redislabs
HOME OF REDIS

WHITE PAPER

RediSearch: A High Performance Search Engine as a Redis Module

Author: Dvir Volk, Senior Architect, Redis Labs

CONTENTS

Abstract

When the Redis Modules API came to being, it was clear that it would provide Redis with capabilities that will take it to new places, far beyond where client applications and Lua scripts are able to. At Redis Labs, we began to explore use cases that will both demonstrate the power of Modules, and allow us to extend Redis in unique ways to new unprecedented uses.

One of the less trivial cases that emerged pretty quickly was building a search engine. Redis is already used as a search engine by many developers via client libraries (we've even worked on a search client library implementation ourselves). However, we knew the power of Redis Modules would enable search features and performance that these libraries cannot offer.

So, among other modules (see www.redismodules.com), we started working on an interesting search engine implementation – built from scratch in C and using Redis as a framework. The result, RediSearch, is now available on Modules Hub, with a substantial feature-set. And in keeping with the Redis tradition of super high performance as a key feature in and of itself, it performs extremely fast in benchmarks.

In this paper, we'll discuss the internal design and features of RediSearch and demonstrate its current capabilities.

RediSearch At-a-Glance

RediSearch is a full-text search engine developed from scratch on top of Redis, using the new Redis Modules API to extend Redis with new commands and capabilities. Its main features include:

- Simple, fast indexing and searching
- Data stored in RAM, using memory-efficient custom data structures
- Support for multiple languages using utf-8 encoding
- Document and field scoring
- Numeric filtering of results
- Query expansion by Stemming
- Exact phrase search
- Filtering results by specific properties (e.g. search "foo" in title only)
- Powerful auto-suggest engine
- Incremental indexing (without the need to optimize or vacuum the index)
- Support for use as a search index for documents stored in another database
- Support for indexing of existing HASH objects already in redis as documents
- Scaling to multiple Redis instances

A Little Taste: RediSearch in Action

Let's look at some of the key concepts of RediSearch using this example over the redis-cli tool:

1. Defining an Index

In order to search effectively, RediSearch needs to know how to index documents. A document may have several fields, each with its own weight (e.g. a title is usually more important than the text itself). The engine can also use numeric fields for filtering (and we may add mode options in the future).

Hence, the first step is to create the index definition, which tells RediSearch how to treat the documents we will add.

In this example, we'll create a product catalog index, where each product has a name, description and price (allowing us to filter products by price).

We use the **FT.CREATE** command, which has the syntax:

```
FT.CREATE <index_name> <field> [<score>|NUMERIC] ...
```

Or in our example:

```
FT.CREATE products name 10.0 description 1.0 price NUMERIC
```

This means that *name* and *description* are treated as text fields with respective scores of 10 and 1, and that price is a numeric field used for filtering.

2. Adding Documents

Now we can add new products to our index with the **FT.ADD** command:

```
 FT.ADD <index_name> <doc_id> <score> FIELDS <field> <value> …
```

Adding a TV for example:

```
FT.ADD products prod1 1.0 FIELDS \
name "Acme 40-Inch 1080p LED TV" \
description "Enjoy enhanced color and clarity with stunning \ Full HD 1080p" \
price 277.99
```

The product is added immediately to the index and can now be found in future searches.

Note: RediSearch supports utf-8 (or plain ASCII) encoded text, and does not check for validity of the input. Its main constraints are that tokenization uses ordinary spaces and punctuation marks, which work for both ASCII and utf-8.

3. Searching

Now that we have added products to our index, searching is very simple:

```
FT.SEARCH products "full hd tv" LIMIT 0 10
```

Or with price filtering between $200 and $300:

```
FT.SEARCH products "full hd tv" FILTER price 200 300
```

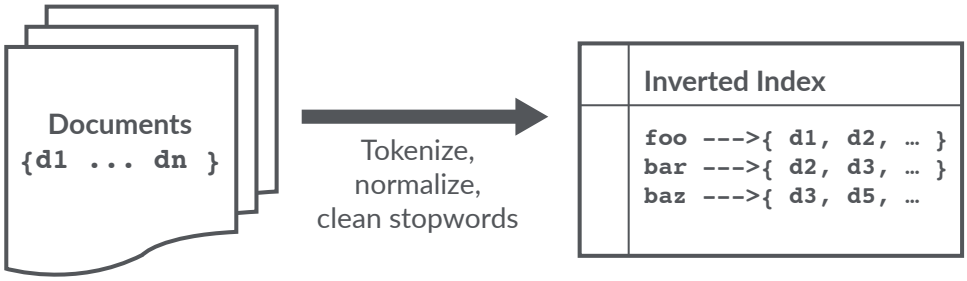
The results are shown in the redis-cli as:

```
1) (integer) 1
2) "prod1"
3) 1) "name"
   2) "Acme 40-Inch 1080p LED TV"
   3) "description"
   4) "Enjoy enhanced color and clarity with stunning Full HD 1080p at twice the res-
olution of standard HD TV"
   5) "price"
   6) "277.99"
```

Now that we've demonstrated how it works, we'll dive in a bit to the design of RediSearch.

The Theory: A Crash Course in Search and Redis

The fundamental part of any search engine is what's called an Inverted Index. In very simple terms (and do read some more about this if you have some time), a search index is a map between words or terms, to the respective documents they appear in.



Creating an Inverted Index from a collection of documents

So if we have two documents, one titled "Hello World" and the other titled "Hello Kitty," a search engine would keep an Inverted Index of the word "hello" that contains two records, one for each of the aforementioned documents.

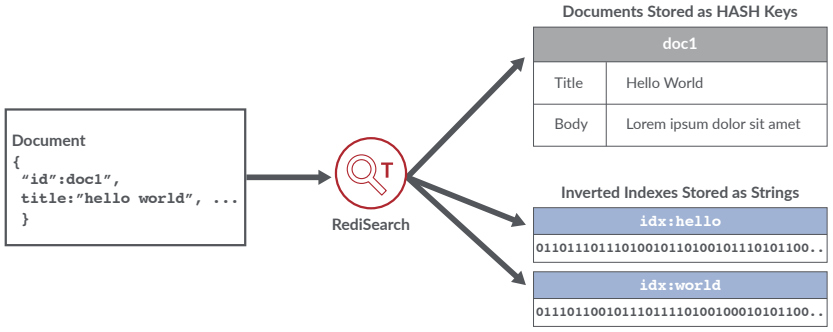
Searching consists of loading, traversing, intersecting (if needed) and sorting these indexes in order to produce the most relevant results. It gets much more complicated, of course, as those indexes contain relevant scores, term positions in the document and more – but that's beyond the scope of this piece.

