UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE LINGUAGENS E SISTEMAS DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Elton Luiz Rasch

UMA APLICAÇÃO PARA CARGA DE DADOS DE MONITORAMENTO DA GEOMETRIA DE LINHAS FÉRREAS

Santa Maria, RS, Brasil
2018

Elton Luiz Rasch

UMA APLICAÇÃO PARA CARGA DE DADOS DE MONITORAMENTO DA GEOMETRIA DE LINHAS FÉRREAS

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Bacharel em Ciência da Computação**.

Orientador: Prof. Dr. João Carlos Damasceno Lima

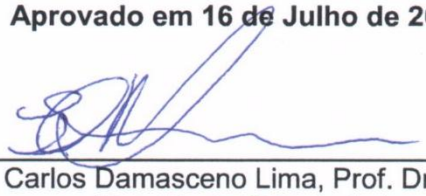
445
Santa Maria, RS
2018

Elton Luiz Rasch

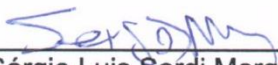
UMA APLICAÇÃO PARA CARGA DE DADOS DE MONITORAMENTO DA GEOMETRIA DE LINHAS FÉRREAS

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Bacharel em Ciência da Computação**.

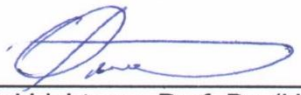
Aprovado em 16 de Julho de 2018



João Carlos Damasceno Lima, Prof. Dr. (UFSM)
(Presidente/Orientador)



Sérgio Luis Sardi Mergen, Prof. Dr. (UFSM)



Daniel Lichtnow, Prof. Dr. (UFSM)

Santa Maria, RS
2018

ABSTRACT

AN APPLICATION TO LOAD GEOMETRIC DATA FROM FROM RAILWAYS

AUTHOR: Elton Luiz Rasch
ADVISOR: João Carlos Damasceno Lima

Along the advancement of technology, new ways of monitoring sensor data are being proposed. In this work, we intend to explore a solution to load, store, retrieve and delete data that reflects the state of railroad tracks, which are collected from sensors installed in a control car. Initially, the nature of the data is analyzed and, considering that it is a series of temporal data, we set out to search for and test an alternative database management system. Thus, the tool "TimescaleDB" was found, which was promising initially, so to increase the performance of the operations of registration, removal and recovery of large volumes of temporal data. In addition, traditional tools for storing data, such as PostgreSQL, in versions 9.6 and 10 were explored. From its tenth version, an instrument to deal with this specific data type was made available, and it was also considered. After this, tests were conducted from the proposed solution, but did not show satisfactory results regarding the use of TimescaleDB, which led to the non-use of the tool.

Keywords: railway’s geometric data; time-series data; database.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Dados de 2014 mostram que o Brasil possui em torno de 30.129 km de linhas férreas. Além disso, há conexões ferroviárias com Argentina, Bolívia e Uruguai e, em seu auge, o país já possuiu 34.207 quilômetros. Porém, devido às crises econômicas e a falta de investimentos em modernização, tanto por parte da iniciativa privada como do poder público, e também ao crescimento do transporte rodoviário, grande parte dessa rede foi desativada. Aliado a isso, a pouca tecnologia nacional concernente a este meio de transporte faz com que o abandono da modalidade seja ainda maior, tendo em vista que as poucas empresas do meio necessitam recorrer a empresas estrangeiras, muitas vezes implicando em altos custos.

Parte deste problema diz respeito à falta de mecanismos adequados ao monitoramento da geometria das linhas férreas, sendo esta a principal motivação do presente trabalho. Nesse sentido, ele representa uma parte de um projeto maior, que visa fornecer uma solução de software e hardware para extrair e monitorar dados das linhas. Este projeto maior já está em andamento há vários anos, em uma parceria estabelecida entre a UFSM e a empresa Rumo Logística, a qual necessita de um meio eficiente para coletar dados referentes ao estado das linhas férreas sob sua concessão. Com a evolução das tecnologias, os problemas foram repensados, dando origem à novas possibilidades para aprimorar as soluções anteriormente executadas.

Para se ter uma ideia do cenário no qual o projeto se insere, é preciso compreender o fluxo dos dados até o software desenvolvido. Os dados são gerados a partir dos “carros controle”, um tipo especial de carro de linha munido de diversos sensores. Para ser mais específico, trata-se de 16 acelerômetros inerciais, 14 sensores de dados analógicos e 16 sensores de dados digitais. Além disso, o carro controle também é desprovido de sistemas de amortecimento, com vistas a dar maior precisão nos dados coletados. Um exemplo de carro controle pode ser visto na Figura 1.

