UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Luigi Tosin Misturini

Análise e implementação de cache para aplicação web

Florianópolis
2021

Luigi Tosin Misturini

Análise e implementação de cache para aplicação web

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Leandro Buss Becker

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Misturini, Luigi
Análise e implementação de cache para aplicação web / Luigi Misturini ; orientador, Leandro Becker, 2021.
61 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2021.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Sistemas web. 3. Cache. 4. Redis. I. Becker, Leandro. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Luigi Tosin Misturini

Análise e implementação de cache para aplicação web

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, 22 de Fevereiro de 2021.

Prof. Hector Bessa Silveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Leandro Buss Becker, Dr.
Orientador
UFSC/CTC/DAS

William Carvalho Bastos, Bel.
Supervisor
Nexti

Prof. Werner Kraus Junior, Dr.
Avaliador
UFSC/CTC/DAS

Prof. Fabio Luis Baldissera, Dr.
Presidente da Banca
UFSC/CTC/DAS

Este trabalho é dedicado ao meu avô, Odir Tosin(in memorian).

AGRADECIMENTOS

Agradeço aos meu pais e familiares, por sempre me apoiarem nas minhas decisões e estarem sempre presentes em qualquer situação.

Aos meu amigos, por me ajudarem em vários momentos durante a graduação.

Gostaria de agradecer a Nexti por permitir tornar possível todas as atividades desse projeto, agradeço a todos da equipe de desenvolvimento e da equipe de suporte por me ajudarem em todas as atividades, relacionadas ou não a esse projeto, e por criarem um ambiente excelente de trabalhar.

Agradecimentos para Lucas, pelas ideias e ajuda com a decisão do projeto, além de orientações sobre onde aplicar o cache e para William, pelo imenso suporte sobre o Time Tracking.

Agradeço especialmente a minha namorada, Gabrielle Meireles, por sua companhia, confiança e apoio que recebi durante todos esses anos.

RESUMO

Tipicamente os sistemas Web utilizam como forma principal de armazenamento de informações os bancos de dados relacionais. Entretanto, muitas vezes podem ser o causador de um gargalo do sistema, seja pelo fato do banco ser muito grande ou ter muitas requisições para processar. Existem algumas formas de melhorar a performance, uma delas a adição de um cache entre a aplicação e o banco de dados, fornecendo uma forma mais rápida de acesso onde as informações serão armazenadas. Além de melhorar a performance do sistema, tal memória cache irá reduzir a carga sobre o banco de dados principal, permitindo que consultas mais elaboradas não necessitem competir por recursos com consultas triviais. Estratégias de cache podem variar dependendo da aplicação e das necessidades, tendo grande vantagem uma maior velocidade de acesso, por usarem a memória principal do sistema e possuírem algoritmos melhor otimizados. Assim o banco de dados em memória foi a tecnologia escolhida para analisar a viabilidade da utilização e melhorar a performance em um microsserviço de apuração de ponto através do Redis, um armazenamento de estrutura de dados de chave-valor de código aberto na memória com o objetivo de melhorar a performance aliviar a carga do banco de dados principal de uma aplicação web. Posteriormente foi analisado a diferença de performance com a utilização de cache na aplicação e seu impacto no consumo de recursos.

Palavras-chave: Redis. Cache. Banco de dados. NoSQL

ABSTRACT

Typically Web systems use relational databases as the main form of information storage. However, they can often cause a system bottleneck, either because the database is very large or has many requests to process. There are some ways to improve performance, one of which is the addition of a cache between the application and the database, providing a faster way of access where the information will be stored. In addition to improving system performance, such cache memory will reduce the load on the main database, allowing more elaborate queries to avoid competing for resources with trivial queries. Cache strategies can vary depending on the application and needs, with great advantage of a higher access speed, because they use the main memory of the system and have better optimized algorithms. Thus, the in-memory database was the technology chosen to analyze the feasibility of use and improve performance in a time tracking microservice through Redis, an open source key-value data structure storage in memory with the objective of to improve performance and alleviate the load on the main database of a web application. Subsequently, the performance difference with the use of cache in the application and its impact on resource consumption was analyzed.

Keywords: Redis. Cache. Database. NoSQL

LISTA DE FIGURAS

Figura 1 – Comparação de performance IMDB Vs RDBMS 15
Figura 2 – Arquitetura de uma IMDB 20
Figura 3 – Hierarquia de memória 21
Figura 4 – Custo de armazenamento 22
Figura 5 – Redis 23
Figura 6 – Uso de banco de dados SQL e NoSQL 24
Figura 7 – Uso de bancos de dados combinados 25
Figura 8 – Estrutura Spring Data 27
Figura 9 – Consumo de CPU no carregamento do período sem cache 44
Figura 10 – Consumo de memória no carregamento do período sem cache . . . 45
Figura 11 – Consumo de CPU ao salvar o período no cache 46
Figura 12 – Consumo de memória ao salvar o período no cache 46
Figura 13 – Consumo de CPU retornando o período do cache 47
Figura 14 – Consumo de memória retornando o período do cache 47
Figura 15 – Consumo de CPU carregando o perfil sem cache 49
Figura 16 – Consumo de memória carregando o perfil sem cache 49
Figura 17 – Consumo de CPU salvando o perfil no cache 50
Figura 18 – Consumo de memória salvando o perfil no cache 50
Figura 19 – Consumo de CPU retornando o perfil pelo cache 51
Figura 20 – Consumo de memória retornando o perfil pelo cache 51

LISTA DE CÓDIGOS

4.1 Inclusão da dependência do Spring Data Redis 29
4.2 Inclusão do conector Jedis 29
4.3 Implementação do gerador de chaves 31
4.4 Arquivo de configuração do Redis 32
4.5 Configuração da interface do Redis 33
4.6 Configuração da interface do Redis para aceitar qualquer conexão . . . 33
4.7 Desativação do modo protegido no Redis 33
4.8 Uso da anotação @EnableCaching 34
4.9 Uso da anotação @Cacheable 35
4.10 Definição de mais de um cache 35
4.11 Definição de chave usando SpEL 35
4.12 Definição de chaves usando gerador de chaves 35
4.13 Uso da anotação @CachePut 36
4.14 Uso da anotação @CacheEvict 37
4.15 Método para retorno das informações do período 38
4.16 Classe criada para adicionar as anotação para o cache 38
4.17 Repositórios criados para uso das anotações de cache 38
4.18 Repositórios criados para uso das anotações de cache 39
4.19 Repositórios criados para uso das anotações de cache 39
4.20 Repositórios criados para uso das anotações de cache 39
4.21 Repositórios criados para uso das anotações de cache 39
4.22 Método adicionado com a anotação de cache para salvar o período . . 40
4.23 Método adicionado com a anotação de cache para salvar o período . . 40
4.24 Anotação para remoção do período do cache 40
4.25 Anotação para remoção do período do cache 40
4.26 Método que retorna o perfil de apuração 41
4.27 Repositório criado para adição das anotações de cache 41
4.28 Cache nas formas de exportação 41
4.29 Cache nos tipos de eventos de apuração 42
4.30 Método de atualização com anotação 42
4.31 Método de remoção com anotação 42
4.32 Método de remoção com anotação 42
6.1 Método modificado para salvar chave no cache 54
6.2 Método modificado para atualizar chave no cache 54
6.3 Classe utilizada para comunicar com o Redis 55

LISTA DE TABELAS

Tabela 1 – Tempos de carregamentos do período 48
Tabela 2 – Tempos de carregamentos do período 52

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AWS	Amazon Web Services
BSD	Berkeley Software Distribution
CPU	Central Processing Unit
CRUD	Create, Read, Update e Delete
DAO	Data Access Object
EJB	Enterprise JavaBeans
HDD	Hard Disk Drive
IMDB	In-Memory Database
J2EE	Java 2 Enterprise Edition
JPA	Java Persistence API
JVM	Java Virtual Machine
NoSQL	Not Only SQL
POM	Project Object Model
POSIX	Portable Operating System Interface
RAM	Random Acess Memory
RDBMS	Relational Database Management Systems
SpEL	Spring Expression Language
SQS	Simple Queue Service
SSD	Solid State Drive
STS	Spring Tool Suite
TCP	Transmission Control Protocol
VA	Vale Alimentação
VR	Vale Refeição
VT	Vale Transporte
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO . . . 14
1.1 OBJETIVOS . . . 15
1.2 ORGANIZAÇÃO DOS CAPÍTULOS . . . 16
2 A NEXTI . . . 17
2.1 TIME TRACKING . . . 17
2.1.1 Período de apuração . . . 18
2.1.2 Perfil de apuração . . . 18
3 TECNOLOGIAS RELACIONADAS . . . 19
3.1 BANCOS DE DADOS EM MEMÓRIA . . . 19
3.1.1 Redis . . . 20
3.2 NOSQL . . . 23
3.3 SPRING . . . 25
3.3.1 Spring Boot . . . 26
3.3.2 Spring Tools Suite . . . 26
3.3.3 Spring Data . . . 26
3.3.3.1 Spring Data Redis . . . 27
3.4 MAVEN . . . 28
3.5 VISUALVM . . . 28
4 DESENVOLVIMENTO . . . 29
4.1 CONFIGURAÇÃO DO PROJETO . . . 29
4.1.1 Arquivo de configuração . . . 30
4.1.2 Jedis Connection Factory . . . 30
4.1.3 Key Generator . . . 30
4.1.4 Redis Template . . . 31
4.2 CONFIGURAÇÃO DO REDIS . . . 33
4.3 CACHE DE MÉTODOS . . . 34
4.3.1 @EnableCaching . . . 34
4.3.2 @Cacheable . . . 34
4.3.3 @CachePut . . . 36
4.3.4 @CacheEvict . . . 36
4.4 INCLUSÃO DO CACHE NA APLICAÇÃO . . . 37
4.4.1 Período de apuração . . . 37
4.4.2 Perfil de apuração . . . 40
5 ANÁLISE SOBRE O CACHE . . . 44
5.1 ANÁLISE DO PERÍODO . . . 44
5.2 ANÁLISE DO PERFIL . . . 48
6 MONITORAMENTO DA APURAÇÃO COM CACHE . . . 53

7 CONCLUSÃO E TRABALHOS FUTUROS 57
7.1 CONCLUSÃO 57
7.2 TRABALHOS FUTUROS 57
REFERÊNCIAS 59

1 INTRODUÇÃO

Aplicações Web são indispensáveis atualmente, promovendo uma grande facilidade de uso através da internet, porém essas aplicações devem oferecer uma boa experiência de uso, não somente em sua forma de utilização como também na performance.

Ter um sistema lento que obriga o usuário a ficar minutos esperando por algo acontecer não é algo admissível, ainda mais nos tempos atuais em que temos grandes velocidades não só de conexão, como também de processamento disponível, com processadores cada vez mais rápidos e com crescente número de núcleos.

Porém em muitos casos, mesmo com essa disponibilidade de processamento, sistemas cresceram tanto e/ou processam tantos dados que ainda assim tem sua performance comprometida, já que a principal forma de armazenamento de dados é na forma de discos rígidos, ou também conhecidos como Hard Disk Drive (HDD), que tem boas performances para disponibilizar dados de forma sequencial, mas pecam em buscar dados espalhados pelo disco(ANDERSON, 2021).

Outro desses gargalos pode ser atribuído ao banco de dados, tendo tecnologias de Relational Database Management Systems (RDBMS) consolidadas no mercado, como MySQL, POSTGRESQL, Oracle entre outros, mas ainda assim podem sofre quando o tamanho do banco cresce a cada dia e a cada novo usuário acessando o banco para ler ou modificar registros.

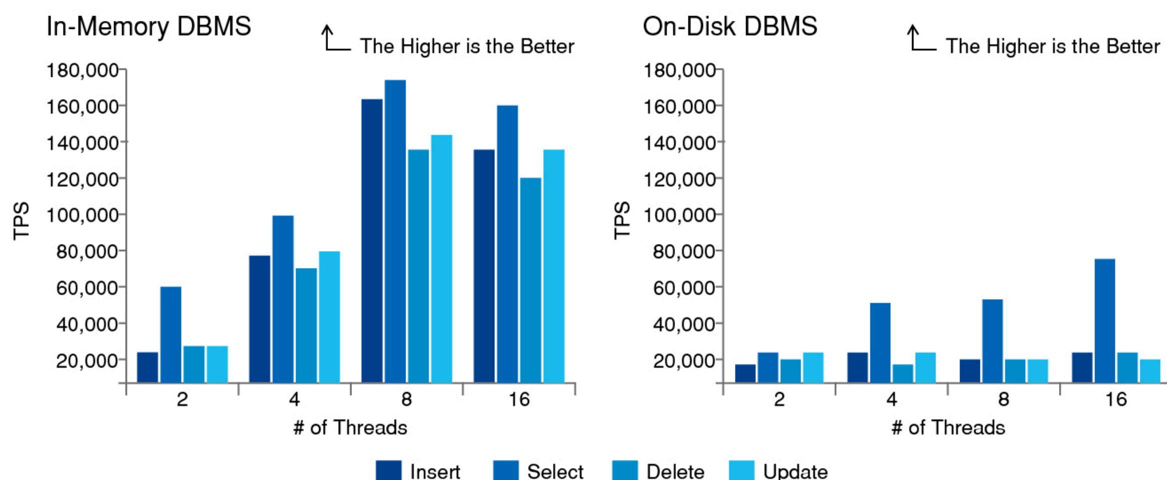
Existem algumas formas de contornar esse problema, uma delas é utilizar uma banco de dados em memória, ou In-Memory Database (IMDB) como cache, armazenando as informações mais comumente usadas para evitar acessos constantes ao banco de dados, além de ter uma velocidade de acesso muito mais rápida do que os bancos tradicionais(JIHAD NAJAJREH, 2017), como consta na figura 1, que armazenam as informações em disco, os bancos em cache utilizam a memória principal do sistema, a Random Acess Memory (RAM).

Além de utilizar uma forma mais eficiente de armazenamento, utilizar uma tecnologia que permite uma melhor escalabilidade de forma horizontal e flexível(STONEBRAKER, 2010), como o Not Only SQL (NoSQL) garante que os problemas enfrentados pelos bancos relacionais não ocorram novamente.

Os bancos de dados NoSQL diferem também em outro ponto, os bancos tradicionais seguem os princípios ACID(Atomic, Consistent, Isolated, and Durable), enquanto que os bancos de dados NoSQL são regidos pelos princípios BASE(Basically Available, Soft State, and Eventually Consistent), que garantem alta disponibilidade de dados, porém sacrificam a consistência desses dados.(DAN PRITCHETT, 2018)(COOK, 2021).

