far, the live personalization of the user experience is the easiest part to code, bringing everything done so far together like a symphony.

As the user logs into the application, Redis stores personalized information. Typically, the information includes:

User account information

User profile data
purchase history, recommendations, user interests

Personalize Data Structures

The Personalize session uses a session store that leverages data structures we've already created.

Personalize data structures

NAME	TYPE	EXAMPLE

Session Store	Hash	`user:<userid>:session:<sessionid>`
		`Elements in user:<userid>:session:<sessionid>`
		`goldmember <yes/no>`
		`purchase_history (pointer to the set that stores the history)`
		`interest_categories (pointer to the set that stores interest categories for this user)`
		`recommendations (pointer to the set that stores items recommended for this user)`

Personalize Methods

`setPersonalizedSession(user, session)`
loads user profile information

When a user arrives, this method computes the recommendations and stores them in the user's session object so they are ready to use. Because Redis works in real-time, the recommendations generated are always up to date.

```
void setPersonalizedSession(String userid, String sessionid){

   // loads user account information - name, address, access controls
   // This information is typically loaded from an identity manager
   // and stored in a Hash data structure inside Redis
   loadUserAccountInformation(userid, sessionid);

   redis.call("hset", "user:"+userid+":session:"+sessionid, "goldmember", isFrequentUser(userid));
   String[] purchaseHistory = mostPurchasedItems(userid, 10); // 10 is an arbitrary number

   // Store purchase history
   redis.call("sadd", "user:"+userid+":session:"+sessionid+":purchase_history", purchaseHistory)
   redis.call("hset", "user:"+userid+":session:"+sessionid, "purchase_history",
          "user:"+userid+":session:"+sessionid+":purchase_history");

   // Compute recommendations by interests categories
   setRecommendationsByInterests(userid);
   // reco_items_by_category:<category>:user:<userid> gives the recommendations by interest

   // Store user interests
   setUserInterestCategoriesByBehavior(String userid);
   redis.call("hset", "user:"+userid+":session:"+sessionid, "interest_categories",
          "interest_categories:"+userid);

   // Compute recommendations by purchase history
   setRecommendationsByPurchaseHistory(userid);
   redis.call("hset", "user:"+userid+":session:"+sessionid, "recommendations",
          "reco_items_for_user:"+userid);


   // Store other personalized information with the tags, userid and sessionid
}
```

Conclusion

Redis provides an incredibly powerful, robust and elegant way to create real-time personalizations that are scalable, no matter how large your inventory or your customer base. Our narration of how pseudo code would implement such an application connects the general principles of Redis to the way they are used in a common use case. We would like to thank Roshan Kumar for his work on the pseudo code.

This paper is intended to get you started on the path to learning to think in Redis and use it as a way to express the structure and implementation of scalable applications. If you found this approach useful, please contact us with suggestions for other use cases and applications that are in need of explanation.

Redis provides an incredibly powerful, robust and elegant way to create real-time personalizations that are scalable, no matter how large your inventory or your customer base.

This paper was created by CITO Research and sponsored by Redis Labs

About CITO Research

CITO Research is a source of news, analysis, research and knowledge for CIOs, CTOs and other IT and business professionals. CITO Research engages in a dialogue with its audience to capture technology trends that are harvested, analyzed and communicated in a sophisticated way to help practitioners solve difficult business problems.

Visit us at http://www.citoresearch.com

Connect with us

redislabs
home of redis