

ARQUITETURA NOSQL

# KEY-VALUE DATABASES (REDIS)

MARCELO MANZANO E REGINA CLAUDIA CANTELE



2

## LISTA DE FIGURAS

Figura 2.1 - Db-Engines NoSQL chave-valor .....	8
Figura 2.2 - Voldemort.....	9
Figura 2.3 - Amazon Dynamo DB.....	10
Figura 2.4 - Riak.....	10
Figura 2.5 - Memcached.....	12
Figura 2.6 - Redis.....	13
Figura 2.7 - Tipos de arquiteturas ReDis.....	15
Figura 2.8 - Arquitetura do nó ReDis.....	17
Figura 2.9 - Arquitetura do nó ReDis.....	19
Figura 2.10 - Sharding.....	22
Figura 2.11 - Checkpoint .....	23
Figura 2.12 - Append only File .....	24
Figura 2.13 - Redis Server .....	25
Figura 2.14 - Resultado dos comandos.....	25
Figura 2.15 - SETs .....	33
Figura 2.16 – Union, difference e intersection .....	34
Figura 2.17 - SETs .....	38
Figura 2.18 - LISTs.....	43
Figura 2.19 - HASHs .....	48
Figura 2.20 - Georreferências .....	50
Figura 2.21 - IoT x Golang x Redis.....	52
Figura 2.22 - Uber 1ª Geração .....	55
Figura 2.23 - Uber 2ª Geração .....	56
Figura 2.24 - Uber 2ª Geração - Latência.....	56
Figura 2.25 - Uber 3ª Geração - Reconstrução .....	57
Figura 2.26 - Uber Plataforma Genérica.....	58
Figura 2.27 - Uber Michelangelo .....	59

## LISTA DE CÓDIGOS-FONTE

Código-fonte 2.1 – Redis String .....	31
Código-fonte 2.2 – Redis String .....	32
Código-fonte 2.3 – Redis String .....	32
Código-fonte 2.4 – Redis TTL .....	32
Código-fonte 2.5 – Redis Listas .....	36
Código-fonte 2.6 – Redis underline .....	37
Código-fonte 2.7 – Redis Sorted List.....	42
Código-fonte 2.8 – Redis Pilhas .....	47
Código-fonte 2.9 – Redis Pilhas .....	47
Código-fonte 2.10 – Redis Pilhas .....	47
Código-fonte 2.11 – Redis Pilhas .....	47
Código-fonte 2.12 – Redis Lista .....	48
Código-fonte 2.13 – Redis Hashed .....	49
Código-fonte 2.14 – Redis Georadius .....	52
Código-fonte 2.15 – Inserção no final da lista .....	53
Código-fonte 2.16 – Obter último valor.....	54

## SUMÁRIO

<b>2 KEY-VALUE DATABASES (REDIS)</b>	<b>5</b>
2.1 Características	5
2.2 Tecnologias	8
2.2.1 Voldemort	8
2.2.2 Amazon Dynamo DB	9
2.2.3 Riak	10
2.2.4 Memcached	12
2.2.5 Redis	13
2.2.5.1 Visão geral da Arquitetura	17
2.2.5.2 Data Path & Clustering	20
2.2.5.3 Sharding	21
2.2.5.4 Persistência de dados	22
2.2.5.5 Instalação Redis	24
2.2.5.6 Estruturas de Dados no Redis	26
2.2.5.7 Trabalhando com Strings	27
2.2.5.8 Trabalhando com Conjuntos (SETs)	32
2.2.5.9 Trabalhando com Conjuntos Ordenados (SETs)	37
2.2.5.10 Trabalhando com LISTS	42
2.2.5.11 Trabalhando com HASHs	48
2.2.5.12 Trabalhando com GEORreferências	50
2.3 Cases	52
2.3.1 Sistema IoT para aquisição de dados com Redis e linguagem Golang	52
2.3.2 Evolução arquitetura uber	54
<b>REFERÊNCIAS</b>	<b>60</b>

## 2 KEY-VALUE DATABASES (REDIS)

### 2.1 Características

Um armazenamento chave-valor é um conceito ou estrutura de dados utilizada há muito tempo em Ciências da Computação, frequentemente considerado o mais simples dos bancos de dados NoSQL.

Sua estrutura constitui-se de uma lista de pares de elementos compostos por uma chave e um valor. Esse modelo pode ser comparado à estrutura de dados chamada Tabela Hash, na qual são associados valores às chaves de busca para permitir um acesso rápido ao seu conteúdo. Alguns sistemas dessa categoria permitem – além dos tipos simples de dados, como numerais e cadeias de caracteres – a utilização de listas e conjuntos de valores de tipos simples.

Os dados armazenados neste tipo de modelo são constituídos por duas partes: uma string, que representa a chave; e os dados a serem armazenados, que representam o valor, criando assim um par “chave-valor” ou “the big hash table”. Uma cadeia de símbolos (chave) leva a um blob de dados arbitrariamente grande (valor).

Os valores podem ser de qualquer tipo de dados. As chaves, por sua vez, podem possuir nomenclaturas bastante flexíveis, de acordo com as necessidades do sistema, ou seja, dois itens de dados vinculados.

A simplicidade deste modelo torna seu armazenamento rápido, fácil de usar, escalável e flexível. No entanto, os sistemas originais chave-valor não foram projetados para permitir que os pesquisadores filtrassem ou controlassem os dados que retornam de uma solicitação – eles não incluíam um mecanismo de pesquisa.

Eles costumam dar suporte a bases com grandes volumes de dados. Em contrapartida, esse modelo de banco de dados não costuma permitir que consultas sejam realizadas sobre os seus dados, mas apenas sobre as chaves de busca. Assim, todo o acesso é feito por meio das chaves de busca e, apenas com a chave, é possível ter acesso ao valor.

Os bancos de dados de chave-valor não estabelecem um esquema ou metadado específico (schema less). Os sistemas de chave-valor tratam os dados como uma coleção única com a chave que representa uma sequência arbitrária, por exemplo, um nome de arquivo, hash ou uma URL de uma página web. Os armazenamentos de chave-valor geralmente usam muito menos memória enquanto salvam e armazenam a mesma quantidade de dados, aumentando o desempenho de certos tipos de cargas de trabalho.

O modelo chave-valor também não agrupa os dados por instâncias, diferentemente do realizado no Modelo Relacional por meio de tabelas. No modelo chave-valor, todos os dados estão armazenados em uma estrutura composta apenas por duas colunas. Essa característica impossibilita a definição de esquemas de dados, sendo possível a introdução de algum metadado apenas por meio da nomenclatura explícita das chaves. Também não existe a possibilidade de realização de consultas mais complexas, como subconsultas. É possível realizar apenas uma consulta por vez e, baseando-se em seu resultado, realizar uma segunda.

Essa perspectiva torna esse modelo de dados mais simples, o que diminui os tempos de resposta, permitindo que a capacidade de armazenamento de suas bases de dados seja uma das maiores dos sistemas enquadrados na categoria NoSQL. Além disso, o modelo não dá suporte a relacionamentos; não existe referência de chaves e, conseqüentemente, não há integridade referencial. Somando-se a isso, não existe uma linguagem de consulta, o que ocasiona algumas limitações na capacidade de busca

Enquanto bancos de dados relacionais lidam com transações de pagamento muito bem, eles possuem dificuldades para lidar com grandes volumes de transações simultâneas. No entanto, os NoSQL chave-valor podem escalar conforme necessário e manipular volumes extremamente altos de tráfego por segundo, fornecendo serviço para milhares de usuários simultâneos. Assim, processam grandes quantidades de dados e um fluxo consistente de operações de leitura / gravação para:

- Gerenciamento de sessões em aplicativos da web: oferecendo aos usuários a opção de salvar e restaurar sessões.

- Preferências do usuário e repositórios de perfis dados pessoais de usuários específicos.
- Recomendações de produtos itens personalizados nos quais um cliente pode estar interessado.
- Cupons, anúncios personalizados adaptados e visualizados pelos clientes em tempo real.
- Jogos on-line com vários jogadores, gerenciando a sessão de cada jogador.
- Gerenciar carrinhos de compras para compradores on-line – até a hora do pagamento.

As empresas que vendem produtos pela Internet geralmente enfrentam os diferentes volumes de acordo com a sazonalidade de festas e promoções, como a Black Friday ou o Natal, em comparação com o resto do ano. A questão é pagar por uma infraestrutura dimensionada para o pico de compras do Natal (e pagar por essa infraestrutura pelo resto do ano) ou correr o risco de não ser capaz de lidar com a corrida do Natal (e travar por várias horas). Supondo que um banco de dados relacional lide com serviços normais durante todo o ano, ter um serviço na nuvem com um banco de dados de chave-valor para a corrida de Natal fornece uma solução eficiente e relativamente barata.

De um modo geral, o segredo dos bancos de dados de chave-valor reside na simplicidade e na velocidade resultante de acesso. A recuperação de dados requer uma solicitação direta (chave) para o objeto na memória (valor) e não há linguagem de consulta. Os dados podem ser armazenados em sistemas distribuídos sem se preocupar com a localização dos índices, o volume de dados ou a desaceleração da rede. Alguns bancos de dados de chave-valor estão usando armazenamento flash e índices secundários, em um esforço para ultrapassar os limites da tecnologia de chave-valor.

Um banco de dados de chave-valor é fácil de construir e dimensionar. Geralmente, oferece excelente desempenho e pode ser otimizado para atender às necessidades de uma organização. Quando um banco de dados de chave-valor é substituído por novos aplicativos, há uma chance maior de operar mais lentamente.

## 2.2 Tecnologias

Muitas tecnologias estão associadas ao NoSQL chave-valor e o site Db-Engines classifica-os por sua popularidade atual. A popularidade usa parâmetros, como número de menções do sistema em sites, interesse geral no sistema no Google Trends, frequência de discussões técnicas sobre o sistema, número de ofertas de emprego nas quais o sistema é mencionado em sites, como Indeed e Simply Hired, número de perfis em redes profissionais mais populares, como LinkedIn e Upwork, e relevância nas redes sociais, como o número de tweets do Twitter.

65 systems in ranking, July 2020

Rank			DBMS	Database Model	Score		
Jul 2020	Jun 2020	Jul 2019			Jul 2020	Jun 2020	Jul 2019
1.	1.	1.	Redis	Key-value, Multi-model	150.05	+4.40	+5.78
2.	2.	2.	Amazon DynamoDB	Multi-model	64.58	-0.29	+8.17
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	30.40	-0.40	+1.32
4.	4.	4.	Memcached	Key-value	25.84	+1.04	-1.22
5.	6.		etcd	Key-value	8.57	+0.52	
6.	5.	5.	Hazelcast	Key-value, Multi-model	8.45	+0.04	+0.18
7.	7.	6.	Aerospike	Key-value, Multi-model	6.82	+0.15	+0.23
8.	8.	7.	Ehcache	Key-value	6.51	+0.23	-0.05
9.	9.	10.	ArangoDB	Multi-model	5.85	+0.47	+1.19
10.	10.	8.	Riak KV	Key-value	5.41	+0.42	-0.65
11.	11.	11.	Ignite	Multi-model	4.95	+0.08	+0.67
12.	12.	9.	OrientDB	Multi-model	4.88	+0.06	-0.81
13.	13.	12.	Oracle NoSQL	Key-value, Multi-model	4.42	+0.20	+0.96
14.	14.	13.	InterSystems Caché	Multi-model	3.44	-0.02	+0.14
15.	15.	15.	Oracle Berkeley DB	Multi-model	3.40	+0.20	+0.36

Figura 2.1 - Db-Engines NoSQL chave-valor  
Fonte: DB-Engines (2020)

### 2.2.1 Voldemort

Este projeto foi inicialmente desenvolvido no LinkedIn em linguagem Java em 2008, e a sua criação tinha por objetivo fornecer suporte do tipo chave-valor para armazenamento de dados do tipo “Quem viu o meu perfil”. Tornou-se código aberto em 2009 e desde então tem sido adotado por várias organizações e projetos, que requerem um armazenamento com um alto nível de disponibilidade e ao mesmo tempo pouca latência. Este sistema provisiona um mecanismo de controle de concorrências multiversão (MVCC), as atualizações realizadas nas réplicas da base



de dados presentes em outros nós são realizadas de forma assíncrona. Assim, este sistema não consegue garantir a consistência dos dados em todos os momentos (CATTELL, 2011), regendo-se essencialmente pelo ponto da consistência eventual do teorema BASE.

Devido à política de particionamento baseada em Hash, este sistema não apresenta suporte nativo para consultas baseadas em intervalos de dados. O sistema Voldemort acaba por ser bastante flexível, de modo que suporta a ligação de outros motores de armazenamento ao sistema, por exemplo, o sistema BerkeleyDB ou MySQL.

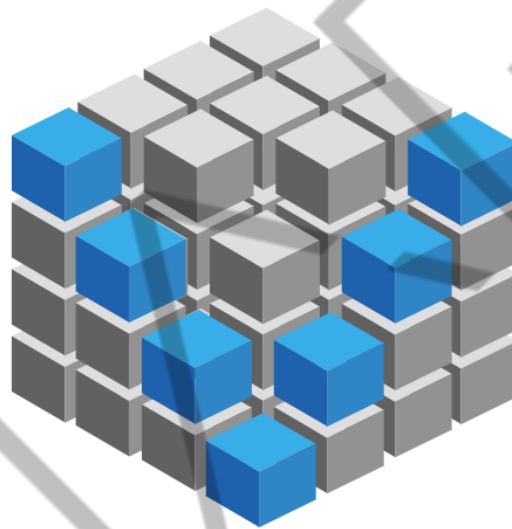


Figura 2.2 - Voldemort  
Fonte: Adaptado por FIAP (2020)

### 2.2.2 Amazon Dynamo DB

Em 2007, a Amazon apresentou o seu sistema de armazenamento do tipo chave-valor que tinha como objetivo suportar a seção do “carrinho de compras” da sua loja on-line. Este sistema precisava ser altamente disponível e confiável, ou seja, teria de funcionar 24 horas por dia, 7 dias por semana, sem falhas e sem afetar o processo de compra por parte de clientes (MCCREARY; KELLY, 2014). No entanto, este sistema nunca foi disponibilizado ao público, sendo apenas utilizado internamente na Amazon. Não obstante esta situação, o artigo publicado teve um grande impacto no movimento NoSQL.



