



Home of Redis

# Redis Enterprise Developer Workshop

## Modules Overview



# Redis Labs – Home of Redis

---



The commercial company behind Open Source Redis



Provider of the **Redis Enterprise (Redis<sup>e</sup>)** technology, platform and products

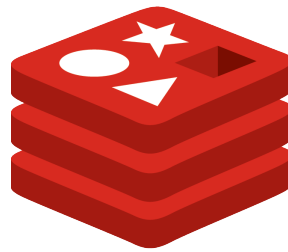
*Founded in 2011*

*HQ in Mountain View CA, R&D center in Tel-Aviv IL*

# Topics

---

- Overview of the Redis Module System
- Redis-ML and Spark-ML modules
- reJSON
- Redisearch
- Redis Graph

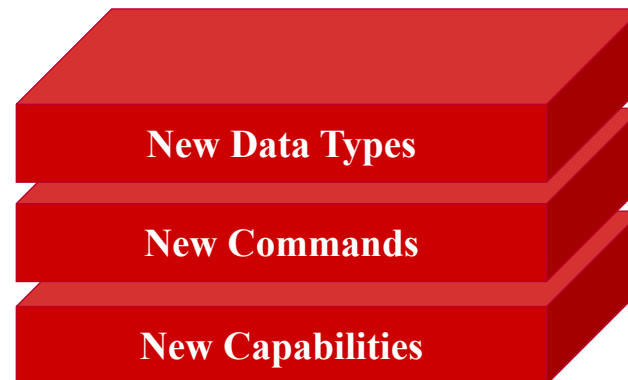
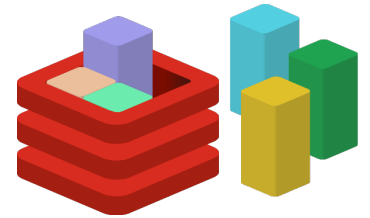


# Overview of Redis Modules



# Redis Modules

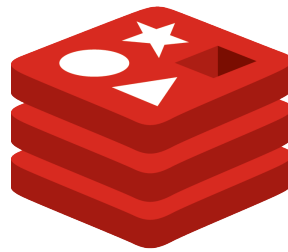
- Any C/C++ program can now run on Redis
- Use existing or add new data-structures
- Enjoy simplicity, infinite scalability and high availability while keeping the native speed of Redis
- Can be created by anyone



# Modules : A Revolutionary Approach

Adapt your database to your data, not the other way around

<b>Neural Redis</b> Simple Neural Network Native to Redis	<b>Redis-ML</b> Machine Learning Model Serving	<b>Redisearch</b> Full Text Search Engine in Redis
<b>ReJSON</b> JSON Engine on Redis. Pre-released	<b>Time Series</b> Time series values aggregation in Redis	<b>Graph</b> Graph database on Redis based on Cypher language
<b>Rate Limiter</b> Based on Generic Cell Rate Algorithm (GCRA)	<b>Crypto Engine Wrapper</b> Secure way to store data in Redis via encrypt/decrypt with various <a href="#">Themis</a> primitives	<b>Secondary Index/RQL</b> Indexing + SQL-like syntax for querying indexes. Pre-released



## Redis-ML and Spark-ML



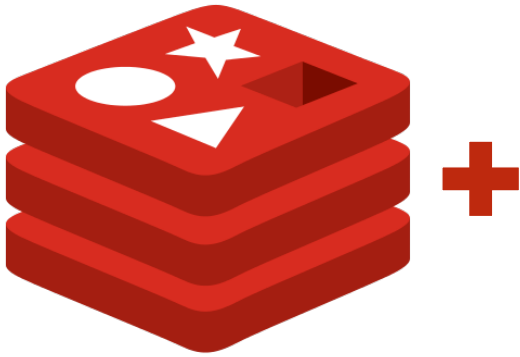
# Redis-ML Module

---

- Predictive models as a native Redis Data Type (4.0 Modules API)
- Predictive model serving engine
- Multiple supported models
- Training framework independent



# Redis ML Module



## Redis Module

Tree Ensembles

Linear Regression

Logistic Regression

Matrix + Vector Operations

More to come...

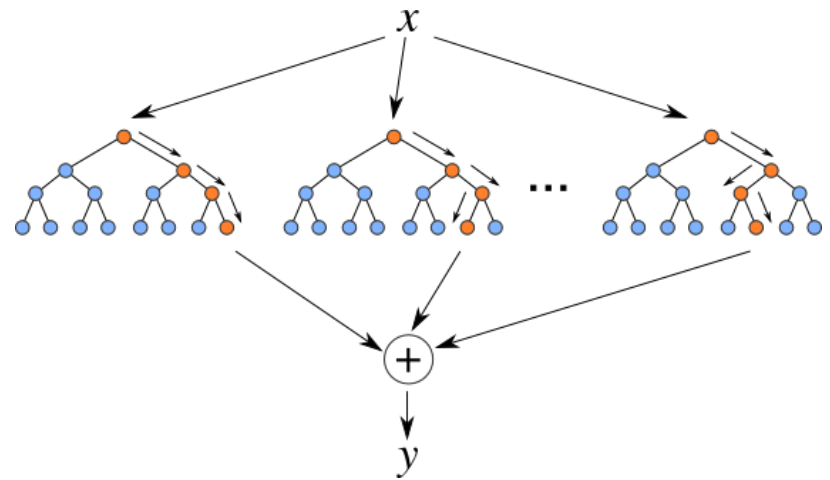
# Redis-Spark Module

---

- Connector between Apache Spark and Redis-ML
- Train models using Apache Spark
- Load trained models into Redis

# Example: Random Forest

- A collection of decision trees
- Splitter Node can be:
  - Categorical (e.g. day == "Sunday")
  - Numerical (e.g. age < 43)
- Decision is taken by the majority of decision trees



# Forest Data Type Commands

---

Add nodes to a tree in a forest:

```
ML.FOREST.ADD <forestId> <treeId> <path>  
  [ [NUMERIC|CATEGORIC] <splitterAttr> <splitterVal> ] |  
  [LEAF] <predVal>
```