Na empresa Nexti, onde o presente trabalho foi realizado, se verificado a necessidade de aliviar a carga sobre o banco de dados principal, o que está prejudicando em

Figura 1 – Comparação de performance IMDB Vs RDBMS



Fonte – Altibase

alguns pontos a performance do sistema, para isso foi implantado a integração de um dos microsserviços que compõem o sistema Nexti, o Time Tracking, com o Redis, um banco de dados em memória que também opera como cache, que se destaca pela performance comparado a outros IMDB's(ABDULLAH TALHA KABAKUSA, 2017)(JIHAD NAJAJREH, 2017), onde foi analisado a viabilidade do uso dessa tecnologia para aprimorar a performance do sistema.

Para complementar a analise da viabilidade, também foi verificado se essa integração traria impactos negativos para o consumo de recursos na aplicação.

1.1 OBJETIVOS

O objetivo dessa monografia é analisar a viabilidade de utilizar um IMDB para melhorar o desempenho do sistema Nexti, mais especificamente em um dos seus microsserviços de apuração de ponto, o Time Tracking, através da implantação da integração desse microsserviço com o Redis e analisar o impacto que essa tecnologia trouxe para o sistema, tanto em tempos de carregamento de informações, como em consumo de recursos.

Serão analisados os *endpoints* da Application Programming Interface (API) usada pelo sistema para verificar quais são passíveis de melhora e armazenar as informações retornadas na base de cache.

Além de tentar melhorar a performance do sistema, será feita uma implementação em caráter experimental para verificar a possibilidade de monitorar a apuração de ponto em lote, tendo a possibilidade de, em caso de uma queda do sistema, retomar do ponto onde foi interrompido, além de informar ao usuário o avanço da apuração e

sua conclusão.

1.2 ORGANIZAÇÃO DOS CAPÍTULOS

A seguinte monografia está divida em 7 capítulos, sendo eles:

O segundo capítulo será uma breve explicação sobre a empresa Nexti, sua história e sobre o sistema de mesmo nome, onde serão apresentados alguns detalhes e a apresentação do microsserviço que será foco no desenvolvimento junto com detalhes sobre as informações gerenciados por ele.

No capítulo 3 serão apresentadas todas as tecnologias relacionadas ao desenvolvimento feito, brevemente explicado na introdução.

O desenvolvimento efetivo será exposto no capítulo 4, onde todas as análises, implementações ou modificações de código serão mostradas em detalhes.

No quinto capitulo será analisado os resultados obtidos com a implementação do banco de dados em memória atuando como cache, mostrando o efeito sobre o tempos de carregamento e verificando quais impactos foram sentidos, sejam positivos ou negativos, no consumo de recursos pela aplicação quando utilizando o cache.

Uma implementação em caráter experimental será exposta no capítulo 6 para verificar se é possível utilizar o Redis para realizar o acompanhamento da apuração em lote do ponto do colaboradores.

Por fim a conclusão será apresentada no capítulo 7, junto com quais serão os trabalhos futuros que foram identificados após a finalização da implementação.

2 A NEXTI

A Nexti nasceu de uma necessidade vinda da Orsegups, em 2011 já existiam sistemas de controle de entrada e saída de viaturas, tendo um bom controle sobre a viaturas foi pensado em aplicar a mesma ideia para o controle de vigilantes, surgindo assim o sistema Presença

Em 2013 foi feita a primeira implantação em umas das regionais da Orsegups, em Lajes, porém o Presença era desenvolvido dentro de um sistema de terceiros, assim em 2014 foi feita uma apresentação para um futuro cliente, onde a Orsegups percebeu que havia uma necessidade desse mercado

A partir disso foi desenvolvido o Minha Presença, que mais tarde, em 2017, se tornaria o Nexti, sistema de mesmo nome da empresa.

O sistema Nexti inicialmente consistia apenas em um controle de presença dos colaboradores, com o tempo foi evoluindo permitindo a possibilidade de efetuar a cobertura de um colaborador que faltou por outro disponível, em seguida foram adicionadas as marcações de ponto e verificação de pontos inconsistentes.

Com o tempo novos módulos foram sendo adicionados, permitindo que os clientes gerem relatórios e controlem a geração de benefícios, como Vale Alimentação (VA)/Vale Refeição (VR)/Vale Transporte (VT).

Em sua maioria, o sistema Nexti é construído com a linguagem Java, sendo utilizado a arquitetura de microsserviços e tendo como *framework* utilizado o Spring.

Como o sistema é divido em microsserviços foi escolhido um deles para ser a implementação inicial do Redis, sendo posteriormente expandido para outros caso seja benéfica a integração com o cache.

O módulo de apuração, chamado de *Time Tracking*, um dos microsserviços que compõem o sistema Nexti, surgiu a partir da necessidade de um dos primeiros clientes do Presença.

Porém só seria idealizado no final de 2017, onde foi iniciado o desenvolvimento efetivo do *Time Tracking*, sendo utilizado pelo primeiro cliente no inicio de 2018, tendo seu desenvolvimento ativo até o momento.

Será o microsserviço escolhido para ser feita a implementação da integração com o Redis para analisar a viabilidade do utilização da base de cache.

2.1 TIME TRACKING

Como explicado anteriormente, um dos microsserviços que compõem o sistema Nexti é o *Time Tracking*, responsável, primariamente, por realizar a apuração das horas trabalhadas dos colaboradores.

Muitas das informações utilizadas durante a apuração são gerenciadas por outros microsserviços, sendo o *Time Tracking* responsável apenas por processar esses

insumos, mas para isso depende de duas informações muito importantes que são exclusivas do *Time Tracking*, sendo elas definidas como Período e Perfil de apuração.

2.1.1 Período de apuração

Consiste em parametrizar um intervalo de tempo em que será feita uma apuração, geralmente consiste em um mês, mas pode ser definido qualquer intervalo.

É possível definir um período global, que irá abranger todos os colaboradores, mas também pode ser criado um período de exceção, onde será definido um forma de filtrar alguns colaboradores com base em critérios pré-definidos, como por exemplo, apenas os colaboradores que estejam em um determinado horário serão contemplados nesse período de exceção.

É no período também que é feita a apuração em lote, ponto principal do *Time Tracking*, onde todas as informações de registro de ponto, trocas de escalas/posto/horários, coberturas e ausências são convertidas em eventos com seus respectivos valores em horas.

A partir do período também é possível exportar os insumos para integração com algum software para gerar a folha de pagamento, a partir de um layout pré definido, onde os eventos de interesse do cliente pode ser escolhidos e atribuídos a um código específico.

2.1.2 Perfil de apuração

Aqui é onde se concentram a maior parte das parametrizações sobre a apuração, a partir de diversas *flag's* disponibilizadas o usuário consegue informar como deve ser feita a apuração, quais eventos devem ser gerados, como devem ser gerados além de outras informações sobre como as informações apuradas devem ser tratadas.

O perfil de apuração também pode ser separado em global e exceção, porém aqui existe a possibilidade de ter exceções primárias e secundárias, as exceções são as mesmas do período e seguem as mesma lógica, mas com a opção exceção secundária possibilita filtragem mais detalhada.

Assim é possível definir um perfil que contemple todos os colaboradores que estão lotados em um posto e que fazem parte de um sindicato, por exemplo.

3 TECNOLOGIAS RELACIONADAS

A seguir serão apresentadas todas as pesquisas feitas sobre as tecnologias que serão utilizadas no projeto a fim de contextualizar os termos e tecnologias abordadas nos capítulos seguintes.

3.1 BANCOS DE DADOS EM MEMÓRIA

Bancos de dados são a principal forma de armazenamento de informações atualmente, essas informações são salvas no HDD, sendo uma forma de armazenamento não volátil e a principal forma de armazenar dados em massa.

Porém em sistemas críticos, como o de telecomunicações é necessário que o acesso a essas informação seja feito de forma extremamente rápida e mesmo com a migração dos HDD’s convencionais, cujo acesso aos dados se faz de forma mecânica, com discos magnéticos e uma cabeça de leitura, para os discos de estado sólidos, os Solid State Drive (SSD)’s, onde não existem partes mecânicos, apenas módulos de memória Flash, ainda não conseguem suprir essa disponibilidade(DEVOPEDIA, 2021).

Aqui entram os bancos de dados em memória RAM, onde ao se utilizar uma forma de armazenamento volátil, porém com velocidades de acesso muito mais rápida, é possível ter um sistema muito responsivo e performático(VERONIKA ABRAMOVA JORGE BERNARDINO, 2014), um exemplo de arquitetura de um IMDB é mostrado na figura 2.

O acesso a memória RAM ocorre na ordem dos nanosegundos, enquanto que o *seektime* de um HDD ocorre na ordem dos milissegundos(JIHAD NAJAJREH, 2017), portanto a memória RAM tem uma velocidade de repostas em torno de 1 milhão de vezes mais rápida, a transferência de dados não segue essa mesma regra, mas tendo um tempo de resposta tão mais rápido já mostra um ganho muito expressivo.

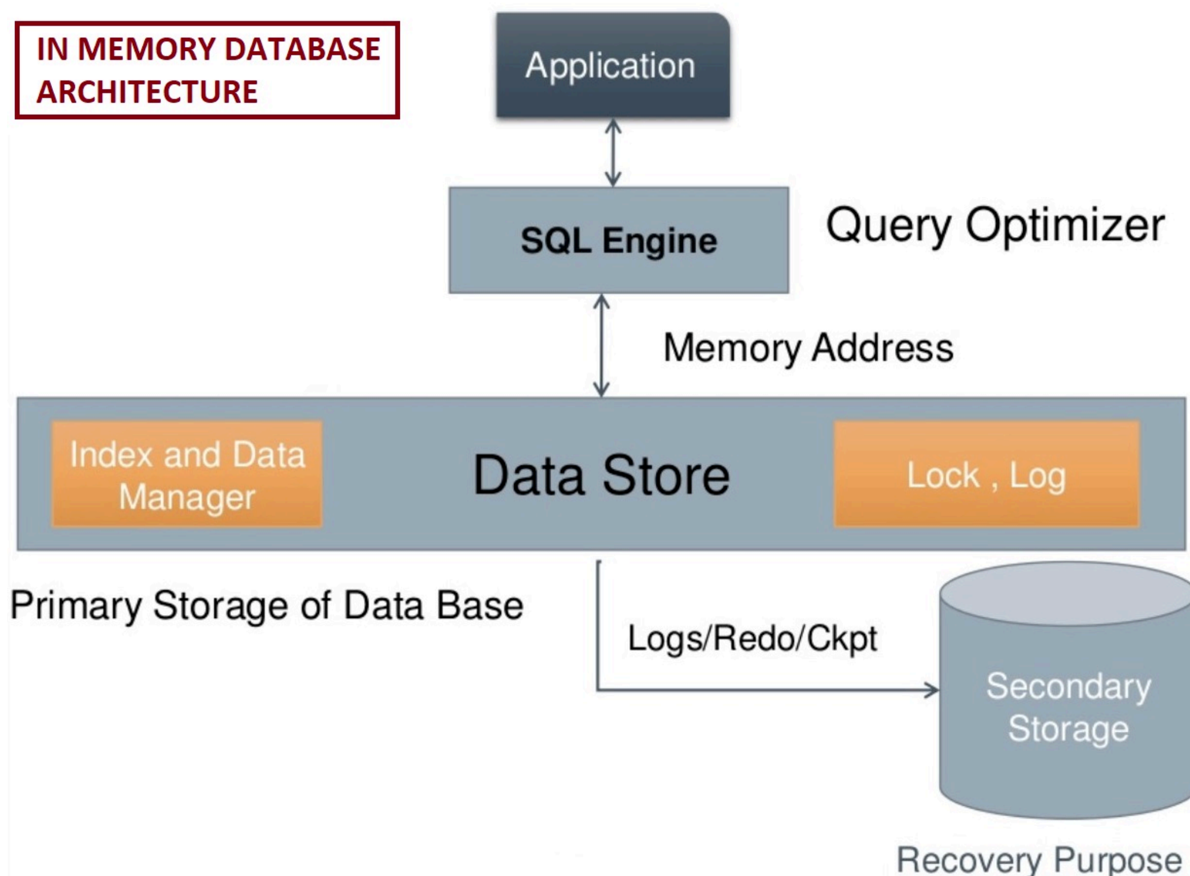
A figura 3 demonstra a diferença entre velocidades e custo dos dispositivos de armazenamento com relação a memória principal.

Com o aumento da quantidade de memória disponível, além da diminuição do custo por Gigabyte da memória RAM, demostrado pelo grafico da figura 4, agora é possível ver Terabytes de memória disponível em grandes servidores permitindo assim que não só algumas informações sejam mantidas em cache, mas também que bancos de dados sejam totalmente disponibilizados em memória.

Porém, os banco de dados tradicionais ainda não podem ser totalmente abandonados, já que a memória RAM é volátil, em caso de falha que ocorra corte de alimentação de energia para a memória, todas as informações são perdidas, sendo assim o armazenamento em disco ainda de muita importância.

Para esse projeto foi utilizado o Redis como banco de dados em memória e cache.