Figura 2.3 - Amazon Dynamo DB  
Fonte: Adaptado por FIAP (2020)

### 2.2.3 Riak

Riak é uma base de dados distribuída, tolerante a falha, de armazenamento do tipo par chave-valor. É um sistema de fonte aberta desde 2009. Este sistema é distribuído pela empresa do Massachusetts, Basho technologies, encontra-se em produção desde 2008, é altamente baseado no sistema Dynamo apresentado pela Amazon e é implementado recorrendo à linguagem Erlang (FINK, 2012).



Figura 2.4 - Riak  
Fonte: Adaptado por FIAP (2020)

Objetos Riak podem ser procurados e armazenados em JSON, o que permite que cada objeto possua vários campos. Neste sistema, objetos podem ser agrupados em “baldes”, tal como nas coleções suportadas pelas bases de dados que recorrem ao armazenamento no formato de documentos (a analisar no subponto seguinte). O sistema Riak não suporta índices ou outros campos de identificação

para além da chave primária. Assim, acaba por não fornecer a mesma profundidade analítica que outros sistemas providenciam (CATTELL, 2011).

Um grande ponto a favor deste sistema é a sua grande capacidade de tolerância a falhas. Servidores podem ser ligados, desligados ou falharem a qualquer momento, sem que afete de qualquer modo o sistema (REDMOND; WILSON, 2012). Um nó que falha não representa um problema de resolução imediata e pode ser tratado apenas quando existir disponibilidade para tal. O sistema Riak possui várias características, das quais destacamos:

- **Processamento Paralelo:** recorrendo à utilização de MapReduce, o sistema apresenta capacidade para compor e decompor consultas transversalmente em qualquer conjunto de servidores, para assim conseguir alcançar a computação e análise em tempo real dos dados armazenados.
- **Ligações:** um sistema Riak pode ser construído para “imitar” parte do comportamento de uma base de dados NoSQL de armazenamento do tipo grafo, para isso recorrendo à utilização de “links”. Um “link” pode ser considerado como uma conexão unilateral entre dois pares chave-valor. Seguir essas conexões irá permitir criar um mapa das relações existentes entre os vários pares chave-valor de uma base de dados.
- **Pesquisa:** O sistema Riak possui um método de pesquisa distribuído e tolerante a falhas que possui a capacidade de realizar análises completas sobre texto. O sistema ainda pode recorrer aos seus “baldes” para providenciar um método de transformação mais rápida de chaves em valores. Rápido, flexível e escalável, é bom para desenvolver aplicativos e trabalhar com outros bancos de dados e aplicativos.
- Um banco de dados, intermediário de mensagens e cache de memória. Ele suporta hashes, sequências de caracteres, listas, bitmaps e HyperLogLog.

### 2.2.4 Memcached

O Memcached caracteriza-se como um sistema genérico de cache em memória. Ele foi construído pensando na melhoria de desempenho de aplicações web através da redução na demanda de requisições ao banco de dados em disco. Brad Fitzpatrick o desenvolveu para melhorar o desempenho do site Livejournal.com através de uma solução melhor de cache. A sua implementação é na linguagem Perl e posteriormente foi reescrito em C.



Figura 2.5 - Memcached  
Fonte: Adaptado por FIAP (2020)

Os aplicativos da Web exigem, na maioria das vezes, acesso a seus dados para tomar decisões rápidas e mostrar instantaneamente o que o usuário busca/deseja. Armazenamentos de chave-valor são ideais para armazenar os dados mais usados, evitando o acesso intensivo ao armazenamento de dados em disco mais lento. O Memcached utiliza uma arquitetura multithread e o controle de concorrência interno é feito através de uma hash-table estática de locks.

MediaWiki é usado pela Wikipedia e muitos outros sites na web e usa Memcached para armazenar em cache valores para reduzir a necessidade de realizar cálculos caros e reduzir a carga nos servidores de banco de dados. Neste exemplo, uma chave exclusiva é usada para armazenar pequenos pedaços de dados arbitrários (cadeias de caracteres e outros objetos) dos resultados de chamadas ao banco de dados, chamadas de API ou renderização de página. O desempenho é aumentado armazenando em cache os resultados de uma consulta

ao banco de dados no Memcached por um período arbitrário, como 5 minutos, e depois consultando o Memcached primeiro pelos resultados, em vez do banco de dados.

### 2.2.5 Redis

Este último banco chave-valor é o que usaremos em nossa disciplina, e você verá por quais razões escolhemos o Redis, portanto detalharemos bem mais sua arquitetura e funcionamento.



Figura 2.6 - Redis  
Fonte: Adaptado por FIAP (2020)

Redis significa *Remote Dictionary Service* e foi criado pelo programador italiano Salvatore Sanfillip, contratado pela empresa VMware para trabalhar no projeto em tempo integral. Este projeto, lançado em 2009, foi implementado recorrendo à linguagem C, é um projeto de código aberto (CATTELL, 2011), trabalha em memória, oferecendo, assim, uma grande performance. Ao mesmo tempo, também fornece uma boa capacidade de replicação e um modelo de dados único para a construção de plataformas/aplicações orientadas para a resolução de problemas. Este sistema tem a seu favor a facilidade de utilização e um conjunto sofisticado de comandos para interação com o sistema. Suas principais características são:

- **Velocidade:** o Redis armazena todo o conjunto de dados na memória principal e é por isso que é extremamente rápido. Ele carrega até 110.000 SETs/segundo e 81.000 GETs/segundo podem ser recuperados em uma caixa Linux de nível de entrada. O Redis suporta o Pipelining de comandos e facilita o uso de vários valores em um único comando para acelerar a comunicação com as bibliotecas do cliente.
- **Persistência:** enquanto todos os dados residem na memória, as alterações são salvas de forma assíncrona no disco usando políticas flexíveis com base no tempo decorrido e/ou no número de atualizações desde o último salvamento. O Redis suporta um modo de persistência de arquivo somente de anexação. Verifique mais sobre Persistência ou leia o `AppendOnlyFileHowto` para obter mais informações.
- **Estruturas de dados:** o Redis suporta vários tipos de estruturas de dados, como strings, hashes, conjuntos, listas, conjuntos classificados com consultas de intervalo, bitmaps, hiperloglogs e índices geoespaciais com consultas radius.
- **Operações atômicas:** operações Redis trabalhando nos diferentes Tipos de Dados são atômicas, então é seguro definir ou aumentar uma chave, adicionar e remover elementos de um conjunto, aumentar um contador etc.
- **Idiomas suportados:** o Redis suporta vários idiomas, como ActionScript, C, C++, C #, Clojure, Common Lisp, D, Dart, Erlang, Go, Haskell, Iax, Java, JavaScript (Node.js), Julia, Lua, Objective-C, Perl, PHP, Dados Puros, Python, R, Raquete, Ruby, Rust, Scala, Smalltalk e Tcl.
- **Replicação master/slave:** Redis segue uma replicação Master/Slave muito simples e rápida. São necessários apenas uma linha no arquivo de configuração para configurá-la e 21 segundos para que um escravo conclua a sincronização inicial do conjunto de chaves de 10 MM em uma instância do Amazon EC2.
- **Sharding:** Redis suporta sharding. É muito fácil distribuir o conjunto de dados em várias instâncias do Redis, como outro armazenamento de valor-chave.

- **Portátil:** o Redis é escrito em ANSI C e funciona na maioria dos sistemas POSIX, como Linux, BSD, Mac OS X, Solaris etc. Redis é relatado para compilar e trabalhar sob o WIN32 se compilado com o Cygwin, mas não há suporte oficial para o Windows atualmente.

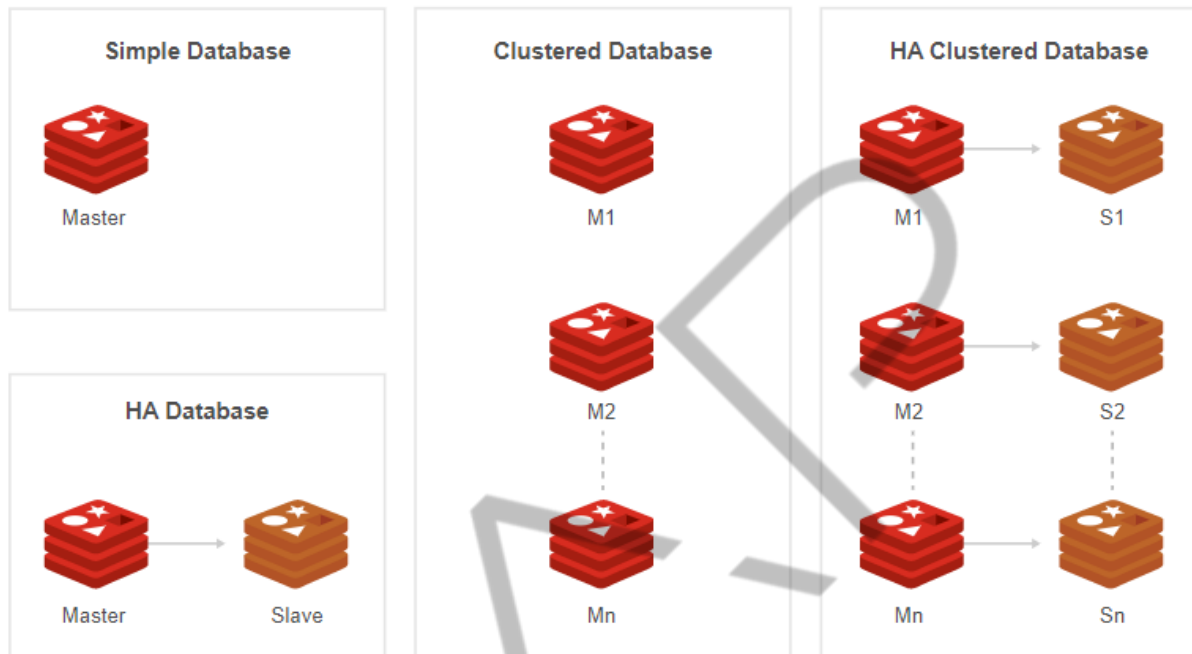


Figura 2.7 - Tipos de arquiteturas Redis  
Fonte: REDIS LABs (2020)

Para além destes fatores, o sistema Redis é especialmente renomeado pela sua rapidez. De acordo com indicadores obtidos em testes de performance, as leituras neste sistema são bastante rápidas e as escritas ainda mais; o sistema é capaz de tratar e gerir mais de cem mil operações por segundo.

Esta velocidade deve-se, em grande parte, ao fato de este sistema manter o conjunto de dados a utilizar em memória. Este sistema é também capaz de oferecer persistência de dados, porque possui um mecanismo que, de uma forma assíncrona, escreve as alterações realizadas, em disco (HAINES, 2009). É possível configurar de quanto em quanto tempo ou depois de quantas transações é que o sistema deve guardar os dados. Deste modo, se um servidor falhar, será grande a possibilidade de não serem perdidas várias transações ou modificações realizadas sobre os dados.

O ponto fraco do sistema Redis é o fato de que o tamanho do conjunto de dados a utilizar tem de caber confortavelmente na memória RAM disponível no sistema (HAINES, 2009), estando limitado nesse aspecto em relação a outras

soluções NoSQL e, mais especificamente, a outros sistemas de armazenamento do tipo par chave-valor. Você pode facilmente construir sistemas complexos sobre o Redis para:

- Esquemas de indexação definidos pelo usuário.
- Filas de mensagens com notificação de novo elemento em tempo real.
- Armazéns de grafos direcionados e não direcionados para sistemas seguintes ou amigos.
- Sistemas de notificação de publicação/assinatura em tempo real.
- Back-ends de análise em tempo real.
- Servidores de filtro Bloom.
- Filas de tarefas e sistemas de tarefas.
- Placares de recordes.
- Sistemas de classificação de usuários.
- Sistemas de armazenamento hierárquico estruturado em árvore.
- Notícias personalizadas individuais ou feeds de dados para seus usuários.

Kyle Davis, gerente técnico de marketing da Redis Labs, escreveu recentemente:

A intenção original do Redis (ou de qualquer repositório de chave-valor) era ter uma chave ou identificador específico para cada pedaço de dados individual. O Redis expandiu rapidamente esse conceito com tipos de dados, nos quais uma única chave pode se referir a vários (até milhões de) dados. À medida que os módulos chegavam ao ecossistema, a ideia de uma chave foi ampliada ainda mais, porque agora um único dado podia abranger várias chaves (para um índice RediSearch, por exemplo). Portanto, quando perguntado se o Redis é um armazenamento de chave-valor, geralmente respondo com 'ele desce da linha de chave-valor dos bancos de dados', mas note que, neste momento, é difícil justificar o Redis apenas como um armazenamento de chave-valor. (DAVIS, [s.d.])



### 2.2.5.1 Visão geral da Arquitetura

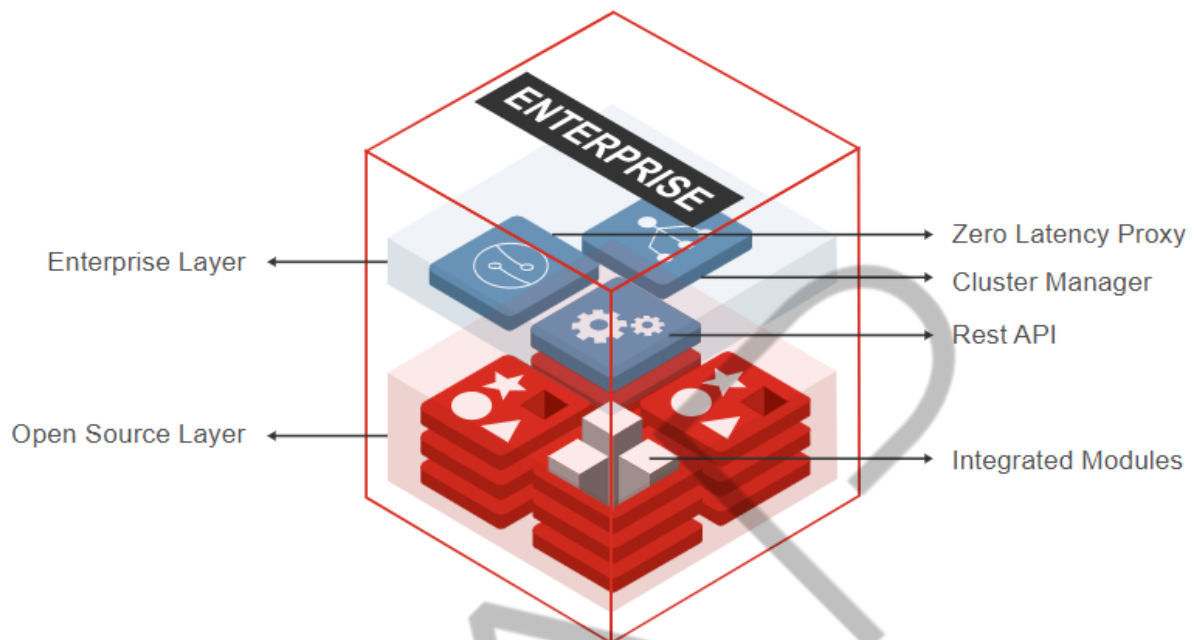


Figura 2.8 - Arquitetura do nó ReDis  
Fonte: REDIS LABs (2020)

O cluster Redis Enterprise é construído em uma arquitetura simétrica, com todos os nós contendo os seguintes componentes:

- **Redis Shard:** uma instância do Redis de código aberto com função mestre ou escravo que faz parte do banco de dados.
- **ZERO-Latency proxy:** o proxy é executado em cada nó do cluster, é escrito em C e é baseado em uma arquitetura sem estado de corte, multithread e sem bloqueio. O proxy lida com as seguintes funcionalidades principais:
  - Oculta a complexidade do cluster do aplicativo/usuário.
  - Mantém o endpoint do banco de dados.
  - Pedidos de encaminhamento.
  - Suporta o protocolo memcached.
  - gerencia a criptografia de dados por meio de SSL.
  - Fornece autenticação SSL forte baseada em cliente.