Perform classification/regression of a feature vector:

```
ML.FOREST.RUN <forestId> <features>  
  [CLASSIFICATION|REGRESSION]
```

# Forest Data Type Example

```
> ML.FOREST.ADD myforest 0 . CATEGORIC sex "male" .L  
  LEAF 1 .R LEAF 0  
OK  
> ML.FOREST.RUN myforest sex:male  
"1"  
> ML.FOREST.RUN myforest sex:something_else  
"0"
```

# Real World Challenge: Ad Serving

---

- Need to serve 20,000 ads/sec @ 50msec data-center latency
- Runs 1k campaigns → 1K random forest
- Each forest has 15K trees
- On average each tree has 7 levels (depth)

# Real World Challenge

---

- Ad serving company
- Need to serve 20,000 ads/sec @ 50msec data-center latency
- Runs 1k campaigns → 1K random forest
- Each forest has 15K trees
- On average each tree has 7 levels (depth)

# Ad Model Serving: Infrastructure Cost Saving

**Cut computing infrastructure  
by 97%**



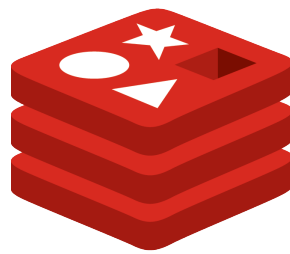
**Homegrown**

**1,247 x c4.8xlarge**



**35 x c4.8xlarge**





reJSON



# What is JSON

- JavaScript Object Notion
- Data interchange format
  - Lightweight
  - Human Readable
  - Simple to Parse
  - Simple to Generate
- ECMA Standard
- Common web development format

```
{  
  "userId": 0,  
  "firstName": "Tague",  
  "lastName": "Griffith",  
  "userName": "tague"  
}
```

# Storing JSON in Redis

## Serialized as Redis String

- GET/SET operations
- $O(N)$  access
- Client deserialization
- No in-place updates

## Deserialized in Hash

- Decode: HMSET
- Encode: HMGET
- $O(1)$  Access
- Client deserialization
- No in-place updates

## reJSON Module

---

- JSON as a native Redis Data Type (4.0 Modules API)
- Keys map to JSON values
  - Scalars
  - Objects
  - Arrays
  - Nested or Not
- Stored as a document tree structure
- Path access to JSON elements
- Atomic, in-place updates

# reJSON Basic Get/Set Operations

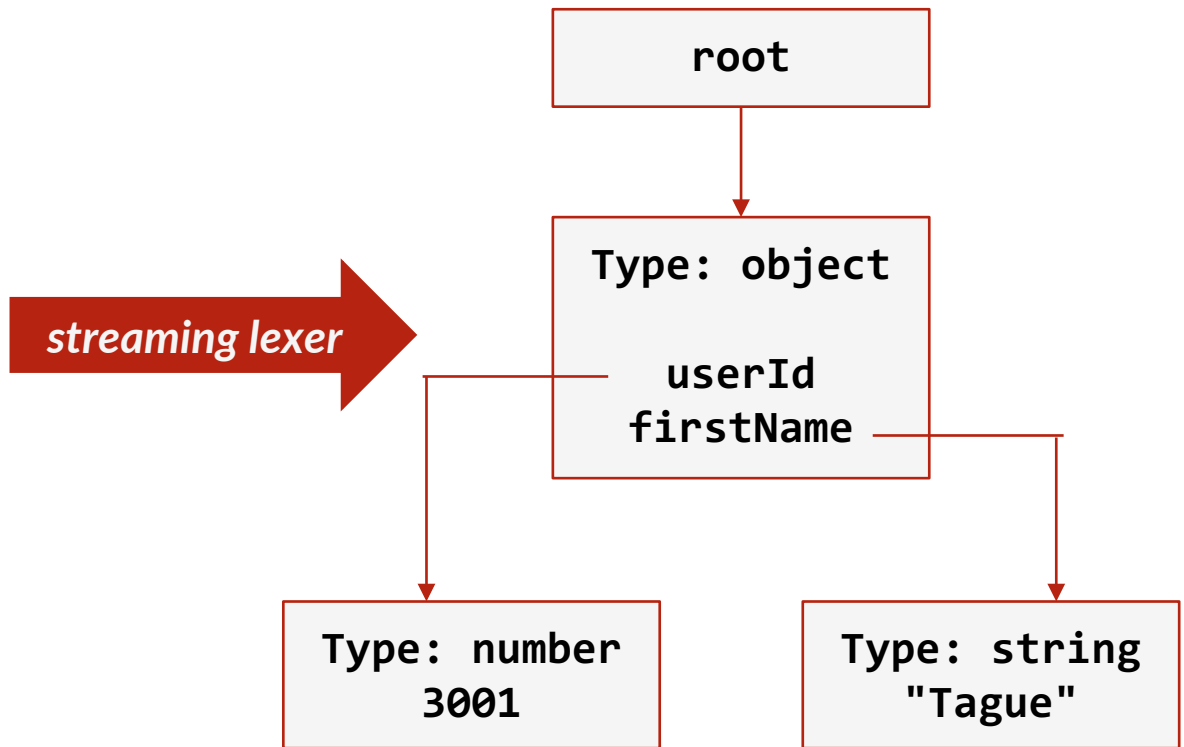
```
redis> JSON.SET json:scalar . '"Hello JSON!"'
OK
redis> JSON.SET json:object . '{"userId": 3001, "firstName": "Tague"}'
OK

redis> JSON.GET json:scalar
"Hello JSON!"
redis> JSON.GET json:object
{"userId":3001,"firstName":"Tague"}
redis> JSON.GET json:object .userId
3001

redis> JSON.SET json:object .userId 2001
OK
Redis> JSON.GET json:object
{"userId":2001,"firstName":"Tague"}
```

