WHITE PAPER

IoT backend with Redis, S3 and Node.js

Stefano Fratini

CONTENTS

Executive Summary

Redis is a perfect solution to support IoT backends when significant amounts of data need to be ingested from multiple devices and be made available via APIs.

Relational database technologies fall short in this scenario and complex and expensive NoSQL solutions are usually suggested. The IoT industry is still in its infancy, though, and in a very competitive environment like this it's common for companies with unproven business models to try to evaluate innovative solutions at a rapid pace.

In these circumstances, cost is a decisive factor in making Redis the technology of choice as it allows for speed, flexibility and simplicity with limited compromises.

The solution outlined in this paper supports collecting and exposing energy readings from smart meters with an extremely low TCO (Total Cost of Ownership) and little to no data maintenance.The techniques exposed here also apply to a multitude of other use cases/scenarios in the IoT space.

History

While the field of IoT matures, solutions to address various use cases must be designed from scratch and will eventually evolve to optimal architectures based on cost and efficiency. The system described below is an evolution of an existing solution with MySQL as the primary datastore and Redis used as a caching layer (caching device IDs and operational data) as well as for real-time data collection.

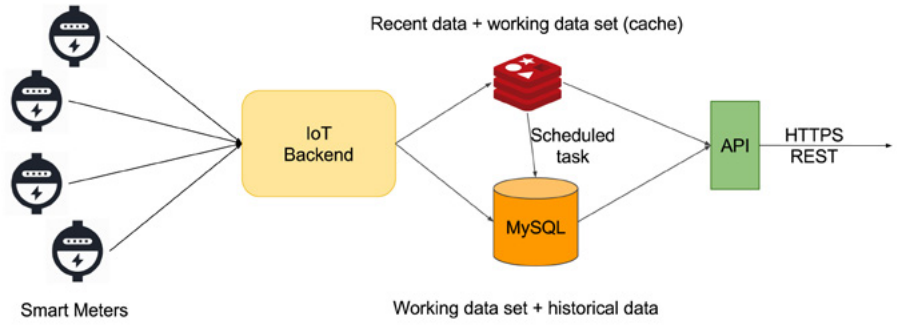


Figure 1. *Old design*

Energy data coming from the devices had the following structure:

```
[
  {
     "eReal": [-4, -11, 1, -5, 0, -3083],
     "eRealNegative": [4,11,0,5,1,3083],
     "eRealPositive": [0,0,1,0,1,0],
     "eReactive": [-5,-5,-8,-7,-2,-4090],
     "eReactiveNegative": [5,5,8,7,2,4090],
     "eReactivePositive": [0,0,0,0,0,0],
     "vRMSMin": [249,249,249,249,249,249],
     "vRMSMax": [250.2,250.2,250.2,250.2,250.2,250.2],
     "iRMSMin": [0.022,0.022,0.022,0.022,0.022,0.087],
     "iRMSMax": [0.023,0.023,0.023,0.023,0.024,0.09],
     "timestamp": 1471389300,
     "duration": 300
  },
  {...}
]
```

And it was:

1. Ingested into Redis Lists,
2. Copied into MySQL tables over time and
3. Expired in Redis.

The old solution worked relatively well but had the following limitations:

- It was built around a hybrid relational design and needed Redis as an ingestion and caching layer for high volume and velocity requirements typical to IoT.
- As energy data was collected in Redis and migrated over time into MySQL, there was a limit to the amount of data MySQL could collect given its lack of support for native sharding.
- Every system that uses caching has to deal with datasets that eventually become complex to manage over time (as they violate the Single Source of Truth principle).

Redis as a caching layer

It's not uncommon to introduce Redis into an existing system as a caching layer to improve performance. Since it runs in-memory, is written in C and runs faster than most other systems, Redis doesn't need a cache in front of it. Usually its simplicity and scalability lead to increased use as the system architecture evolves.