Until now, Inverted Indexes on top of Redis were always modeled with Redis' native data types—usually Sorted Sets, where the document ID was stored as the "element" and the relevance as the "score." This worked great because it allowed developers to use Redis' intersection and union to perform multi-word searches, but the approach has its limitations: it doesn't support exact phrase search, it has a big memory overhead, and can be very slow with big records intersections.

The Design: Departing From Redis' Data Structures

Our first design choice with RediSearch was to model Inverted Indexes using a custom data structure. This allows for more efficient encoding of the data, and possibly faster traversal and intersection of the index. In addition, encoding word positions in the index enables exact phrase searches. Although the Modules API did not support custom data types when this work began, we were able to use another, simpler feature: Redis DMA (Direct Memory Access) Strings.

This feature is simple, yet very powerful. It basically allows modules to allocate data on Redis string keys, then get a direct pointer to the data allocated by each key, without copying or serializing it. This ensures very fast access to huge amounts of memory. Since, from the module's perspective, the string value is exposed simply as a chunk of memory, it can be used as a data structure (especially if it is in serialized form).



Storing document content and index data on different redis data types

Encoding the Inverted Index

Luckily, Inverted Indexes are usually stored in serialized compressed form, and read one record at a time to compress them. This makes DMA strings a perfect fit for our needs: we save the Inverted Indexes in Redis using a technique that combines Delta Encoding and Varint Encoding to encode entries—minimizing space used for indexes, while keeping decompression and traversal efficient.

For each entry we encode:

- The document ID (as a delta from the previous document)
- The term frequency, factored by the document's rank
- Flags that can be used to filter only specific fields
- An Offset Vector, with all the document offsets of the word

This lets us encode a single index entry in as little as 6 bytes (Note: this is the best case. Depending on the number of a word's occurrences in the document, this can get much higher). On top of that, we keep extra data structures, like a Skip Index of the Inverted Index, for faster intersections.

But all these binary structures are modeled, from the Redis perspective, as simple strings. When a search is performed, we simply get a pointer to that string from Redis, and from there on, we read the index records as a stream of binary encoded data. In fact, the internal index code has no concept of it running on top of Redis, and in unit tests it runs on plain malloc'd memory.

We also keep metadata about the index itself and the document content in traditional Redis structures, since Redis does a great job of storing these types of data.

Document and result ranking

In general, search engine users always want the most relevant results for their search term. This is achieved through various methods, typically by considering both the quality of documents, and the relevance of the search terms to those documents. Google's revolutionary (at the time) PageRank algorithm is a good example of document quality measurement.

In the case of RediSearch, there is no automatic algorithm to deduce the quality of a document. Rather, it is left to the user to provide a quality score for each indexed document. As seen above, every document entered into the engine using **FT.ADD** is given a user-assigned rank normalized between 0 and 1.0. Currently, this score is immutable and cannot change after the document is added to the index.

But of course, document ranking is not enough. We combine the document score with TF-IDF scoring of each word in order to rank the results. Other relevance ranking methods will be added in the future.

The frequency of a term in a document is calculated using a separate optional weight for each field, allowing the user to specify the importance of certain fields. For example, the title of a page might be 10 times more important than the body, and thus we assign it a factor of 10, and the body a factor of 1. This ensures that terms appearing in the title influence search results much more than terms in the body.

On top of that, in the case of multiple word intersect type queries (foo AND bar), we take the minimal distance between the terms in the query, and factor that into the ranking. The closer the terms are to each other, the higher the result. When searching, we keep a priority queue of the top N results requested, and return them sorted by rank.

Storing Documents

Our engine can index “documents”, which are basically a list of field-value pairs. The index knows how to index each field, but that’s not enough—we need to actually store the data for retrieval. In a simple use case, you can just push documents to RediSearch, and it creates a complete index and document database, from which you can retrieve whole documents.

For storing documents, we use simple Redis HASH keys, where each document is represented by a single key, and each property and its value by a HASH key and element. For retrieval, we simply perform an HGETALL query on each retrieved document, returning its entire data. If the user needs to retrieve a specific document by its id, a simple HGETALL can be performed by the user.

It is not mandatory to save the raw document content when indexing a document, and you can even use Redis as a search index for another database. In that case, you simply tell Redis not to store the data by adding the **NOSAVE** modifier to the indexing command.

Indexing Existing Redis Data

Since documents are stored as HASH objects in redis, it is trivial for RediSearch to index HASH objects already in redis, and treat them as documents.

If the user has already created HASH keys in redis with HSET or HMSET, RediSearch can index these documents automatically. This is useful for creating indexes on existing redis databases.

The command FT.ADDHASH tells RediSearch which HASH object should be added to an index, and the module reads that object, and indexes the fields in it that correlate to the index’s own field definition.

Numeric Filters

As shown in the example above, RediSearch supports numeric range filtering of results. This is done by defining fields in the index as NUMERIC. Numeric values in documents for these fields will be indexed and used for filtering the results in runtime. Currently this is implemented using sorted sets, but that implementation is likely to change in the future.

When searching, we can add FILTER predicates to the query (see example), limiting the results to documents with specific numeric values. The syntax for **FILTER** follows the syntax of ZRANGEBYSCORE limits, and can be exclusive or inclusive.

Query Execution Engine

We use a chained-iterator based approach to query execution, similar to Python generators in concept. The index can create “iterators” with a uniform interface that can combine the following actions:

- **read** the index of a word
- **union** multiple iterators
- **intersect** multiple iterators, or
- intersect multiple iterators only if the result is an **exact phrase**

All these iterators are lazy evaluated, entry-by-entry, with constant memory overhead.

To execute a query, we build an execution tree – a chain of these iterators based on parsing the query text and filter predicates. The resulting tree has a "root" iterator, which is read by the query execution engine and filtered for the top N results. The root iterator would in turn iterate its children (if it has any), and so forth.