Figura 2 – Arquitetura de uma IMDB



Fonte – Kodamasimham

3.1.1 Redis

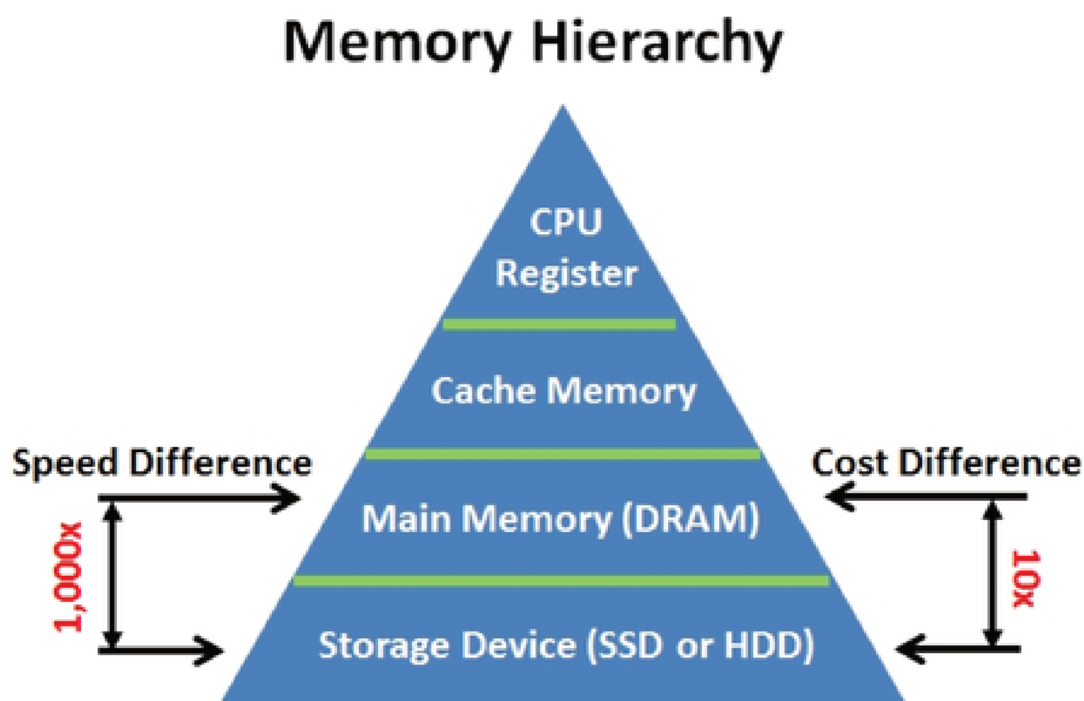
Acrônimo de *REmote DIctionary Server*(REDIS, 2020a) é um armazenamento NoSQL de estruturas de dados chave-valor em memória ,que foi criado por Salvatore Sanfiippo, quando buscava melhorar a escalabilidade de sua *startup* italiana, e liberado de forma *open source* em 2009 sobre a licença Berkeley Software Distribution (BSD).

Usado como base de dados, cache, demonstado na figura 5, e *message broker*. o Redis é altamente replicável(REDIS, 2021b) e extremamente performático(REDIS, 2021a).

Escrito em código C e compatível com mais de 30 linguagens de programação faz do Redis o mais popular armazenador de estruturas de chave-valor, além de suportar essa estrutura de chave-valor, o Redis também tem suporte a vários outros tipo de dados mais complexos, como os seguintes:

- Strings
- Hashes

Figura 3 – Hierarquia de memória



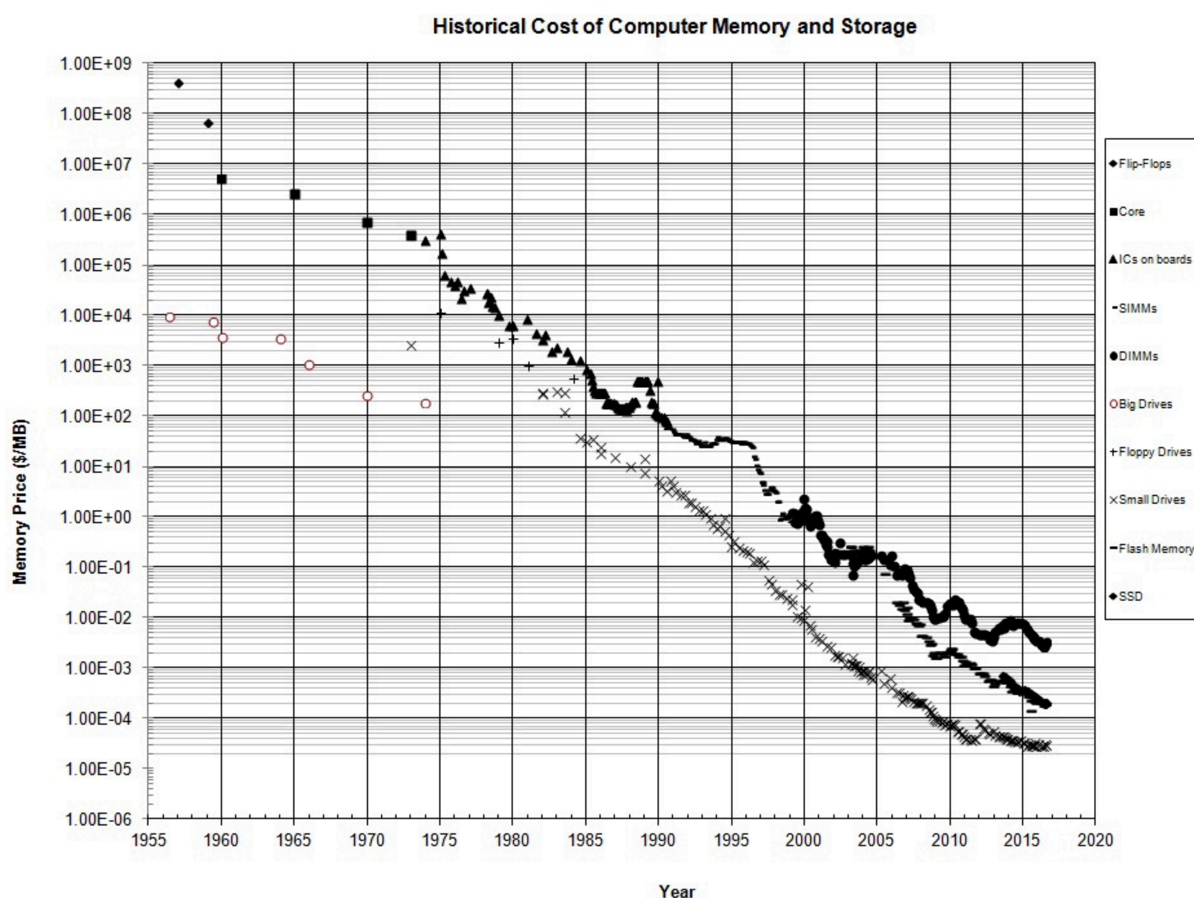
Fonte – BeSang Inc.

- Listas
- Sets
- Bitmaps
- Hyperloglogs
- Dados geoespaciais
- Streams

O Redis possui suporte a maioria dos sistemas compatíveis com o padrão Portable Operating System Interface (POSIX), como Linux, porém não possui uma *build* oficial para Windows.

Apesar de ser uma base de dados em memória, é possível que os dados sejam salvos em disco, quando necessário, outra característica importante é que todos os comandos são executados de forma atômica, devido a natureza *single-threaded* do Redis.

Figura 4 – Custo de armazenamento



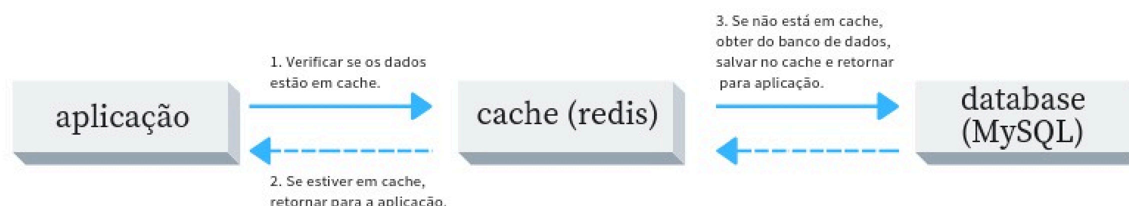
Fonte – J.C. McCallum

O funcionamento do Redis é baseado no modelo cliente-servidor, sendo um servidor Transmission Control Protocol (TCP), recebendo comandos de um cliente por uma conexão via *socket*(DAKAR, 2015).

Alguns dos casos de uso incluem:

- Armazenamento em cache
- Gerenciamento de sessões
- Classificação em tempo real
- Limite de taxa
- Filas
- Chat e sistemas de mensagens
- Placares de jogos

Figura 5 – Redis



Fonte – https://www.treinaweb.com.br

O Redis ainda emprega uma arquitetura de replicação principal e oferece suporte à replicação assíncrona, permitindo que os dados seja replicados em vários servidores melhorando a performance de leitura, distribuindo as solicitações entre os vários servidores e permite a recuperação quando o principal servidor sofre uma queda.

Mesmo sendo um banco em memória o Redis permite que os dados sejam salvos no disco, com backups *point in time*, o que permite a restauração dos dados, possivelmente sem muitas perdas, já que as operações que modificam algum tipo de dados corresponde a 10%-15%(DEVOPEDIA, 2021) das transações feitas no banco de dados.

3.2 NOSQL

Com a evolução da internet muitas novas formas de dados foram surgindo e com isso, tratar esses dados começou a se tornar uma tarefa complexa e custosa, assim em 1998 surgiu o termo NoSQL, utilizado para definir um banco de dados de código aberto e que não utilizasse o modelo relacional, até então padrão entre os bancos de dados.

O termo NoSQL surgiu em 1998, mas só foi popularizado em 2006 quando o Google citou o termo em um artigo(FAYCHANG, 2006) em uma época que os bancos de dados relacionais não suportavam mais a carga de dados gerada pela internet.

A ideia do NoSQL é se diferenciar dos bancos de dados relacionais, não sendo necessários que os bancos de dados NoSQL seja semelhantes, são denominados assim apenas por se diferenciarem dos bancos tradicionais e serem a resposta para as limitações dos RDBMS.

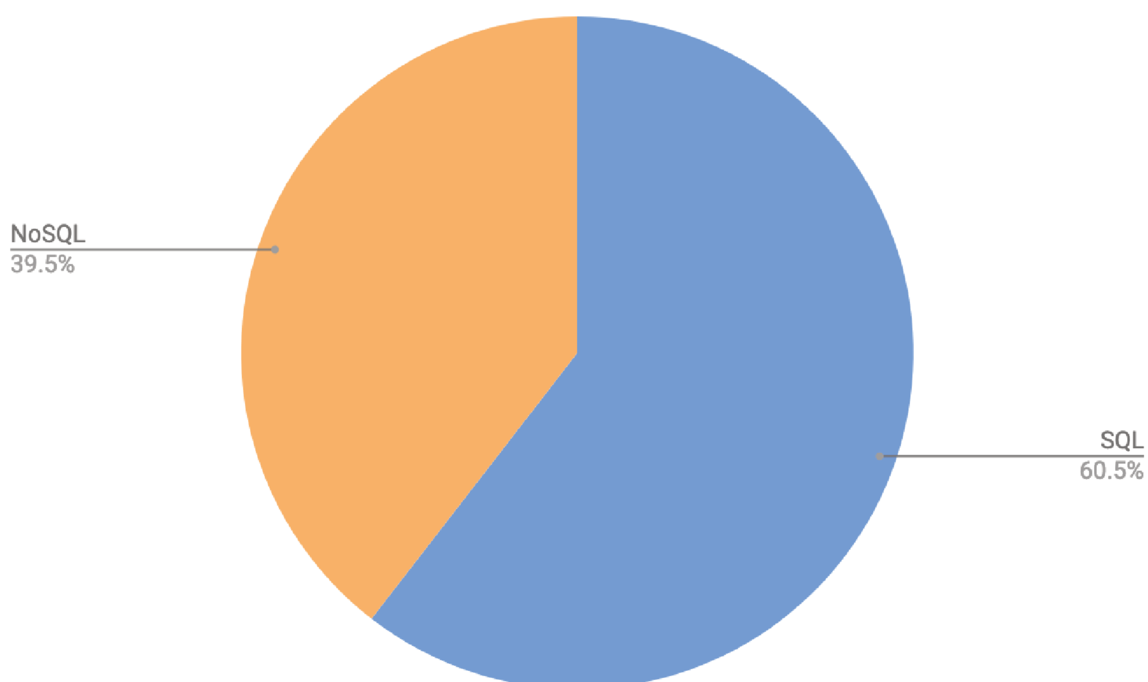
Bancos de dados não-relacionais podem suportar diversos tipos de dados estruturados, semi-estruturados ou até mesmo desestruturados e ainda dados híbridos tendo custo reduzido, baixa complexidade e uma ótima performance(ALSHAFIE GAFAAR MHMOUD MOHMMED, 2017).

As necessidades de lojas onlines, redes sociais, comunicação e serviços oferecidos pela web que fizeram surgir os requisitos hoje supridos pelos bancos NoSQL, entre essas necessidades estão:

- Escalabilidade
- Alta disponibilidade
- Utilização e alocação otimizada de recursos
- Capacidade de armazenamento virtualmente ilimitada
- Hospedagem compartilhada

Mesmo tendo muitas vantagens sobre os RDBMS, os bancos de dados NoSQL ainda não dominam totalmente o mercado, como pode ser visto na figura 6, principalmente por serem uma tecnologia que teve sua popularidade ainda recentemente.

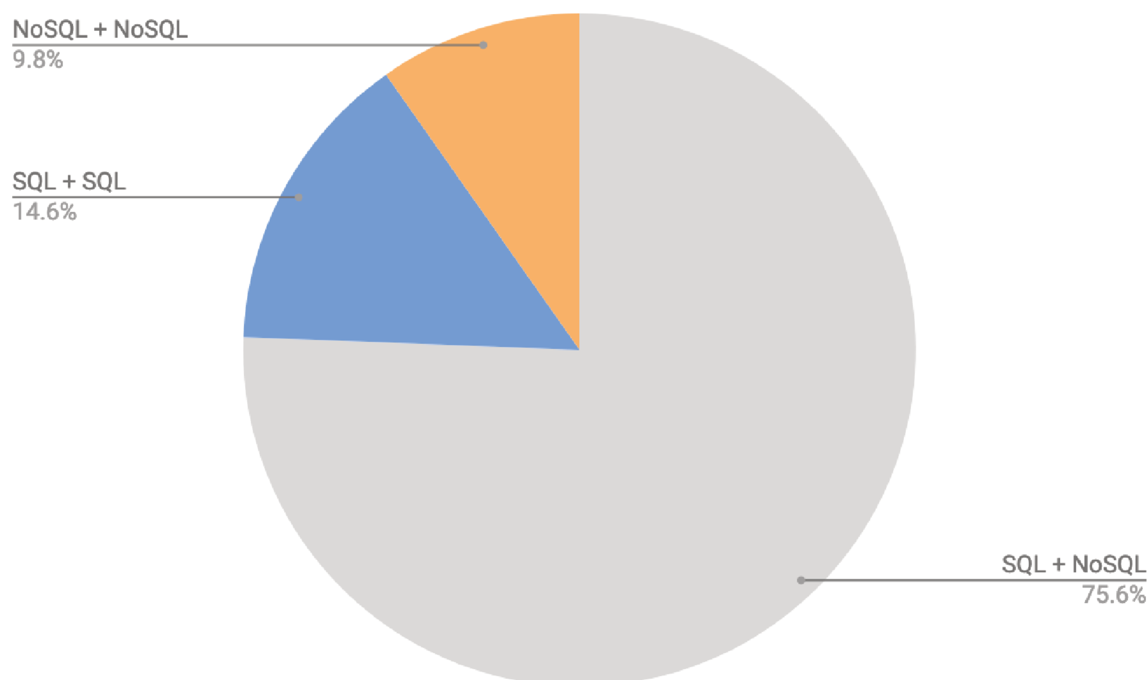
Figura 6 – Uso de banco de dados SQL e NoSQL



Fonte – http://highscalability.com

Muitas empresas estão migrando de de bases de dados legadas, como Oracle, mas ainda não estão totalmente abandonando os RDBMS tradicionais mas sim combinar os dois tipos de banco de dados para ter as vantagens das duas tecnologias, como mostrado na figura 7

Figura 7 – Uso de bancos de dados combinados



Fonte – http://highscalability.com

3.3 SPRING

Spring é um *framework* Java visando facilitar o desenvolvimento de aplicações, utilizando conceitos de inversão de controle e injeção de dependência, criado por causa de dificuldade de desenvolvimento de aplicações corporativas utilizando Java 2 Enterprise Edition (J2EE), que ainda era uma plataforma em desenvolvimento, e possuía ótimas ideias para o desenvolvimento de aplicações leves e distribuídas, porém com limitações que acabavam tornando a aplicação pesada e que continha muito mais do que o necessário.