As new features and advanced data structures are implemented, Redis becomes the primary datastore for more and more data types. Data architectures are the hardest to evolve and this approach, while yielding great operational results, usually leads to unclean or brittle designs over time.

A new solution

The main objectives for a new solution were:

- A cleaner design without the need for data maintenance,
- Scalability beyond MySQL storage capabilities and
- Remove caching as much as possible to simplify data consistency.

The new solution is outlined below:

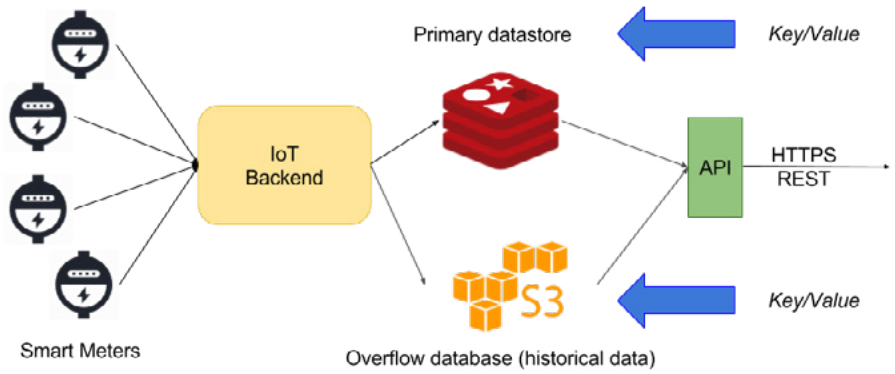


Figure 2. *New solution*

With this design, Redis has become the primary datastore for all entities and Amazon S3 has replaced MySQL as the overflow database for historical energy data -- eliminating the scalability and data maintenance restrictions of the previous solution.

The data model

There are multiple ways to map data entities to Redis keys and values, although there seem to be a few common conventions used in the industry. Below is an example of the naming conventions used to map objects, indexes and energy data in Redis for this application:

Type	Redis Type	Example
Objects (Users, Devices, Accounts)	Hash	backend:dev:devices:D704206006772
Indexes	Sets or Sorted Sets (ZSETs) of entity IDs	backend:dev:devices:D704206006772:clients
Energy Data	Lists of JSON encoded data	backend:dev:energy:D704206006772:1494979200

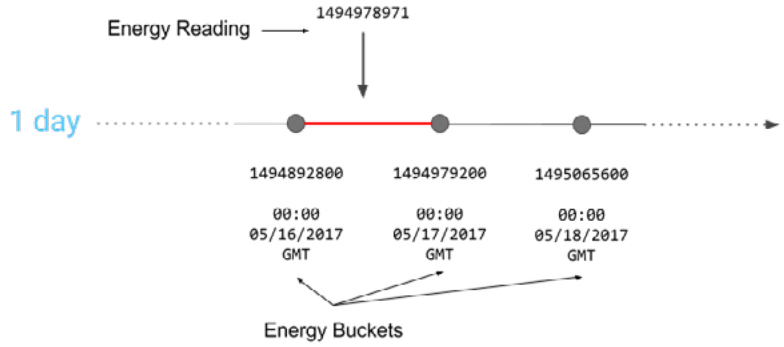
Most objects that would normally be stored in a relational table are now stored using Hash-es and accessed using indexes (Sets or Sorted Sets).

Energy data is ingested into LISTs, with each element being a JSON-encoded string as explained below.

Data Ingestion

Similar to the previous system, the new design uses Redis **as a time series database**.

We basically divide the time into blocks, with each block defined by its starting timestamp. Energy data is accumulated in each block until the current one becomes stale.



Energy data hits the backend though UDP packets and is stored in Redis Lists whose key is identified by the time bucket (like `backend:dev:energy:D704206006772:1494979200`).