Here are a few examples of how different queries are parsed into execution plans:

Case	Example	Execution Chain
single word query	hello	`READ("hello")`
phrase (AND) query	hello world	`INTERSECT { READ("hello"), READ("world") }`
Phrase with stemming	Hello worlds	`INTERSECT { READ("hello"), UNION { read("world"), read("worlds") } }`
Exact term	“hello world”	`EXACT_INTERSECT { READ("hello"), READ("world") }`
Numeric Filter	tv, FILTER price 0 300	INTERSECT { READ("tv"), NUMERIC { 0 <= price <= 300 } }

Stemming and Query Expansion

Query expansion is a technique used in search engines, whereby a user’s search term will prompt the engine to also search for other terms that might yield relevant results. Common techniques are synonyms, fuzzy matching of words spelled a little differently, and stemming. Currently (but this will change in the future), RediSearch only enables stemming support.

Stemming is a technique of reducing a word to its “stem” or root form, such as taking a plural word to singular, or removing other inflection types. For example, the words “driving,” “drives” and “driver” are all based on the word “drive.”

Redis uses the Snowball stemming library, which provides stemming for over 15 languages. When indexing a document containing any of those words, RediSearch will index them twice – once as the original term, and once as the stem.

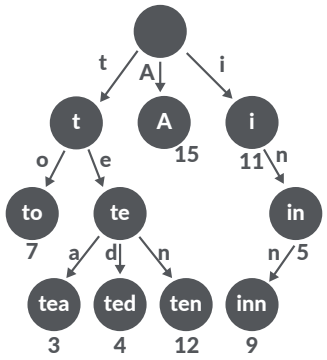
The same is done in query time. Searching for “driver” will also search for the stem “drive,” thus yielding results for “driving” and “drives” as well. Priority is given to the exact same inflection of the word, but search results will also contain other inflections, usually ranked lower.

This feature can be disabled at query time by adding the modifier **VERBATIM** to the **FT.SEARCH** command after the query itself.

Note: Currently, RediSearch does not attempt to guess the document or query language. If search terms are not English, you should set the language for indexing and querying using the LANGUAGE modifier. The list of supported language can be found in the module’s documentation.

Auto-Complete and Fuzzy Suggestions

Another important feature for RediSearch is its auto-complete or suggest commands. This allows you to create dictionaries of weighted terms, and then query them for completion suggestions to a given user prefix. For example, if a user starts to put the term "Icd tv" into a dictionary, sending the prefix "Ic" will return the full term as a result. The dictionary is modeled as a trie (prefix tree) with weights, which is traversed to find the top suffixes of a prefix.



Example prefix tree. Source: Wikipedia.

RediSearch also allows for Fuzzy Suggestions, meaning you can get suggestions to prefixes even if the user makes a typo in their prefix. This is enabled using a Levenshtein Automaton, allowing efficient searching of the dictionary for all terms within a maximal Levenshtein Distance of a term or prefix. Then suggestions are weighted based on both their original score and their distance from the prefix typed by the user.

However, searching for fuzzy prefixes (especially very short ones) will traverse an enormous number of suggestions. In fact, fuzzy suggestions for any single letter will traverse the entire dictionary, so we recommend using this feature carefully, in consideration of the performance penalty it incurs. Since Redis is single threaded, blocking it for any amount of time means that no other queries can be processed at that time.

Note: Currently, fuzzy searching will work correctly with the Latin alphabet only. However, the general approach of auto-complete suggestions works on any utf-8 encoded text.

An example of creating and searching an auto-complete dictionary in RediSearch:

1. Adding some terms and their weights into a dictionary called "completer" (notice that the return value is the size of the dictionary after insertion):

```
127.0.0.1:6379> FT.SUGADD completer "foo" 1
(integer) 1
127.0.0.1:6379> FT.SUGADD completer "football" 5
(integer) 2
127.0.0.1:6379> FT.SUGADD completer "fawlty towers" 7
(integer) 3
127.0.0.1:6379> FT.SUGADD completer "foo bar" 2
(integer) 4
127.0.0.1:6379> FT.SUGADD completer "fortune 500" 1
(integer) 5
```

2. A simple prefix search, including scores:

```
127.0.0.1:6379> FT.SUGGET completer foo
1) "football"
2) "foo"
3) "foo bar"
127.0.0.1:6379> FT.SUGGET completer foo WITHSCORES
1) "football"
2) "2.0412414073944092"
3) "foo"
4) "1"
5) "foo bar"
6) "0.89442718029022217"
```

3. Fuzzy prefix searches (note that we get results whose prefix is at most 1 Levenshtein Distance from the given prefix):

```
127.0.0.1:6379> FT.SUGGET completer foo FUZZY
1) "football"
2) "foo"
3) "foo bar"
4) "fortune 500"
127.0.0.1:6379> FT.SUGGET completer faulty FUZZY
1) "fawlty towers"
```

Scaling RediSearch

While RediSearch is very fast and memory efficient, if an index is big enough, at some point it will be too slow or consume too much memory. Then, it will have to be scaled out and partitioned to several machines.

Traditional clusters map different keys to different "shards" to achieve this. However, in search indexes, this approach is not practical. If we mapped each word's index to a different shard, we would end up needing to intersect records from different servers for multi-term queries.

The way to address this challenge is to employ a technique called Index Partitioning, which is very simple at its core:

- The index is split across many machines/shards by **document ID**
- Every shard has a complete index of all the documents mapped to it
- We query **all shards concurrently**, and use client side logic to merge these results into a single result
- When searching for documents, we send the same query to N shards, each holding a sub index of 1/N documents
- Since we're only interested in the top K results of all shards, each shard returns just its own top K results
- We then merge the N lists of K elements, and extract the top K elements from the merged list

This is pretty straightforward, however the current Redis Modules API does not allow it to happen on the server side, so we've implemented a special client in Go, for the purpose of benchmarking. This client is publicly available as a reference implementation. It needs to be ported to other programming languages, in order to scale RediSearch while working with them.

There are two approaches being considered, to solve this issue:

4. Creating a gateway server to RediSearch, which will automate this and provide a simplified entry point, doing all the aforementioned logic behind the scenes.
5. In time, a cluster API for Redis Modules is planned. Once it is available, this logic could be moved into the module itself, and no special client will be needed.