# ReJSON tree data structure

```
JSON.SET key .  
{  
  "userId": 3001,  
  "firstName": "Tague"  
}
```



# reJSON Commands

<b>General</b>	JSON.DEL, JSON.GET, JSON.MGET, JSON.SET, JSON.TYPE
<b>Numbers</b>	JSON.NUMINCRBY, JSON.NUMMULTBY
<b>Strings</b>	JSON.STRAPPEND, JSON.STRLEN
<b>Objects</b>	JSON.OBJKEYS, JSON.OBJLEN
<b>Arrays</b>	JSON.ARRAPPEND, JSON.ARRINDEX, JSON.ARRINSERT, JSON.ARRLEN, JSON.ARRPOP, JSON.ARRTRIM, JSON.RESP

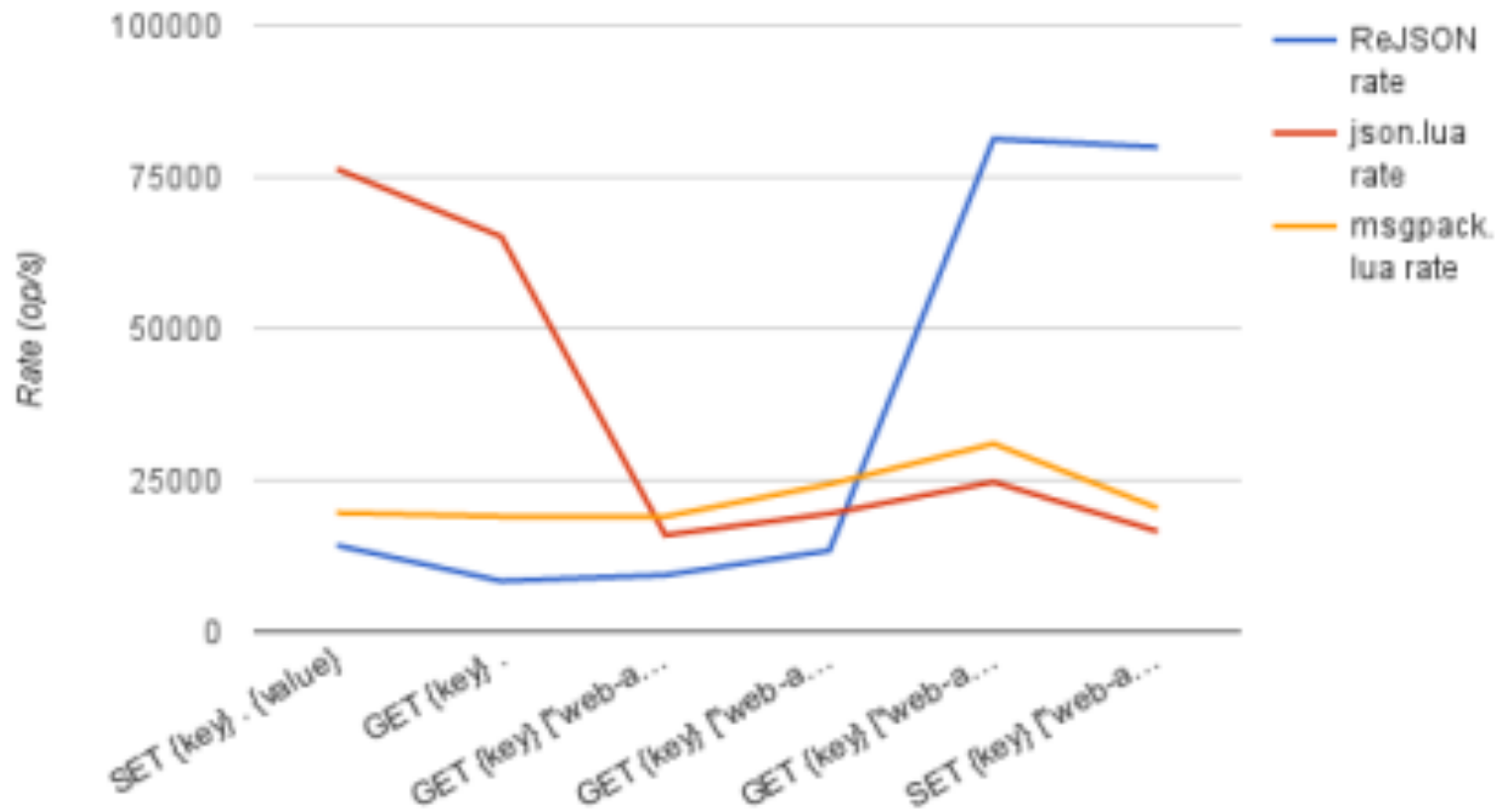
# Access reJSON API

---

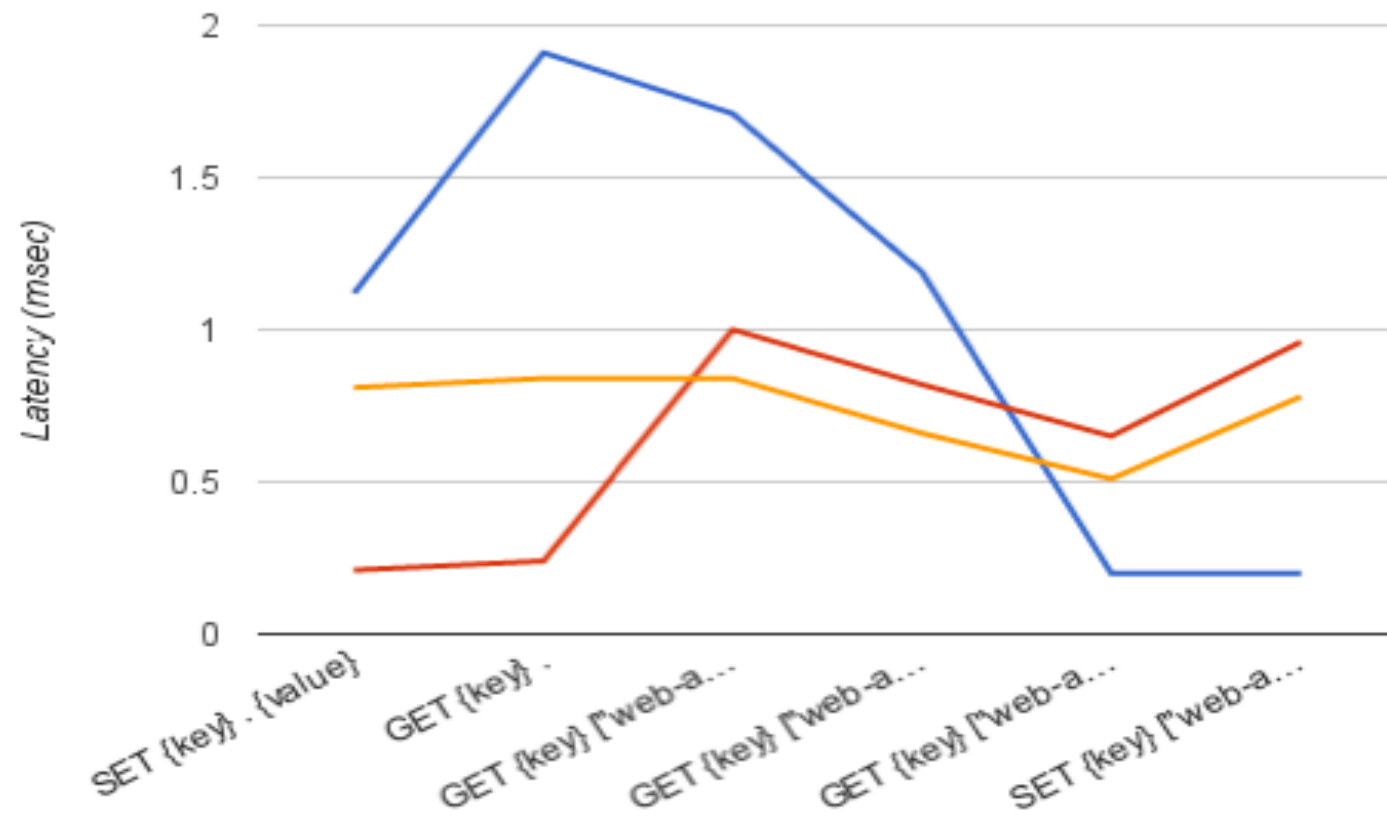
- Python
  - rejson-py
  - redis-py execute command
- Java
  - JReJSON
  - JedisConnection.execute

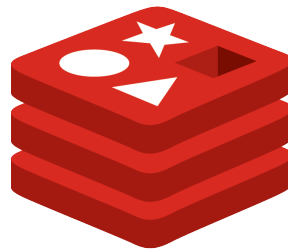