- Permite a aceleração do Redis por meio de pipelining e gerenciamento de conexão.
- **Cluster Management:** este componente contém um conjunto de processos distribuídos que juntos gerenciam todo o ciclo de vida do cluster. A entidade do gerenciador de cluster é completamente separada dos componentes do caminho de dados (proxies e fragmentos Redis) e tem as seguintes responsabilidades:
  - Provisionamento e desprovisionamento de banco de dados.
  - Gestão de recursos.
  - Processos de vigilância.
  - Escalonamento automático.
  - Refragmentação.
  - Reequilibrando.
- **API REST segura:** todas as operações de gerenciamento e controle no cluster Redis Enterprise são realizadas por meio de uma API segura e dedicada, que é resistente a ataques e fornece melhor controle das operações de administração do cluster. Uma das principais vantagens que essa interface oferece é a capacidade de provisionar e desprovisionar recursos do Redis a uma taxa muito alta e quase sem dependência da infraestrutura subjacente. Isso o torna muito adequado para a nova geração de ambientes baseados em microsserviços.

No caso de uma instalação em cluster, o Redis apresenta uma arquitetura “shared nothing”, linearmente escalável, “multitenant” (multilocatário) e simétrica – como podemos ver abaixo:

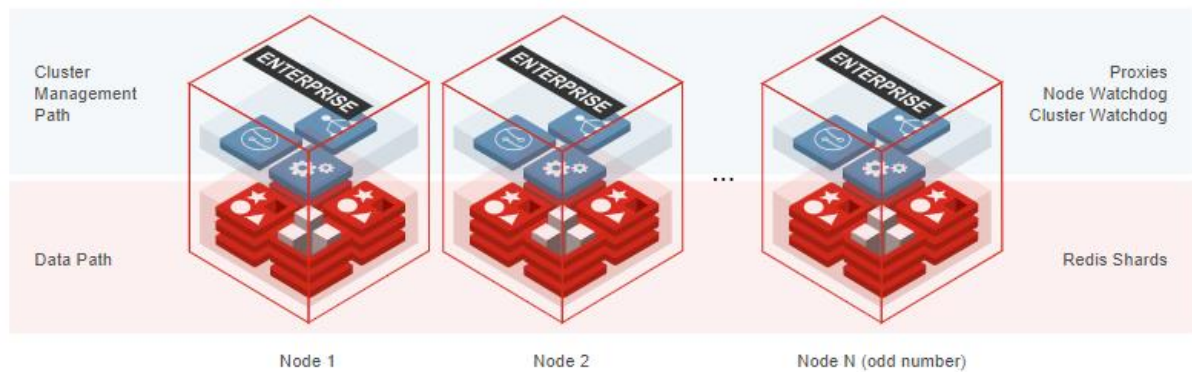


Figura 2.9 - Arquitetura do nó ReDis  
Fonte: REDIS LABs (2020)

O cluster Redis Enterprise é construído em uma separação completa entre os componentes de data path (ou seja, proxies e fragmentos) e os componentes do caminho de controle/gerenciamento (ou seja, processos de gerenciamento de cluster), o que fornece uma série de benefícios importantes:

- **Desempenho:** entidades de caminho de dados não precisam lidar com funções de controle e gerenciamento. A arquitetura garante que todos os ciclos de processamento sejam dedicados a atender às solicitações dos usuários, o que melhora o desempenho geral. Por exemplo, cada fragmento do Redis em um cluster do Redis Enterprise funciona como se fosse uma instância autônoma do Redis. O shard não precisa monitorar outras instâncias do Redis, não precisa lidar com eventos de falha ou partição e não tem conhecimento de quais slots de hash estão sendo gerenciados.
- **Disponibilidade:** o aplicativo continua acessando dados de seu banco de dados Redis, mesmo quando ocorre fragmentação, refragmentação e reequilíbrio. Nenhuma mudança manual é necessária para garantir o acesso aos dados.
- **Segurança:** o Redis Enterprise evita que comandos de configuração sejam executados por meio das APIs regulares do Redis. Qualquer operação de configuração é permitida por meio de uma interface segura de UI, CLI ou API que segue controles de autorização baseados em funções. A arquitetura baseada em proxy garante que apenas conexões

certificadas possam ser criadas com cada shard e apenas solicitações certificadas possam ser recebidas por shards do Redis.

- **Capacidade de gerenciamento:** provisionamento de banco de dados, alterações de configuração, atualizações de software e muito mais são feitos com um único comando (via UI ou API) de maneira distribuída e sem interromper o tráfego do usuário.

#### 2.2.5.2 Data Path & Clustering

O “data path” é baseado em vários proxies multitarefa de latência zero que residem em cada um dos nós do cluster para mascarar a complexidade subjacente do sistema. Esse modelo suporta vários proxies por banco de dados Redis e permite o uso de qualquer cliente Redis com reconhecimento de cluster ou regular. Isso permite que a base de código do usuário funcione como está. Cada proxy encaminha solicitações de clientes para os servidores Redis relevantes (shards). Quando novos fragmentos são adicionados a um banco de dados, o proxy lida com as novas regras de roteamento para as solicitações do aplicativo de forma transparente, dimensionando imediatamente o desempenho do banco de dados e a capacidade de memória sem quaisquer alterações no próprio aplicativo.

Os proxies também suportam os protocolos de texto e binários do Memcached e executam a tradução em andamento entre a semântica do protocolo Memcached e Redis. Isso permite que os usuários do Memcached desfrutem de muitos dos recursos que não estão disponíveis com o Memcached de software livre, como replicação integrada, failover automático, backups de persistência de dados e dimensionamento (out/in) sem perder dados.

Seu gerenciador de cluster é uma função de controle sofisticada que fornece recursos como resharding, rebalancing, failover automático, reconhecimento de rack, provisionamento de banco de dados, gerenciamento de recursos, configuração de persistência de dados, backup e recuperação. O gerenciador de cluster emprega vários mecanismos de watchdog no nível de nó do cluster e no nível do processo, o que garante uma resposta instantânea a eventos como falhas de nó, rack e data center e pode manipular vários eventos de failover ao mesmo tempo. Como o

gerenciador do cluster é totalmente desacoplado dos componentes do caminho de dados, as alterações em seus componentes de software não afetam os componentes do caminho de dados.

Uma arquitetura “shared nothing” maximiza o desempenho de cada banco de dados. A migração automática de shards (discutido na sequência) entre nós é executada quando necessário.

O proxy de latência zero utiliza vários mecanismos para melhorar o desempenho, incluindo pipelining just-in-time, conexões de soquete, pool de conexão e multiplexação. Melhorias na persistência de dados AOF (append only file – discutido na sequência) e algoritmos de reescrita otimizados.

A replicação sem disco é empregada entre os fragmentos mestre e escravo, em vez de usar o fluxo de dados padrão baseado em arquivo. Os aprimoramentos no nível do sistema de arquivos permitem o acesso ideal ao armazenamento e suportam operações de gravação intermitente sem degradar (ou bloquear) o desempenho do banco de dados.

### **2.2.5.3 Sharding**

No Redis, a fragmentação de dados (sharding) é a técnica para dividir todos os dados em várias instâncias do Redis para que cada instância contenha apenas um subconjunto das chaves. Tal processo permite atenuar o crescimento dos dados adicionando mais e mais instâncias e dividindo os dados em partes menores (fragmentos ou partições). Além disso, também significa que mais e mais poder de computação está disponível para lidar com seus dados, dando suporte eficaz ao dimensionamento horizontal.

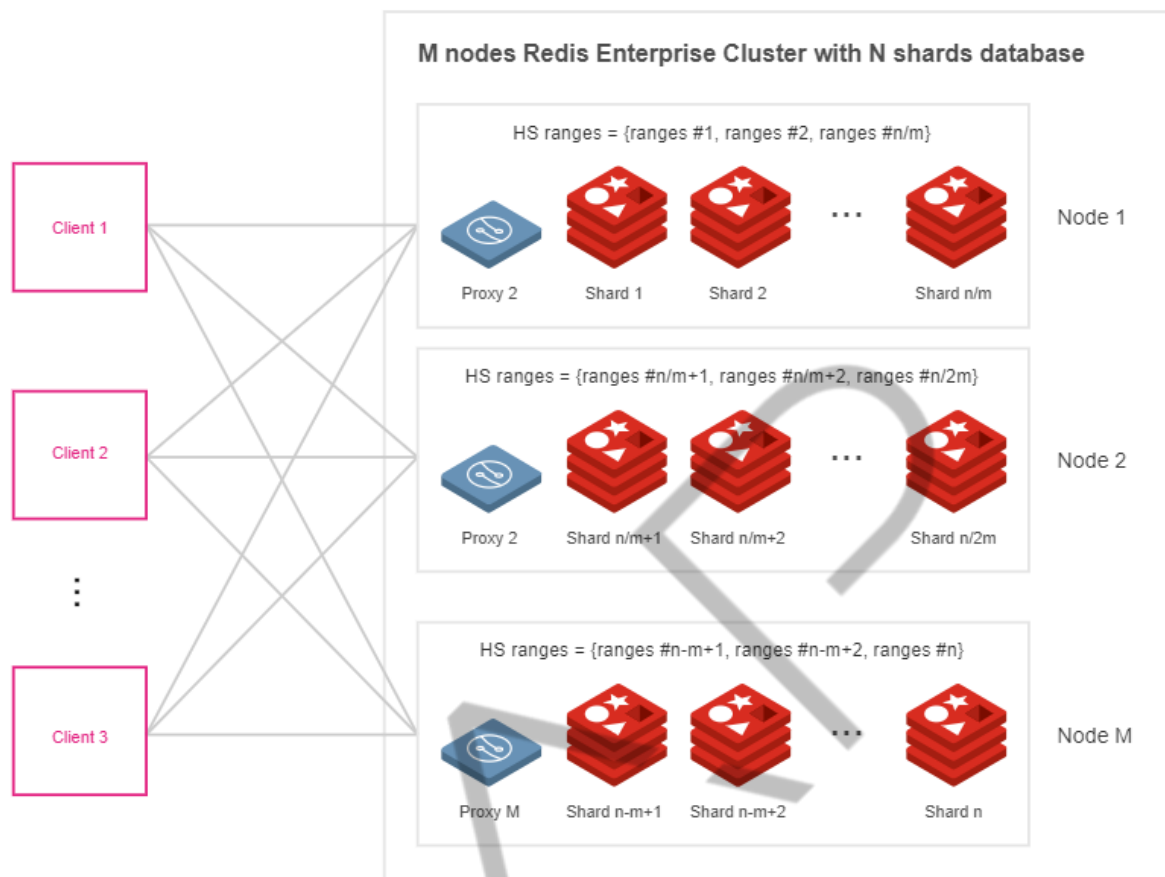


Figura 2.10 - Sharding  
Fonte: REDIS LABs (2020)

#### 2.2.5.4 Persistência de dados

No Redis, podemos criar uma cópia point-in-time dos dados na memória criando um snapshot. Após a criação, esses snapshots podem ser submetidos a backup, copiados para outros servidores para criar um clone do servidor ou deixados para uma reinicialização futura.

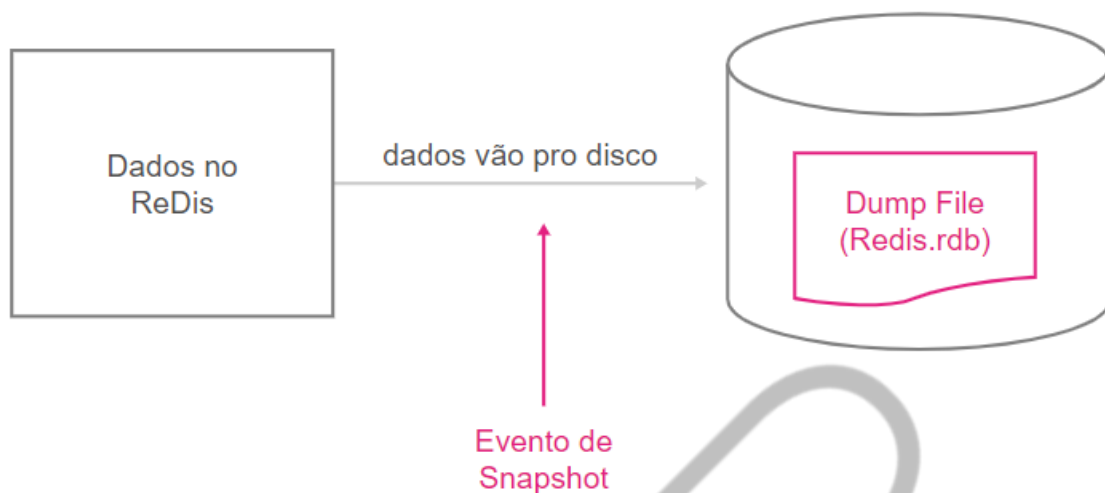


Figura 2.11 - Checkpoint  
Fonte: Elaborado pelo autor (2020)

No lado da configuração, os snapshots são gravados no arquivo referenciado no parâmetro *dbfilename* na configuração e armazenados no caminho referido como *dir*. Até que o próximo snapshot seja executado, os dados gravados no Redis desde o último snapshot iniciado (e concluído) seriam perdidos se houvesse um travamento causado pelo Redis, o sistema ou o hardware.