Benchmark Results

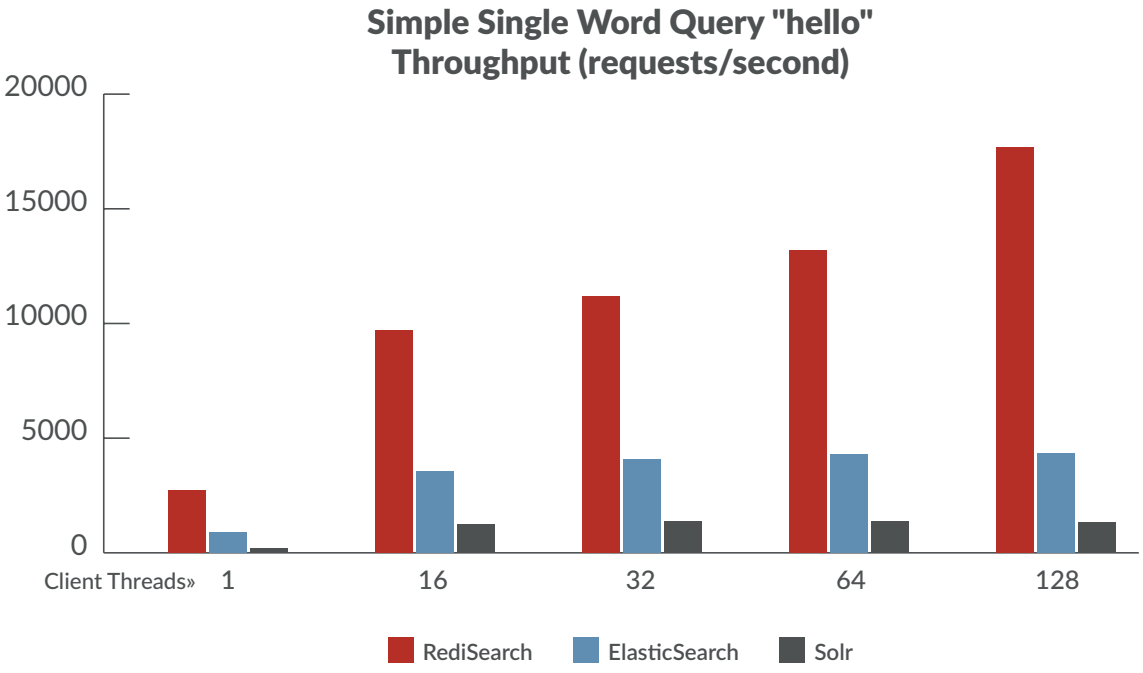
To assess the performance of RediSearch compared to other open source search engines, we've created a set of benchmarks measuring latency and throughput – while indexing and querying the same set of data. Our benchmark results show that RediSearch performs between 120% to 500% faster as compared to ElasticSearch and Solr.

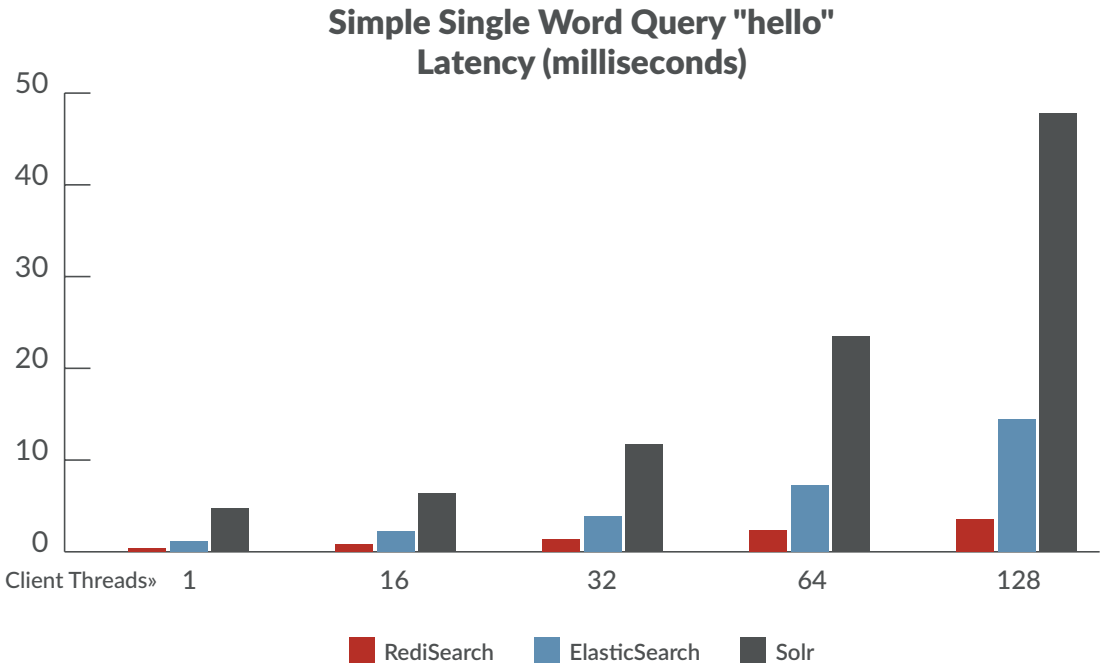
The Benchmark Setup

- **The data set:** A dump of all English abstracts available from Wikipedia pages, containing about 5.1 million short abstracts.
- **The benchmark:** We ran several queries with different profiles against all search engines. Each query was executed with 1, 8, 16, 32 and 64 concurrent clients running in parallel. We also ran an autocomplete benchmark, which tested the top 1100 most popular 2 and 3-letter prefixes from the data set with the same client concurrency profiles.
- **The physical setup:** Two c4.4xlarge AWS EC2 instances, each with 16 cores, 30GB of RAM and SSD EBS storage. One acts as the client, and the other runs the servers.
- **The engines tested:**
  - **RediSearch:** 5 shards running on 5 Redis masters, with no load balancing, redundancy or built-in caching. This setup used **at most 5 CPU cores** of the server machine.
  - **ElasticSearch:** One instance with 5 shards, with filter caching disabled. ElasticSearch utilized **all 16 CPU cores** during the benchmark, since it is multi-threaded.
  - **Solr:** Two instances of solr-cloud, each with 2 shards running on it. Caching was completely disabled, and Solr utilized all 16 CPU cores during the benchmark as well.
- **The client:** An application written in Go, utilizing as many CPU cores as possible, using popular Go clients for Redis, ElasticSearch and Solr. The source for it is publicly available on github.