# Throughput



# Latency





**Redisearch**



# Redisearch Module

---

- Developed by Redis Labs using 4.0 Module API
- Optimized Data structures
- Fast index and search
- Text, numeric and geo filters
- Non-blocking updates and deletes
- Scale to billions of documents

# Redisearch Fundamentals

---

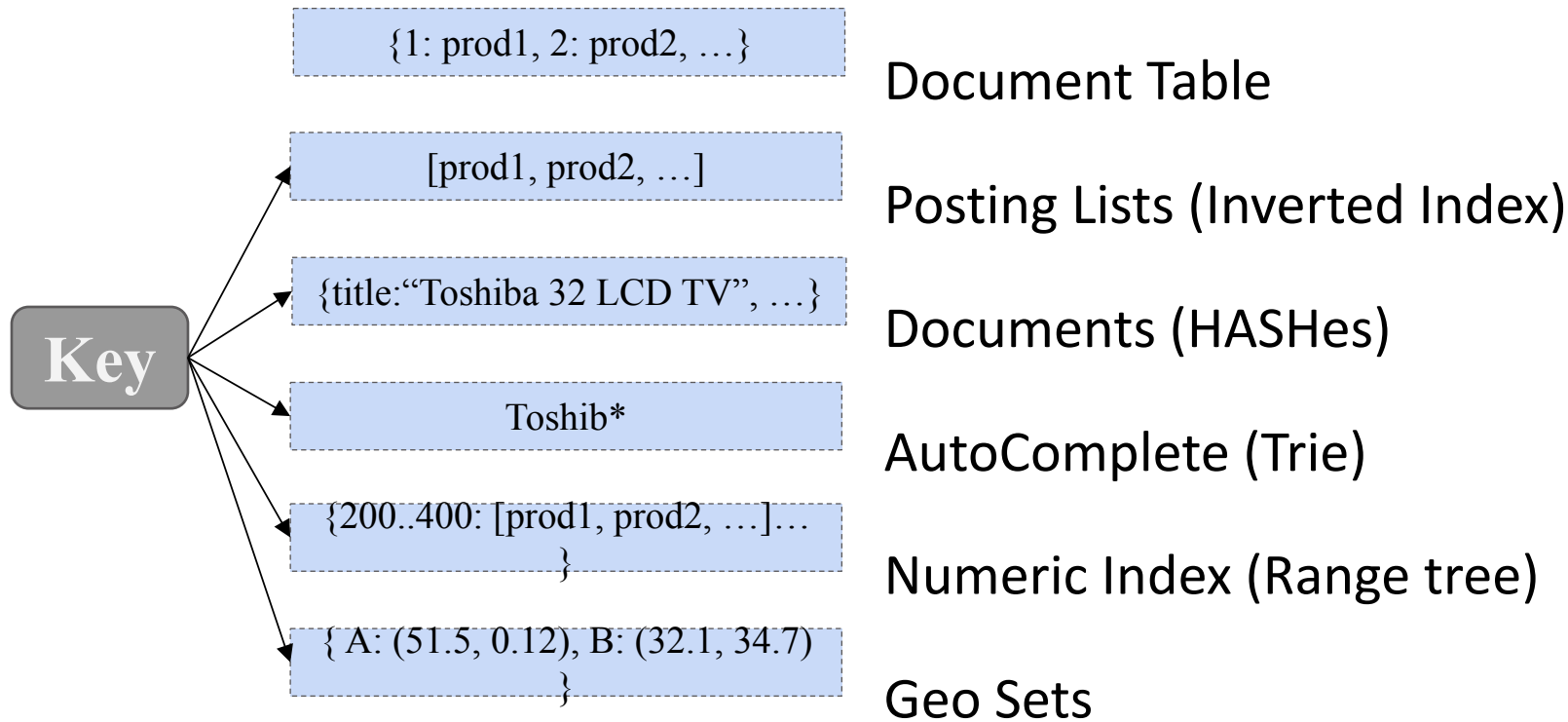
```
> FT.CREATE products SCHEMA title TEXT price NUMERIC

> FT.ADD products prod1 1.0 FIELDS title "Toshiba 32 LCD TV" price 250
> FT.ADD products prod2 1.0 FIELDS title "Samsung 42 LCD TV" price 350

> FT.SEARCH products "lcd tv"
1) (integer) 2
2) "prod1"
   ...
4) "prod2"
   ...

> FT.SEARCH products "lcd tv" FILTER price 300 400
1) (integer) 1
   "prod2"
```