Os dados gerados pelos sensores são disponibilizados em três tipos de arquivos e, através de softwares executados pelos próprios usuários, são gerados gráficos e relatórios. A partir do software desenvolvido neste trabalho, denominado “Import”, estes dados passam a ser carregados por um funcionário em um banco de

Figura 1 - Carro controle

Fonte: Canal CesarBlumenau - YouTube[1]

dados de um servidor disponível no próprio carro controle e, posteriormente, enviados por meio de uma conexão à uma central localizada na cidade de Curitiba - PR[2]. Os demais softwares continuam gerando relatórios e gráficos com base nesses dados, porém também passam a chamá-los do próprio banco de dados, ao invés de realizar a leitura dos arquivos, como ocorre no cenário atual[3].

Fundamentalmente, esses dados podem ser agrupados em uma categoria: dados temporais. A análise desse tipo de dados constitui assunto primordial para o trabalho, e pode ser vista com detalhes no capítulo 2.

Dado este panorama geral, é possível agora estabelecer o objetivo principal e também a parte considerada prática do projeto: a elaboração do Import. Já os objetivos específicos, constituíram-se de uma aproximação aos conceitos que permeiam a aplicação, em particular o de banco de dados, bem como alternativas que eventualmente pudessem ser utilizadas para aumentar o desempenho da aplicação.

---

[1] Disponível em: <https://www.youtube.com/watch?v=nCamG1mVfNk> Acesso: 05 de junho de 2018.
[2] Essa conexão poderá ser do tipo 3G, se disponível no momento da medição, ou Wi-Fi, quando estiver nas dependências da empresa.
[3] Um exemplo dos gráficos gerados a partir dos dados coletados pode ser visto no Anexo II.

As seções do presente trabalho estão estruturadas de modo que se forneça, nesta seção, um panorama da solução existente, a fim de delinear o problema que se pretende resolver. No capítulo 2, é realizado um levantamento teórico dos principais conceitos abordados durante a elaboração do software, em particular os conceitos envolvidos na otimização de bancos de dados que guardam séries de dados temporais e virtualização. A seguir, no capítulo 3, serão abordados os testes conduzidos para verificar a consonância das soluções disponíveis com o projeto. Já no capítulo 4, será abordado o detalhamento do software e, por fim, será fornecida uma conclusão do trabalho.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo serão abordados temas que permearam a concepção e a implantação da aplicação. Inicialmente, será trazida à tona a temática dos tipos de dados que a aplicação manipula (seção 2.1). Em seguida serão abordadas possíveis soluções para o armazenamento desses tipos de dados, a saber, a ferramenta TimescaleDB (seção 2.2) e o banco de dados PostgreSQL (seção 2.3). Por fim, ainda será abordada a utilização da técnica de virtualização (seção 2.4) e sua utilização no projeto.

2.1 DADOS TEMPORAIS E SGBDs

Todo sistema de monitoramento (onde também se encaixa o presente trabalho), deve antecipar o máximo de problemas possíveis. Fundamentalmente, Joshi aponta três benefícios centrais aos quais os desenvolvedores devem estar atentos, a saber: (1) a identificação de gargalos, (2) a capacidade que o sistema oferece e (3) a sua segurança e confiabilidade (JOSHI, 2012, p.4). Tendo em vista que a aplicação efetua a carga de dados para um banco de dados, é de suma importância efetuar a decisão correta na escolha do Sistema de Gerenciamento de Banco de Dados (SGBD) utilizado, tendo em vista a maximização destas características. Como tal, este deve possuir como funções primárias realizar a guarda segura de dados e retorná-los como resposta para as demandas de outras aplicações. Embora existam diversas opções disponíveis, a escolha se limitou dentre as opções que não representassem custos adicionais ao projeto, como a aquisição de licenças de uso, por exemplo.

Ainda de acordo com Joshi, podemos comparar soluções de softwares utilizando os seguintes critérios: funcionalidade, custo, adoção pelo mercado, suporte, manutenção, desempenho, escalabilidade, usabilidade, segurança, flexibilidade, interoperabilidade e implicações legais (JOSHI, 2012, p. 6). Entretanto, para o caso concreto aqui tratado, algumas destas características se sobrepõe às outras. Dentre os requisitos para a escolha, salienta-se a necessidade de ser um banco de dados estável e robusto, além de suportar relacionamento entre as tabelas, não implicar em custos adicionais (preferencialmente de código aberto).