O Spring se diferenciou por não depender de um servidor de execução, tendo apenas a JVM como base, e tendo a simplicidade como ponto principal, proporcionando utilizar apenas aquilo que é realmente necessário para o projeto, removendo muitos comportamentos obrigatório do J2EE e Enterprise JavaBeans (EJB)'s, tornando a arquitetura leve e escalável.

Tendo os conceitos de inversão de controle e injeção de dependências delegados a um *container*, que representa o núcleo do *framework*, responsável por criar e gerenciar os componentes da aplicação.

3.3.1 Spring Boot

O Spring boot é um projeto que faz parte do ecossistema do Spring que ajuda a criar aplicações *standalone*, removendo a complexidade da configuração do projeto, a partir de dependências já pré definidas.

Escolhendo módulos através de *starters*, que são dependências que agrupam outras dependências, são incluídos no POM.XML do projeto.

Mesmo tendo essa dependências agrupadas, o que deixa o arquivo mais organizado, não impede o usuário de fazer as suas próprias customizações.

Removendo a necessidade de se preocupar com as dependências do projeto, o Spring Boot permite ao desenvolvedor dedicar mais tempo a elaboração das regras de negócio e desenvolvimento do projeto.

O Spring Boot ajuda a simplificar a forma de inicialização de aplicações Java Web também, considerando o fluxo normal de inicialização:

Empacotar a aplicação → Escolher e baixar um *webserver* → Configurar o *webserver* → *deploy* da aplicação e inicialização do *webserver*

O processo com o Spring Boot se torna:

Empacotar a aplicação → Executar com um simples comando

O Spring Boot se encarrega do resto inicializando e configurando um *webserver* embarcado e faz o *deploy* da aplicação nesse *webserver*.

3.3.2 Spring Tools Suite

O Spring Tool Suite (STS) é uma IDE baseada no Eclipse, usada para desenvolvimento de aplicações Spring, também pode ser instalado como um *plugin* em uma instalação do Eclipse JEE.

A versão *standalone* já vem com contem todos os módulos do Java EE, assim todas *features* do Eclipse Java EE também estão disponíveis no STS.

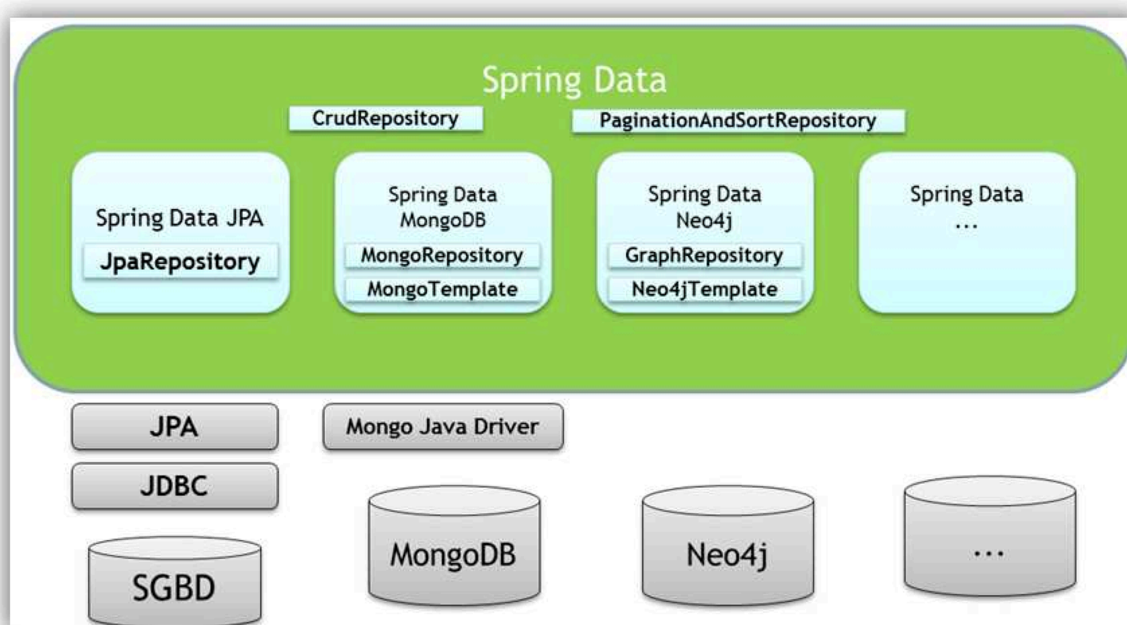
3.3.3 Spring Data

Com o objetivo de facilitar e unificar o acesso diferentes tecnologias de armazenamento de dados, o Spring Data é um projeto que faz parte do ecossistema do Spring(TRELLE, 2013).

Geralmente temos Data Access Object (DAO)'s específicos para um tipo de objeto com operações de Create, Read, Update e Delete (CRUD), já o Spring Data disponibiliza interfaces genéricas para esses aspectos incluindo as especificidades de cada banco de dados.

A figura 8 mostra de forma geral a arquitetura do Spring Data.

Figura 8 – Estrutura Spring Data



Fonte – https://www.igti.com.br

No projeto é usado o Java Persistence API (JPA), através do EclipseLink, para mapeamento objeto-relacional, o que facilita a implementação do Spring Data, já que todas as anotações da API do JPA são reutilizadas pela implementação do Spring Data.

3.3.3.1 Spring Data Redis

Para facilitar a comunicação entre a aplicação e o banco de dados é utilizado o pacote Spring Data Redis, que oferece fácil configuração e acesso ao Redis a partir de uma aplicação Spring.

Oferecendo tanto abstrações de alto e baixo nível para interagir com a base de dados livra o usuário de preocupações com a infraestrutura.

3.4 MAVEN

O Maven é uma ferramento desenvolvida pela Apache Software Foundation desde 2003, desenvolvida para automatizar a construção de aplicações desenvolvidas em Java(ROBINSON, 2021).

Por ser uma ferramente extremamente popular é bastante estável e possui diversas funcionalidades e *plugins*.

Para utilizar o Maven cada projeto terá seu arquivo Project Object Model (POM), que nada mais é que um arquivo eXtensible Markup Language (XML) que contem as informações do projeto, como o nome do projeto, versão, tipo de empacotamento, dependências, *plugins*, etc.

3.5 VISUALVM

O VisualVM é umconjunto de ferramentas que permite a visualização de dados detalhados sobre aplicações Java enquanto estão sendo executadas na Java Virtual Machine (JVM), muitas das ferramentas antes usadas separadamente agora estão unificadas no VisualVm, como o JConsole, jstat e jmap permitindo ver informações de diversas aplicações Java de forma uniforme.

É possível ainda adicionar *plugins* para expandir o numero de ferramentas para análise, sendo um desses *plugins*, Tracer, utilizado para gerar os gráficos.

Será usado essa ferramenta para analisar o consumo de recursos da JVM antes e depois da adição do cache para saber se além de um ganho em performance em tempo de carregamento, a quantidade de recursos tem modificação.

4 DESENVOLVIMENTO

Nesse capítulo será exposto o desenvolvimento realizado para integrar o micros-serviço do Time Tracking com o Redis, em um escopo geral foi feita a configuração do projeto do Time Tracking para incluir as dependências do Spring Data Redis, que permite que uma aplicação Spring se conecte com o Redis, mas essa conexão depende de um conector, que também foi adicionado nas dependências.

Após as inclusões da dependências foi feita a configuração das propriedades de conexão com o Redis, através de um arquivo de configuração criado, onde são definidos os parâmetros usados para a conexão entre a aplicação e o cache, além desse arquivo de configuração foi desenvolvido também uma nova classe que é a responsável por gerenciar as criações de chaves usadas no cache.

Por fim foi feita a inclusão das anotações de cache nos métodos já implementados no Time Tracking, a seguir será feita a demonstração dos passos realizados de forma mais detalhada contendo as implementações e configurações realizadas.

4.1 CONFIGURAÇÃO DO PROJETO

O primeiro passo para incluir o suporte ao Redis no projeto é incluir a dependência do Spring Data Redis, essa inclusão é feita através de uma nova entrada no aquivo POM.XML do projeto, como é mostrado abaixo.

```
1 <dependency>
2     <groupId>org.springframework.data</groupId>
3   <artifactId>spring-data-redis</artifactId>
4 </dependency>
```

Trecho de código 4.1 – Inclusão da dependência do Spring Data Redis

Com essa nova entrada no arquivo de dependências a ferramenta Maven realiza o download da dependência automaticamente e é disponibilizado no *buildpath* da aplicação para uso.

Em seguida é necessário incluir a dependência de algum cliente para fazer a conexão com o Redis, existem vários clientes (REDIS, 2020b) disponíveis, desde clientes leves e eficientes até mais robustos e *thread safe*, para o projeto foi escolhido o Jedis, por ser leve e simples de usar.

Novamente é feito uma nova entrada no aqruivo POM.XML do projeto, assim permitindo o acesso a dependências do Jedis para ser usada.

```
1 <dependency>
2   <groupId>redis.clients</groupId>
3   <artifactId>jedis</artifactId>
4 </dependency>
```

Trecho de código 4.2 – Inclusão do conector Jedis

4.1.1 Arquivo de configuração

Para utilizar o Redis na aplicação é necessário configurar o cliente de conexão, no caso o Jedis, foi criado um arquivo de configuração chamado *RedisConfig.java*, onde a classe *RedisConfig* é definida, sendo ela uma implementação da classe *CachingConfigSupport*, dentro dessa classe alguns métodos foram criados a fim de definir as propriedades de conexão e também métodos auxiliares usados nas interações com o Redis, a seguir será feita uma breve explicação sobre esses métodos desenvolvidos e seu propósito.

4.1.2 Jedis Connection Factory

Esse método é necessário pois é onde são feitas as configurações de conexão com o cache, para gerenciar as conexões com a base de dados é criado um *factory*, por padrão é utilizado as informações padrões de conexão, com o host sendo localhost, porta 6379 e sem uso de senha, mas essa configurações são facilmente trocadas utilizando os respectivos métodos para troca dessas informações no arquivo de configuração.

Como o desenvolvimento para analise será feito totalmente em ambiente local, não será configurado senha, o que não deve ser replicado quando for migrado para ambiente de produção.

4.1.3 Key Generator

Como o armazenamento no Redis é do tipo chave valor, é necessário ter uma forma eficiente de gerar chaves sem que ocorram conflitos de chaves para valores diferentes, por padrão, quando não é definido um *Key Generator* é utilizado o nome e os parâmetros do método invocado para construir uma chave para identificar o valor que será salvo no Redis.

Para ter mais controle das chaves geradas uma nova classe foi implementada e adicionada ao projeto, a implementação é mostrada no trecho de código XX.

A implementação desse gerador de chaves espera receber o nome do método e os parâmetros e o objeto alvo do método, com isso é possível criar uma chave personalizada com base no tipo de dados que está sendo retornado, assim desconsiderando parâmetros que possam vir a causar conflitos com outras chaves ou que não agregam para a exclusividade da chave ou que não fazem sentido serem utilizados como chave, como a senha de login de um usuário.

```
1  public class CustomKeyGenerator implements KeyGenerator {
2
3    private static final Object EMPTY = SimpleKey.EMPTY;
4
5    @Override
6    public Object generate(Object target, Method method, Object... params)
       {
7      if (params.length > 0) {
8        return StringUtils.arrayToDelimitedString(params, "_");
9      }
10     if(params.length == 0) {
11       return method.getName();
12     }
13     return CustomKeyGenerator.EMPTY;
14   }
15 }
```

Trecho de código 4.3 – Implementação do gerador de chaves

Essa classe é utilizada no método *keyGenerator()*, do arquivo de configuração do Redis, para fornecer o gerador de chaves padrão a ser utilizado na criação das chaves utilizadas no cache.

4.1.4 Redis Template

A principal classe para interação de dados com o Redis é o *Redis Template*, responsável por realizar a serialização e a desserialização entre os objetos e os dados binários armazenados no Redis(RAI, 2018).

Por padrão é utilizado o serializador do próprio Java, mas é possível utilizar outros serializadores.

RedisTemplate suporta diferentes métodos para operações com dados:

- ValueOpeartions<K,V> opsForValue() retorna ValueOperations, que realiza operações em valores simples.
- ListOpeartions<K,V> opsForList() retorna ListOperations, que realiza operações em listas.
- SetOpeartions<K,V> opsForSet() retorna SetOperations, que realiza operações em sets.
- HashOpeartions<K,V> opsForHash() retorna HashOperations, que realiza operações em valores hash.

Para operações que exigem uma alta interação com *string's* é recomendado utilizar StringRedisTemplate, que é otimizado para lidar com elas.

O Redis Template é uma classe própria do *core* do Spring, não foi feita implementação dela e está apenas exposta para conhecimento de como é feito o acesso a dados do Redis através dela pelo Spring.

A seguir é mostrada a classe completa de configuração criada, onde os métodos acima explicados são todos utilizados.

```
1  EnableRedisRepositories
2  @Configuration
3  public class RedisConfig extends CachingConfigurerSupport {
4
5    @Bean
6    public JedisConnectionFactory jedisConnectionFactory() {
7      JedisConnectionFactory jedisConnectionFactory = new
     JedisConnectionFactory();
8      jedisConnectionFactory.setHostName("localhost");
9      jedisConnectionFactory.setPort(6379);
10     return jedisConnectionFactory;
11   }
12
13   @Override
14   @Bean("customKeyGenerator")
15   public KeyGenerator keyGenerator() {
16     return new CustomKeyGenerator();
17   }
18
19   @Bean
20   public RedisTemplate<?, ?> redisTemplate() {
21     RedisTemplate<?, ?> redisTemplate = new RedisTemplate<>();
22     redisTemplate.setConnectionFactory(jedisConnectionFactory());
23     redisTemplate.afterPropertiesSet();
24     return redisTemplate;
25   }
26
27   @Override
28   @Bean
29   public CacheManager cacheManager() {
30         return new RedisCacheManager(redisTemplate());
31     }
```

Trecho de código 4.4 – Arquivo de configuração do Redis

O método *cacheManager()* utiliza uma implementação específica do Redis, por isso não é feito seu detalhamento.