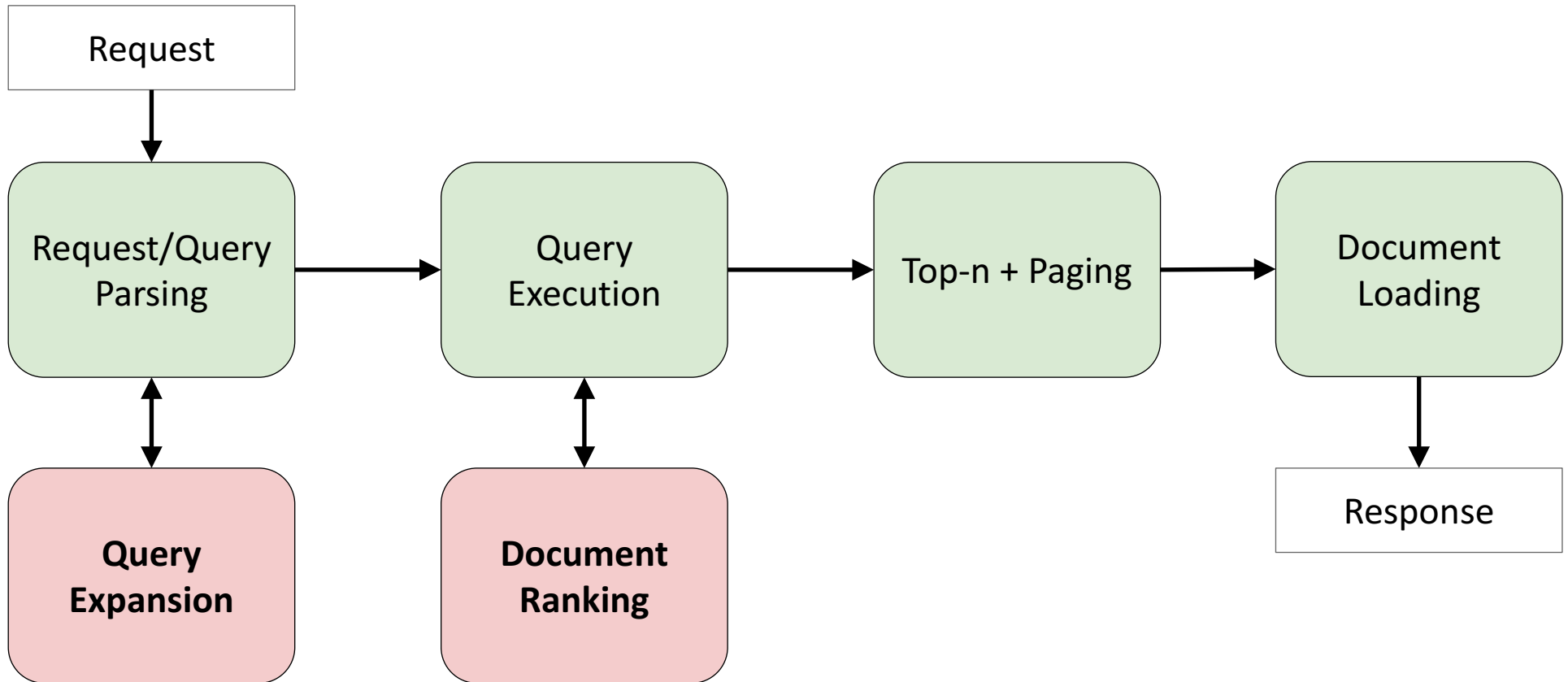
# Redisearch Data Structures



# Redisearch Query

```
FT.SEARCH {index} {query} [NOCONTENT] [VERBATIM]
[NOSTOPWORDS] [WITHSCORES] [WITHPAYLOADS]
[FILTER {numeric_field} {min} {max}] ...
[GEOFILTER {geo_field} {lon} {lat} {radius} m|km|mi|ft]
[INKEYS {num} {key} ... ]
[INFIELDS {num} {field} ... ]
[SLOP {slop}] [INORDER]
[LANGUAGE {language}]
[EXPANDER {expander}]
[SCORER {scorer}]
[PAYLOAD {payload}]
[SORTBY {field} [ASC|DESC]]
[LIMIT offset num]
```

# Redisearch Search Request Processing





# Evaluating Queries

---

- Document at-a-time approach
- Indexes are decoded as iterators, one object per round
- Top N results copied into a heap & sorted on the fly

# Scoring Results

---

- Each document in the query's result is evaluated by a **Scoring Function**
- **Top-N** scored documents are selected
- The default uses:
  - Term Frequency–Inverse Document Frequency (TF-IDF)
  - User given a priori document score
  - Proximity boosting
- And you can write your own function!