Quando a quantidade de dados que armazenamos no Redis tende a ser inferior a alguns gigabytes (em geral em ambientes big data), snapshots podem ser a resposta certa. Mas à medida que nosso uso de memória Redis cresce, o mesmo acontece com o tempo para executar uma operação de bifurcação para o BGSAVE.

Para evitar que a bifurcação cause tais problemas, podemos desabilitar o salvamento automático inteiramente. Quando o salvamento automático é desativado, precisamos chamar manualmente BGSAVE (que geraria uma certa contenção e congelamento do sistema no processo de gravação), ou podemos chamar SAVE. Com SAVE, o Redis bloqueia até que o salvamento seja concluído, mas porque não há bifurcação, não há atraso de bifurcação.

Os snapshots são muito úteis quando temos que lidar com perda massiva de dados em potencial, pois para muitos aplicativos, 15 minutos ou uma hora de perda de processamento de dados é tempo demais. Para permitir que o Redis mantenha as informações mais atualizadas possível, sobre os dados contidos na memória,

armazenamos em disco, porém usamos o formato AOF – append only file, que em termos de I/O é muito mais rápido.

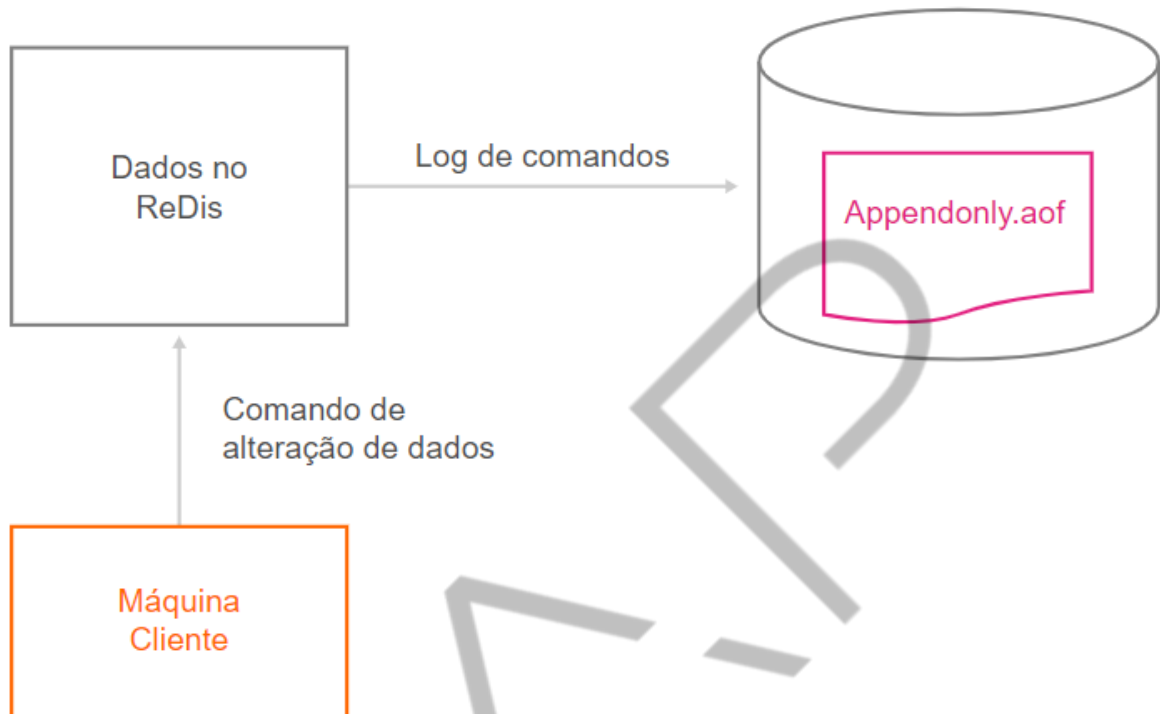


Figura 2.12 - Append only File  
Fonte: Elaborado pelo autor (2020)

Neste caso, como o arquivo é aberto somente para append, não há necessidade de controles de validação do conteúdo existente no arquivo ou mesmo da abertura total dele. Em contrapartida, o arquivo cresce muito rápido e isso consome espaço em disco, levando o DBA a ter um cuidado maior com esse ponto e, no caso de falha, o tempo de reinicialização do banco Redis é mais lento, uma vez que o arquivo AOF teria todas as versões temporais dos dados, mesmo os dados removidos ou alterados.

#### 2.2.5.5 Instalação Redis

Utilizaremos uma versão MS Windows para realizar os comandos básicos. Baixe o pacote Redis-x64-3.2.100.zip, conforme link abaixo.

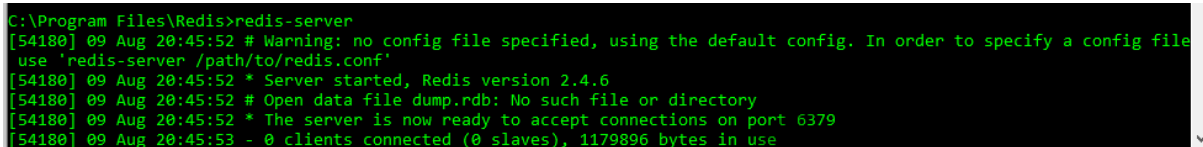


VERSÃO ATUAL DO REDIS PARA WINDOWS (Suporta os comandos atuais):

<https://github.com/MicrosoftArchive/redis/releases/download/win-3.2.100/Redis-x64-3.2.100.msi>

Abrir cmd no diretório C:\Program Files\Redis

Inicializar o redis-server e minimizar a janela.

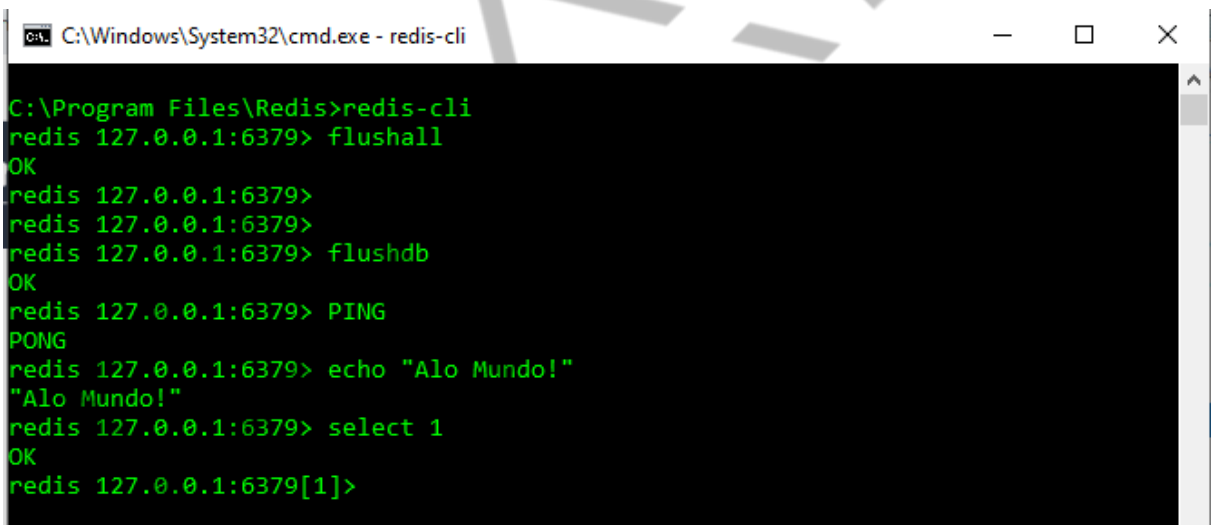


```
C:\Program Files\Redis>redis-server
[54180] 09 Aug 20:45:52 # Warning: no config file specified, using the default config. In order to specify a config file
use 'redis-server /path/to/redis.conf'
[54180] 09 Aug 20:45:52 * Server started, Redis version 2.4.6
[54180] 09 Aug 20:45:52 # Open data file dump.rdb: No such file or directory
[54180] 09 Aug 20:45:52 * The server is now ready to accept connections on port 6379
[54180] 09 Aug 20:45:53 - 0 clients connected (0 slaves), 1179896 bytes in use
```

Figura 2.13 - Redis Server  
Fonte: Redis (2020)

Abrir cmd no diretório C:\Program Files\Redis

Inicializar o redis-cli para executarmos os comandos de linha.



```
C:\Windows\System32\cmd.exe - redis-cli
C:\Program Files\Redis>redis-cli
redis 127.0.0.1:6379> flushall
OK
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> flushdb
OK
redis 127.0.0.1:6379> PING
PONG
redis 127.0.0.1:6379> echo "Alo Mundo!"
"Alo Mundo!"
redis 127.0.0.1:6379> select 1
OK
redis 127.0.0.1:6379[1]>
```

Figura 2.14 - Resultado dos comandos  
Fonte: REDIS (2020)

Outra forma de utilizar o CLI do REDIS é extrair os arquivos baixados (descompactar) e abrir os dois executáveis (redis-cli.exe) e (redis-server.exe). Pronto! Agora teste os comandos deste material e divirta-se!

### 2.2.5.6 Estruturas de Dados no Redis

O Redis não é somente um banco de armazenamento simples key-value, é na verdade um servidor de estruturas de dados, suportando diferentes tipos de valores. O que isso significa é que, enquanto nos bancos tradicionais de armazenamentos key-value você associa as chaves de string a valores de string, no Redis o valor não é limitado a uma string simples, podendo conter estruturas de dados mais complexas. A seguir está a lista das principais estruturas de dados suportadas pelo Redis:

- **Binary-Safe Strings:** todas as chaves no Redis são binary-safe strings, ou seja, você pode usar qualquer sequência binária como uma chave, de uma string como "oi" ao conteúdo de um arquivo JPEG.
- **LISTs:** coleções de elementos de string ordenadas de acordo com a ordem de inserção. Eles são basicamente listas vinculadas.
- **SETs:** coleções de elementos únicos e não ordenados de strings.
- **Sorted SETs:** semelhantes a conjuntos, mas cada elemento de sequência está associado a um valor numérico flutuante, chamado pontuação. Os elementos são sempre ordenados por sua pontuação, portanto, ao contrário de Sets, é possível recuperar um intervalo de elementos (por exemplo, você pode perguntar: me dê o top 10 ou o 10 inferior).
- **HASHs:** que são mapas compostos de campos associados a valores. O campo e o valor são strings. Isso é muito semelhante aos hashes Ruby ou Python.
- **BITMAPs:** é possível, usando comandos especiais, manipular valores de String como uma matriz de bits: você pode definir e limpar bits individuais, contar todos os bits definidos como 1, localizar o primeiro conjunto ou indefinido, e assim por diante.
- **HYPERLOGLOGs:** esta é uma estrutura de dados probabilística que é usada para estimar a cardinalidade de um conjunto.

**Algumas outras regras sobre chaves:**

Chaves muito longas não são uma boa ideia. Por exemplo, uma chave de 1024 bytes é uma má ideia, não apenas em termos de memória, mas também porque a consulta da chave no conjunto de dados pode exigir várias comparações de chaves, o que pode se tornar lento e pesado em termos computacionais. Mesmo quando a tarefa em questão é para coincidir com a existência de um valor grande, hash-IO (por exemplo, com SHA1) é uma ideia melhor, especialmente a partir da perspectiva de memória e largura de banda.

Chaves muito curtas não costumam ser uma boa ideia. Não faz sentido escrever "u1000f" como chave, se você puder escrever "Usuario: 1000: seguidores". O último é mais legível e o espaço adicionado é menor, comparado ao espaço usado pelo próprio objeto-chave e o objeto de valor. Enquanto chaves curtas, obviamente, consomem um pouco menos de memória, seu trabalho é encontrar o equilíbrio certo.

Tente ficar com um esquema. Por exemplo, "object-type: id" é uma boa ideia, como em "Usuario: 1000". Pontos ou traços são frequentemente usados para campos com várias palavras, como em "Citação: 1234: enviar.para" ou "Citação: 1234: enviar-para". O tamanho máximo permitido da chave é de 512 MB.

#### 2.2.5.7 Trabalhando com Strings

String é o tipo mais simples de valor que você pode associar a uma chave. Como as chaves Redis são strings, quando usamos o tipo string como um valor também, estamos mapeando uma string para outra string. O tipo de dados da string é útil para vários casos de uso, como o armazenamento em cache de fragmentos ou páginas HTML. Para começar, vamos executar alguns comandos prévios, por exemplo, apagar todos os dados do banco:

Sintaxe: FLUSHALL ou FLUSHDB

Onde:

**ALL** – apaga os dados de todos os bancos da instance em execução.

**DB** – apaga os dados somente do banco de trabalho escolhido.

Agora, vamos escolher um banco para trabalhar:

Sintaxe: `SELECT 1`

Vamos iniciar nossa jornada criando objetos simples, ou seja, strings:

Sintaxe: `SET key value [expiration EX seconds|PX milliseconds] [NX|XX]`

Onde:

**EX** - define o tempo de expiração especificado, em segundos.

**PX** - define o tempo de expiração especificado, em milissegundos.

**NX** - define a chave somente se ela ainda não existir.

**XX** - somente configure a chave, se já existir.

Teste (reproduza os comandos abaixo):

**SET** mykey myvalue

**SET** 10 'Aula de ReDis'

**GET** mykey

**GET** 10

Como você percebeu, o comando **SET** cria pares chave-valor (simples, neste caso), e o comando **GET** retorna os valores armazenados nas chaves informadas. Imagine agora que se precise sobrescrever parte da string armazenada em uma chave, para isso usamos o comando **SETRANGE**:

Sintaxe: `SETRANGE key offset value`

Onde:

**key** – é a chave que será usada para alteração do valor.

**offset** – posição numérica de onde será iniciada a alteração de string.

**value** – string de substituição.

Teste (reproduza os comandos abaixo):

**SET** mykey 'Ola mundo cruel'

**SETRANGE** mykey 4 'ventos'

**GET** mykey

Viu? Esse comando simplesmente vai na string de valor, na posição 4 (no exemplo acima) e “joga por cima” a nova string. Graças ao **SETRANGE** e aos comandos **GETRANGE** análogos, você pode usar strings Redis como um array linear com acesso aleatório. Este é um armazenamento muito rápido e eficiente em muitos casos de uso do mundo real.