Uma possibilidade inicial são bancos de dados orientados a documentos (NoSQL). Contudo, estes costumam apresentar pouco ou nenhum suporte à relacionamentos, e a implementação dessa característica precisa ser feita pelo próprio desenvolvedor do sistema, conforme aponta Solheim:

> "Comparado ao banco de dados relacional tradicional (SQL) um banco de dados orientado a documentos (NoSQL) tem pouco ou nenhum suporte à relações entre objetos. Um mecanismo de armazenamento NoSQL persiste e recupera documentos (muitas vezes no formato JSON) e qualquer relação entre seus documentos é necessário ser implementada por você mesmo. Sem qualquer suporte direto a relações, você se encontra ocupado com a lógica e a manutenção dos objetos e relações em sua camada de aplicação" (SOLHEIM, 2018, tradução nossa[4]).

Dentro das características destacadas é possível citar o PostgreSQL, considerado um dos SGBD bastante sólido e seguro atualmente. Dentre os recursos, podemos destacar: consultas complexas; chaves estrangeiras; integridade transacional; controle de concorrência; *triggers* e *views*. Além disso, trata-se de uma ferramenta de código aberto e conta com uma ampla comunidade de colaboradores, está disponível para os principais sistemas operacionais, e não há limitações para o uso comercial. Dentre as diversas interfaces disponíveis, a aplicação se utilizou da Libpq, que consiste em uma biblioteca que permite aos programas clientes escritos em C passarem consultas ao PostgreSQL.

Para o projeto, era necessário que o banco de dados pudesse lidar também com grandes quantidades de registros, tanto para inserção quanto para consulta e descarte dos mesmos. Nesse sentido, a técnica de particionamento fornece um incremento de desempenho bastante útil. De acordo com o próprio manual do PostgreSQL, essa técnica se refere à "dividir o que é uma grande tabela lógica em várias tabelas físicas menores"[5]. Assim o particionamento facilita o gerenciamento de grandes tabelas ou índices, permitindo o acesso e o gerenciamento de subconjuntos

[4] "Compared to a traditional relational database (SQL), a document oriented (NoSQL) database has poor or non-existent support for relations between objects (data schema). A NoSQL datastore persists and retrieves documents (often in JSON format) and any relationships between your documents is something you must implement yourself. Without any direct support for relations, you're stuck with the logic and maintenance of objects / relations in your application layer."

[5] https://www.postgresql.org/docs/9.1/static/ddl-partitioning.html (acesso: 15/05/2018)

de dados de forma muito mais rápida e eficaz, ao mesmo tempo em que mantém a integridade geral dos dados.

2.2 PARTICIONAMENTO

Basicamente existem dois tipos de particionamento: vertical e horizontal. No particionamento vertical, busca-se dividir uma tabela em várias tabelas que contêm menos colunas. Dentro deste, há dois tipos de particionamento: a normalização e divisão de linhas. A normalização é um o processo padrão e bastante difundido, utilizado para remover colunas redundantes de uma tabela e colocá-las em tabelas secundárias, vinculadas à tabela primária pela relação da chave primária e chave estrangeira. Assim, economiza-se principalmente espaço em disco, uma vez que colunas com valores nulos são eliminados. Já o particionamento de linhas divide a tabela original verticalmente em tabelas com menos colunas. Cada linha lógica em uma tabela dividida coincide com a mesma linha lógica das outras tabelas conforme é identificado por uma coluna UNIQUE KEY idêntica, em todas as tabelas particionadas.

Já o particionamento horizontal consiste em dividir uma tabela em várias tabelas, cada qual contendo o mesmo número de colunas, mas com um menor número de linhas. Por exemplo, uma tabela que contém 1 bilhão de linhas pode ser particionada horizontalmente em 12 tabelas, com cada tabela menor representando um mês de dados de um ano específico. Qualquer consulta, ao requerer dados de um mês específico, fará referência somente à tabela apropriada. Contudo, determinar como particionar as tabelas horizontalmente depende de como os dados são analisados. Como regra básica, busca-se particionar as tabelas de forma que as consultas façam referência ao menor número possível de tabelas.

Existem no mercado diversas soluções baseadas no particionamento horizontal, em particular, sistemas que realizam esse particionamento com dados oriundos de sensores e que possuem uma coluna de registro de *timestamp*, e são conhecidas como “*Time Series Databases*” (TSDB). Podemos defini-las como sistemas otimizados para lidar como séries de dados temporais, vetores de números indexados por tempo ou intervalos de tempo. Em alguns sistemas essas séries são conhecidas como “rastro”, “curvas” ou “perfis”. Por exemplo, ao registrar o log dos

valores de uma temperatura através do tempo, podemos chamar a série de rastro de temperatura (IBM, p. 5).

