Common Redis Use Cases

TECHNICAL ENABLEMENT

Use Cases

1. Caching
2. Session Store
3. Leaderboards
4. Message Processing and Delivery
5. Indexing and Querying

Caching

Caching Intro

Distributed systems require low latency and scalability

Challenges of a disk-based database

- **Slow data retrieval:** Even with optimizations, pulling data from disk plus processing increases response times
- **Scaling is costly:** In high read scenarios it may require a lot of additional resources and still not get close to an in-memory DB
- **Data access too complex:** relational dbs are optimized for relations not access

Benefits of adding a Redis caching layer

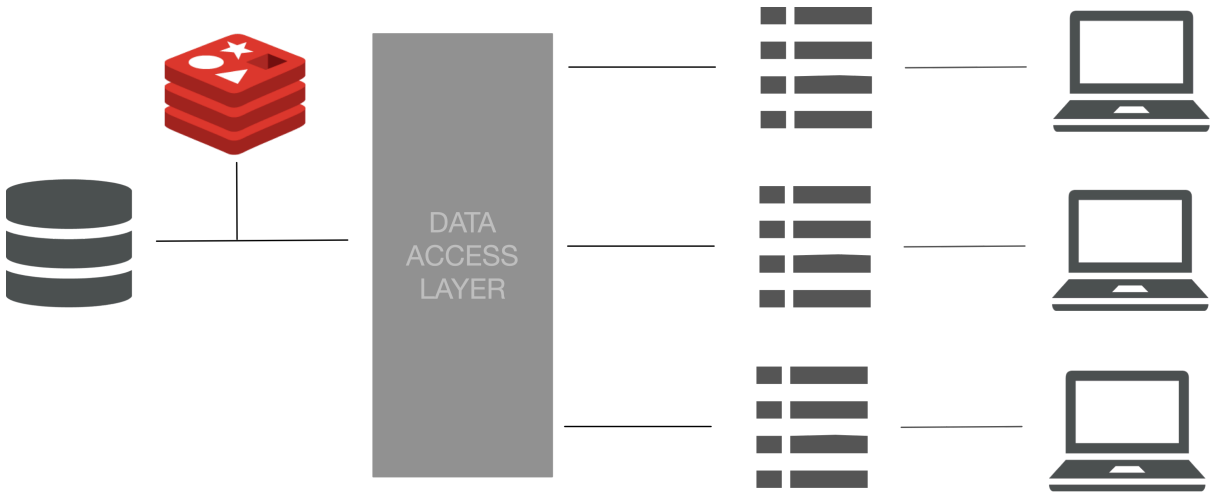
- **Fast:** extremely fast - sub-millisecond latency
- **High Throughput:** high throughput - millions of ops per second
- **Accessible:** less application processing and complexity
- **Efficient:** increased performance out of less hardware

Caching Intro

Applications use caches to store common, repeatedly read objects

What to cache?

- Object Store: database results, HTTP Response Object, rendered page
- Simple key/value string pairs: product data, scores
- Native Data Structure: store & retrieve data from natively supported data structures
- Files: images or other binary files



Caching Patterns: Look-Aside

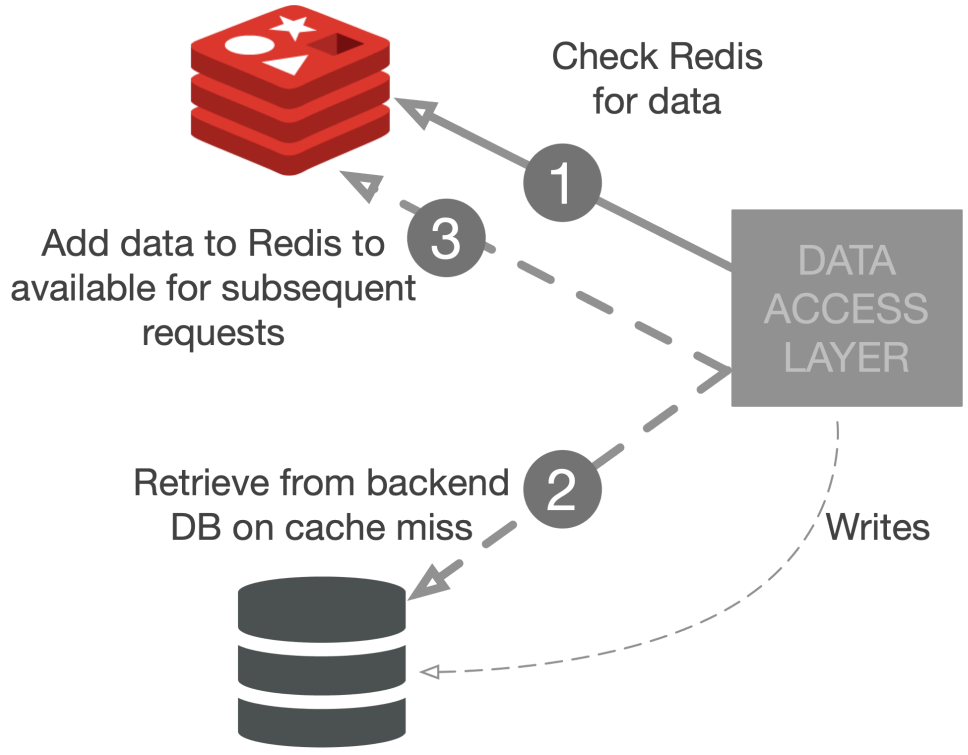
Data is written into cache only when requested, cache is only used for reads

Advantages

- The cache does not need to hold all the data, only what's requested
- Fault tolerant: if a failure occurs, cache is easily re-populated

Disadvantages

- Networking intensive: a single cache miss created multiple network hops
- Stale data needs to be taken into account by the application