Vamos agora para o comando **APPEND**, que se a chave já existir e for uma string, esse comando anexará o valor no final da string. Se a chave não existir, ela será criada e definida como uma string vazia, portanto, **APPEND** será semelhante a **SET** neste caso especial.

Sintaxe: **APPEND** key value

Onde:

**key** – é a chave que será usada para alteração do valor.

**value** – string de adição.

Teste (reproduza os comandos abaixo):

**SET** mykey2 'Ola mundo cruel'

**APPEND** mykey2 'demais'

**GET** mykey2

Agora, e se sua chave for uma string numérica que precisa ser incrementada? O Redis tem a solução – comandos **INCR** e **INCRBY**:

Sintaxe: **INCR** key ou **INCRBY** key number

Onde:

**key** – é a chave que será usada para alteração do valor.

**number** – valor a ser incrementado.

O comando INCR avalia o valor da string como um inteiro, incrementa-o de um em um, e finalmente, define o valor obtido como o novo valor. Existem outros comandos semelhantes, como INCRBY, DECR e DECRBY. Internamente, é sempre o mesmo comando, agindo de uma maneira um pouco diferente. Já no comando INCRBY, podemos informar qual o valor a ser incrementado, em vez de ser um a um.

Teste (reproduza os comandos abaixo):

**SET** counter 100

**INCR** counter

**INCR** counter

**INCRBY** counter 50

Além desses comandos, podemos fazer uso também dos comandos **TYPE** e **DEL**, que servem para trazer o tipo de valor que está armazenado em uma chave e, deleta a chave, respectivamente.

Teste (reproduza os comandos abaixo):

**SET** kkey x

**TYPE** kkey

**DEL** kkey

**TYPE** mykey

Outra função muito usada em sites de e-commerce, carrinhos de compra, cotações de dólares etc. é o comando EXPIRE, que funciona independentemente do tipo de valor. Basicamente, você pode definir um tempo-limite para uma chave, e

quando o tempo de vida transcorre, a chave é automaticamente destruída, exatamente como se o usuário chamasse o comando **DEL** com a chave.

Teste (reproduza os comandos abaixo):

**SET** key some-value

**EXPIRE** key 5

**GET** key --- execute esse comando imediatamente.

**GET** key --- execute esse comando após uns 5-6 segundos.

Uma forma de ver, de maneira cronológica decrescente, o tempo da chave acabando, é usar o comando **TTL**:

Teste (reproduza os comandos abaixo):

**SET** key 100 EX 10

**TTL** key --- repita o comando até que a chave expire.

Como pôde ver, o exemplo acima define uma chave com o valor da string 100, com uma validade de dez segundos. Posteriormente, o comando **TTL** é chamado para verificar o tempo restante para a chave. Para definir e verificar a validade em milissegundos, verifique os comandos **PEXPIRE** e **PTTL** e a lista completa de opções **SET**. Para obter uma lista completa dos comandos de manipulação de strings, acesse: <<https://redis.io/commands/set>>. Faça agora alguns exercícios:

a) Definindo strings atômicos

```
set volei_fem Meninas_do_Volei
get volei_fem
mset volei_fem_92 Barcelona volei_fem_96 Atlanta
keys *
get volei_fem_92
```

Código-fonte 2.1 – Redis String  
Fonte: Redis (2020)

Baseado nos resultados, como você definiria os comandos “novos” **MSET** e **KEYS** acima? Dica: **M** é de multi.

## b) Alterando e acrescentando valores:

```
get volei_fem
setrange volei_fem 17 Especial
set futebol_Brasil Time_70
append futebol_Brasil ' apenas craques'
```

Código-fonte 2.2 – Redis String  
Fonte: Redis (2020)

## c) Excluindo chaves:

```
set gremio campeao
get gremio
del gremio
get gremio
```

Código-fonte 2.3 – Redis String  
Fonte: Redis (2020)

## d) Brincando com o tempo

```
set inter 'Manga Claudio Figueroa Marinho Perez Vacaria
Cacapava Carpegiani Falcao Valdomiro Claudomiro Dario Lula'
get inter
expire inter 120
ttl inter
set gremio campeao ex 30
ttl gremio
get gremio
```

Código-fonte 2.4 – Redis TTL  
Fonte: Redis (2020)

### 2.2.5.8 Trabalhando com Conjuntos (SETs)

SETs são coleções **não ordenadas** de strings. Também é possível fazer várias outras operações em relação a conjuntos como teste, se um determinado elemento já existir, realizando a interseção, união ou diferença entre vários conjuntos e assim por diante, ou seja, pense a partir dessa nova estrutura de dados, que vamos trabalhar com conjuntos de valores, como fazíamos em matemática no ensino primário.



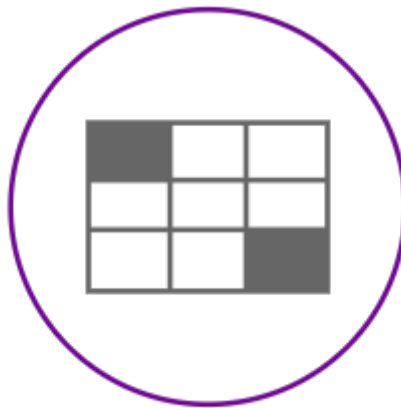


Figura 2.15 - SETs  
Fonte: SETs (2020)

Como vamos trabalhar com conjuntos, obviamente, poderemos adicionar mais de um valor (string) para uma chave, correto? Vamos observar o uso do comando **SADD**:

Sintaxe: **SADD** key member [member]

Onde:

**key** - é a chave que será usada para inserção do valor.

**member** – valor(es) a ser(em) adicionado(s).

Teste (reproduza os comandos abaixo):

```
SADD myset 1 2 3
```

```
SMEMBERS myset
```

Como visto, o comando para retornar os valores de uma chave, quando trabalhamos com SETs (conjuntos) não é o comando **GET** e sim o comando **SMEMBERS**. Repare que os comandos para trabalhar com a estrutura de dados SET são todos prefixionados por **S**, ou seja SET.

Já que estamos trabalhando com conjuntos, podemos brincar de UNION, DIFFERENCE e INTERSECTION, veja o exemplo gráfico e os comandos na sequência:

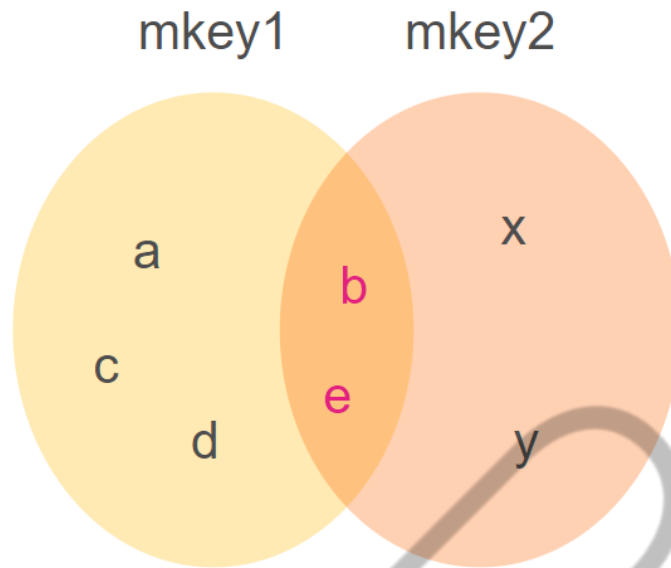


Figura 2.16 – Union, difference e intersection  
Fonte: adaptado de Redis (2020)

Sintaxe: `SUNION` | `SDIFF` | `SINTER` key [key]

Onde:

**key** - é a chave que será usada nas comparações.

Teste (reproduza os comandos abaixo):

Primeiramente crie os 2 conjuntos ilustrados acima mkey1 e mkey2.

```
SADD mkey1 a b c d e
```

```
SADD mkey2 b e x y
```

Liste agora os membros:

```
SMEMBERS mkey1
```

```
SMEMBERS mkey2
```

Agora, teste algumas operações:

```
SUNION mkey1 mkey2
```

```
SDIFF mkey1 mkey2
```

**SINTER** mkey1 mkey2

Como você pode ver, o comando **SUNION** retorna os membros do conjunto resultantes da união de todos os conjuntos dados. O comando **SDIFF** retorna os membros do conjunto resultantes da diferença entre o primeiro conjunto e todos os conjuntos sucessivos, e o comando **SINTER** retorna os membros do conjunto resultante da interseção de todos os conjuntos dados. Outro comando para se trabalhar com conjuntos (SET) bem útil é o **SISMEMBER**:

Sintaxe: **SISMEMBER** key member

Onde:

**key** - é a chave que será usada nas comparações.

**member** – valor a ser pesquisado.

Teste (reproduza os comandos abaixo):

**SADD** mst 1 2 3

**SISMEMBER** mst 3

**SISMEMBER** mst 30

Interessante, não? Pois conjuntos são bons para expressar relações entre objetos. Por exemplo, podemos facilmente usar conjuntos para implementar tags. Uma maneira simples de modelar esse problema é ter um conjunto para cada objeto que queremos marcar. O conjunto contém os IDs das tags associadas ao objeto. Vamos analisar agora o comando **SPOP**:

Sintaxe: **SPOP** key [count]

Onde:

**key** - é a chave que será usada nas comparações.

**count** – quantidade de valores a serem retornados randomicamente.

Teste (reproduza os comandos abaixo):

**SADD** mstA 1 2 3

**SPOP** mstA

Você entendeu sua proposta? Esse comando, **SPOP**, remove e retorna um ou mais elementos aleatórios do armazenamento de valor definido na chave. Essa operação é semelhante a **SRANDMEMBER**, que retorna um ou mais elementos aleatórios de um conjunto, mas não o remove.

Teste (reproduza os comandos abaixo):

A continuação do exemplo acima, veja quantos elementos ficaram no SET:

**SMEMBERS** mstA

Super, não é? Para outros comandos de conjunto (SET), acesse a documentação online Redis: <<https://redis.io/commands/sadd/>>. Faça agora alguns exercícios:

#### a) Listas

```
sadd time_70 Felix Carlos_Alberto Brito Piazza Everaldo  
Clodoaldo Gerson Rivellino Jairzinho Tostao Pele  
smembers time_70  
sadd time_74 Leao Ze_Maria Luis_Pereira Marinho_Perez  
Marinho_Chagas Piazza Carpegiani Rivellino Paulo_Cesar  
Valdomiro Jairzinho Dirceu  
smembers time_74
```

Código-fonte 2.5 – Redis Listas  
Fonte: Redis (2020)

#### b) Cuidado com underline!

```
sadd time_70 Felix Carlos_Alberto Brito Piazza Everaldo  
Clodoaldo Gerson Rivellino Jairzinho Tostao Pele Rivellino  
sadd time_70 Brito  
sadd time_70 Paulo_Cesar
```

```
sinter time_70 time_74  
sinterstore times_70_74 time_70 time_74  
smembers times_70_74  
sunion time_70 time_74
```

```
sdiff time_70 time_74
```

```
sismember time_70 Pele  
sismember time_70 Rivellino  
sismember time_74 Pele  
sismember time_74 Rivellino
```

Código-fonte 2.6 – Redis underline  
Fonte: Redis (2020)

O que será que o comando abaixo faz?

**sscan** time\_70 0 match C\*

Execute-o e tente explicar sua funcionalidade.

Nota: Comando disponível a partir da versão 2.8.0.

fonte: <https://redis.io/commands/sscan>

IMPORTANTE!

Teste online disponível em: <https://try.redis.io/>

### 2.2.5.9 Trabalhando com Conjuntos Ordenados (SETs)

Conjuntos ordenados são um tipo de agrupamento de dados semelhantes a uma mistura entre um Conjunto e um Hash (será visto posteriormente). Da mesma forma que os SETs, os conjuntos ordenados são compostos de elementos sequenciais únicos e não repetitivos, portanto, em certo sentido, um conjunto ordenado também é um conjunto.

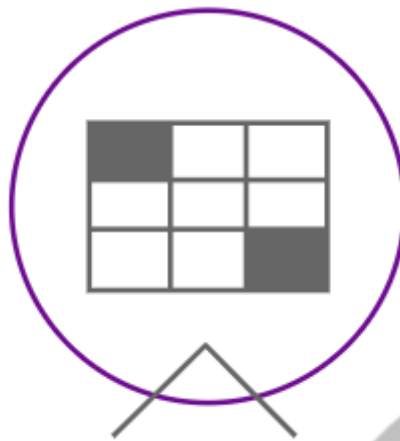


Figura 2.17 - SETs  
Fonte: SETs (2020)

No entanto, enquanto os elementos dentro dos conjuntos não são ordenados, cada elemento em um conjunto ordenado é associado a um valor de ponto flutuante, chamado score (é por isso que o tipo também é semelhante a um **HASH**, pois cada elemento é mapeado para um valor).

Além disso, os elementos de um conjunto ordenado são classificados (para que não sejam ordenados on-the-fly, a ordem é uma peculiaridade da estrutura de dados usada para representar conjuntos de classificação). Eles são ordenados de acordo com a seguinte regra: se A e B forem dois elementos com uma pontuação diferente, então  $A > B$  se  $A.score > B.score$ .

Se A e B tiverem exatamente a mesma pontuação, então  $A > B$  se a string A for lexicograficamente maior que a string B. As cadeias A e B não podem ser iguais, pois os conjuntos classificados têm apenas elementos exclusivos.

Como ambas as estruturas de dados são conjuntos (**SETs**), para diferenciarmos os conjuntos não ordenados dos conjuntos ordenados, usaremos os comandos prefixados por **Z**, ou seja, para adicionar membros a um conjunto ordenado, usaremos o comando **ZADD** que adiciona todos os membros especificados com as pontuações especificadas ao conjunto ordenado armazenado na chave. É possível especificar vários pares de pontuação/membro. Se um membro especificado já for um membro do conjunto classificado, a pontuação será atualizada e o elemento será reinserido na posição correta para garantir a ordem correta.

Se a chave não existir, um novo conjunto classificado com os membros especificados como membros únicos será criado, como se o conjunto classificado

estivesse vazio. Se a chave existir, mas não contiver um conjunto classificado, será retornado um erro. Os valores de pontuação devem ser a representação de string de um número de ponto flutuante de precisão dupla. + inf e -inf são valores válidos também.