# Auto Complete

---

- Custom data type - trie based
- Fuzzy completions
- Manual dictionary building and scoring
- Unicode aware case-folding

# Scaling Redisearch

---

- Partition by document id
- Each partition contains a full index of  $1/N$  of the documents
- All terms for a sub-index are stored on the same shard
- This means we need a query coordinator:
  - Distribute the queries to all shards
  - Merge the results

# Redisearch Sharing



Redis

+



Search Engine

+



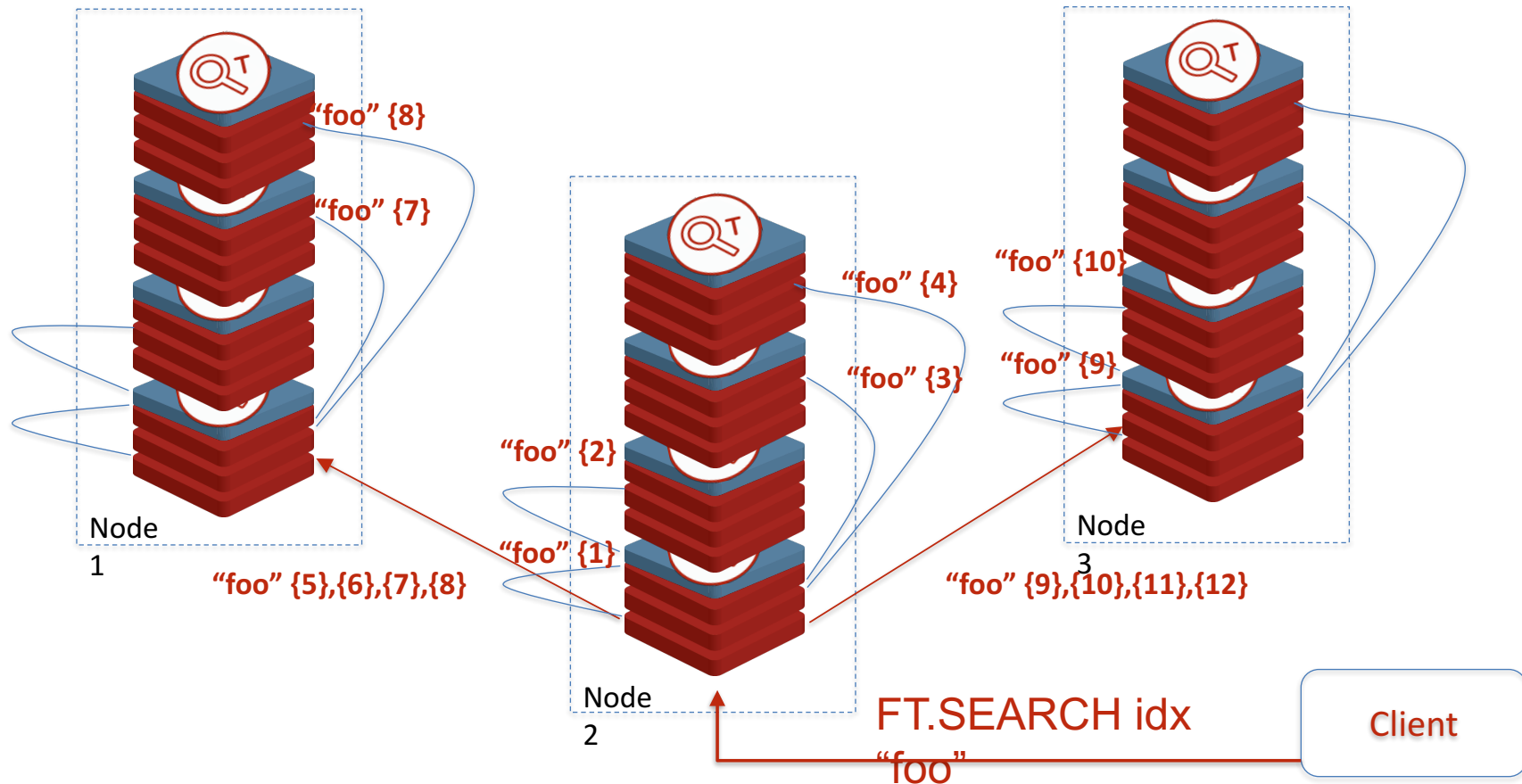
Coordinator

=

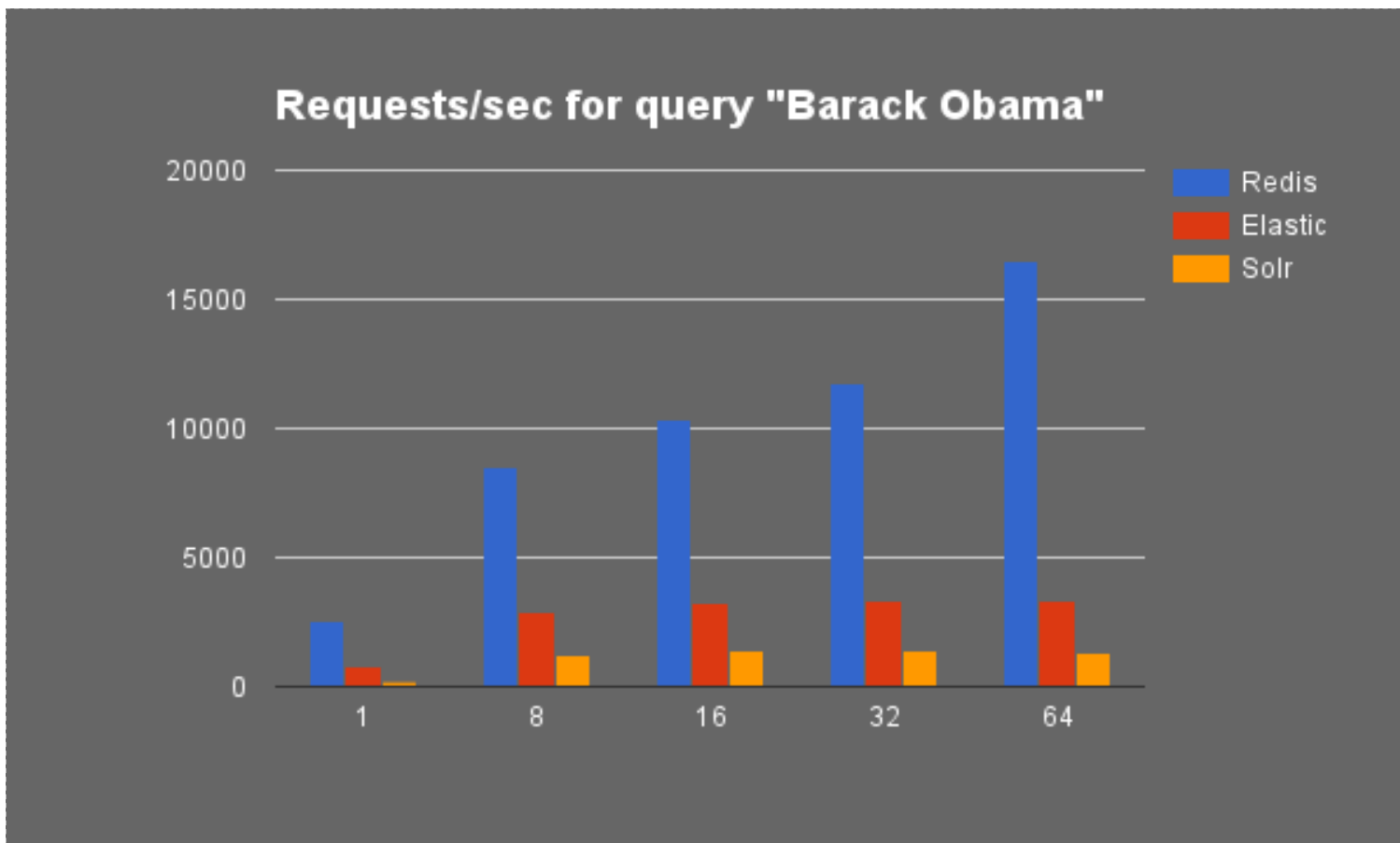