Caching Patterns: Write-Through with RedisGears RGSync

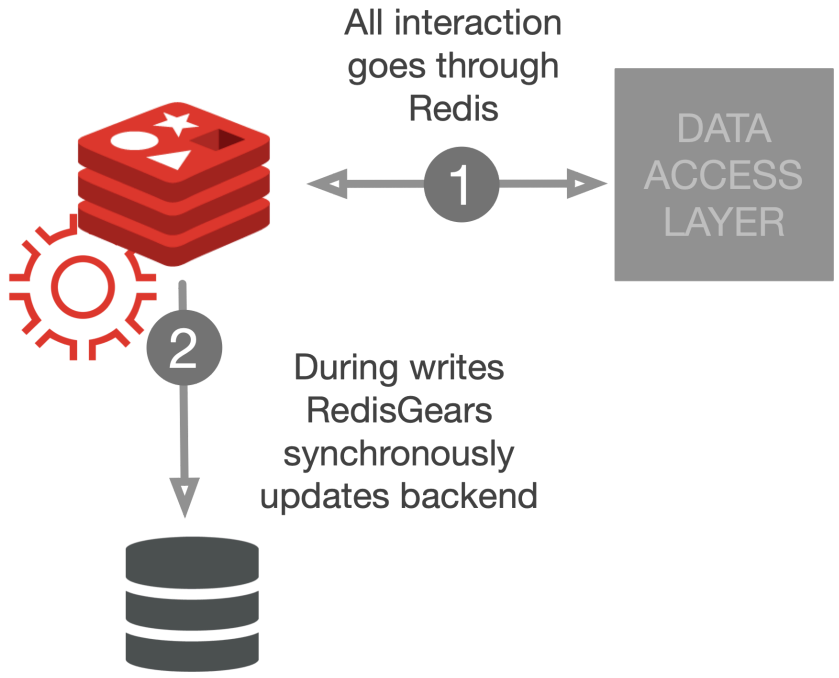
Write-through pattern but implementation is using RedisGears, cache is used for reads and writes

Advantages

- Less networking hops
- Data is in sync and available
- Application simplicity

Disadvantages

- System performance cost during writes
- All the data must live in the cache
- Need to pre-populate data (use look-aside or CDC)



NOTE RedisGears RGSync is a built-in 'recipe' enabling write-through

Caching Patterns: Write-Behind with RedisGears RGWriteBehind

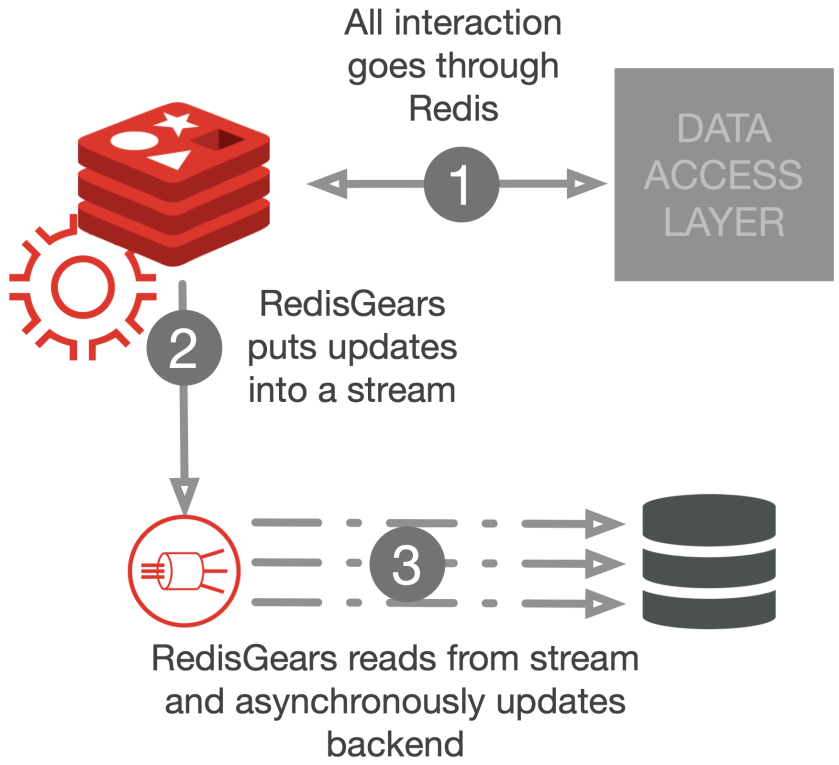
Data is written into the cache first and then later into the backend DB, cache is used for reads and writes

Advantages

- The cache is is always up to date
- Simplicity for the application
- Fewer network hops

Disadvantages

- Backend eventually consistent
- All the data must live in the cache



Caching Patterns: Read-Through with RedisGears

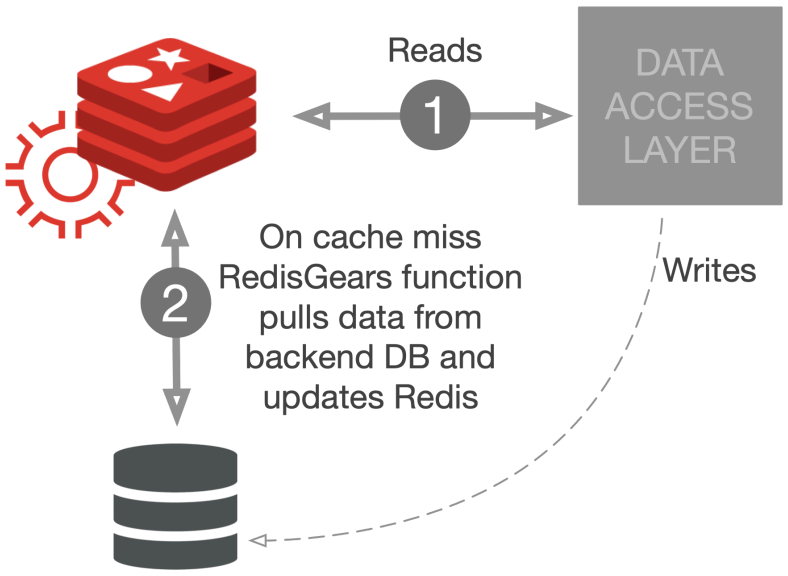
Data is written into the backend DB and read from the cache, cache is used for reads only