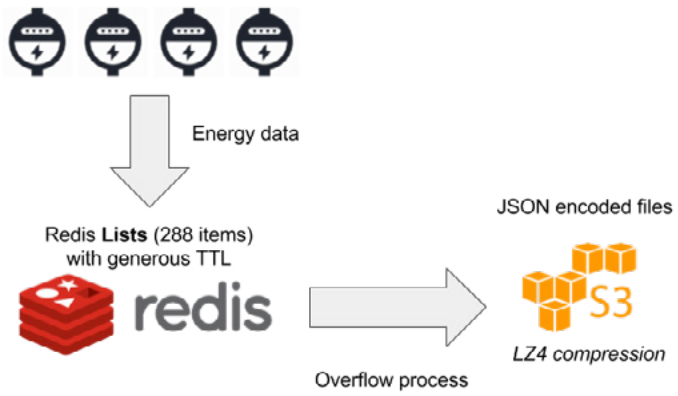


Figure 3. Data flow

Each list ID is also stored in a pending SET (backend:dev:energy:pending), which is scanned every few minutes for stale buckets. The scan operation (SSCAN) on the SET is O(n) time and space-wise, but as we continuously run it and remove elements from the SET, we are guaranteed that the SET size remains bounded. While SSCAN is an O(n) operation for a complete iteration, it is split up among multiple calls with each call being O(1), making it an ideal way of interacting with this type of structure.

Once a stale bucket is identified, its content gets compressed using the lightweight LZ4 algorithm and uploaded to Amazon S3. The choice of LZ4 is ideal in this scenario, because this advanced algorithm compresses the data roughly by half while increasing CPU usage only marginally. This creates a balanced compromise between storage cost on Amazon S3 and speed of access for querying.

Querying

Querying is somehow the slightly more complex aspect of this architecture. Any API call can query for historical energy data (hosted in S3) as well as current data (hosted in Redis). The APIs need to be smart enough to:

- Decompose any incoming energy query into the corresponding data buckets that need to be reached,
- Determine which datastore to hit: only Redis, only Amazon S3 or both and
- Merge the data before returning it to the client, while avoiding duplication.

All of the above is transparently handled by the API layer and hidden to the actual API clients.

Queries related to accounts, groups or devices always hit only Redis instead.

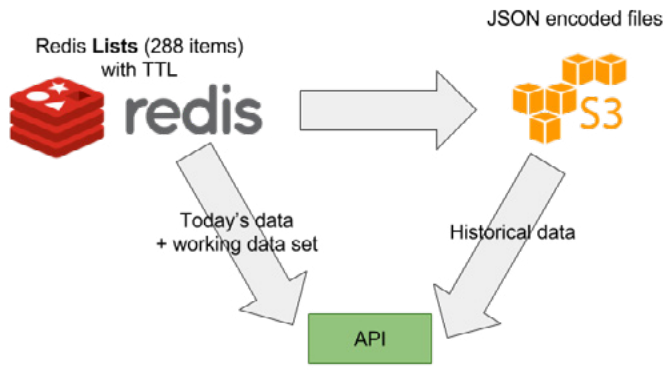


Figure 4. Query handling in the new solution

Review of the new design

The new architecture has the following characteristics:

- There is a learning curve in designing a data model which isn't relational.
- Querying for energy data is complex, as we are querying two different datastores (although they are both Key/Value).
- It's important to consider the possibility that not all necessary indexes may be available from the beginning, and so we will need to build new indexes in Redis over time (see section below).
- Amazon S3 is a Key Value datastore, which brings some limitations in terms of transactions per seconds (depending on the naming conventions of the JSON-encoded files it stores).

Unforeseen Indexes in Redis

As a matter of fact, we don't always know in advance all the possible ways to query the data. The same problem happens with traditional RDBMS-es, but it is masked by the use of a declarative query language (SQL). With RDBMs-es, devising the best strategy to handle unforeseen queries becomes challenging, and sometimes brute force methods like resorting to on-disk full table scans become necessary.