4.2 CONFIGURAÇÃO DO REDIS

Como o Redis não possui uma versão oficial para Windows, a plataforma escolhida para uso do Redis no desenvolvimento foi o Ubuntu 20.04, após a instalação foi necessário fazer alguma modificações nos arquivos de configuração, que serão mostradas a seguir.

O Redis deve ser acessado apenas por clientes confiáveis dentro da rede(REDIS. . . , 2021), o que significa que não é recomendado deixar o Redis exposto diretamente á internet ou aberto a conexões de clientes não confiáveis.

Para isso deve ser atrelado ao Redis apenas uma interface, sendo definida no arquivo de configuração redis.conf da seguinte maneira.

```
1 bind 127.0.0.1
```

Trecho de código 4.5 – Configuração da interface do Redis

Com isso o Redis irá apenas responder as requisições vindas da interface de *loopback*, porem para facilitar o desenvolvimento o Redis será atrelado a qualquer interface, fazendo a mudança para

```
1 bind 0.0.0.0
```

Trecho de código 4.6 – Configuração da interface do Redis para aceitar qualquer conexão

Essa mudança deve ser feita com cautela, por ser executado dentro de uma rede local confiável, sem acesso direto á internet e sendo executado localmente no mesmo computador de desenvolvimento essa configuração pode ser feito, em outras situações pode ser extremamente arriscado e irá deixar o Redis totalmente desprotegido.

Já pensando nisso existe mais uma camada de proteção no Redis, quando uma instância do Redis é executada com as configurações padrões e sem uma definição de senha, ele entra automaticamente em modo de proteção, ou *protected mode*, onde o Redis irá apenas responder as requisições vindas da interface de *loopback*, enviando uma mensagem de erro para todas as outras tentativas de conexões.

Novamente, por executar em ambiente seguro, será desabilitada essa opção, sendo feita a seguinte modificação

```
1 protected-mode no
```

Trecho de código 4.7 – Desativação do modo protegido no Redis

Com essas modificações no arquivo de configuração é simplificado o acesso ao Redis para desenvolvimento.

4.3 CACHE DE MÉTODOS

Para fazer o cache serão usadas algumas anotações acima da assinatura do métodos já existentes no Time Tracking, as anotações surgiram na versão 5 do Java(ORACLE, 2021) e são usadas para fornecer *meta data*(JENKOV, 2021) para o código Java, não afetando a execução do código diretamente, mas algumas serão usadas para esse propósito, como as anotações de cache, fazendo com que o método em que a anotação foi aplicada não seja executado e sim retornando as informações salvas no cache.

Anotações ainda podem ter elementos relacionadas a elas, sendo como parâmetros ou atributos, algumas anotações de cache irão utilizar esses elementos para definir informações, como o nome do cache, a chave utilizada ou um gerador de chaves encarregado de realizar a criação dessa chave.

Nas seções a seguir são demonstrados quais anotações são utilizadas, seu funcionamento e como devem ser configuradas.

4.3.1 @EnableCaching

Essa anotação é a primeira a ser adicionada ao projeto e deve ser incluída na classe principal do projeto, para habilitar o processamento das anotações de cache pelo Spring.

```
1 @SpringBootApplication
2 @EnableCaching
3 public class CachingApplication {
4
5   public static void main(String[] args) {
6     SpringApplication.run(CachingApplication.class, args);
7   }
8 }
```

Trecho de código 4.8 – Uso da anotação @EnableCaching

Quando essa anotação for incluída irá ativar um pós processamento que irá inspecionar cada bean Spring em busca de anotações de cache em métodos públicos, caso uma anotação for encontrada será criado um proxy que irá monitorar e interceptar as chamadas dos métodos e executar as ações de cache de acordo do a anotação.

4.3.2 @Cacheable

É utilizada quando queremos que o retorno do método com essa anotação seja incluído no cache, quando ocorrer a chamada do método com essa anotação será consultado no cache se existe a chave atribuída essa chamada de método, que pode ser por exemplo, o nome do método concatenado com os parâmetros passados ao método

Se houver uma chave válida correspondente será retornado o valor contido na chave.Caso não exista uma chave válida, o método é executado normalmente e no retorno será incluído no cache esse valor com a respectiva chave, para que em futuras chamadas do método seja retornado do cache.

Essa anotação é recomendada em métodos de busca de informações no banco, onde não ocorrem alterações de dados, como demonstrado no excerto abaixo.

```
1 @Cacheable("usuarios")
2 public Usuario buscarUsuario(Long id) {
3 ...
4 }
```

Trecho de código 4.9 – Uso da anotação @Cacheable

Junto com a anotação deve ser definido um nome onde os valores serão armazenados, um nome para o cache propriamente dito, que será onde será verificado a existência da chave.

Mais de um cache pode ser definido, permitindo que os valores sejam armazenados em mais de um lugar, se for o caso todos os caches serão checados pela existência da chave.

```
1 @Cacheable({"usuarios", "clientes"})
2 public Usuario buscarUsuario(Long id) {
3 ...
4 }
```

Trecho de código 4.10 – Definição de mais de um cache

Além do nome do cache pode ser indicada a forma da geração da chave que será usada para salvar a informação no cache, aqui existem duas formas, uma delas é definindo usando Spring Expression Language (SpEL), ou indicando um gerador de chaves, as duas formas são mostradas abaixo.

```
1 @Cacheable(cacheNames="usuarios", key="#id")
2 public Usuario buscarUsuario(Long id) {
3 ...
4 }
```

Trecho de código 4.11 – Definição de chave usando SpEL

```
1 @Cacheable(cacheNames="usuarios", keyGenerator="meuGeradorDeChaves")
2 public Usuario buscarUsuario(Long id) {
3 ...
4 }
```

Trecho de código 4.12 – Definição de chaves usando gerador de chaves

Como discutido anteriormente, para a implementação foi preferido o uso do gerador de chaves, para ter uma padronização da forma de geração de chaves.

É possível também utilizar o cache de forma condicional, ou seja, as informações só serão armazenadas quando certas condições forem cumpridas, como por exemplo, usuários que acessam o sistema com pouca frequência não serão salvos.

Não foi identificada uma situação em que o uso do cache condicional fosse necessário no projeto.

4.3.3 @CachePut

Tem um comportamento similar ao @Cacheable, porém quando ocorre a chamada de um método com essa anotação não é verificado o cache em busca de uma chave correspondente, a execução do método ocorre de forma normal, somente ao final da execução o retorno é inserido no cache com sua respectiva chave.

Essa anotação é indicada quando ocorre a alteração de alguma informação que já está salva no banco e é necessário atualizar o cache para que as informações não sejam divergentes, ou quando uma nova informação será inserida no banco, como um novo usuário, por exemplo.

Dessa forma quando essa nova informação for salva no banco também já se torna disponível no cache.

Essa anotação suporta as mesmas opções que o método @Cacheable, porém essa anotação não deve ser usada no mesmo método em que já houver uma anotação @Cacheable, pois apesar de terem comportamentos semelhantes, podem ter efeitos indesejados se usados em conjunto, já que a anotação @CachePut executa normalmente o método, enquanto a anotação @Cacheable substitui a execução pela consulta em cache.

```
1 @Cacheable(cacheNames="usuarios", key="#usuario.id")
2 public Usuario atualizarUsuario(Usuario usuario) {
3 ...
4 }
```

Trecho de código 4.13 – Uso da anotação @CachePut

4.3.4 @CacheEvict

Essa anotação é usada para remover informações do cache, quando o método anotado for executado será removida do cache a chave correspondente.

Geralmente usada em métodos que removem informações do banco ou quando queremos que após algum processamento de dados as informações contidas no cache sejam limpas para garantir que as informações sejam retornadas atualizadas do banco.

Essa anotação deve receber um, ou mais caches a serem limpos, e a chave a ser removida, ainda suporta a opção de remover todas as chaves de um respectivo cache.

```
1 @CacheEvict(cacheNames="usuarios", allEntries=true)
2 public void deletarUsuariosInativos(List<Long> usuariosInativos){
3 ...
4 }
```

Trecho de código 4.14 – Uso da anotação @CacheEvict

É importante apontar que o cache deve ser usado apenas em métodos em que o retorno seja constante com base nos parâmetros fornecidos, ou seja, os resultado da execução do método não pode mudar com o tempo considerando que os mesmo parâmetros foram passados na chamada.

A pesquisa de um usuário por seu id único é um bom exemplo desse comportamento, muito dificilmente essa informação será modificada, assim garantindo que as informações presentes no cache são condizentes com as informações existentes no banco de dados.

Mas isso não quer dizer que dados que podem sofrer modificações não possam ser armazenados em cache, basta fazer a correta utilização das anotações mencionadas acima para que quando alguma informação for atualizada no banco, que seja atualizada também no cache.

Um exemplo em que o cache não é recomendado é em métodos que realizam algum cálculo baseado em tempo transcorrido, dependendo de quando o método for executado o resultado pode ser diferente.

4.4 INCLUSÃO DO CACHE NA APLICAÇÃO

Para adicionar o cache na aplicação do Time Tracking foi verificado quais *endpoints* são consultados para retornar as informações, e encontrado os métodos que são executados pelo *endpoint* acessado, a ideia não é implementar novos métodos ou modificar a execução deles, apenas encontrar o melhor ponto para aplicação do cache, mas como será exposto nas seções seguintes, algumas implementações foram necessárias para possibilitar a correta aplicação da anotação de cache.

4.4.1 Período de apuração

Como o período é uma das entidades mais simples que compõem o módulo do Time Tracking, foi a escolhida para a primeira implementação, então utilizando o console de desenvolvedor do navegador é verificado qual endpoint é utilizado para para retornar essa informações, o método executado é mostrado a seguir.

```
1 public TimeTrackingPeriod load(String id) {
2 ...
3 TimeTrackingPeriod newTimeTrackingPeriod = repo.findOne(entityId);
4 loadBind(newTimeTrackingPeriod);
5 ...
6 }
```

Trecho de código 4.15 – Método para retorno das informações do período

Aqui é feita uma chamada para o repositório do perfil, onde é retornado do banco o perfil com base no Id, sendo um bom lugar para aplicar o cache, porém o método *getById* é um método que pertence a interface JPARepository, onde não temos acesso a assinatura do método para fazer a inclusão da anotação.

Para contornar esse problema é necessário criar uma nova interface que irá estender a interface anterior, assim é possível fazer a sobrescrita do método e incluir a anotação de cache @Cacheable, como mostrado a seguir:

```
1
2 public interface TimeTrackingPeriodRepository extends JpaRepository<
    TimeTrackingPeriod, Long>
3 {
4 @Override
5 @Cacheable(value = "timeTrackingPeriod", key = "new String(#p0)")
6 TimeTrackingPeriod findOne(Long id);
7 }
```

Trecho de código 4.16 – Classe criada para adicionar as anotação para o cache

Já temos a implementação do cache para o período, porém ainda é possível adicionar mais informações ao cache, logo em seguida ao método que retorna as informações do período é feita a busca pelas exceções primárias e secundárias,sendo o método *loadBind* responsável por essa consulta, novamente as consultas são feitas pelo Id, então é adicionado o cache em todas os métodos de consulta.

Como todos os métodos são nativos da interface, é feito o mesmo procedimento anterior de criar uma nova interface e fazer a sobrescrita para permitir a adição da anotação de cache.

```
1 public interface CompanyRepository extends JpaRepository<Company, Long>
    {
2 @Override
3 @Cacheable(value = "company", keyGenerator = "customKeyGenerator")
4 Company findOne(Long id);
5 }
```

Trecho de código 4.17 – Repositórios criados para uso das anotações de cache

```java
public interface WorkplaceRepository extends JpaRepository<Workplace,
    Long>, WorkplaceRepositoryCustom {
@Override
@Cacheable(value = "workplace", keyGenerator = "customKeyGenerator")
Workplace findOne(Long id);
}
```

Trecho de código 4.18 – Repositórios criados para uso das anotações de cache

```java
public interface TradeUnionRepository extends JpaRepository<TradeUnion,
    Long> {
@Override
@Cacheable(value = "tradeUnion", keyGenerator = "customKeyGenerator")
TradeUnion findOne(Long id);
}
```

Trecho de código 4.19 – Repositórios criados para uso das anotações de cache

```java
public interface PersonRepository extends JpaRepository<Person, Long>,
    PersonRepositoryCustom {
@Override
@Cacheable(value = "person", keyGenerator = "customKeyGenerator")
Person findOne(Long id);
}
```

Trecho de código 4.20 – Repositórios criados para uso das anotações de cache

```java
public interface BusinessUnitRepository extends JpaRepository<
    BusinessUnit, Long> {
@Override
@Cacheable(value = "businessUnit", keyGenerator = "customKeyGenerator")
BusinessUnit findOne(Long id);
}
```

Trecho de código 4.21 – Repositórios criados para uso das anotações de cache

Com isso temos o cache aplicado no retorno das informações do banco, porém quando o período sofre alguma alteração, essa alteração deve ser aplicada nas informações salvas no cache, ou mesmo quando o período for removido do sistema, deve ser removido do cache também, assim é feito o mesmo procedimento anterior, mas agora na alteração e exclusão do período.

Acessando um período e salvando alguma alteração, é verificado qual endpoint é acessando via console do navegador.

O método invocado é então analisado para encontrar onde é salvo a entidade no banco, aqui novamente usando o método nativo, é feita a implementação já mencionada anteriormente.

```
1 public TimeTrackingPeriod update(TimeTrackingPeriodDto
      timeTrackingPeriodDto, String id) {
2 ...
3 TimeTrackingPeriod timeTrackingPeriodNew = repo.saveAndFlush(
      timeTrackingPeriod);
4 ...
5 return timeTrackingPeriodNew;
6 }
```

Trecho de código 4.22 – Método adicionado com a anotação de cache para salvar o período

```
1 @Override
2 @CachePut(value = "timeTrackingPeriod", key = "new String(#p0.id)")
3 <S extends TimeTrackingPeriod> S saveAndFlush(S entity);
```

Trecho de código 4.23 – Método adicionado com a anotação de cache para salvar o período

Da mesma forma é feita a análise e implementação para a remoção do período, vendo qual endpoint é acessado.