Advantages

- The cache is is updated when needed
- Simplicity for the application

Disadvantages

- Overall Redis DB performance hit on cache-miss
- Stale data needs to be taken into account by the application



NOTE read-through is not an official RedisGears recipe yet

Caching Patterns: CDC Cache Replica

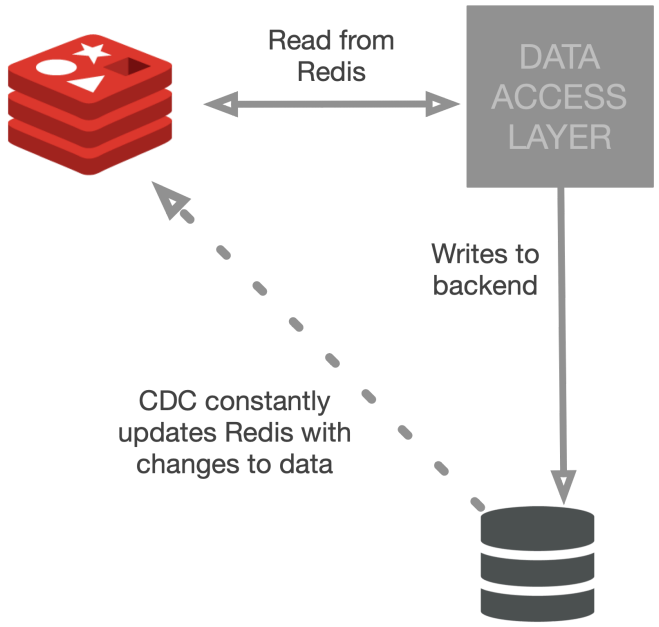
Data is written into the backend DB and CDC (Change-Data-Capture) replicates to the cache, cache is used for reads only

Advantages

- Cache is warm
- Simplicity for the application
- No stale data

Disadvantages

- May encounter a cache-miss pre CDC replication
- All the data must live in the cache



NOTE Can be combined with read-through via RedisGears to avoid the cache-miss

Session Store

Sessions Overview

Distributed applications = distributed sessions

Key Features that make Redis a fit

A session store needs to be fast requiring an in-memory solution like Redis. The user/client is waiting for the retrieval of their details before their unique data/experience can be returned. Writes also need to be fast as sessions need to be updated frequently.

While a session is considered active it must always be available. High availability and persistence may be required to recover from failure scenarios both of which Redis offers.

Session Store vs Cache

Isn't a session store just another type of cache?

Differences

- Sessions are unique to a user/clients while a cache data is global to an application
- Cached data can be lost and rebuilt from source DB while session is active data in Redis is the source of truth
- Sessions have state that is updated over time while cache exists in the latest state only
- Sessions have more requirements to ensure state is not lost through high availability and persistence
- Cached data is meant to improve application (database) performance

Sessions Examples

Common examples of sessions that could be stored in Redis

Use Case	Data Stored	Features
Login State	last access, source (browser, app), type (long term, browser based)	TTL, HA, persistance
Navigation	breadcrumb trail access	HA
Shopping Cart	items, last update	HA, persistence
API Tokens	app ID, scopes, start time	TTL, HA

Leaderboards

Leaderboards Requirements

Unique throughput requirements are not suited to traditional solutions

Gaming leaderboards enable players to track performance against each other. They have a social component of encouraging competition but also may contribute to the the overall logic of a game for tuning or matching playes with similar skill level.

Leaderboards require every participant's score be updated and calculated against the overall group.

This demands simultaneous updates from thousands/millions on active participants making this a high-write scenario.

The results need to be accurately displayed by thousands/millions of users with periodic polls for updates making this a high-read

Relational databases do a poor job collecting and distributing information to thousands or millions active users in real time due to disk bottelnecks.

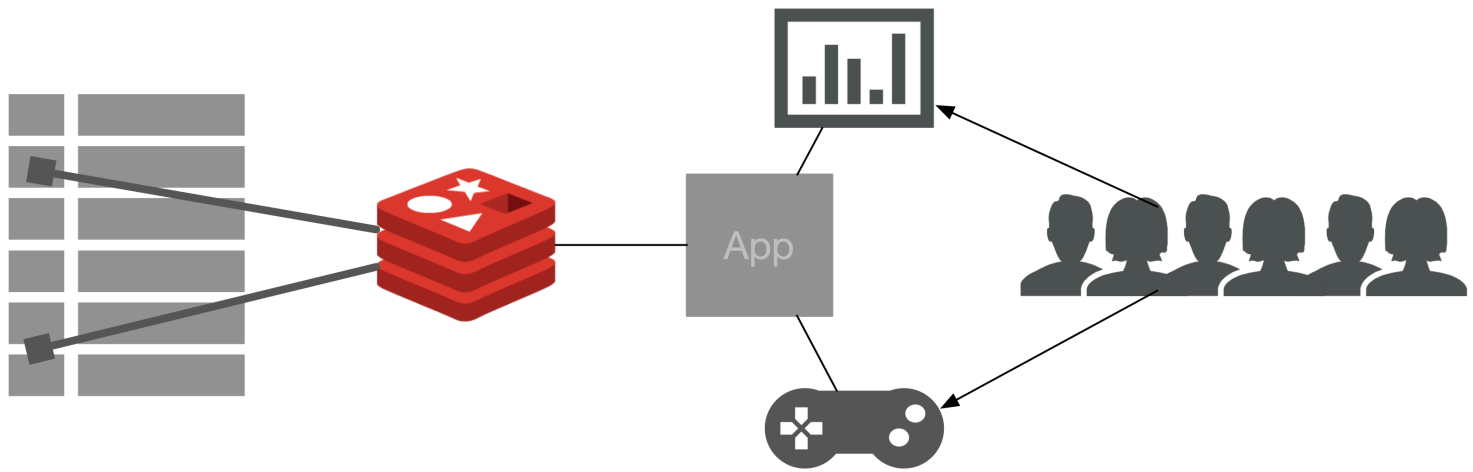
The traditional structure of a RDBMS adds additional unnecessary complexity for the use case.