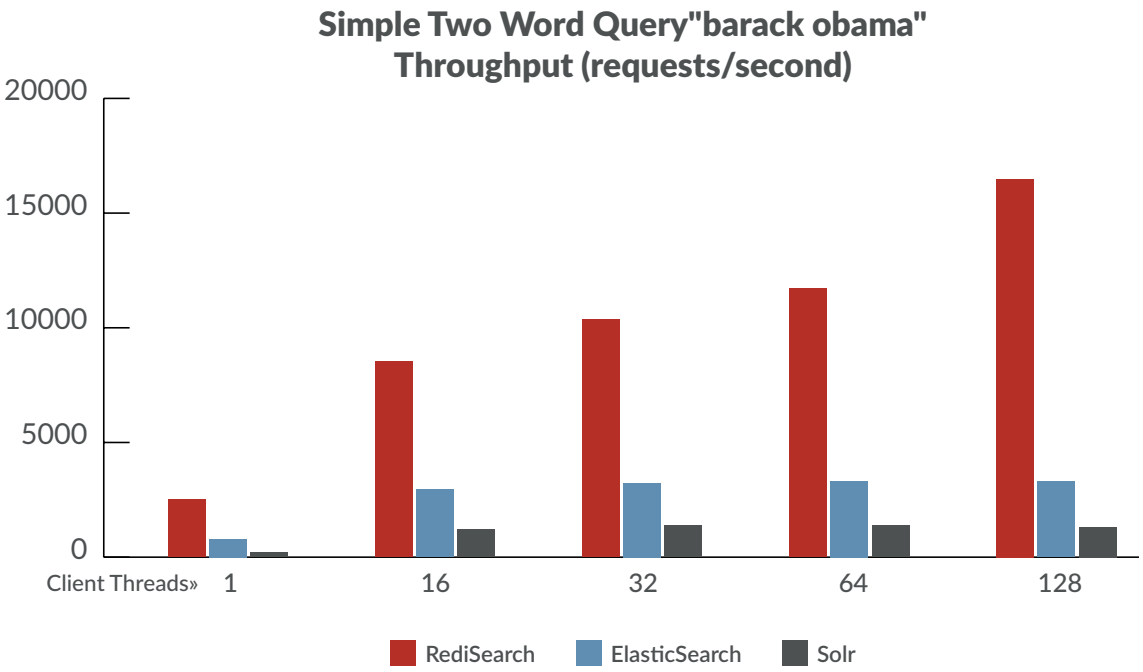
The Results

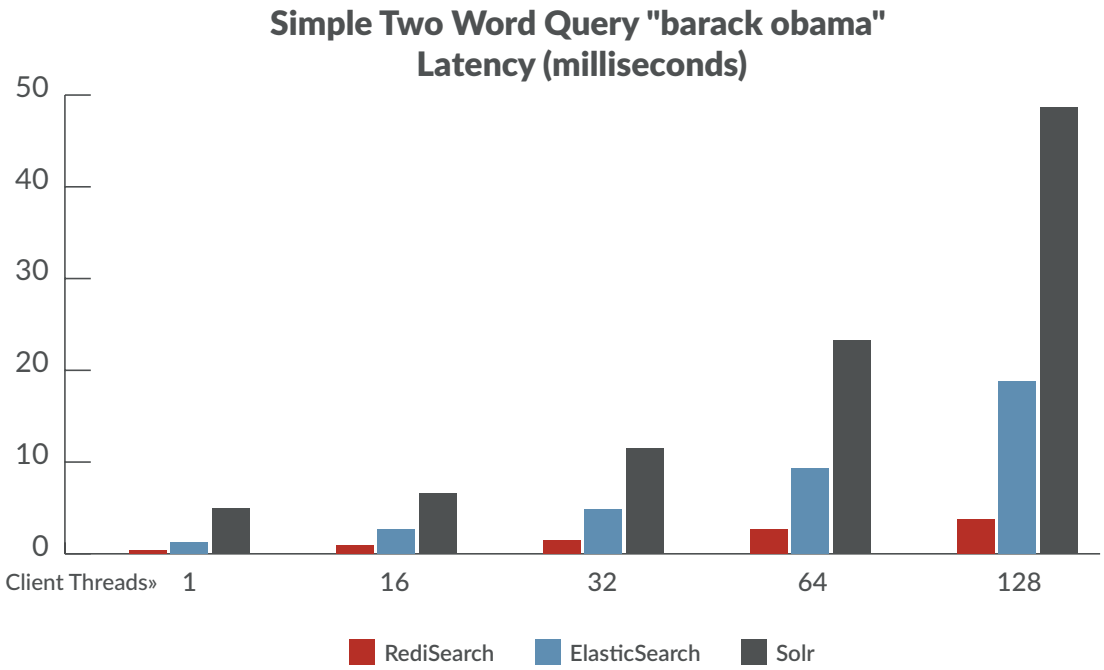
Benchmark 1: Easy single-word query - **hello**