Sintaxe: `ZADD key [NX|XX] [CH] [INCR] score member [score member ...]`

Onde:

**key** - é a chave que será usada nas comparações.

**NX** – atualiza apenas elementos que já existem, não adiciona novos.

**XX** – não atualiza elementos já existentes, somente adiciona.

**INCR** – faz o ZADD agir como o ZINCRBY.

**score** – critério de peso ou ordenação do valor.

**member** – valor a ser inserido.

Teste (reproduza os comandos abaixo):

```
ZADD myzset 2 "two" 3 "three"
```

Vamos criar agora uma lista de “hackers” históricos, usando o ano de seus “feitos” como score:

```
ZADD hackers 1940 "Alan Kay"
```

```
ZADD hackers 1957 "Sophie Wilson"
```

```
ZADD hackers 1953 "Richard Stallman"
```

```
ZADD hackers 1949 "Anita Borg"
```

```
ZADD hackers 1965 "Yukihiro Matsumoto"
```

```
ZADD hackers 1914 "Hedy Lamarr"
```

```
ZADD hackers 1916 "Claude Shannon"
```

```
ZADD hackers 1969 "Linus Torvalds"
```

```
ZADD hackers 1912 "Alan Turing"
```

Como você pode ver, o **ZADD** é semelhante ao **SADD**, mas recebe um argumento adicional (colocado antes do elemento a ser adicionado), que é a pontuação. O **ZADD** também é flexível, portanto, você está livre para especificar vários pares de valores de pontuação, mesmo que isso não seja usado no exemplo acima. Agora, e para trazer elementos de um conjunto ordenado?

Sintaxe: **ZRANGE** key start stop [WITHSCORES]

Onde:

**key** - é a chave que será usada nas comparações.

**start** – posição inicial da busca.

**stop** – posição final da busca.

**WITHSCORES** – retorna também os scores de cada valor.

Teste (reproduza os comandos abaixo):

**ZRANGE** hackers 0 -1

**ZRANGE** hackers 0 -1 withscores

Notou a diferença? No caso acima o 0 e -1 significa desde o índice do elemento 0 até o último elemento (-1 funciona aqui exatamente como no caso do comando **LRANGE**), elementos classificados, o Redis não precisa fazer nenhum trabalho, já está tudo ordenado. Outra forma de acessar seus elementos é assim:

Sintaxe: **ZRANGEBYSCORE** key min max [WITHSCORES] [LIMIT offset count]

Onde:

**key** - é a chave que será usada nas comparações.

**start** – posição inicial da busca.

**stop** – posição final da busca.

**WITHSCORES** – retorna também os scores de cada valor.



**LIMIT** – retorna somente um número limitado de resultados.

Teste (reproduza os comandos abaixo):

**ZREMRANGEBYSCORE** hackers 1940 1960

**ZRANGEBYSCORE** hackers -inf 1950

Ou seja, os comandos acima vão retornar todos os elementos no conjunto classificado na chave com uma pontuação entre min e max (incluindo elementos com pontuação igual a min ou max). Os elementos são considerados ordenados de baixa a alta pontuação.

Os elementos com a mesma pontuação são retornados em ordem lexicográfica (isso segue de uma propriedade da implementação do conjunto classificado no Redis e não envolve computação adicional).

O argumento **LIMIT** opcional pode ser usado para obter apenas um intervalo dos elementos correspondentes (semelhante ao deslocamento **SELECT LIMIT**, contagem em SQL). Lembre-se de que, se o deslocamento for grande, o conjunto classificado precisará ser percorrido pelos elementos de deslocamento antes de chegar aos elementos a serem retornados, o que pode adicionar complexidade de tempo  $O(N)$ . O argumento opcional **WITHSCORES** faz com que o comando retorne o elemento e sua pontuação, em vez do elemento sozinho.

Intervalos exclusivos e infinitos min. e max. podem ser -inf e +inf, para que você não precise saber a pontuação mais alta ou mais baixa no conjunto classificado para obter todos os elementos de uma pontuação ou até uma determinada pontuação. Por padrão, o intervalo especificado por min e max é fechado (inclusive). Já que temos um score, podemos “ranquear” os valores:

Sintaxe: **ZRANK** key member

Onde:

**key** - é a chave que será usada nas comparações.

**member** – valor a qual se deseja verificar o rank.

Teste (reproduza os comandos abaixo):

**ZRANK** hackers "Anita Borg"

**ZREVRANK** hackers "Anita Borg"

Para outros comandos sobre conjuntos ordenados, acesse a documentação on-line Redis: <<https://redis.io/commands/zadd>>. Vamos fazer um pouco mais de exercícios sobre este tema?

```
zadd formula_1 3 Senna
zrange formula_1 0 -1
zadd formula_1 2 Fittipaldi
zadd formula_1 1 Hill
zadd formula_1 3 Piquet
zadd formula_1 7 Schumacher
zadd formula_1 4 Prost
zadd formula_1 2 Hakkinen
```

```
zrevrange formula_1 0 -1
zrangebyscore formula_1 2 3
zrangebyscore formula_1 2 3 withscores
zrank formula_1 Piquet
```

Código-fonte 2.7 – Redis Sorted List  
Fonte: Redis (2020)

O que será que o comando abaixo faz?

**zscan** formula\_1 0 match \*i\*

#### 2.2.5.10 Trabalhando com LISTS

Para explicar o tipo de dados LIST, é melhor começar com um pouco de teoria, já que o termo LIST é frequentemente usado de maneira imprópria pelas pessoas da tecnologia da informação. Por exemplo, "Listas de Python" não são o que o nome pode sugerir (Listas vinculadas), mas sim Arrays (o mesmo tipo de dado é chamado Array in Ruby na verdade).



Figura 2.18 - LISTs  
Fonte: LISTs (2020)

De um ponto de vista muito geral, uma Lista é apenas uma sequência de elementos ordenados: 10,20,1,2,3 é uma lista. Mas as propriedades de uma lista implementada usando uma matriz são muito diferentes das propriedades de uma lista implementada usando uma lista vinculada.

As listas Redis são implementadas por meio de listas vinculadas. Isso significa que, mesmo se você tiver milhões de elementos em uma lista, a operação de adicionar um novo elemento na cabeça ou na cauda da lista é executada em tempo constante. A velocidade de adicionar um novo elemento com o comando **LPUSH** ao início de uma lista com dez elementos é o mesmo que adicionar um elemento ao cabeçalho da lista com 10 milhões de elementos.

Qual é o lado negativo? Acessar um elemento por índice é muito rápido em listas implementadas com um Array (acesso indexado em tempo constante) e não tão rápido em listas implementadas por listas vinculadas (a operação requer uma quantidade de trabalho proporcional ao índice do elemento acessado).

As listas Redis são implementadas com listas vinculadas porque, para um sistema de banco de dados, é crucial poder adicionar elementos a uma lista muito longa de maneira muito rápida. Outra grande vantagem, como você verá daqui a pouco, é que as Listas Redis podem ser tiradas em tempo constante.

Quando o acesso rápido ao meio de uma grande coleção de elementos é importante, há uma estrutura de dados diferente que pode ser usada, chamada de conjuntos de classificação.

Sintaxe: LPUSH | RPUSH key value [value]

Onde:

**key** - é a chave que será usada para inserção/consulta do valor.

**value** - valor a ser inserido.

Cuidado aqui, você notou que identificamos o tipo de estrutura de dados no Redis, pelo prefixo do comando, certo? Neste caso, os prefixos L e R indicam: Left e Right respectivamente.

Teste (reproduza os comandos abaixo):

**RPUSH** mylist A B

**LPUSH** mylist first

Ou seja, LPUSH – insere todos os valores especificados no topo da lista armazenada na chave. Se a chave não existir, ela será criada como uma lista vazia antes de executar as operações de envio. Quando a chave contém um valor que não é uma lista, um erro é retornado.

É possível enviar vários elementos usando uma única chamada de comando, especificando vários argumentos no final do comando. Os elementos são inseridos um após o outro no início da lista, do elemento mais à esquerda até o elemento mais à direita. Assim, por exemplo, o comando **LPUSH** mylist a b c resultará em uma lista contendo c como primeiro elemento, b como segundo elemento e a como terceiro elemento.

O comando **RPUSH**, insere todos os valores especificados na parte final da lista (à direita) armazenada na chave. Se a chave não existir, ela será criada como uma lista vazia antes de executar a operação de envio. Quando a chave contém um valor que não é uma lista, um erro é retornado.

É possível enviar vários elementos usando uma única chamada de comando, especificando vários argumentos no final do comando. Os elementos são inseridos um após o outro na parte final da lista, do elemento mais à esquerda até o elemento mais à direita. Assim, por exemplo, o comando **RPUSH** mylist a b c resultará em

uma lista contendo a como primeiro elemento, b como segundo elemento e c como terceiro elemento.

Para listar os elementos, vamos usar o comando **LRange**:

Teste (reproduza os comandos abaixo):

**LRange** mylist 0 -1

Vamos criar outra lista:

**RPush** mylist 1 2 3 4 5 "foo bar"

**LRange** mylist 0 -1

Resumindo, observe que **LRange** leva dois índices, o primeiro e o último elemento do intervalo a retornar. Ambos os índices podem ser negativos, dizendo ao Redis para começar a contar a partir do final: então -1 é o último elemento, -2 é o penúltimo elemento da lista e assim por diante. Como você pode ver, o **RPush** anexou os elementos à direita da lista, enquanto o **LPush** final anexou o elemento à esquerda.

Lembram do comando POP?

Sintaxe: **LPOP** | **RPOP** key

Onde:

**key** - é a chave que será usada para a operação.

Teste (reproduza os comandos abaixo):

**RPush** mylist a b c

**RPOP** mylist

**RPOP** mylist

**RPOP** mylist

Carrinho de compras lhe parece familiar neste contexto? O que temos então, o comando **LPOP**, remove e retorna o primeiro elemento da lista armazenada na

chave, e o comando RPOP por sua vez, remove e retorna o último elemento da lista armazenada na chave. Ah, há também o comando TRIM:

Sintaxe: **LTRIM** key start stop

Onde:

**key** - é a chave que será usada para a operação.

**start** – posição inicial.

**stop** – posição final para a operação do trim.

Teste (reproduza os comandos abaixo):

**RPUSH** mylst2 1 2 3 4 5

**LTRIM** mylst2 0 2

**LRANGE** mylst2 0 -1

Resumindo, podemos “aparar” uma lista existente para que ela contenha apenas o intervalo especificado de elementos listados. Tanto o **start** quanto o **stop** são índices baseados em zero, onde 0 é o primeiro elemento da lista (o head), 1 o próximo elemento e assim por diante.

Por exemplo: **LTRIM** foobar 0 2 modificará a lista armazenada no foobar para que somente os três primeiros elementos da lista permaneçam. Os valores de **start** e **stop** também podem ser números negativos indicando offsets do final da lista, onde

-1 é o último elemento da lista, -2 o penúltimo elemento e assim por diante.

Índices fora do intervalo não produzirão um erro: se start for maior que o final da lista, ou start > end, o resultado será uma lista vazia (o que faz com que a chave seja removida). Se o final for maior que o final da lista, o Redis o tratará como o último elemento da lista. Imaginando o seguinte exemplo:

**LPUSH** mylist <some element>

**LTRIM** mylist 0 999

A combinação acima adiciona um novo elemento e leva apenas os 1000 elementos mais novos para a lista. Com o **LRange**, você pode acessar os principais itens sem precisar memorizar dados muito antigos. Para outros comandos sobre listas de valores no Redis, acesse sua documentação online: <<https://redis.io/commands/lpush>>. Vamos exercitar um pouco mais?

```
lpush cor Branco Laranja Roxo Azul Amarelo Vermelho Verde
Preto
lrange cor 0 -1
```

```
lpush volei_fem_prata Ana_Moser Ida Ana_Flaviana Ana_Paula
Fernanda_Venturini Ana_Lucia Ana_Maria_Volpone Fofao Leila
lrange volei_fem_prata 0 -1
lrange volei_fem_prata 0 -4
lrange volei_fem_prata 3 -1
lrange volei_fem_prata 3 -3
```

```
rpush volei_fem_prata Hilma
lpush volei_fem_prata Marcia_Fu
```

Código-fonte 2.8 – Redis Pilhas  
Fonte: Redis (2020)

a) 2 vezes 2 entradas:

```
rpush volei_fem_prata Marcia_Fu
rpush volei_fem_prata Ida
```

Código-fonte 2.9 – Redis Pilhas  
Fonte: Redis (2020)

b) Sacando a Ana Moser:

```
rpop volei_fem_prata
lrange volei_fem_prata 0 -1
```

Código-fonte 2.10 – Redis Pilhas  
Fonte: Redis (2020)

c) Sacando a Ida e a Leila:

```
rpopt volei_fem_prata
lpop volei_fem_prata
```

Código-fonte 2.11 – Redis Pilhas  
Fonte: Redis (2020)

d) Podando a lista:

```
ltrim volei_fem_prata 0 2
```

Código-fonte 2.12 – Redis Lista  
Fonte: Redis (2020)

### 2.2.5.11 Trabalhando com HASHs

Os **HASHs** Redis têm exatamente a aparência esperada de um "hash", com pares de valor de campo, embora os **HASHs** sejam úteis para representar objetos, na verdade, o número de campos que você pode colocar dentro de um **HASH** não tem limites práticos (além da memória disponível), portanto, você pode usar **HASHs** de muitas maneiras diferentes dentro de seu aplicativo.



Figura 2.19 - HASHs  
Fonte: HASHs (2020)

O comando **HMSET** define vários campos do hash, enquanto o **HGET** recupera um único campo. O **HMGET** é semelhante ao **HGET**, mas retorna uma matriz de valores. Em suma, são estruturas muito parecidas com tabelas.

Teste (reproduza os comandos abaixo):

**HMSET** user:1000 username antirez birthyear 1977 verified 1

**HGET** user:1000 username

**HGET** user:1000 birthyear

**HMGET** user:1000 username birthyear endereço

**HGETALL** user:1000



Você reparou que o comando **HASH** permite que se adicione a uma chave pares de valores, veja acima. Adicionamos à chave user:1000 os pares (username=antirez), (birthyear=1977), (verified=1), ou seja, quase uma estrutura tabular.

a) Podemos usar o **INCR** e **INCRBY** nos **HASHs** também, veja Hashed:

```
hmset volei_fem_ouro Fofao levantadora Fabi libero Mari oposta  
Sheilla ponta Paula_Pequeno ponta Thaisa cerntral  
hmget volei_fem_ouro Mari  
hmset ti 'Alan Kay' 1940 'Sophie Wilson' 1957 'Richard  
Stallman' 1953 'Anita Borg' 1949 'Yukihiro Matsumoto' 1965  
'Hedy Lamarr' 1914 'Linus Tovalds' 1969 'Alan Turing' 1912  
'Ada Augusta King' 1815 'Jean Jennings Bartik' 1924  
hmget ti 'Hedy Lamarr' 'Ada Augusta King'
```

Código-fonte 2.13 – Redis Hashed

Fonte: Redis (2020)

Misturando tudo até então:

Trabalhando com dados Expirados:

**expire** formula\_1 120

**ttl** formula\_1

**zrange** formula\_1 3 -1

**sadd** time\_74 Leao Ze\_Maria Marinho\_Perez Luis\_Pereira Marinho\_Chagas  
Piazza Rivellino Valdomiro Jairzinho Paulo\_Cesar Dirceu

Trabalhando com conjuntos de dados:

**sinter** time\_70 time\_74

**sinterstore** times\_70\_74 time\_70 time\_74

**smembers** times\_70\_74

**sunion** time\_70 time\_74

**sdiff** time\_70 time\_74

### 2.2.5.12 Trabalhando com GEOreferências

As georreferências no Redis são tratadas e armazenadas praticamente em um formato de listas ordenadas (ZADD), os comandos do tipo GEO armazenarão pares de coordenadas geográficas.