Leaderboards on Redis

A perfect fit for what this use case needs

Sorted Sets Data Type Matches the Requirements

- In-memory performance provides high throughput for reads and writes
- Add or update to a member score efficiently without external index maintenance
- Parsing of results is simple and fits use case without additional complexity



Message Processing and Delivery

Messaging Intro

In general what are messaging patterns trying to solve?

> *In software architecture, a messaging pattern is a network-oriented architectural pattern which describes how two different parts of a message passing system connect and communicate with each other.*
>
> *~ Wikipedia*

Distributed Systems Require Messaging

A good messaging infrastructure will allow you to connect the components and services of a distributed systems in a loosely coupled way. Asynchronous messaging provides many benefits but also brings design hurdles such as the ordering of messages, message management, idempotency, and more.

Message Queues

Messages like events can be sent into a LIST for processing in order

Architecture

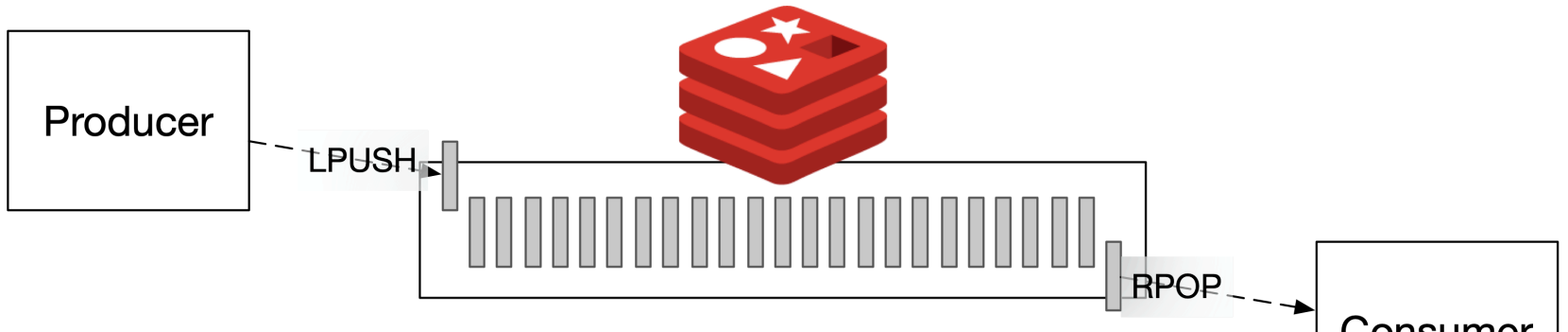
- typically batches/stores messages until they are removed
- First-In-First-Out (FIFO) pattern
- ensures only one consumer can dequeue a message at a time

Use-Cases

- decouples processing
- buffer or batch work
- smooth and/or throttle erratic workloads
- inter-service communication

Delivery

- at-least once delivery guarantee
- pull-based delivery
- messages are delivered in order



Priority Message Queues

Messages like events can be sent into a SORTED SET for processing in order

Architecture

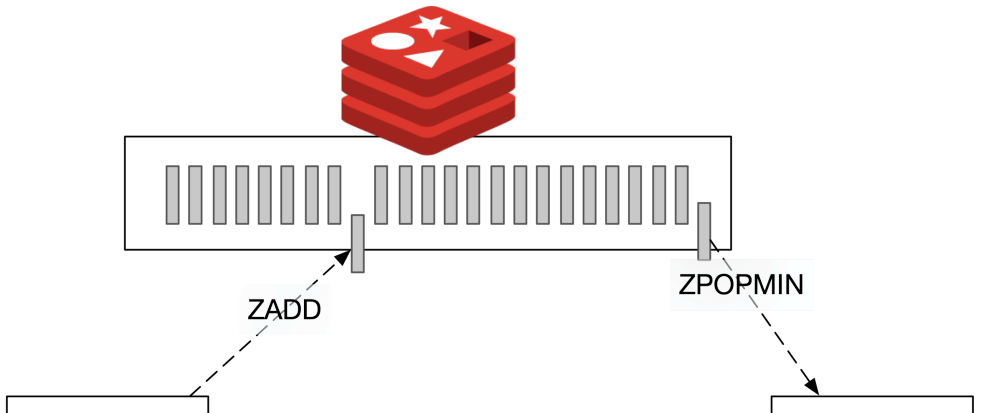
- typically batches/stores messages until they are removed
- ensures only one consumer can dequeue a message at a time
- order maintained by Redis

Use-Cases

- decouples processing
- buffer or batch work
- smooth and/or throttle erratic workloads

Delivery

- at-least once delivery guarantee
- pull-based delivery
- messages are delivered in order of priority