E o método invocado, tendo agora a anotação @CacheEvict aplicada ao método de remoção da entidade do banco de dados e também removemos a entidade do cache.

Mesmo não sendo efetivamente removido do banco, tendo apenas uma remoção lógica, não é necessário manter essas informações no cache, sendo assim removida.

```
1 public TimeTrackingPeriod remove(Long id) {
2 ...
3 this.save(obj);
4 ...
5 return obj;
6 }
```

Trecho de código 4.24 – Anotação para remoção do período do cache

```
1 @Override
2 @CacheEvict(value = "timeTrackingPeriod", key = "new String(#p0.id)")
3 <S extends TimeTrackingPeriod> S save(S entity);
```

Trecho de código 4.25 – Anotação para remoção do período do cache

Com isso finalizamos a implantação do cache no período de apuração, não somente o retorno das informações mas também a atualização e remoção.

4.4.2 Perfil de apuração

O perfil de apuração é uma entidade muito mais complexa que o período de apuração, mas que foi notado uma certa demora para carregar os dados, então é um

bom ponto onde aplicar o cache.

Novamente é acessado o sistema e com a ajuda do console é mapeado qual endpoint é responsável por retornar os dados do perfil.

Acessando esse endpoint encontramos o método responsável por retornar as informações.

```
1 public TimeTrackingProfile load(String id) {
2 try {
3 ...
4 final TimeTrackingProfile newTimeTrackingProfile = repo.findOne(entityId
    );
5 ...
6 return newTimeTrackingProfile;
7 }
8 }
```

Trecho de código 4.26 – Método que retorna o perfil de apuração

Vemos que o método de busca da entidade novamente usa o método nativo, assim aplicamos a mesma estratégia que foi utilizada no período de apuração, é criado uma nova interface que fará a implementação da interface nativa do JPA e assim aplicando a anotação de cache.

```
1
2 public interface TimeTrackingProfileRepository extends JpaRepository<
    TimeTrackingProfile, Long> {
3 @Override
4 @Cacheable(value = "timeTrackingProfile", key="new String(#p0)")
5 TimeTrackingProfile findOne(Long id);
6 }
```

Trecho de código 4.27 – Repositório criado para adição das anotações de cache

Em seguida é carregado a forma de exportação de dados utilizada na exportação de insumos para folha de pagamento, outra informação que pode ser aplicada o cache, como mostrado a seguir.

```
1 public interface TimeTrackingProfileExportEventFormatRepository extends
    JpaRepository<TimeTrackingProfileExportEventFormat, Long>,
    JpaSpecificationExecutor<TimeTrackingProfileExportEventFormat> {
2
3 @Override
4 @Cacheable(value = "timeTrackingProfileExportEventFormat", key = "new
    String(#p0)")
5 TimeTrackingProfileExportEventFormat findOne(Long id);
6 }
```

Trecho de código 4.28 – Cache nas formas de exportação

A ultima parte do método que faz consulta ao banco de dados retorna os tipos de eventos de apuração que podem ser gerados, como são dados que dificilmente serão alterados, é um excelente ponto onde pode adicionado o cache.

Seguindo a mesma lógica aplicada anteriormente é incluído o cache nesse método de busca.

```
1 @Cacheable(value = "timeTrackingType")
2 private List<TimeTrackingType> listAllTimeTrackingTypes() {
3 return timeTrackingTypeRepository.findByAll().stream().map(
    TimeTrackingType::new).collect(Collectors.toList()).stream().filter(
    obj -> !obj.getId().equals(90L)).collect(Collectors.toList());
4 }
```

Trecho de código 4.29 – Cache nos tipos de eventos de apuração

Assim finalizando o cache do perfil de apuração, mas novamente devemos incluir no métodos de atualização e remoção do perfil a suas respectivas anotações para manter as informações do banco de acordo com as informações no cache, seguimos para a atualização das informações.

Seguimos os mesmos passos anteriormente, acessando um perfil e descobrindo o endpoint acessado.

Incluindo a anotação @CachePut.

```
1 @Override
2 @CachePut(value = "timeTrackingProfile", key="new String(#p0.id)")
3 <S extends TimeTrackingProfile> S saveAndFlush(S entity);
```

Trecho de código 4.30 – Método de atualização com anotação

Próximo passo é finalizar com a adição da anotação @CacheEvict na remoção do perfil, seguindo os mesmo passos de anteriormente, com o endpoint acessado.

```
1 @Transactional
2 public TimeTrackingProfile update(TimeTrackingProfile
    timeTrackingProfile, String id) {
3 ...
4 repo.saveAndFlush(timeTrackingProfile);
5 }
6 return null;
7 }
```

Trecho de código 4.31 – Método de remoção com anotação

Método atualizado com a anotação.

```
1 @Override
2 @CacheEvict(value = "timeTrackingProfile", key="new String(#p0.id)")
3 <S extends TimeTrackingProfile> S save(S entity);
```

Trecho de código 4.32 – Método de remoção com anotação

Com isso é finalizado a implementação do cache no perfil de apuração, o passo seguinte será analisado os efeitos da adição do cache na performance, tanto na velocidade de carregamento, como no consumo de recursos.

5 ANÁLISE SOBRE O CACHE

Após a implementação do cache nos *endpoints* foi feita a análise dos efeitos da integração do Time Tracking com o Redis, serão levados em consideração dois ponto nessa análise, primeiramente será o tempo de carregamento das informações e também será verificado se houve mudança nos recursos consumidos pela JVM ao utilizar o cache, sendo o uso de Central Processing Unit (CPU) e o consumo de memória levados em consideração.

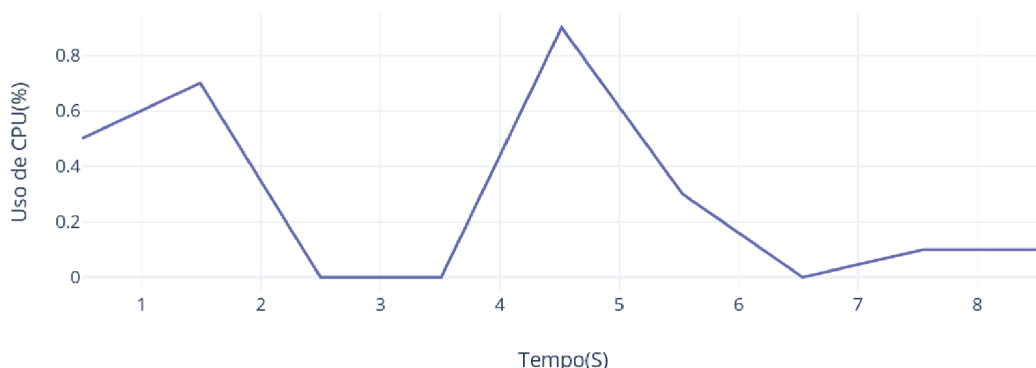
Como o consumo de memória varia bastante e é muito difícil iniciar o monitoramento sempre com a mesma quantidade de memória em utilização, o que será levado em conta será a diferença entre a quantidade de memória inicial e a final em cada análise.

5.1 ANÁLISE DO PERÍODO

Para carregar as informações do período de apuração é consultado apenas um *endpoint*, então será apenas esse considerado na análise, começando pelo tempo de carregamento, mostrado na tabela 1.

Sem o cache na aplicação as informações são recuperadas direto do banco de dados em 833.17 milissegundos, seguindo para o consumo de recursos, os gráficos, na figura 9 mostram o uso de memória da JVM e o uso de CPU.

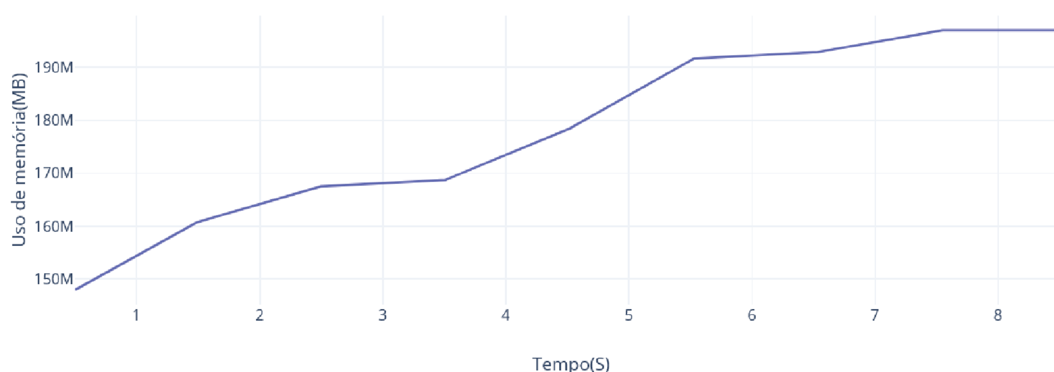
Figura 9 – Consumo de CPU no carregamento do período sem cache



Podemos ver que o consumo de recursos é bem baixo, tendo apenas um pico de 0,9% no máximo em CPU, enquanto pelo restante do tempo o consumo apenas fica oscilando entre 0 e 0,1%.

Em questão de memória carregar o período direto do banco de dados causou um aumento de em torno 49 MB de memória, tendo seu ponto máximo em 196 MB em contrapartida do inicial de 147 MB, como mostrado na figura 10

Figura 10 – Consumo de memória no carregamento do período sem cache



Seguindo com a análise, agora com o cache aplicado ao método, primeiramente com a recuperação dos dados sem as informações presentes em cache, para comparar se houve um impacto negativo quando as informações precisam ser salvas no cache a primeira vez, assim temos que o tempo de carregamento foi em torno de 844 ms, mostrado na tabela 1, sendo muito próximo do carregamento sem cache, então a adição do cache, e consequentemente o passo extra de salvar as informações na memória, não afetou de forma significativa o carregamento das informações.

Em termos de recursos foi verificado que com o passo extra de salvar as informações no cache pode ter consumido um pouco mais de recursos, tendo picos de 0,5% em apenas um ponto e o pico máximo de 1,1% de CPU mostrado na figura 11, que foi muito próximo ao carregamento sem cache, então há uma diferença de consumo de recursos, porém muito pequena para ser considerado como algo que irá influenciar na aplicação.

Apesar não ser muito significante, considerando que o período é uma entidade relativamente simples, é necessário ter em mente que entidades muito complexas podem ter um efeito mais significante.

Considerando o consumo de memória pela JVM houve uma queda em relação ao carregamento quando não foi usado o cache, tendo uma diferença de 26,5 MB de memória, mostrado na figura 12

Agora acessando novamente o período com a recuperação das informações pelo cache temos o seguinte vemos uma melhora de tempo de carregamento signifi-

Figura 11 – Consumo de CPU ao salvar o período no cache

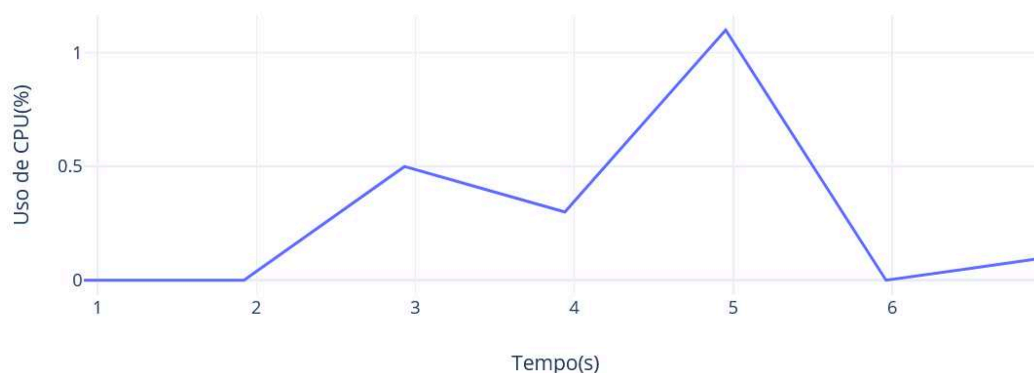
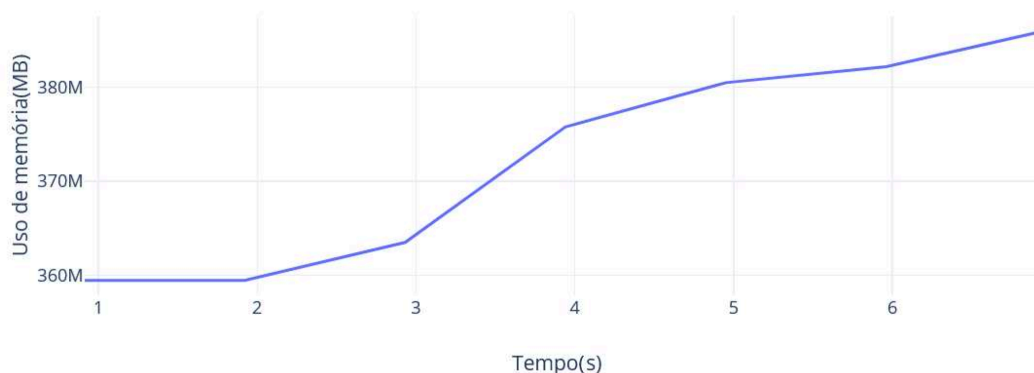


Figura 12 – Consumo de memória ao salvar o período no cache

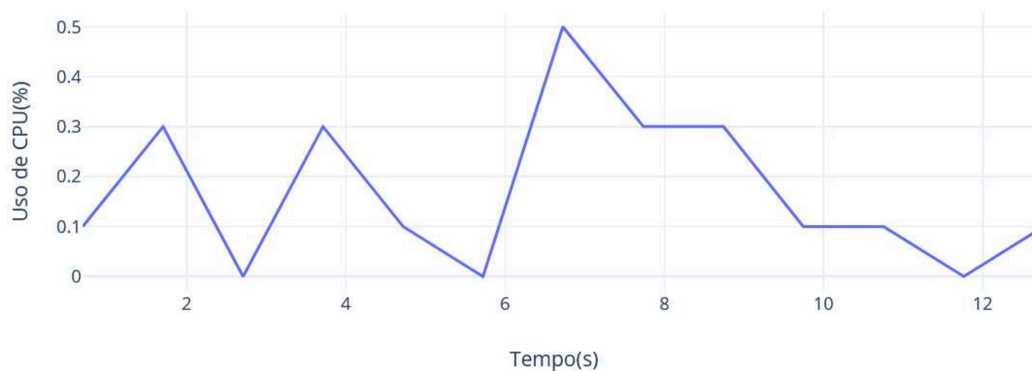


cante, tendo 17,77 milissegundos em tempo de carregamento, demostrado na tabela 1