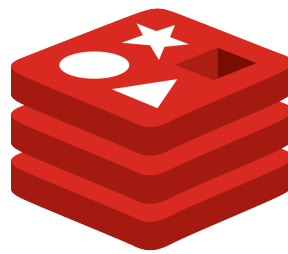
Redisearch Shard

# Redisearch Cluster Architecture



# Performance



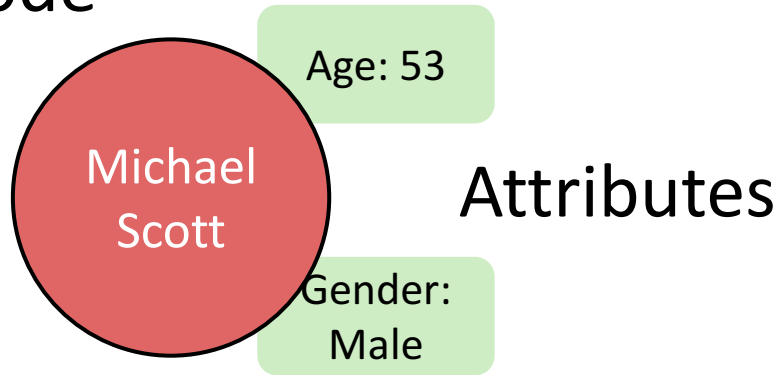


# Redis Graph

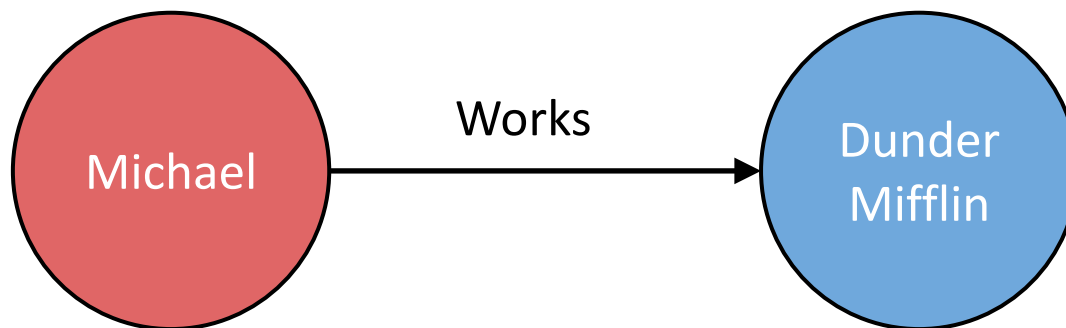


# Basic Graph Data Model

## Node



## Relation



# Adding Data

---

```
GRAPH.CREATENODE <Graph_Name> <Label> <Attributes>
```

```
GRAPH.CREATENODE the_office employee name "michael scott" age 53 gender male  
status single
```

```
GRAPH.ADDEDGE <Graph_Name> <SRC_NODE> <Relation> <DEST_NODE> <Attributes>
```

```
GRAPH.ADDEDGE the_office michael scott boss jim halpert <Attributes>
```