Publish-Subscribe

Messages are broadcast through native Redis pubsub

Architecture

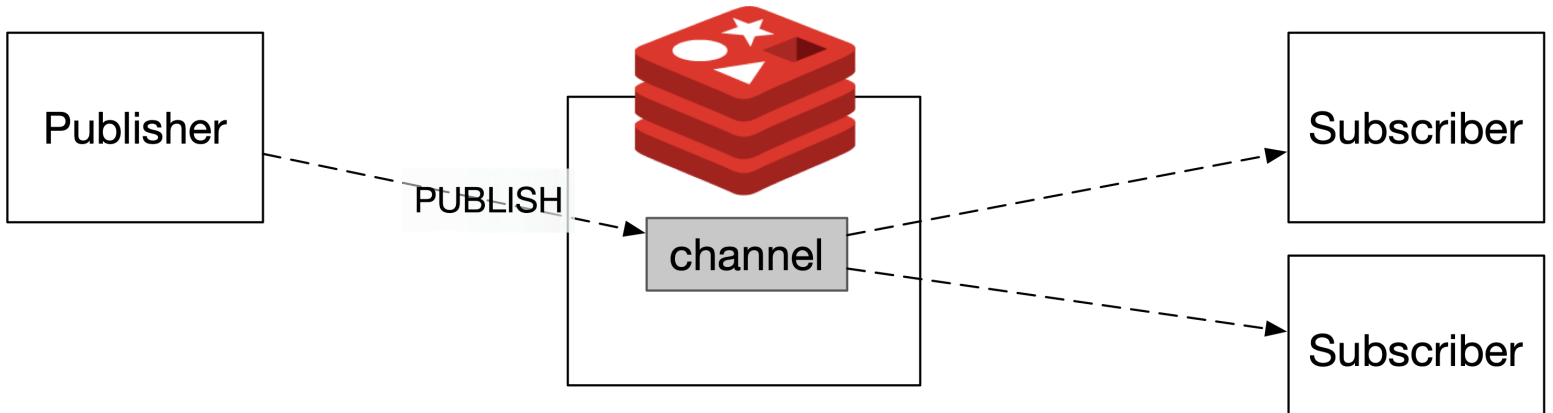
- typically pushes messages out immediately (no batching)
- broadcast (fan out) pattern and fire-and-forget pattern
- parallel consumption/processing of the same message

Use-Cases

- decouples processing
- instant event notifications for distributed applications
- inter-service communication
- alerting / reporting

Delivery

- at-most once delivery guarantee
- push-based delivery
- order is not guaranteed



Streams

Messages are pushed in an append-only log-like data structure which naturally guarantees ordering by time

Architecture

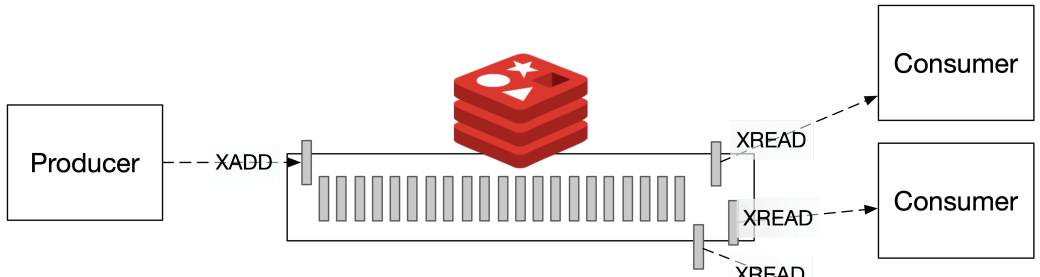
- broadcast pattern (fan out)
- parallel processing / consumption of the same message
- consumers could pull messages out immediately or replay messages from the past

Use-Cases

- decouples processing
- messaging backbone for event-driven architectures
- inter-service communication
- data ingestion / near-real time ETL

Delivery

- at-least once and/or exactly-once (some platforms/configurations) delivery guarantee
- pull-based delivery
- messages are delivered in order (assuming no partitioning)



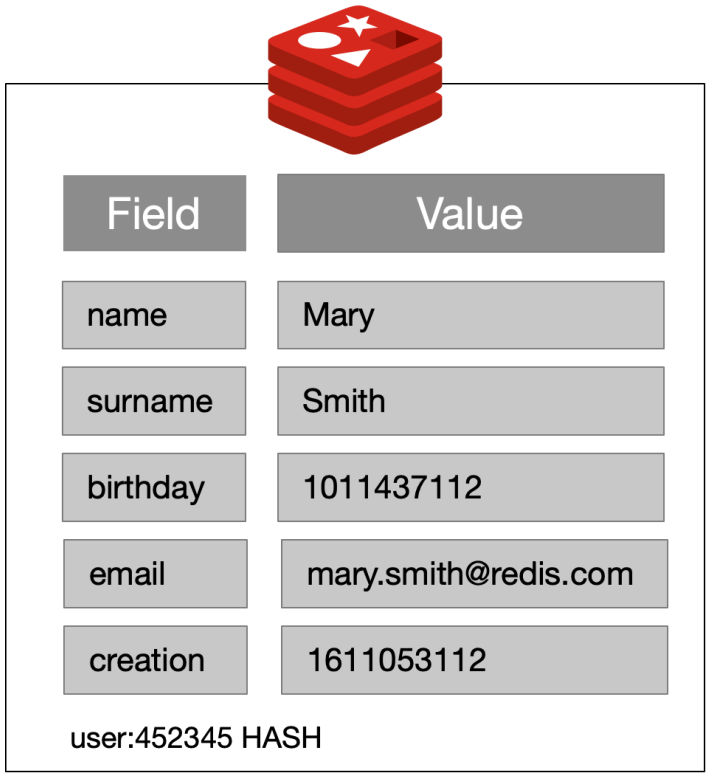
Indexing and Querying

Introduction

An index allows a query to efficiently retrieve data from a database

Characteristics of an index

- alternate access path for retrieving data
- mapping from a secondary attribute (the search key) to the primary key (the id) of an object
- has a physical structure (index structure)
- leverage the right index structure for your query/request