Figura 2.20 - Georreferências  
Fonte: Redis (2020)

O comando aceita argumentos no formato padrão x, y, portanto, a longitude deve ser especificada antes da latitude. Geolocalização – apesar de ser restrita para versões mais recentes, adicionamos aqui a sintaxe.

**Sintaxe: GEOADD** key longitude latitude member

Onde:

Adiciona os itens geo espaciais especificados (latitude, longitude, nome) à chave especificada.

Teste (reproduza os comandos abaixo):

```
GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669  
"Catania"
```

No comando acima, estamos adicionando a chave SICILY, duas coordenadas (Lat Lon), uma para Palermo e outra para Catania. Caso precisemos calcular a distância entre essas duas coordenadas:

Sintaxe: **GEODIST** key member1 member2 (unit)

Onde:

Retorna a distância entre dois membros no índice geo espacial representado pelo conjunto classificado.

**Unit** - unidade de medida sendo, **m (metros)**, km (quilômetros), mi (milhas), ft (feet).

Teste (reproduza os comandos abaixo):

**GEODIST** Sicily Palermo Catania

E se precisarmos criar um círculo em uma coordenada geográfica para ver sua abrangência, especificando uma unidade para compor seu raio?

Sintaxe: **GEORADIUS** key longitude latitude radius [withcord][withdist]...

Onde:

Retorna os membros de um conjunto classificado preenchido com informações geo espaciais usando **GEOADD**, que estão dentro das bordas da área especificada com a localização central e a distância máxima do centro (o raio).

Teste (reproduza os comandos abaixo):

**GEORADIUS** Sicily 15 37 200 km WITHDIST

Para comandos adicionais sobre o tema, acesse a documentação on-line Redis: <<https://redis.io/commands/geoadd>>. Agora, vamos praticar:

```
geoadd Bra -22.54 -43.12 "Rio_de_Janeiro"
geoadd Bra -23.32 -46.38 "Sao_Paulo"
geoadd Bra -30.01 -51.13 "Porto_Alegre"
geoadd Bra -25.25 -49.16 "Curitiba"
geoadd Bra -27.35 -48.32 "Florianopolis"
geoadd Bra -26.18 -48.50 "Joinville"
geoadd Bra -23.05 -48.55 "Avaré"
```

```

geoadd Bra -23.32 -54.35 "Foz_do_Iguacu" -29.45 -57.05
"Uruguaiana"
geohash Bra Sao_Paulo Rio_de_Janeiro Porto_Alegre Curitiba
geodist Bra Sao_Paulo Porto_Alegre km
geodist Bra Uruguaiana Porto_Alegre km
geopos Bra Foz_do_Iguacu Rio_de_Janeiro Piracicaba
georadius Bra -25 -49 700 km
georadius Bra -25 -49 200 km
georadius Bra -25 -49 700 km withdist
georadius Bra -25 -49 700 km withdist withcoord

```

Código-fonte 2.14 – Redis Georadius  
Fonte: Redis (2020)

## 2.3 Cases

### 2.3.1 Sistema IoT para aquisição de dados com Redis e linguagem Golang

Cintra (2019) apresenta uma solução num cenário em que vários sensores dispostos local ou remotamente enviam dados de suas medições periodicamente para um servidor. Uma rotina coletora (GOLANG) armazena os dados no REDIS para que a aplicação possa processar e apresentar as informações em um dashboard.

Em resumo, uma rede de sensores (1), conectados por diferentes protocolos de comunicação (2), administrados por um concentrador (3), com rotinas desenvolvidas na linguagem Golang grava via TCP no cache (4), mais especificamente no NoSQL Redis os dados gerados por ela.

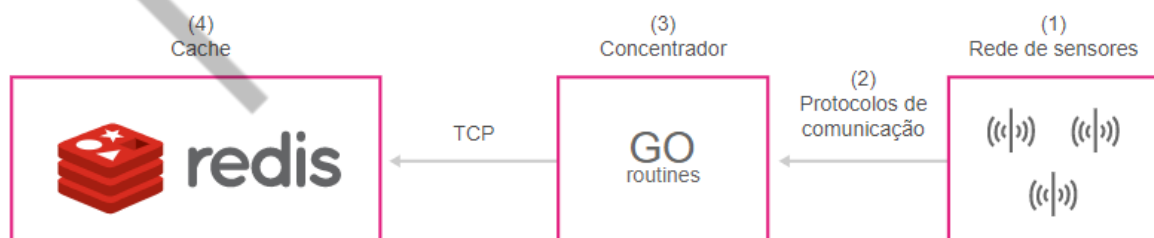


Figura 2.21 - IoT x Golang x Redis  
Fonte: Cintra (2019)

- Os sensores estão conectados aos dispositivos microcontroladores que enviam os dados para o concentrador. Não existem restrições quanto aos

modelos de placas utilizadas, podendo ser Arduino, ESP8266, ESP32, PIC etc.

- Para a comunicação entre os dispositivos e o concentrador estão previstos alguns padrões como Wi-fi/Ethernet (Websocket/HTTP/REST), RF/LORA, Comunicação serial/Bluetooth e GSM/GPRS.
- Rotina em GO que recebe os dados dos sensores, processa e envia para o cache REDIS.
- Armazena os dados dos sensores de forma temporária ou permanente. Caso seja necessário, o cache pode ser espelhado (replicado).

Os dados armazenados dos sensores são a identificação do sensor que está enviando a informação; a data e hora de envio; e o valor da medição. Os outros dados como qual o tipo de sensor, localização, unidade de medida e outras ficarão a cargo da aplicação supervisória. A identificação do sensor é o ID do sensor na forma de uma String composta de 3 letras e uma sequência numérica unidas pelo separador pipe (“|”).

**Exemplo:** Um sensor de temperatura, não importa qual padrão ou modelo, poderá ter o ID “TMP|1” ou um sensor de umidade poderá ter o ID “UMD|100”. Os IDs não se repetem, ou seja, não pode haver dois sensores com a mesma identificação, como em uma chave primária. A data e hora de envio será armazenada no formato Unix Time Stamp. O valor da medição é uma string armazenando um valor numérico.

Dentre as estruturas de dados disponibilizadas pelo REDIS, escolhemos as listas ligadas pois as operações de inserção e remoção no final da lista são extremamente rápidas, que é o que precisamos. Um exemplo das operações de inserção e remoção da lista: Supondo que, no dia “01/01/2020” à “01:00:05”, o sensor de código “TMP|5” enviou o valor “25.25”.

A chave da nossa lista será simplesmente o ID do sensor. Portanto, teremos uma lista para cada sensor. Convertendo a data para Unix, temos: “1577840405”. Então o comando para armazenar esses dados no fim da lista seria esse:

```
RPUSH TMP|5 1577840405|25.25
```

Código-fonte 2.15 – Inserção no final da lista  
Fonte: Cintra (2019)

Para obter o último valor enviado pelo sensor “TMP|5”, emitimos o seguinte comando:

```
RPOP TMP|5
```

Código-fonte 2.16 – Obter último valor  
Fonte: Cintra (2019)

### 2.3.2 Evolução arquitetura uber

Shiftehfir (2018) apresentou a evolução da arquitetura de dados da Uber separando-a em quatro gerações.

#### **Primeira Geração - Ambiente Analítico Tradicional:**

A primeira geração da plataforma de dados do Uber agregou todos os dados do Uber em um só lugar e forneceu interface SQL padrão para os usuários acessarem os dados. Isso resultou em um grande número de novas equipes usando a análise de dados como base para suas decisões de tecnologia e produto. Em poucos meses, o tamanho de dados analíticos cresceu para dezenas de Terabytes e o número de usuários aumentou para várias centenas. O uso de SQL como uma interface padrão simples permitiu que os operadores da cidade interagissem facilmente com os dados sem conhecer as tecnologias subjacentes.

A tecnologia Vertica foi escolhida para o data warehouse por causa de seu design rápido, escalonável e orientado a colunas. Foram desenvolvidos vários trabalhos ETL (Extrair, Transformar e Carregar) para carregar diferentes fontes de dados, como dados vindos AWS S3, bancos de dados OLTP, logs de serviço, entre outros.

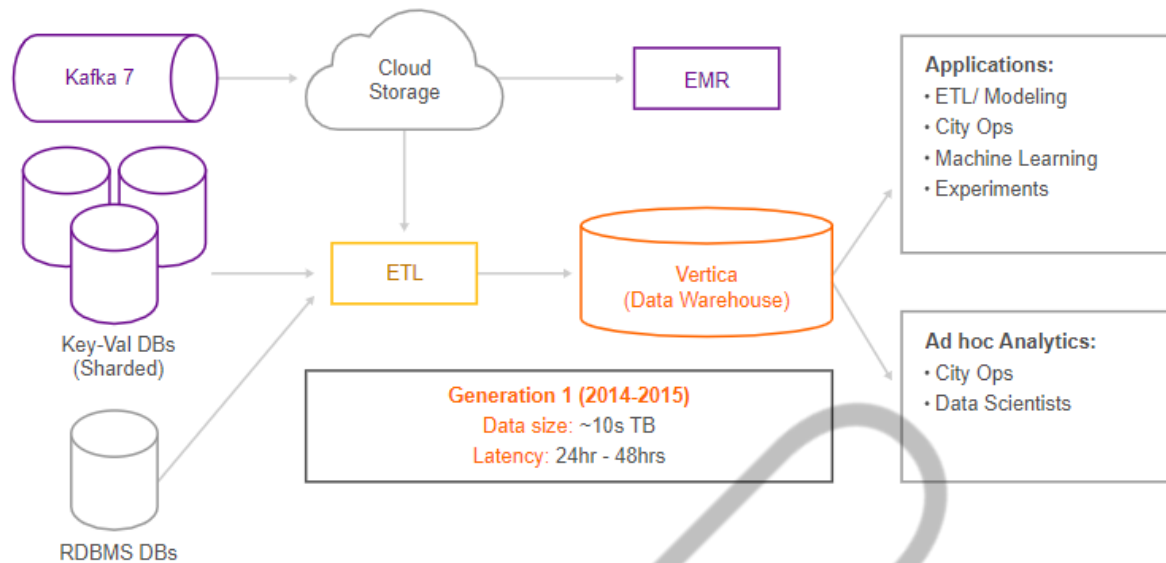


Figura 2.22 - Uber 1ª Geração  
Fonte: Shiftehfar (2018)

### Segunda Geração - chegada Hadoop:

A segunda geração da plataforma de dados do Uber aproveitou o Hadoop para permitir a escalabilidade horizontal. Incorporando tecnologias como Parquet, Spark e Hive, dezenas de petabytes de dados foram ingeridos, armazenados e servidos.

Logo atingiu dezenas de petabytes de dados. Diariamente, havia dezenas de terabytes de novos dados adicionados ao data lake, e a plataforma de dados cresceu para mais de 10.000 vcores com mais de 100.000 jobs em lote em execução em qualquer dia. Isso fez do data lake Hadoop uma fonte da verdade centralizada para todos os dados analíticos do Uber.

### Generation 2 (2015 - 2016) - The arrival of Hadoop

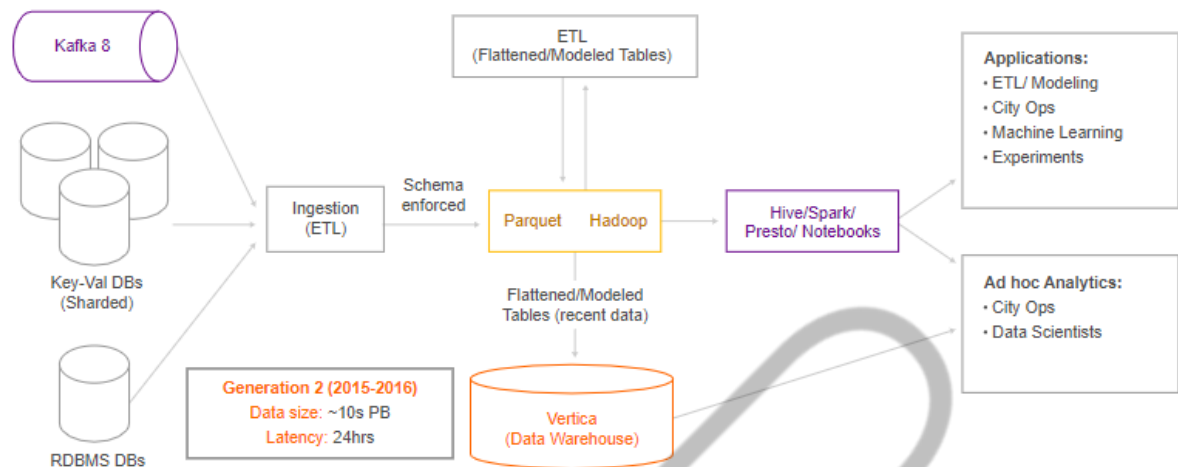


Figura 2.23 - Uber 2ª Geração  
Fonte: Shiftahfar (2018)

O Hadoop habilitava o armazenamento de vários petabytes de dados na plataforma, e a latência para novos dados ainda era superior a um dia, um atraso devido à ingestão de grandes tabelas de origem upstream que levavam várias horas para processar.