# HexaStore Storage Format

---

S

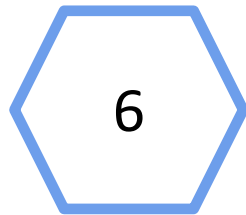
Subject

P

Predicate

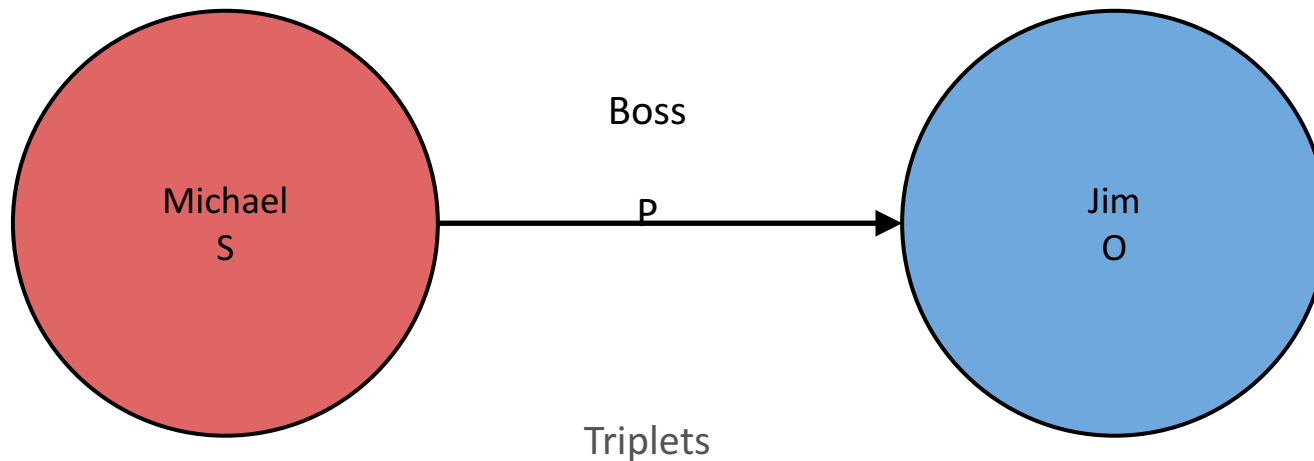
O

Object



SPO OSP  
SOP PSO  
OPS POS

# HexaStore Example



Triplets

**SPO:**Michael:Boss:Jim  
**SOP:**Michael:Jim:Boss  
**OPS:**Jim:Boss:Michael  
**OSP:**Jim:Michael:Boss  
**PSO:**Boss:Michael:Jim  
**POS:**Boss:Jim:Michael

# Converting Queries to Predicates

Who does Michael manages?

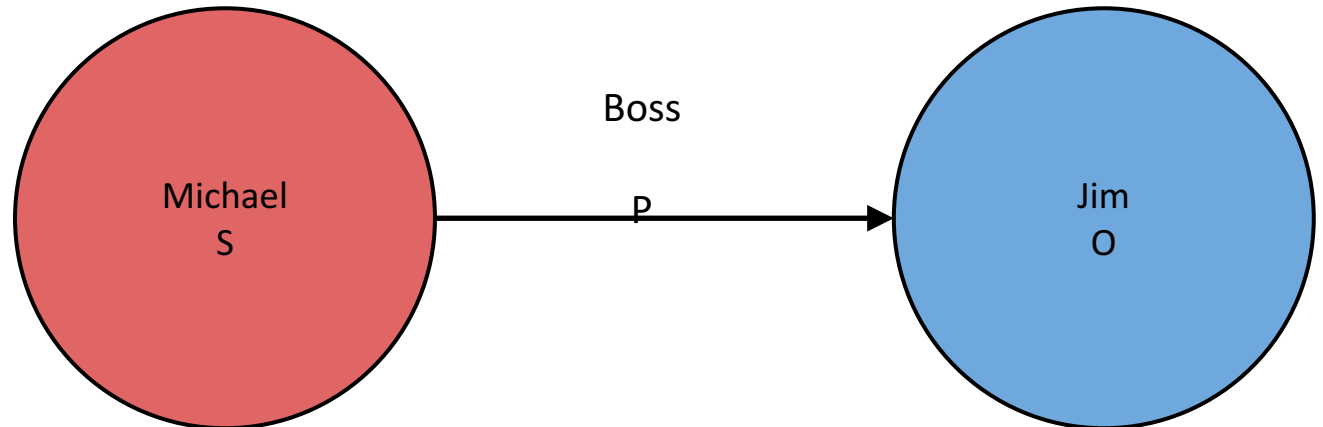
**SPO:Michael:Boss:\***

Who manages Jim?

**OPS:Jim:Boss:\***

Who manages who?

**PSO:Boss:\***



# Query Language Cypher

---

```
MATCH <Graph_name> <graph_pattern>  
WHERE <filters>  
RETURN <entities>  
ORDER BY <properties>  
LIMIT <count>
```

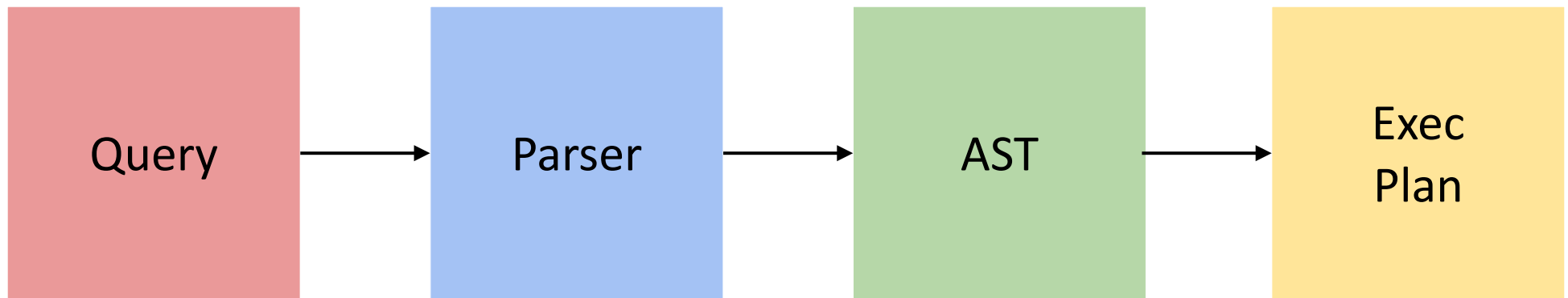
## Cypher Example

---

```
MATCH (Michael:employee {name:michael  
scott})-[boss]->(E:employee)-[visit]->  
(c:country)  
WHERE (E.age > Michael.age AND c.continent  
= Europe)  
RETURN E.name, count(c.name) AS  
countriesVisited  
ORDER BY countriesVisited, E.age DESC  
LIMIT 5
```

# Query Execution

---





# Benchmark

---

- 150K inserts per second
- 15K queries per second

# redislabs

Home of Redis