Building indexes on a traditional RDBMS is also simplified as it is built-in functionality, but this works relatively well for limited datasets only.

When Redis is the primary datastore there are a few options to query data for which there is no index (in the form of a either a SET or a ZSET):

Option	Pros	Cons
1) Use SCAN across a subset of the Redis keys or Sets (SSCAN) or Sorted Sets/ZSETs (ZSCAN)	No need to modify the data model	SCAN commands are O(n) time-wise for complete iterations which isn't ideal for potentially large datasets, however each call is O(1) making it less taxing on your server than a single O(n) command.
	It works well for small or bounded datasets	
2) Build new indexes as needed	It's the most scalable option even for large datasets	Requires some sort of offline scripting and changes to application code

If datasets are small, scanning through our dataset can be viable in the beginning, but option two should always be the long-term preferred solution.

After all, if we find ourselves continuously facing missing indexes, our dataset may be relational in nature and not well-suited for a NoSQL solution like Redis.

Amazon S3 Limitations

Amazon S3 is an infinitely scalable key value datastore offered as a managed service by Amazon Web Services. Given its design, performance and pricing, it makes a great companion for Redis even though the throughput it offers is comparable to all other disk-based solutions. For this specific application (as for many other comparable use cases), historical data does not require the same frequency of access, so Amazon S3's affordability and scalability far outweigh its limitations.

API management with Redis

One of the advantages of introducing Redis into any data architecture is its versatility. Any API layer requires some sort of API management and commercial products covering these requirements are often prohibitively expensive.

In practice, API management boils down to a limited set of features:

- Authentication
- Misuse and throttling prevention
- Usage statistics

Luckily, all of the above is very simple with Redis. Using the same approach as with the energy data time series, we can divide the time into intervals, store counters linked to each interval and increment them as requests come in.

```
let now = parseInt(new Date().getTime()/1000);
let tpsTs = now - now % 5; // 5 seconds ⟶  we divide the time in buckets
let tpsKey = redisKeys.getApiKeyTpsCounterKey(apiKey, tpsTs);

let pipeline = this.redisClient.pipeline();   ↘ apiKeys:key_08f...:tps:1488931200
pipeline.incr(tpsKey);
pipeline.expire(tpsKey, 60); // 1 minute
pipeline.get(tpsKey);                          ↘ usage data available for some time
pipeline.execute((err, res) -> { ... })
                                               ↘ we get the current tps back
                                                 so that we can throttle if necessary
↘ We use pipelining to improve
  performance
```

This way, we can always determine the current usage for an API key, prevent misuse, enable throttling and automatically expire usage data (using the built in EXPIRE command). This requires just a few lines of code, and avoids a performance penalty or the need to introduce another technology and/or tool in our architecture.

Achievements

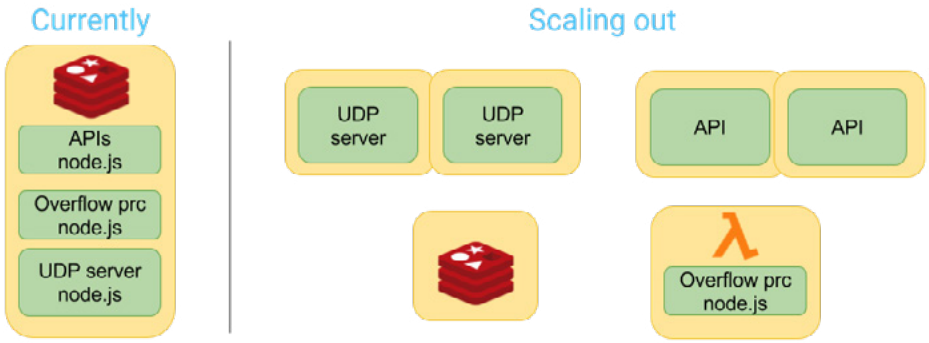
One of the most striking achievements of using Redis as a primary datastore is the reduced Total Cost of Ownership of the overall solution. The architecture described in this paper:

- Supports more than 50k smart meters sending data every 5 minutes,
- Uses an inexpensive Amazon EC2 m3.medium instance and
- Allows over 100k req/day to the APIs on the very same instance.