No manual do Informix, TSDB proprietário desenvolvido pela IBM, há ainda a distinção entre os tipos de dados temporais (IBM, p. 9). Por um lado, temos os regulares, que possuem um intervalo entre dados fixo, de modo a se comportarem de modo previsível. Nesse sentido, não é necessário armazenar os timestamps propriamente ditos, apenas um *offset* de um valor timestamp de referência. Por outro lado, séries de dados temporais irregulares são valores que podem ser gerados de modo arbitrário e não previsíveis, bastante comuns em sistemas orientados a eventos. Embora exista uma certa regularidade nos dados fornecidos pelos sensores dos carros-controle, não existe a certeza de que não haja interferência de qualquer tipo que possa afetar os sensores, e portanto, os dados deles oriundos serão tratados como irregulares.

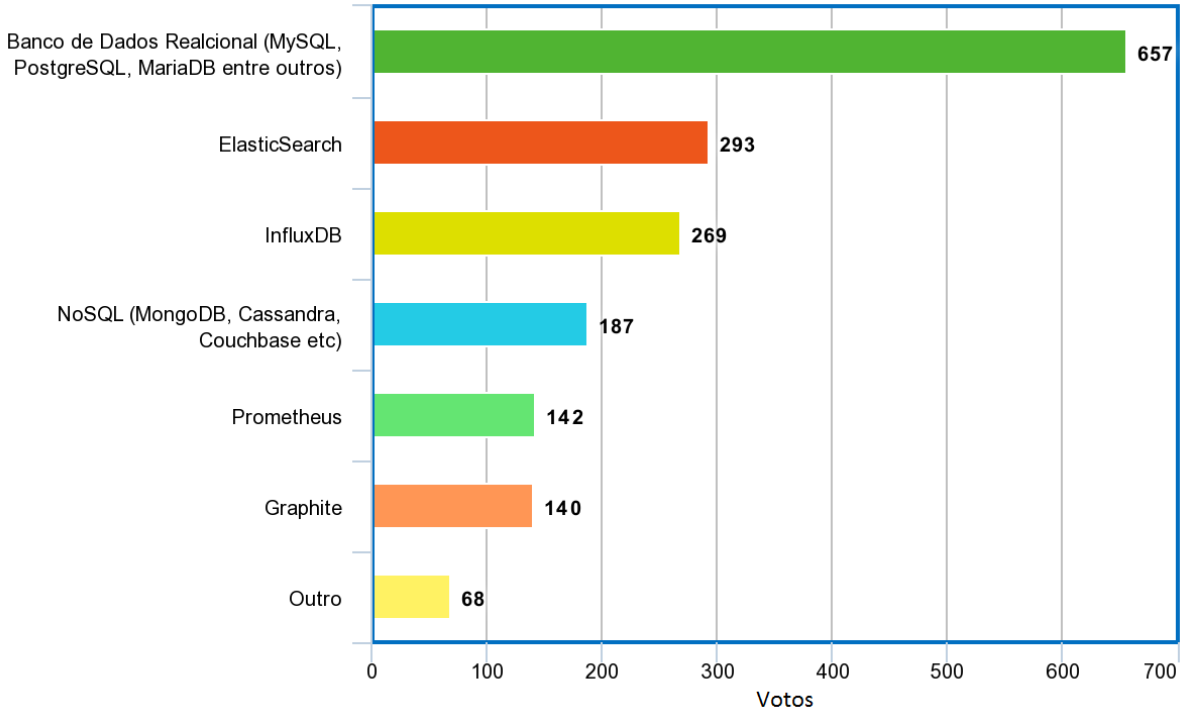
Tendo em vista a quantidade de registros tomados como entrada no Import, a utilização do particionamento horizontal se revelou bastante promissora, uma vez que foi possível dividir as tabelas com base no timestamp gerado no momento da coleta dos dados pelos sensores. Para se ter uma ideia do volume de dados, são coletados cerca de 10.000 registros por quilômetro, e a cada dia, o carro percorre cerca de 80 km coletando dados, gerando aproximadamente 36 MB de dados. Assim, são coletados cerca de 24.000.000 de registros no período de um mês.