Em seguida foi verificado se o consumo de recursos sofreu alguma modificação, sendo mostrado na figura 13

Aqui vemos que o consumo de recurso possui uma ligeira queda em relação ao carregamento de cache, o pico máximo dessa vez foi de 0,5%, tendo outros dois picos de 0,3% em uso de CPU demonstrado que a consulta e retorno das informações na base de cache mão causa mais consumo de recursos, mesmo tendo um leve queda no consumo de CPU ainda não é possível concluir que seja mais eficiente, em termo

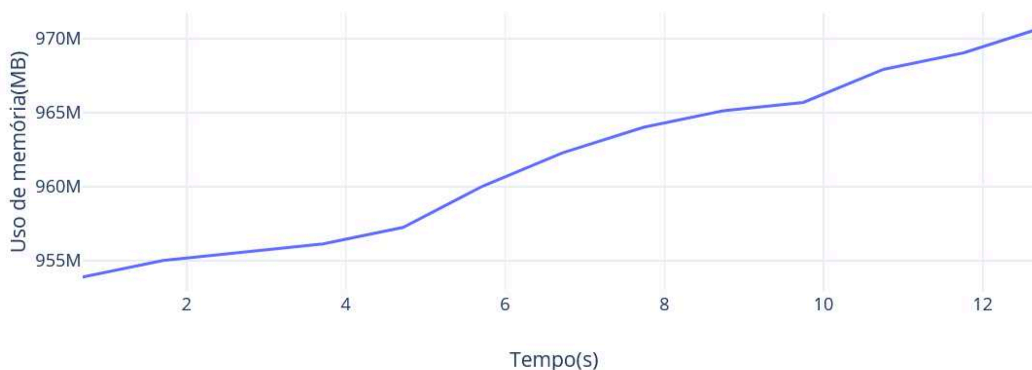
Figura 13 – Consumo de CPU retornando o período do cache



de uso de CPU, utilizar a base de cache, mas em tempo de resposta definitivamente houve uma melhora expressiva.

Seguindo agora para o consumo de memória, houve mais uma queda em relação ao diferencial entre o inicio e o fim do monitoramento, havendo apenas uma diferença de 16 MB de memória consumida pela JVM, demonstrado na figura 14

Figura 14 – Consumo de memória retornando o período do cache



Com essa análise é possível concluir que o impacto do cache na aplicação foi beneficente para o tempo de carregamento, sem prejudicar o consumo de CPU e memória, porém considerando a simplicidade do período, não possui muitas informações

relacionadas e mesmo sem o cache já era carregado em tempos aceitáveis, ainda não é possível concluir se há melhoria no consumo de recursos, na análise seguinte, utilizando o perfil é possível que uma diferença maior seja encontrada.

Tabela 1 – Tempos de carregamentos do período

Tipo	Tempo de carregamento
Sem cache(Não salvando)	833.17ms
Sem cache(Salvando)	907.1ms
Com cache	17.77ms

Fonte – O autor.

5.2 ANÁLISE DO PERFIL

Acessar as informações do perfil consome diversos enpoints, todos eles tiveram o cache aplicado, porem somente será analisado o tempo de resposta do *endpoint* que efetivamente retorna o perfil de apuração para exibição, por ser a entidade mais complexa.

Assim iniciando com o tempo de carregamento do perfil sem cache, sendo mostrado na tabela 2

Aqui é verificado que o cache pode ser uma ótima opção, pois o tempo de carregamento é de 19,23 segundos, um tempo muito alto, porém considerando que a aplicação está sendo executada de forma local com o banco hospedado na Amazon Web Services (AWS) é o principal motivo da demora.

Em seguida foi analisado o consumo de recursos retornando o perfil diretamente do banco de dados principal, mostrado na figura 15

Houve um pico de consumo de CPU em 1,9% quando foi feita a pesquisa do perfil no banco de dado, considerando a complexidade do objeto é de se esperar que ocorra um consumo relativamente maior que o período, em outros momento ocorreram oscilações entre 0.3% e 0.7%, o que é dentro do esperado.

Em consumo de memória houve também um aumento significativo em relação ao consumo quando carregado o período, tendo aqui uma diferença de quase 116 MB de memória a mais causada pelo retorno do perfil do banco de dados.

Agora novamente, seguindo o mesmo processo anteriormente, começando com o carregamento do perfil sem estar presente no cache, mas salvando para a consulta seguinte, foi verificado que o tempo de carregamento teve uma leve variação, sendo um pouco menor, mostrado na tabela 2

Agora vendo como foi o consumo de recursos com o passo extra de salvar em cache as informações, temos os seguintes valores, mostrados na figura 17

Figura 15 – Consumo de CPU carregando o perfil sem cache

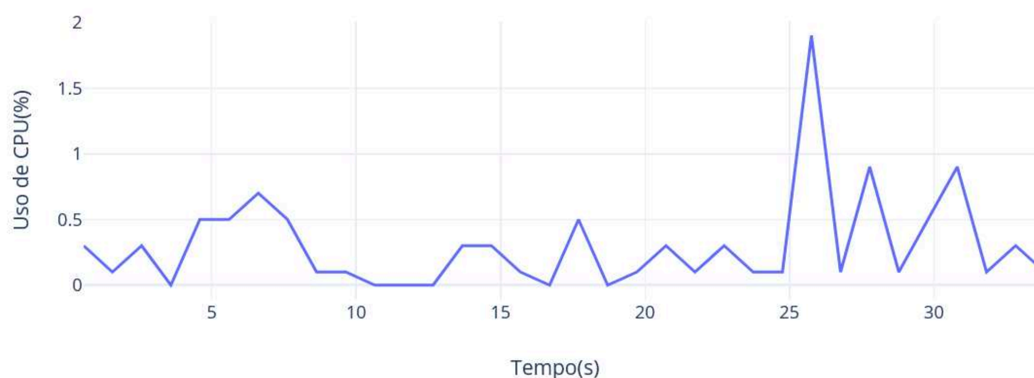
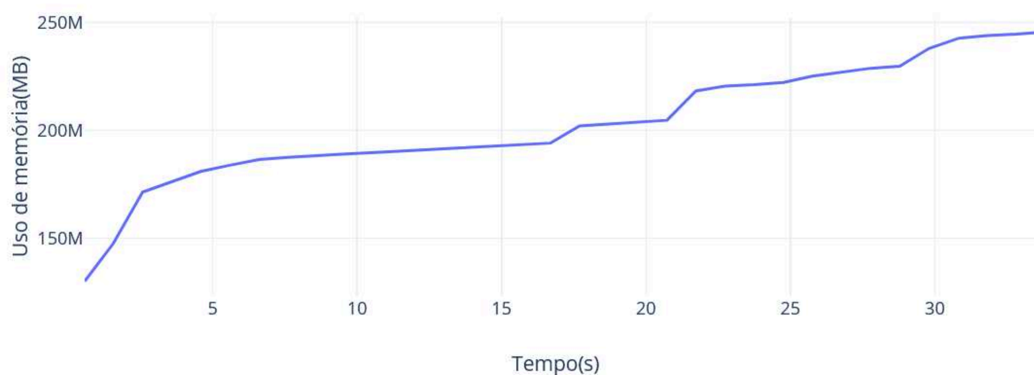


Figura 16 – Consumo de memória carregando o perfil sem cache



Em termos de CPU o consumo ficou dentro do esperado, apresentando uma leve queda, tendo apenas um pico de 1.4%, comportamento oposto ao verificado no período, onde houve um leve aumento, porém devido a pequena diferença não pode ser considerado como algo expressivo.

Novamente houve queda do consumo de memória quando é necessários salvar no cache os dados retornados, tendo uma diferença de 80 MB entre os valores iniciais e finais.

Novamente acessando as informações, mas agora com a pesquisa feita na base de cache temos um tempo de carregamento com uma grande redução, sendo em torno

Figura 17 – Consumo de CPU salvando o perfil no cache

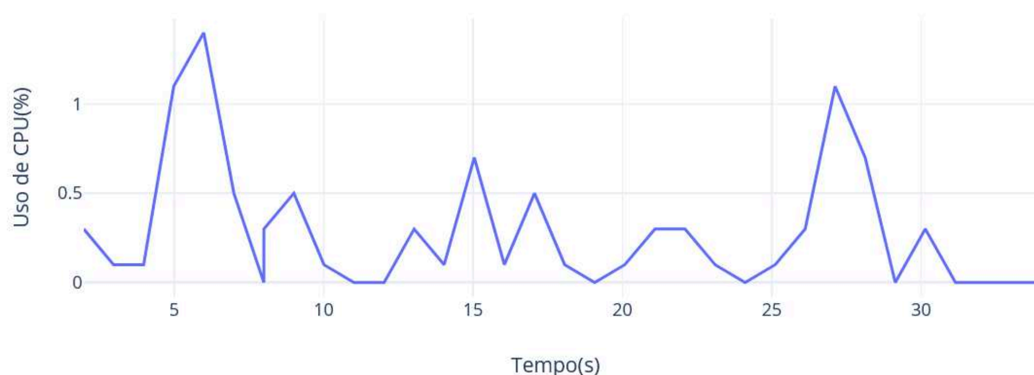


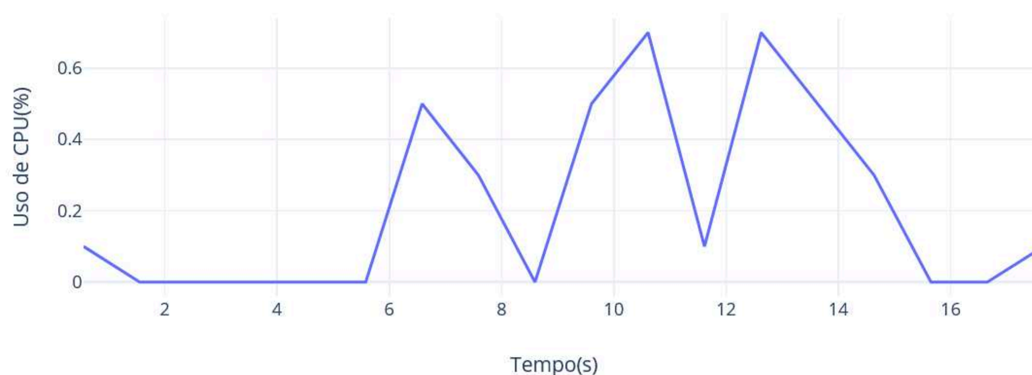
Figura 18 – Consumo de memória salvando o perfil no cache

de 10 vezes menor, como mostrado no tabela 2.

Aqui vemos uma grande diferença entre os tempos de carregamento, uma queda de 19.23 segundos para 682.08 milissegundos, uma melhora muito significativa, e para finalizar com a analise, temos os seguintes consumos de recursos, mostrados na figura 19

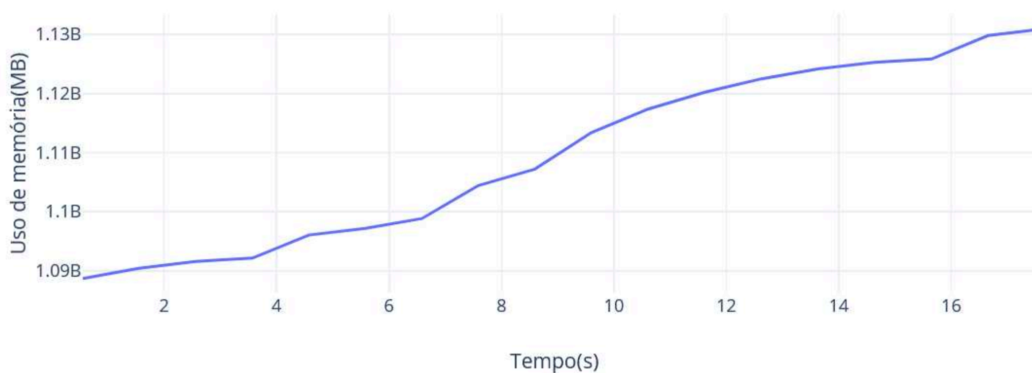
Aqui foi verificado mais uma redução em relação ao uso de CPU ao carregar do cache, onde houve um pico de apenas 0,7% de uso, em contraste aos 1.9% ao carregar direto do banco, uma pequena melhora que também se mostra no consumo de memória, tendo uma queda novamente, sendo de apenas 40 MB quando se utiliza

Figura 19 – Consumo de CPU retornando o perfil pelo cache



o cache, caindo mais que a metade de quando não foi utilizado.

Figura 20 – Consumo de memória retornando o perfil pelo cache



Aplicando o cache na aplicação se mostrou benéfico em questão de tempos de carregamento das informações analisadas, havendo também o mesmo efeito em consumo de memória, mesmo que em pequena escala, porém é bom notar que mesmo adicionando uma camada entre o banco de dados e a aplicação não foi analisado alguma piora em nenhum quesito analisado.

Tabela 2 – Tempos de carregamentos do período

Tipo	**Tempo de carregamento**
Sem cache(Não salvando)	19.23s
Sem cache(Salvando)	18.71s
Com cache	682.08ms

Fonte – O autor.

6 MONITORAMENTO DA APURAÇÃO COM CACHE

Além de melhorar a performance do sistema foi verificado a possibilidade de utilizar o cache de forma a acompanhar o progresso de apuração de ponto em lote de um período de apuração, onde todas as informações relacionadas a um colaborador são coletadas do banco e enviadas para a geração do eventos diários com base nas batidas de ponto do colaborador.

A forma atual de apuração requer que cada apuração de colaborador seja enviada para uma fila de mensagens para ser apurada, sendo usado o Simple Queue Service (SQS) da AWS, dessa forma não é possível saber o andamento da apuração e em casos em que ocorre a queda do sistema a apuração deve ser reiniciada, pois as apurações que estavam sendo processadas podem estar incompletas, o que ira gerar inconsistências quando os dados forem exportados para processamento da folha de pagamento.

A solução escolhida foi de ao enviar uma mensagem para a fila SQS, também incluir uma chave no cache, sendo inicialmente uma combinação do id do colaborador sendo apurado, do id do período sendo apurado e por fim o id do cliente que está realizando essa apuração, garantindo assim que não ocorram conflitos entre chaves.

Essa escolha de chaves foi feito de forma temporária, sendo verificada a acurácia do acompanhamento da apuração em lote, uma chave definitiva será escolhida.