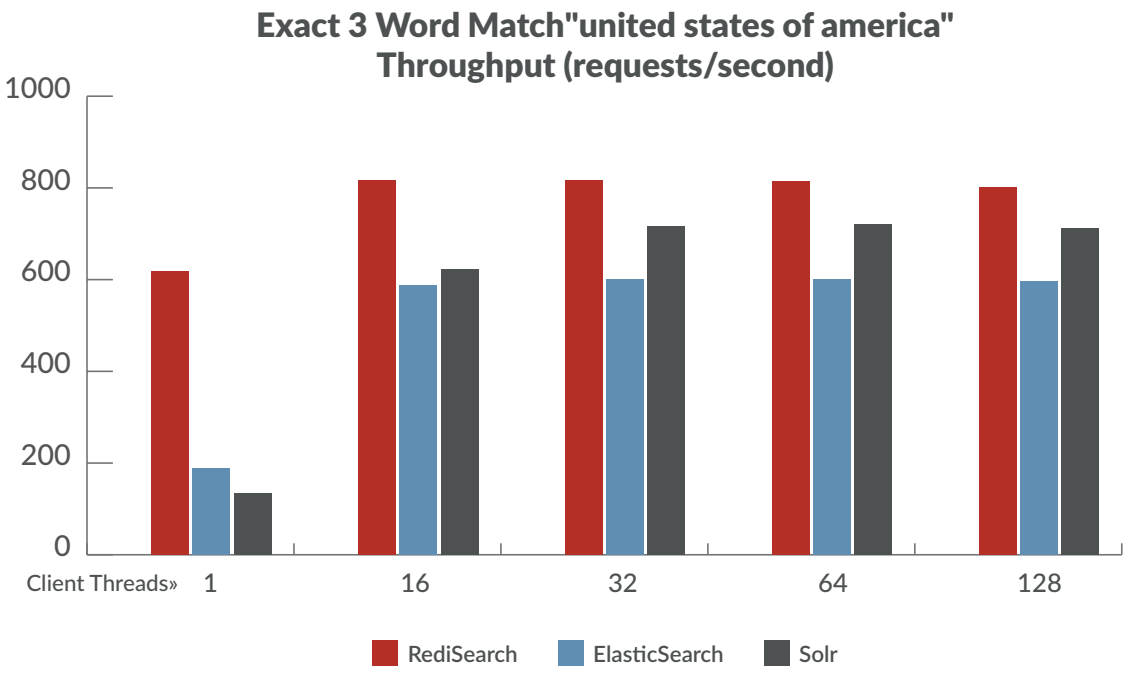


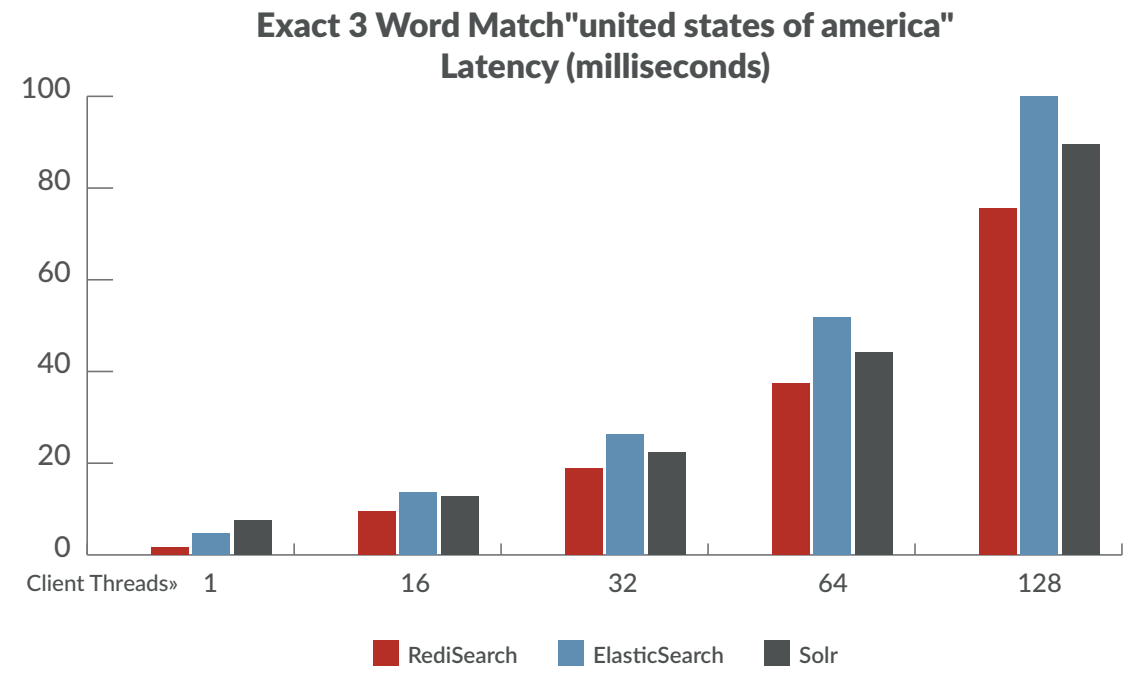
Benchmark 2: Simple two word query - **barack obama**