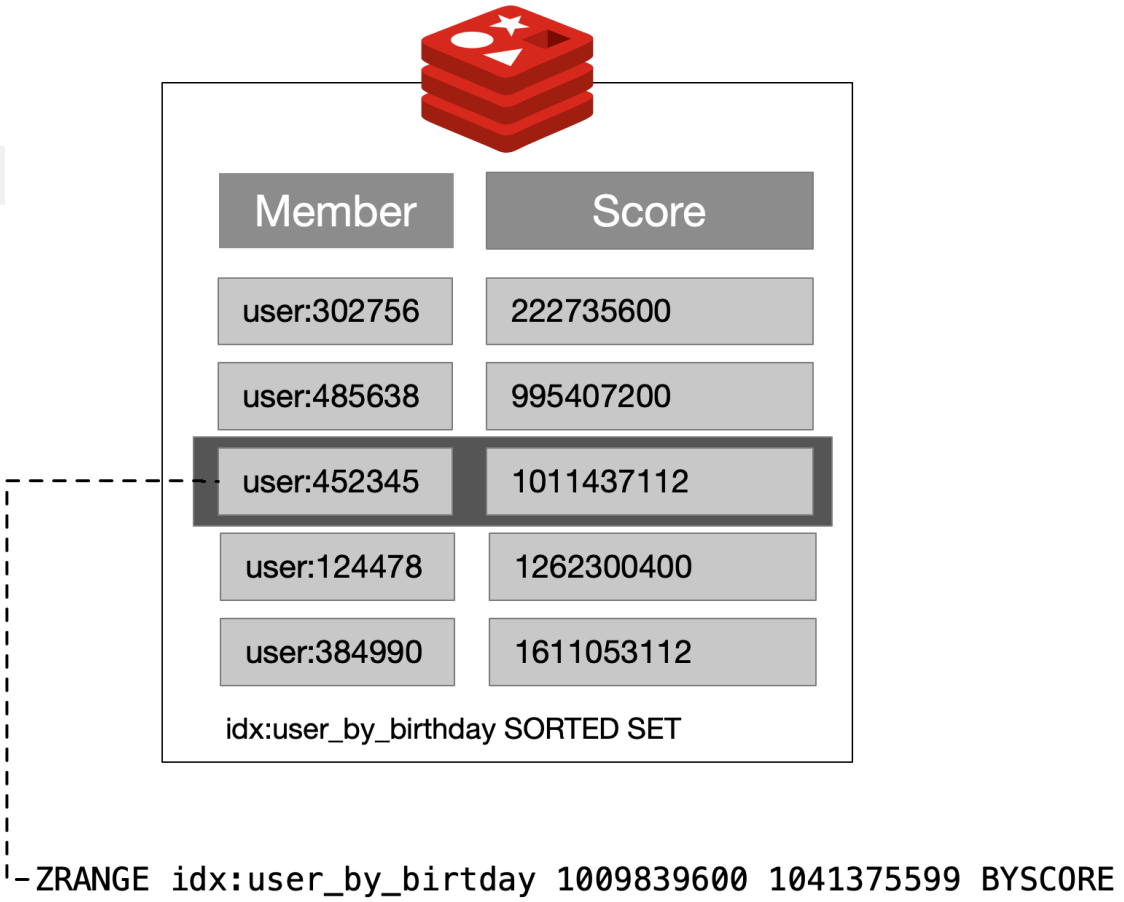
Indexing in Redis

- Secondary index using core data structures
- Inverted index for full text search using RediSearch module

Numerical Index

Secondary index using sorted sets in Redis

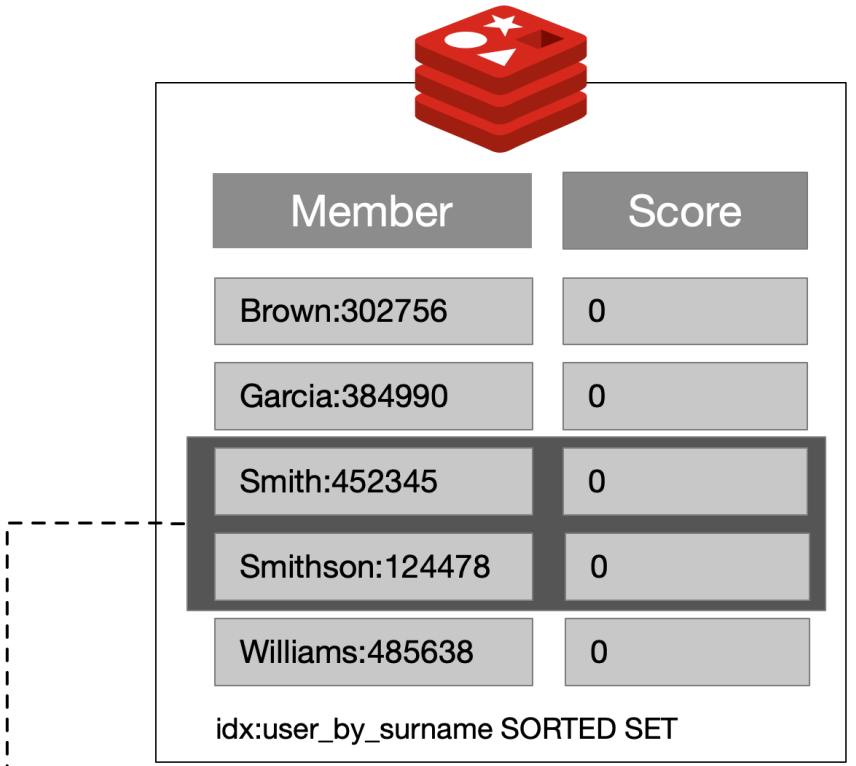
- set of elements ordered by a score (floating point number)
- range query using `ZRANGE` with option `BYSCORE`
- using the `BYSCORE` and `WITHSCORES` options of `ZRANGE` to return the scores as well
- use cases: finding items based on a numeric value or range