Para essa chave será atribuído um valor booleano falso, indicando que a mensagem foi efetivamente enviada para a fila SQS, mas ainda não sendo processada, com isso já temos a métrica de quantos colaboradores já estão em fila de apuração, podendo ser comparado com o total de colaboradores contemplados pelo período em questão.

Ao finalizar a apuração do colaborador será atualizado o valor da chave para o valor verdadeiro, após serem salvos no bancos os eventos relacionados a apuração, garantindo assim que foi concluída com sucesso e os dados foram efetivamente salvos.

Para essa tarefa foi desenvolvida uma classe que seria responsável por incluir, atualizar, remover e consultar o Redis, sendo ela chamada *SqsTrackerService*, que está em sua totalidade no trecho de código 6.3.

Nessa classe é utilizada a interface *HashOperations*, que foi brevemente citada no capitulo 4, seção 4.1.4, sendo responsável por fornecer todas as operações necessárias para interação com o Redis, por isso não será detalhado o funcionamento da classe.

Alguns dos métodos implementados na classe *SqsTrackerService* serão explicados a seguir, outros porem, são autoexplicativos e não serão abordados.

- clearTimeTrackingFilterRedis(): Esse método ainda não foi implementado, mas servirá ao propósito de limpar todas as chaves relacionadas a alguma apuração,

assim caso seja necessário reapurar algum período, a limpeza das chaves deve ser feita antes, ou nenhum colaborador será apurado, afinal todas as chaves já estarão com o valor verdadeiro, indicando que já houve a apuração.

- hashSize(): Esse método tem como objetivo retornar a quantidade de chaves que estão armazenadas no cache.
- valuesInKey():Esse método retornar todas as chaves que estão armazenadas em cache.
- generateHashKeyTimeTrackingFilter(): Método responsável por criar as chaves mencionadas anteriormente, considerando o id do cliente, o id do período e o id do colaborador que está sendo apurado.

O método modificado para inserção da chave indicando o envio da mensagem para o SQS é demonstrado a seguir.

```
1 public void convertAndSendTimeTrackingSync(TimeTrackingFilter
    timeTrackingFilter) {
2 ...
3   Boolean done = sqsTrackerRepository.getTimeTrackingFilterRedis(
    timeTrackingFilter);
4   if (Objects.isNull(done) || !NextiUtil.safeBooleanValue(done)) {
5     sqsTemplate.convertAndSend(timeTrackingQueueBatch,
    timeTrackingFilter);
6     putTimeTrackingFilterInCache(timeTrackingFilter, Boolean.FALSE);
7   }
8 ...
9 }
```

Trecho de código 6.1 – Método modificado para salvar chave no cache

Na finalização da apuração a chave é atualizada, o trecho modificado é o abaixo.

```
1 public void listenerBatch(TimeTrackingFilter timeTrackingFilter) {
2 ...
3   log.info("###TIMETRACKINGBATCH END Lote - {} - {} - {} - {} ms",
    timeTrackingFilter.getCustomerId(), timeTrackingFilter.getPerson(),
    timeTrackingFilter.getProfile(), elapsedTime);
4   sqsTrackerRepository.saveTimeTrackingFilterRedis(timeTrackingFilter,
    Boolean.TRUE);
5   Map<String, Boolean> done = sqsTrackerRepository.valuesInKey("
    timeTrackingFilter");
6 }
```

Trecho de código 6.2 – Método modificado para atualizar chave no cache

Para verificar o funcionamento da apuração foi executado novamente em ambiente local a apuração em lote de um período, foi escolhido um período com cerca de

640 colaboradores, ao iniciar a apuração foi verificado junto ao Redis que as chaves estavam sendo criadas e posteriormente atualizadas ao final da apuração.

Para simular uma queda do sistema foi parada a execução do microsserviço através da interface do STS, ao reexecutar o microsserviço as mensagens que ainda se encontravam em fila no SQS foram apuradas, mas como o envio ainda estava ocorrendo foi abortado, reiniciando a apuração em lote todas as mensagens que já tinhas sido atualizadas no cache foram ignoradas, sendo apenas enviadas para a fila do SQS a que ainda estavam no cache com o valor falso ou que ainda não estavam presentes, com isso foi verificado que a apuração não precisou ser feita totalmente do começo, como ocorre atualmente.

```
1  @Service
2  public class SqsTrackerService {
3
4    private static final String TIME_TRACKING_FILTER_KEY = "
      timeTrackingFilter";
5
6    private RedisTemplate<String, Boolean> redisTemplate;
7
8    private HashOperations<String, String, Boolean> hashOperations;
9
10   @Autowired
11   public SqsTrackerService(RedisTemplate<String, Boolean> redisTemplate)
       {
12     this.redisTemplate = redisTemplate;
13     this.hashOperations = this.redisTemplate.opsForHash();
14   }
15
16   public void saveTimeTrackingFilterRedis(TimeTrackingFilter
      timeTrackingFilter, Boolean status) {
17     hashOperations.put(TIME_TRACKING_FILTER_KEY,
      generateHashKeyTimeTrackingItem(timeTrackingFilter), status);
18   }
19
20   public Boolean getTimeTrackingFilterRedis(TimeTrackingFilter
      timeTrackingFilter) {
21     return hashOperations.get(TIME_TRACKING_FILTER_KEY,
      generateHashKeyTimeTrackingItem(timeTrackingFilter));
22   }
23
24   public void deleteTimeTrackingFilterRedis(TimeTrackingFilter
      timeTrackingFilter) {
25     hashOperations.delete(TIME_TRACKING_FILTER_KEY,
      generateHashKeyTimeTrackingItem(timeTrackingFilter));
26   }
27
```

```
28    public void clearTimeTrackingFilterRedis() {
29    }
30
31    public Long hashSize(String key) {
32      return hashOperations.size(key);
33    }
34
35    public Map<String, Boolean> valuesInKey(String key) {
36      return hashOperations.entries(key);
37    }
38
39    private String generateHashKeyTimeTrackingItem(TimeTrackingFilter
       timeTrackingFilter) {
40      StringBuilder hashKey = new StringBuilder();
41      hashKey.append(timeTrackingFilter.getCustomerId())
42      .append("_")
43      .append(timeTrackingFilter.getPeriod().getId())
44      .append("_")
45      .append(timeTrackingFilter.getPerson().getId());
46      return hashKey.toString();
47    }
```

Trecho de código 6.3 – Classe utilizada para comunicar com o Redis

Mesmo sendo em caráter experimental, essa implementação permite uma forma de acompanhar o andamento da apuração em lote, e principalmente, permitir o resumo da apuração quando ocorrer alguma indisponibilidade no sistema.

Utilizando a base de cache foi possível realizar essa implementação sem modificar o funcionamento atual do sistema, permitindo uma melhora da rastreabilidade sem comprometer as implementações já existentes.

Existem outros microsserviços que também utilizam de alguma forma uma fila SQS, assim é possível expandir essa melhoria para outras partes do sistema.

7 CONCLUSÃO E TRABALHOS FUTUROS

7.1 CONCLUSÃO

Necessitando aliviar a carga do banco de dados principal do sistema Nexti foi implementado uma base de dados em memória para servir de cache para a aplicação, onde os dados mais utilizados ou aqueles que não sofrem tantas modificações são armazenados, evitando assim que o banco de dados principal seja requisitado com consultas triviais.

Além de reduzir a carga sobre o banco de dados principal, como a base de cache se encontra em memória, os tempos de resposta são muito mais rápidos, conferindo além do alivio sobre o banco, também mais responsividade do sistema.

Finalizada a implantação do banco de dados foi feita uma analise sobre os efeitos do cache não só para tempos de carregamento, mas também sobre os efeito no consumo de recursos para verificar se removendo a carga sobre o banco de dados poderiam haver efeitos colaterais, como uso maior de CPU ou memória.

Após a análise foi concluído que a implantação do banco de dados em memória como cache foi um sucesso, houve grandes ganhos em resposta do sistema, tornando a aplicação mais otimizada, e sem sacrificar o consumo de recursos, sendo até possível esperar que a aplicação do cache em pontos mais "pesados"do sistema possam trazes uma diminuição em relação à memória utilizada quando as informações forem retornadas do cache, em relação as consultas diretas ao banco de dados.

Além de melhorar a performance, o cache também foi utilizado para verificar a possibilidade de uso como uma ferramenta de acompanhamento da apuração em lote, que atualmente é feita de maneira assíncrona e impede que seja feito o monitoramento de forma fácil, e permitir que quando ocorram imprevistos, e o sistema venha a ter uma baixa, possibilite a retomada do ponto onde foi abortada.

Essa implementação e mostrou promissora, sendo possível acompanhar o progresso, mostrando quantos colaboradores já estão em fila de apuração e quantos já foram apurados, além de se ocorra uma baixa no sistema, as chaves salvas permitem excluir da reapuração aqueles que já tiveram seus eventos gerados e salvos no banco de dados.

7.2 TRABALHOS FUTUROS

Como o cache foi aplicado a apenas um dos microsserviços que compõem o sistema Nexti, fica como trabalho futuro expandir para os outros microsserviços também utilizarem essa ferramenta, como todas as configurações já estão concluídas sobre o Redis, basta encontrar os pontos mais afetados dos microsserviços e replicar todas as implementações já realizadas no Time Tracking.

O principal ponto de trabalhos futuros é realizar a implantação do cache em ambiente de produção, tornando assim o cache uma parte do sistema Nexti, removendo uma parte da carga de pesquisas no banco de dados principal e promovendo uma melhor experiência para o usuário final.

Outro ponto para possível trabalho futuro é utilizar outra função do Redis para melhorar em mais um quesito o sistema Nexti, ser o principal ponto de troca de mensagens entre os microsserviços, já que o Redis pode ser utilizado como *message broker*, removendo assim a dependência do SQS, que no momento esta se mostrando outro gargalo para o sistema, onde o fluxo de mensagens já está se tornando insuficiente para os parâmetros atuais, e onde o Redis se mostra muito mais eficiente.

REFERÊNCIAS

ABDULLAH TALHA KABAKUSA, Resul Kara. A performance evaluation of in-memory databases. **Journal of King Saud University**, v. 29, n. 4, p. 520–525, 2017. Disponível em: https://www.sciencedirect.com/science/article/pii/S1319157816300453.

ALSHAFIE GAFAAR MHMOUD MOHMMED, Saife Eldin Fath Osman. SQL vs NoSQL. **Journal of Multidiciplinary Engineering Science Studies**, v. 3, n. 5, p. 1790–1792, 2017. Disponível em: https://www.researchgate.net/publication/327834151_SQL_vs_NoSQL.

ANDERSON, Kristi. **2019 Database Trends**. [*S.l.*: *s.n.*]. Disponível em: http://highscalability.com/blog/2019/3/6/2019-database-trends-sql-vs-nosql-top-databases-single-vs-mu.html. Acesso em: 19 jan. 2021.

COOK, John D. **ACID versus BASE for database transactions**. [*S.l.*: *s.n.*]. Disponível em: http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/. Acesso em: 18 jan. 2021.

DAKAR, Romulo. **Redis – o que é e para que serve?** [*S.l.*: *s.n.*], 2015. Disponível em: http://desenvolvedor.ninja/redis-o-que-e-e-para-que-serve. Acesso em: 10 nov. 2020.

DAN PRITCHETT, Ebay. Base: An Acid Alternative. **OpenJournalof Databases**, v. 6, n. 3, p. 48–55, 2018. Disponível em: https://queue.acm.org/detail.cfm?id=1394128.

DEVOPEDIA. **In-Memory Database**. [*S.l.*: *s.n.*]. Disponível em: https://devopedia.org/in-memory-database. Acesso em: 19 jan. 2021.

FAYCHANG. Bigtable:A Distributed Storage System for Structured Data, 2006. Disponível em: https://static.googleusercontent.com/media/research.google.com/pt-BR//archive/bigtable-osdi06.pdf.

JENKOV, Jakob. **Java Annotations**. [*S.l.*: *s.n.*]. Disponível em: http://tutorials.jenkov.com/java/annotations.html. Acesso em: 19 jan. 2021.

JIHAD NAJAJREH, Faisal Khamayseh. Contemporary Improvements of In-Memory Databases: A Survey. **8th International Conference on Information Technology**, 2017. Disponível em: http://scholar.ppu.edu/bitstream/handle/123456789/423/Contemporary%5C%20In%5C%20Memory.pdf.

ORACLE. **Annotations**. [*S.l.*: *s.n.*]. Disponível em: https://redis.io/topics/security. Acesso em: 11 jan. 2021.

RAI, Arvind. **Spring Data Redis Example**. [*S.l.*: *s.n.*], 2018. Disponível em: https://www.concretepage.com/spring-4/spring-data-redis-example. Acesso em: 10 nov. 2020.

REDIS. **How fast is Redis?** [*S.l.*: *s.n.*]. Disponível em: https://redis.io/topics/benchmarks. Acesso em: 19 jan. 2021.

REDIS. **Redis**. [*S.l.*: *s.n.*]. Disponível em: https://redis.io/. Acesso em: 4 set. 2020.

REDIS. **Redis Clients**. [*S.l.*: *s.n.*]. Disponível em: https://redis.io/clients#java/. Acesso em: 4 set. 2020.

REDIS. **Replication**. [*S.l.*: *s.n.*]. Disponível em: https://redis.io/topics/replication. Acesso em: 19 jan. 2021.

REDIS Security. [*S.l.*: *s.n.*]. Disponível em: https://redis.io/topics/security. Acesso em: 11 jan. 2021.

ROBINSON, Scott. **What is Maven?** [*S.l.*: *s.n.*]. Disponível em: https://stackabuse.com/what-is-maven. Acesso em: 7 jan. 2021.

STONEBRAKER, Michael. SQL Databases v. NoSQL Databases. **COMMUNICATIONS OF THEACM**, v. 53, n. 4, p. 10–11, 2010. Disponível em: https://www.labouseur.com/courses/db/Stonebraker-SQL-vs-NoSQL-2010.pdf.

TRELLE, Tobias. **Spring Data: A solução mais geral para persistência?** [*S.l.*: *s.n.*], 2013. Disponível em: https://www.infoq.com/br/articles/spring-data-intro/. Acesso em: 6 nov. 2020.

VERONIKA ABRAMOVA JORGE BERNARDINO, Pedro Furtado. Which NoSQL Database? A Performance Overview. **OpenJournalof Databases**, v. 1, n. 2, p. 17–24, 2014. Disponível em: `https://www.ronpub.com/OJDB-v1i2n02_Abramova.pdf`.