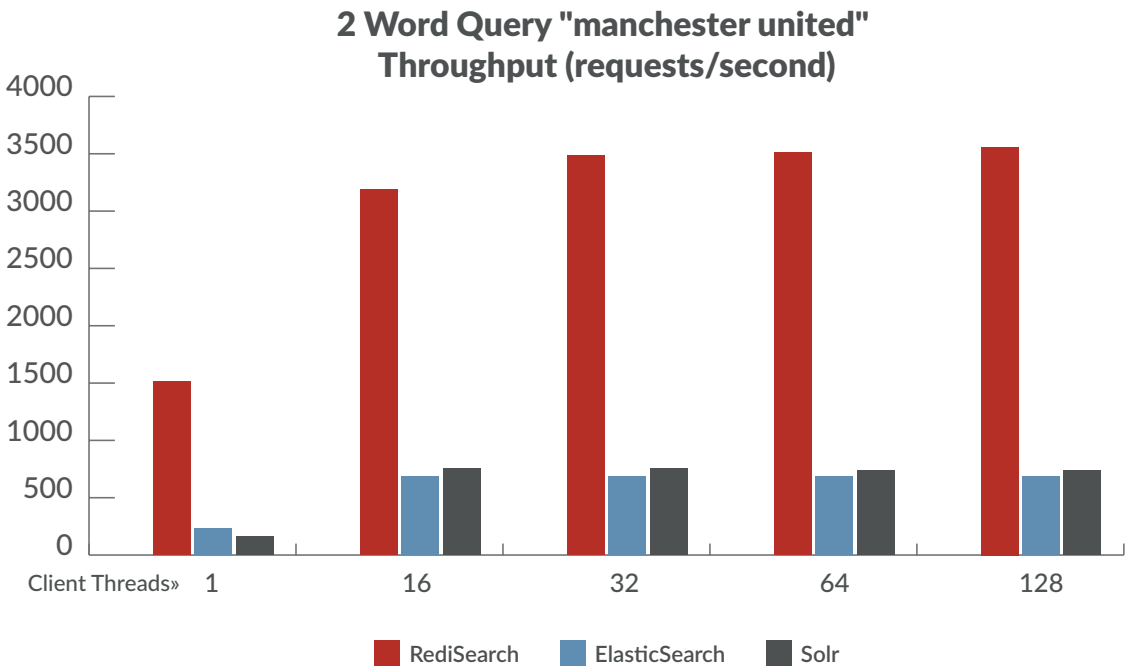


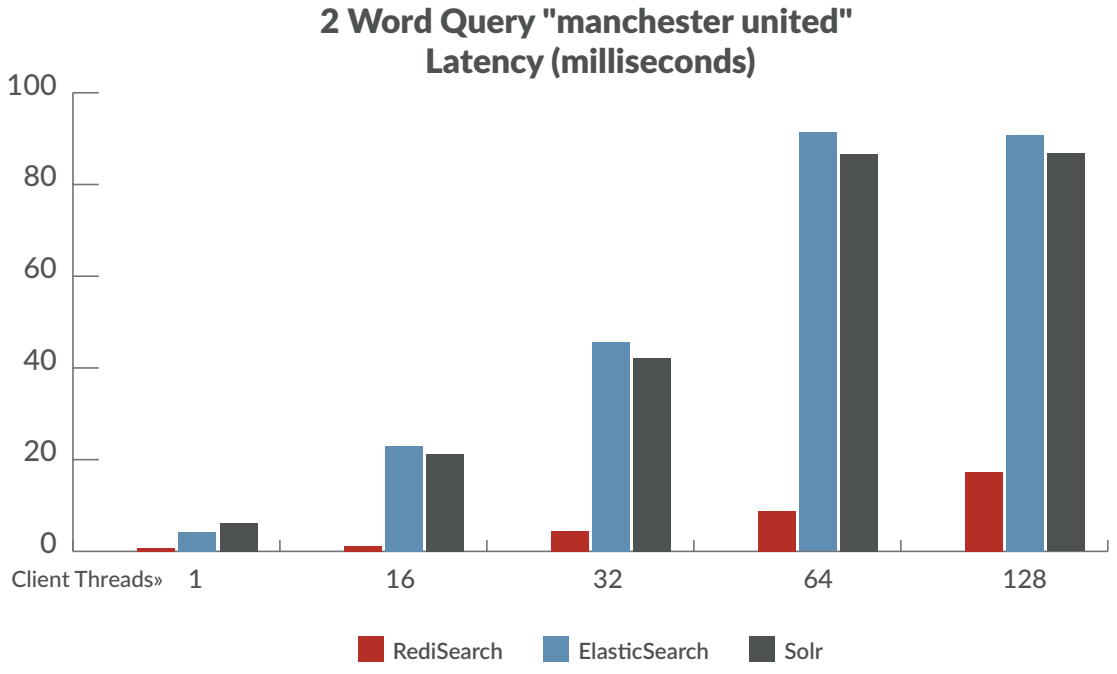
Benchmark 3: Exact word match query - **“united states of america”**



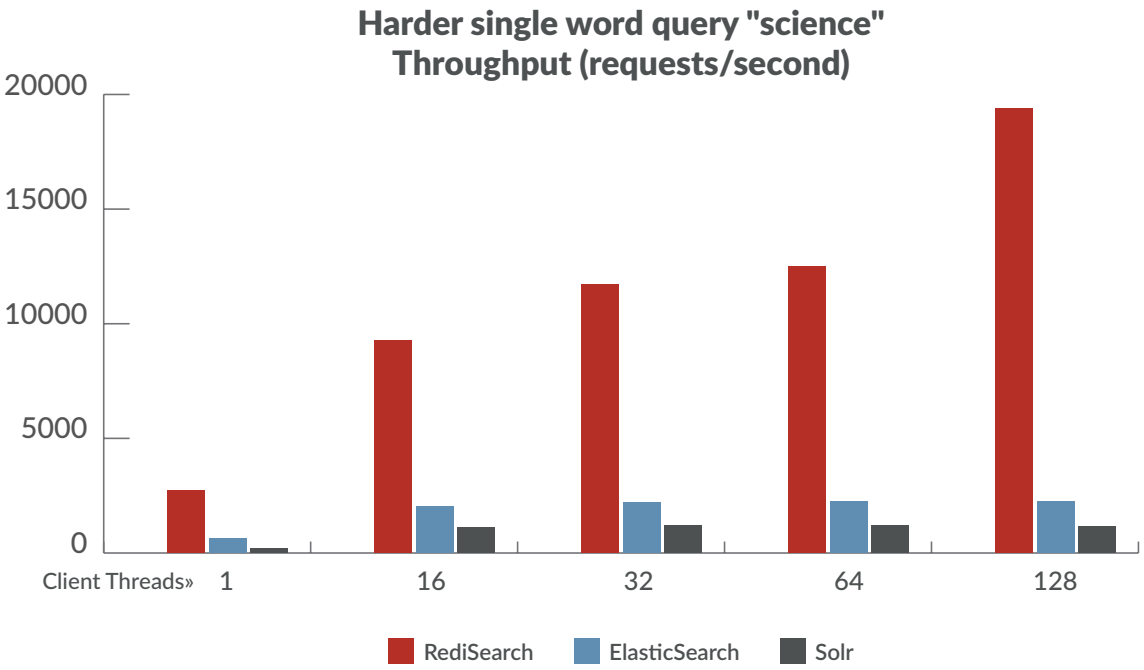


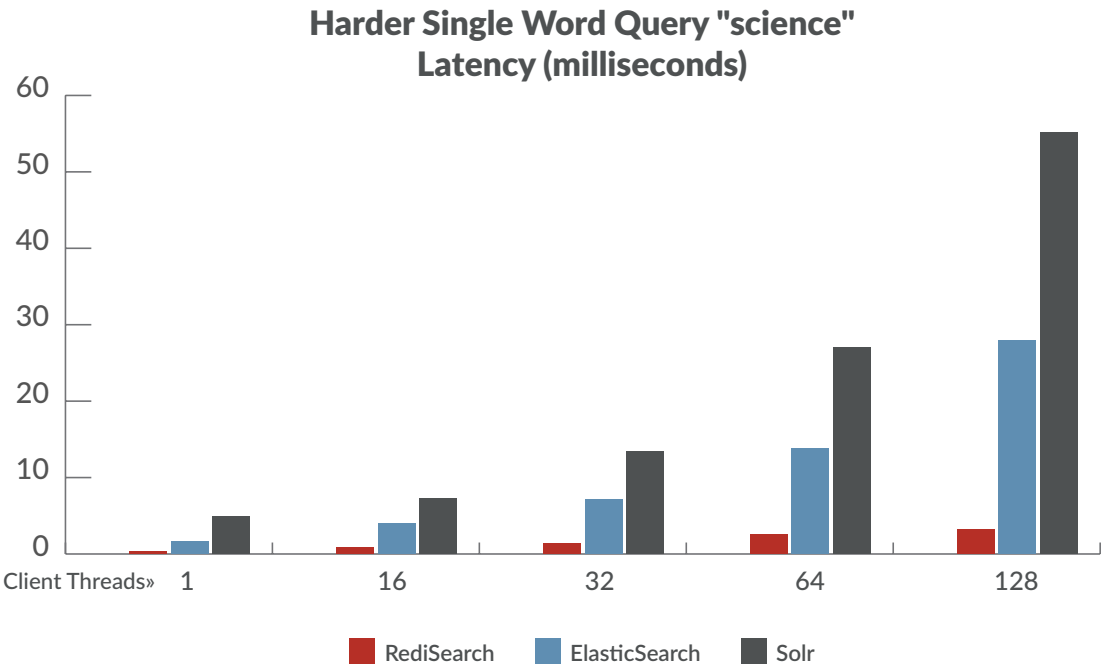
Benchmark 4: Two word query - **manchester united**