Lexicographical indexes

Secondary index using sorted sets in Redis

- when elements are added with the same score, they are sorted lexicographically
- commands like `ZRANGE` with option `BYLEX` and `ZLEXCOUNT` are able to query and count ranges in a lexicographically fashion
- equivalent to time complexity of traditional RDBMS indexes
- use cases: prefix searches like auto-complete

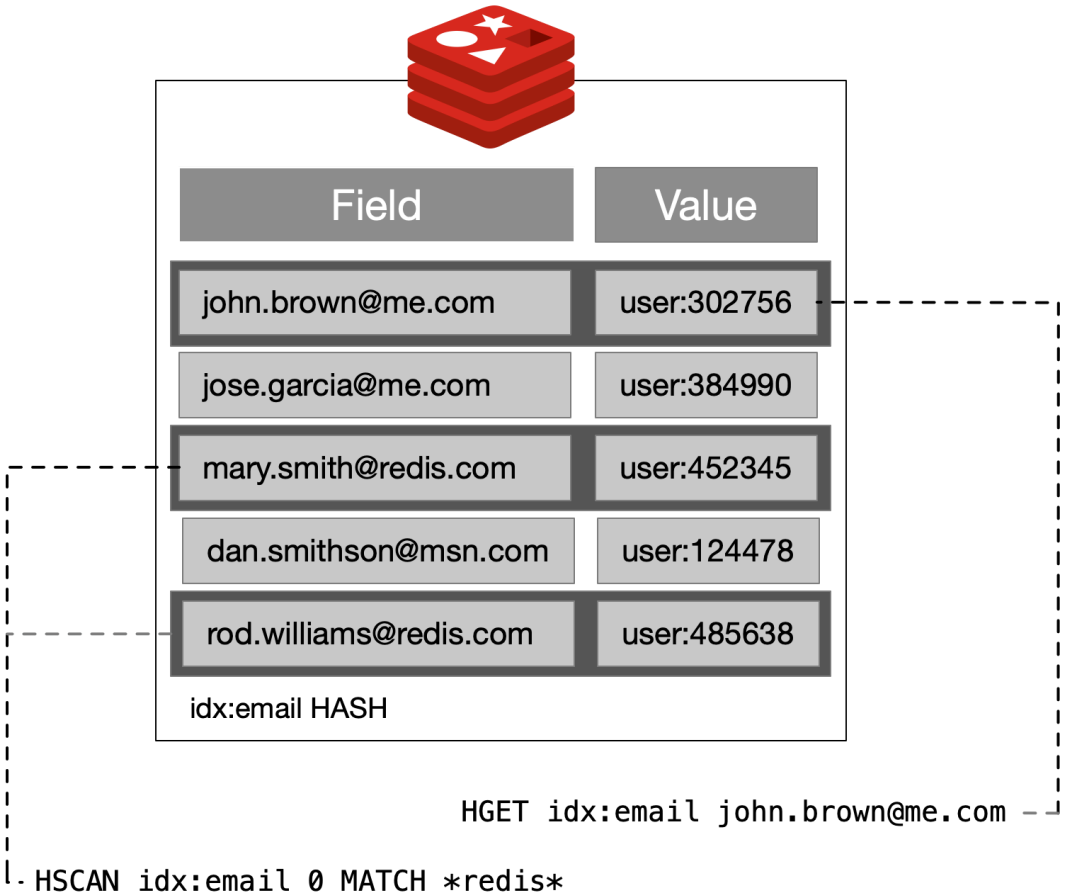



```
ZRANGE idx:user_by_surname [Smi "[Smi\xff" BYLEX
```

Hash Index

Secondary index using hashes in Redis

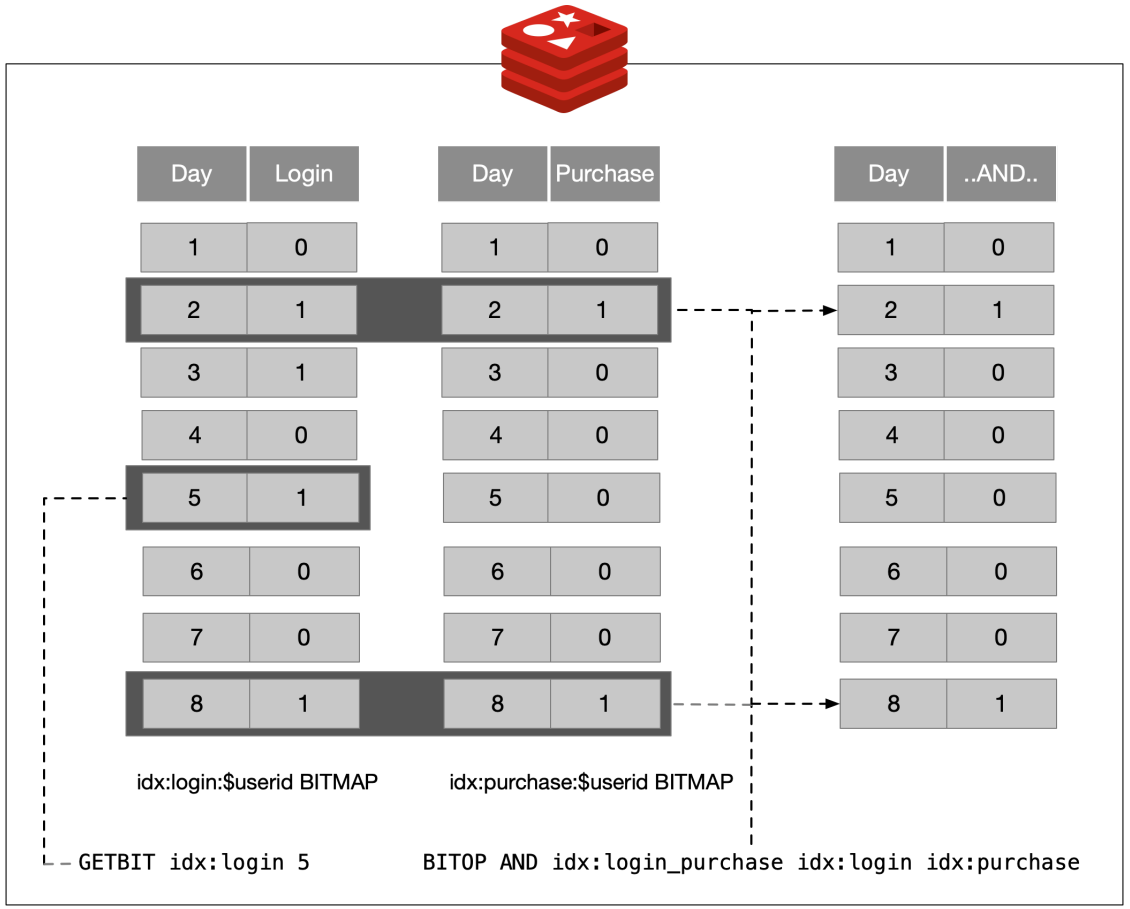
- each field in the hash is a unique member
- exact-match using `HGET` where a field value equals an exact value
- scanning using `HSCAN` with a search pattern using `MATCH` filter
- use cases: exact match or pattern match to link to secondary identifier(s)