All of this is accomplished at a **TCO of <100 US dollars per month**. As an added bonus, it doesn't require any data maintenance, since all historical data is managed by Amazon S3 and policy buckets that autorotate it.

Scaling out the solution

The solution outlined above was designed to limit costs without sacrificing scalability. Its different components have been developed as microservices hosted on one single server (to start). The diagram below outlines a possible solution as we scale out:



In this architecture:

- Redis is hosted separately from the backend code so that we can easily introduce high availability via Redis Sentinel or 3rd party hosting via Redis.
- The UDP servers collecting data from the devices and the API servers are hosted on different servers and can be scaled out separately depending on the load.
- The overflow process of migrating data from Redis to Amazon S3 is not time critical, and can be easily run as a scheduled job in a serverless environment like Amazon Lambda.

More on scaling out

Using Redis Enterprise (either in the cloud as Redis Cloud or as downloadable software), it is possible to reduce any effort you might spend managing shards, high availability and large datasets. With Redis Enterprise, scaling is handled automatically without you having to worry about defining key<->shard distributions at the application level. Redis Enterprise also supports scaling out across a pool of servers, and utilizing all the cores on each server. (Redis is single-threaded, but Redis Enterprise manages multiple instances of Redis on each node, behind the scenes, thereby maximising resource utilization.)

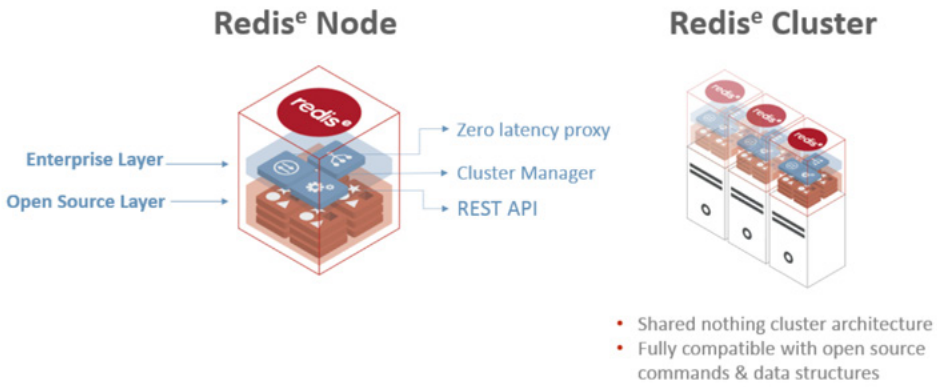


Figure 5. *Redis Enterprise Architecture*

With built-in cluster and node watchdogs, Redis Enterprise constantly monitors Redis processes and triggers failover within seconds to replicated instances across racks/zones/datacenters and even clouds -- making high availability simple.

Conclusions

- IoT and Redis are built around **efficiency**, which is the key for **low TCO**.
- A good system is designed with deep knowledge of its bottlenecks (there is no magic), and Redis has an amazing documentation around both the scalability of each data structure and the Big-O notation of each and every command.
- Modeling data in a non-relational way requires a new way of thinking, which can be intimidating at first but gets easier and easier to master over time. Redis still remains way easier to use than alternative Key/Value datastores like Cassandra, while offering higher throughput.
- It's always a good approach to build incrementally better systems in order to capitalize on operational knowledge. Redis worked great in a hybrid relational / non-relational model, as well as with a pure NoSQL approach.
- IoT is still in its infancy and is a R&D field where performance and freedom of experimentation are of the utmost importance. Redis plays really nicely from both points of view.

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com