### Generation 2 (2015 - 2016) - The arrival of Hadoop

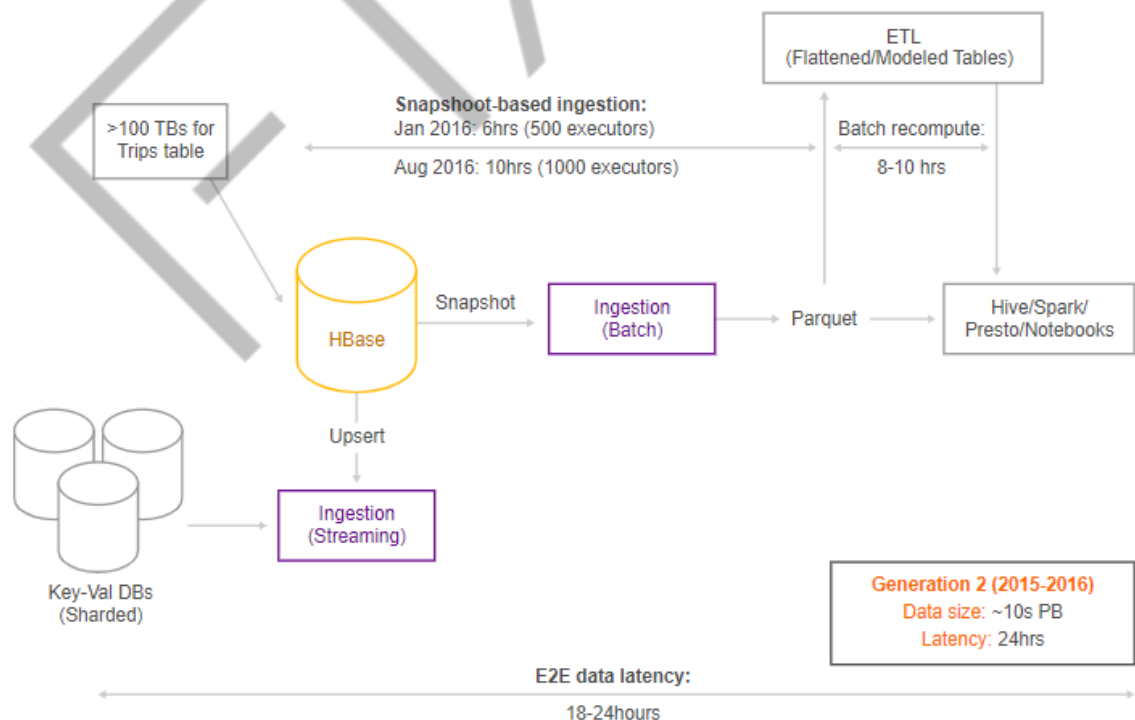


Figura 2.24 - Uber 2ª Geração - Latência  
Fonte: Shiftahfar (2018)



### Terceira Geração – nova arquitetura:

Adicionada à camada Hadoop Upserts e Incremental (Hudi), uma biblioteca de código aberto do Spark que fornece uma camada de abstração sobre HDFS e Parquet para dar suporte às operações de atualização e exclusão necessárias. O Hudi pode ser usado a partir de qualquer trabalho do Spark, tem escalabilidade horizontal e depende apenas do HDFS para operar.

Hudi permite atualizar, inserir e excluir dados Parquet existentes no Hadoop. Além disso, o Hudi permite que os usuários de dados extraiam incrementalmente apenas os dados alterados, melhorando significativamente a eficiência da consulta e permitindo atualizações incrementais de tabelas modeladas derivadas.

#### Generation 3 (2017 - present) - Let's rebuild for long term

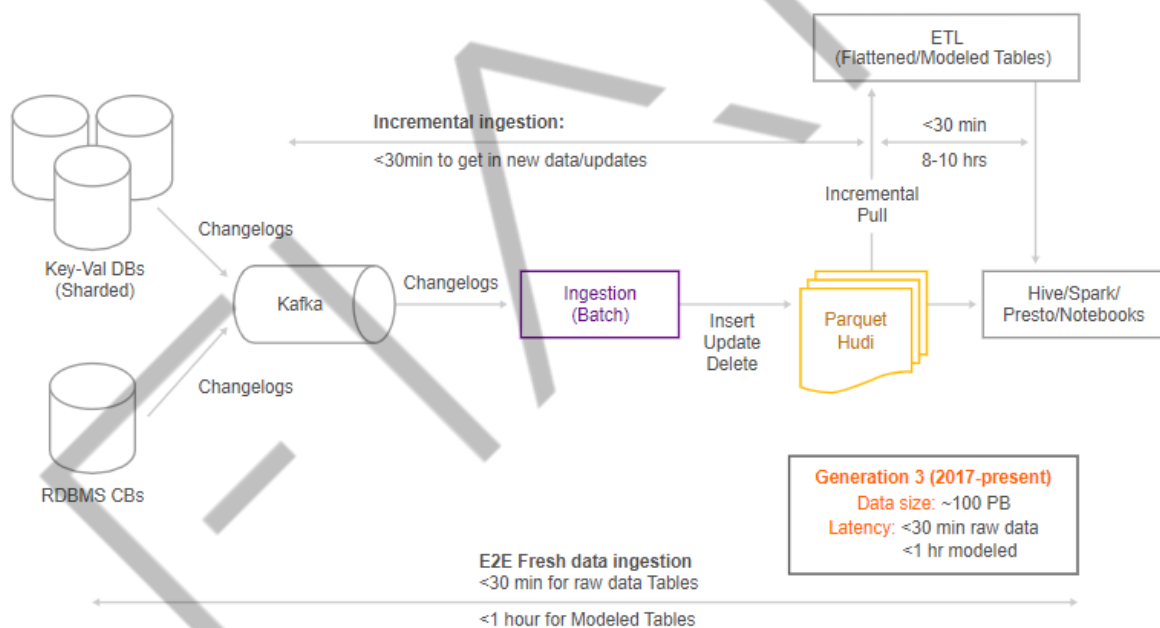


Figura 2.25 - Uber 3ª Geração - Reconstrução  
 Fonte: Shiftehfar (2018)

A plataforma de ingestão de dados, Marmaray, é executada em minilotes e coleta os changelogs de armazenamento upstream do Kafka, aplicando-os sobre os dados existentes no Hadoop usando Hudi. Os trabalhos do Spark de ingestão são executados a cada 10-15 minutos, fornecendo uma latência de dados brutos de 30 minutos no Hadoop (tendo espaço para 1-2 falhas ou novas tentativas de trabalho de ingestão).

Para evitar ineficiências resultantes da ingestão dos mesmos dados de origem no Hadoop mais de uma vez, a configuração não permite quaisquer transformações durante a ingestão de dados brutos, resultando na decisão de tornar a estrutura de ingestão de dados brutos uma plataforma EL em oposição a uma plataforma ETL tradicional. Sob este modelo, os usuários são encorajados a executar as operações de transformação desejadas no Hadoop e no modo em lote após os dados upstream chegarem em seu formato bruto.

### Quarta Geração – novos desafios

A construção de uma plataforma de transferência de dados mais extensível permitiu agregar facilmente todos os pipelines de dados de uma forma padrão em um serviço, bem como oferecer suporte à conectividade entre diversas fontes de dados e coletores de dados.

Generic Any-to-Any Data platform

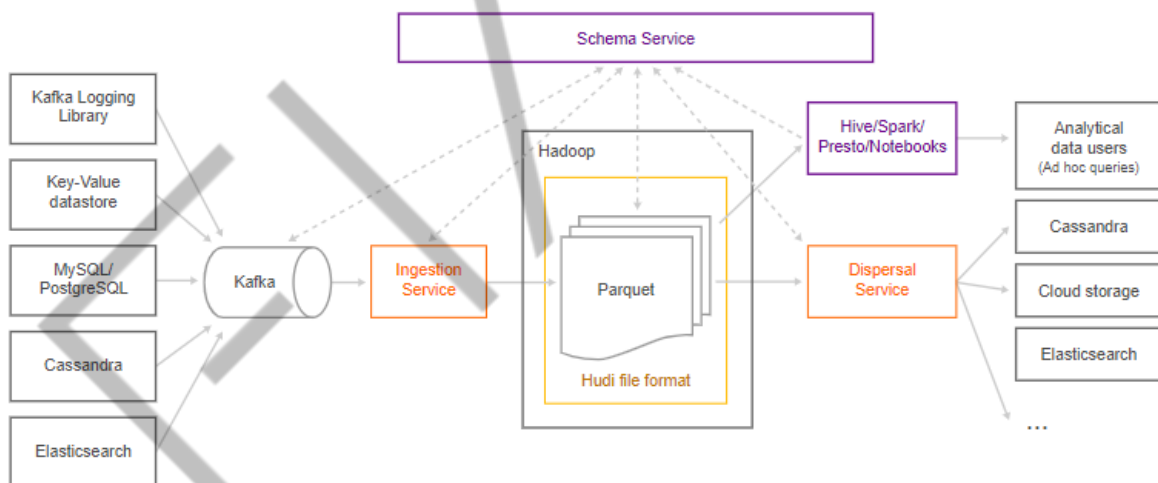


Figura 2.26 - Uber Plataforma Genérica  
Fonte: Shiftehfar (2018)

E mais recentemente, Holler e Mui (2019) apresentaram a Plataforma Michelangelo – uma plataforma de aprendizado de máquina (ML) da Uber, que oferece suporte ao treinamento e ao atendimento de milhares de modelos em produção em toda a empresa. Projetado para cobrir o fluxo de trabalho de ML de ponta a ponta, o sistema atualmente suporta aprendizado de máquina clássico, previsão de série temporal e modelos deep learning para muitos casos de uso, que

vão desde a geração de previsões de mercado, resposta a tíquetes de suporte ao cliente até cálculos precisos de tempos estimados de chegada (ETAs) e ativando nosso recurso One-Click Chat usando modelos de processamento de linguagem natural (NLP) no aplicativo de motorista. A maioria dos modelos Michelangelo baseia-se no Apache Spark MLlib, uma biblioteca de aprendizado de máquina escalonável para o Apache Spark.

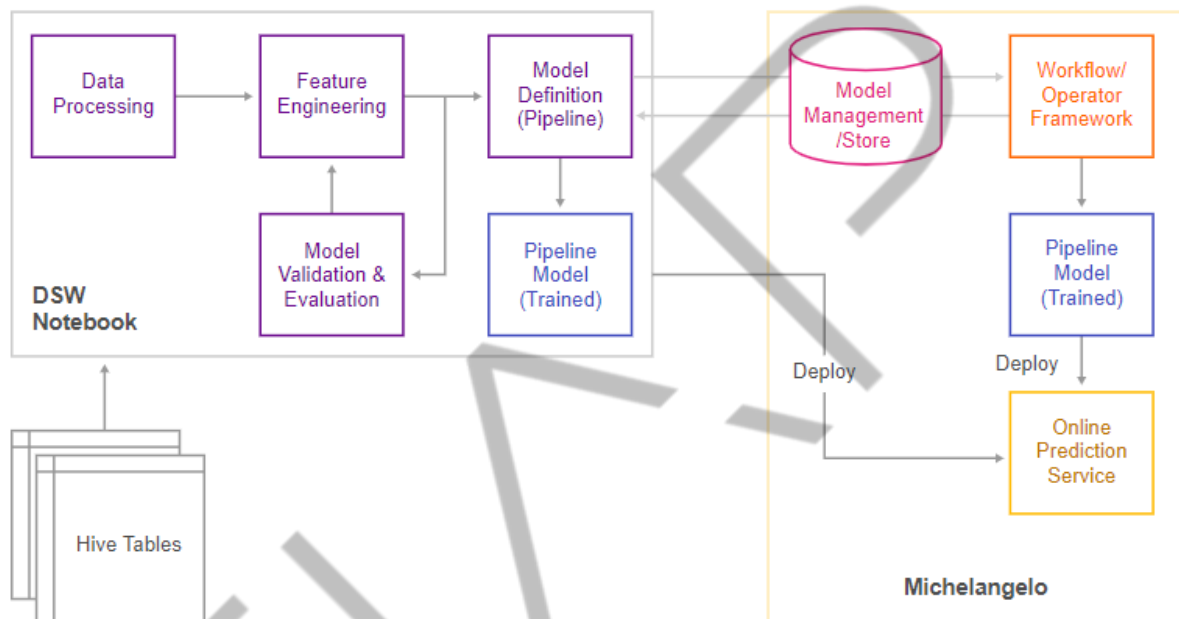


Figura 2.27 - Uber Michelangelo  
Fonte: Holler e Mui (2019)

## REFERÊNCIAS

CATTELL, R. Scalable sql and nosql data stores. **ACM SIGMOD Rec**, v. 39, n. 4, 2011, p. 12-27.

CHELLIAH, I. **Mathematical Set Operations in Python**. 2020. Disponível em: <<https://medium.com/better-programming/mathematical-set-operations-in-python-e065aac07413>>. Acesso em: 05 out 2020.

CINTRA, J. **Sistema IoT para Aquisição de Dados com REDIS e Linguagem GO**. 2019. Disponível em: <<https://josecintra.com/blog/iot-aquisicao-dados-sensores-redis-golang/>>. Acesso em: 16 fev. 2021.

DB-ENGINES. **DB-Engines Website**. 2020. Disponível em: <<https://db-engines.com/en/>>. Acesso em: 20 jul. 2020.

FINK, B. Distributed computation on dynamo-style distributed storage: riak pipe. **Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop**, 2012, p. 43-50.

HAINES, K. **Engine Yard Blog: To Redis or Not To Redis?** 2009. Disponível em: <<https://blog.engineyard.com/2009/key-value-stores-for-ruby-part-4-to-redis-or-not-to-redis>>. Acesso em: 20 jul. 2020.

HOLLER, A.; MUI, M. **Evolving Michelangelo Model Representation for Flexibility at Scale**. 2019. Disponível em: <<https://eng.uber.com/michelangelo-machine-learning-model-representation>>. Acesso em: 15 ago. 2020.

MCCREARY, D.; KELLY, A. **Making Sense of NoSQL: A Guide for Managers and the Rest of Us**. 1.ed. Nova York: Manning Publications, 2014.

REDIS. **Documentation**. 2020. Disponível em: <<https://redis.io/documentation#documentation/>>. Acesso em: 15 ago. 2020.

REDISLABS. Listen to Kyle Davis and Loris Cro Discuss Their New Book. 2020. Disponível em: <<https://redislabs.com/blog/redis-microservices-for-dummies-podcast-kyle-davis-loris-cro-the-new-stackj/>>. Acesso em: 15 out. 2020.

SHIFTEHFAR, R. **Uber's Big Data Platform: 100+ Petabytes with Minute Latency**. 2018. Disponível em: <<https://eng.uber.com/uber-big-data-platform/>>. Acesso em: 15 ago. 2020.

EMEND