Bitmap Index

Secondard index using bitfields in Redis

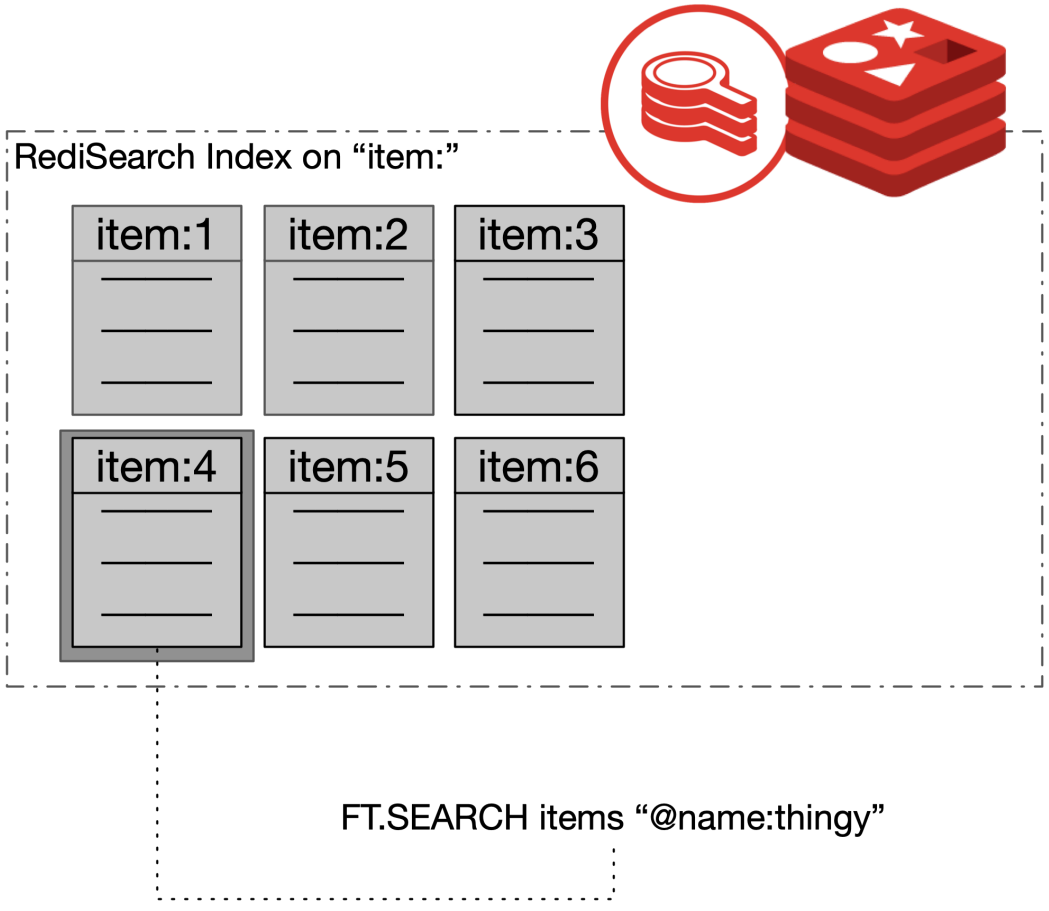
- memory efficient
- Redis strings support bitfield index
- use a bit array in a Redis string via the `SETBIT`, `BITFIELD`, and `GETBIT` commands
- need multiple strings for indexes that have more than two values
- better for boolean fields or items with low cardinality
- can use `BITCOUNT` for an aggregated count of indexes set
- can use `BITOP` to aggregate indexes using bitwise operations (AND, OR, XOR and NOT)
- use cases: ID has X, large amount of



Full Text Search

Inverted index using the RediSearch module

- index any `HASH` based on key pattern with a customized schema
- JSON documents indexing using the RedisJSON module
- incremental full-text indexing
- complex queries, prefix-based searches
- exact phrase search, Slop based search
- prefix based searches
- limiting searches to specific document fields
- numeric filters and ranges
- geo filtering using Redis' Geo-commands
- spell-checking based on dictionaries or known words in a full-text index
- auto-completion based on dictionaries



USER:1 NAME A
REDIS> PING "P
VOCATIONAL ART