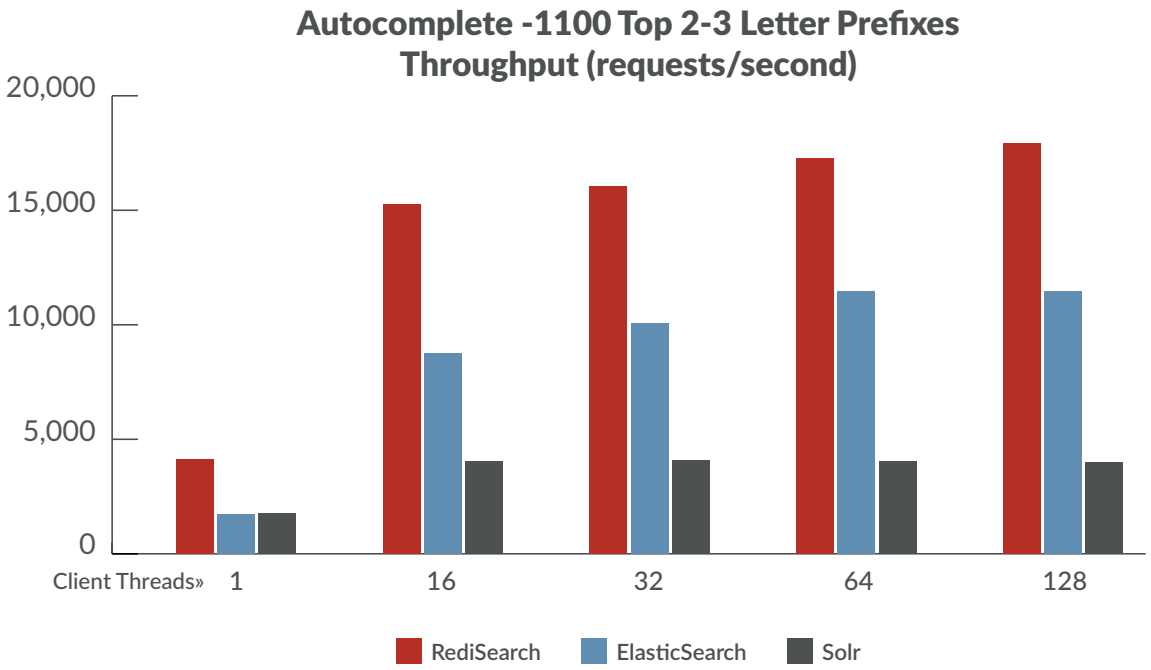


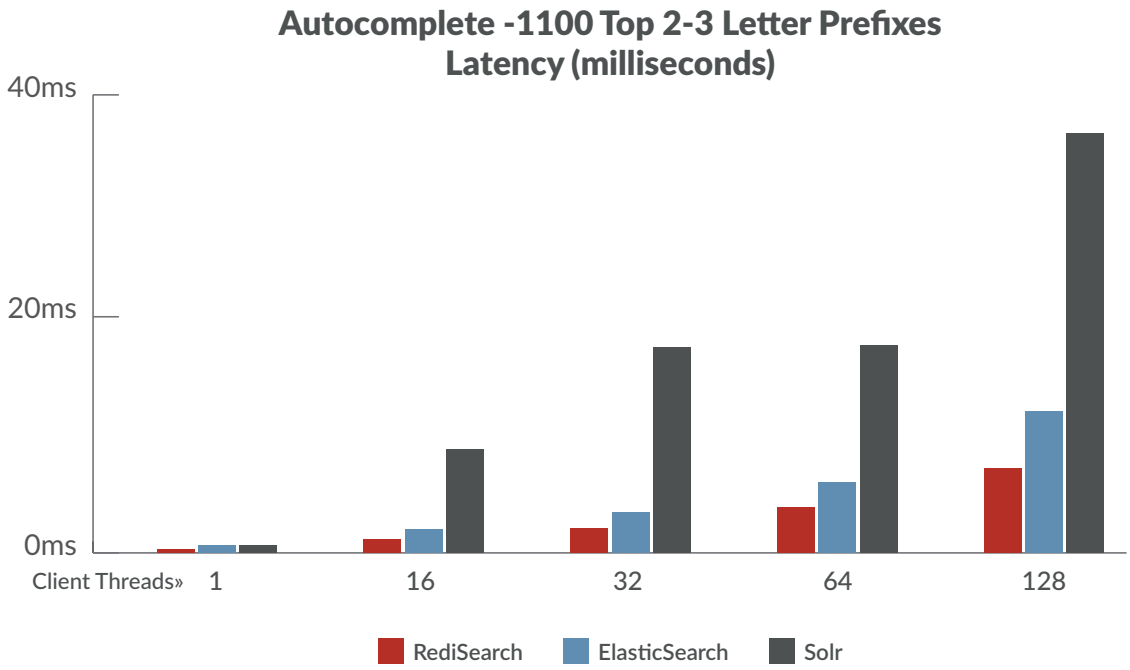
Benchmark 5: Difficult (too many results) query-single word - **science**





Benchmark 6: Autocomplete -**1100 top 2-3 letter prefixes on Wikipedia**





Conclusion

We have demonstrated that utilizing the Redis Modules API, we can create a feature rich and high performance search engine on top of Redis.

As can be seen from the benchmarks, Redis outperforms other search engines using this module, by great margins in some cases - while keeping lower latencies.

While RediSearch is still gaining features compared to ElasticSearch and Solr, its performance advantages make it a viable alternative for many use cases, and over time we intend to enhance it with the help of the open source community.

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