Ademais, tendo em vista que a aplicação responsável por gerar os gráficos se utilizará de uma grande quantidade de registros, a depender do período selecionado, é importante que se traga para a memória primária o menor número de tabelas possíveis.[6]

Entretanto, ainda é necessário considerar os bancos de dados NoSQL. De fato, uma pesquisa realizada em 2017 mostrou que esse tipo de banco de dados é utilizado por cerca de 10% dos usuários quando se trata de soluções para dados baseados em timestamp (PERCONA, 2017). A pergunta realizada na pesquisa foi: “Qual banco de dados você está utilizando para armazenar suas séries de dados temporais”. O resultado pode ser visto na Figura 2:

[6] Um exemplo dos gráficos gerados pode ser visto no Anexo II.

Figura 2 - Distribuição dos votos dos participantes da pesquisa sobre tipos de SGBDs utilizados no armazenamento de séries de dados temporais.



Fonte: PERCONA, 2017 (adaptado).

Para saber porque tais ferramentas são utilizadas, precisamos pensar nos problemas relacionados aos demais SGBDs, em particular aos relacionais. Freedman, professor da Universidade de Princenton, faz uma análise desses motivos:

> "Tipicamente, o motivo para adotar bancos de dados de séries temporais NoSQL é a escala. Enquanto bancos de dados relacionais possuem muitos recursos que a maioria dos bancos de dados NoSQL não possui (suporte para índices secundários robustos, predicados complexos, uma linguagem de pesquisa rica, JOINS etc.), estes são difíceis de escalar". (FREEDMAN, 2017, tradução nossa[7]).

Assim, ele acredita que resolvendo o problema da escalabilidade não há mais razões para escolher sistemas NoSQL em ao invés dos relacionais. Basicamente, existem dois meios para escalar um banco de dados: (1) adicionar mais máquinas e/ou criar um cluster, e (2) aumentar a capacidade de armazenamento em uma

[7] "Typically, the reason for adopting NoSQL time-series databases comes down to scale. While relational databases have many useful features that most NoSQL databases do not (robust secondary index support; complex predicates; a rich query language; JOINs, etc), they are difficult to scale."

máquina. Como no projeto não existe a capacidade da aquisição de outras máquinas, o foco será na segunda opção.

A primeira constatação a ser feita em bancos de dados relacionais é a relação entre memória RAM (tipicamente mais cara) e o armazenamento (tipicamente mais barato). Ocorre que nos bancos de dados relacionais costumeiramente uma tabela é armazenada como uma coleção de páginas de tamanho fixo, sobre as quais são montadas árvores do tipo B para indexar estes dados, facilitando a localização dos mesmos. Quando tanto os dados quanto as árvores são pequenos, é possível armazenar ambos na memória RAM. Contudo, quando se trata de um grande volume de dados, é provável que estes ultrapassem a capacidade da memória RAM, de modo que apenas partes sejam carregadas de cada vez. Assim, há um aumento nas operações de entrada/saída, uma vez que para cada carregamento parcial de dados, há uma recorrência ao armazenamento secundário, uma operação conhecida como *SWAP* (Molina, p. 28). Esse acesso ocorre resgatando blocos de dados, os quais podem ser customizados quanto ao tamanho, para que se busque mais ou menos blocos por acesso[8]. Porém, mesmo para resgatar um único registro, um bloco inteiro será trazido à memória primária.

Para resolver este problema, a maioria dos bancos de dados NoSQL utiliza árvores LSM (*Log-Structured Merge Tree*), as quais apenas inserem ou resgatam páginas inteiras do armazenamento secundário. Na inserção, por exemplo, ao invés de registrar pequenos blocos várias vezes, a informação fica contida em sua totalidade em uma tabela na memória primária, e esta é enfim armazenada na memória secundária como uma tabela de strings (*Stored String Table* ou SST). Similarmente, no resgate de informações, também são geradas tabelas inteiras na memória principal.

Contudo isto traz à tona dois problemas. O primeiro, é que haverá uma exigência muito maior de memória primária. Diferentemente das árvores B, nas árvores LSM não há um índice global para prover uma ordem sobre todas as chaves, e assim procurar um valor se torna muito mais complexo. Em primeiro lugar, é necessário checar na tabela da memória primária pela última versão da chave e, caso

[8] Embora modificar o tamanho dos blocos possa parecer útil, isso afeta principalmente duas coisas: o tempo de busca para localizar os blocos e a fragmentação dos dados, embora este último não seja muito impactante quando há o uso de SSDs, e mesmo estes tipicamente possuem seus próprios tamanhos de blocos.

não seja encontrada ali, é necessário procurar nas tabelas da memória secundária. Novamente, para evitar constantes operações de entrada/saída para acessar a memória secundária, índices sobre as tabelas SST são mantidos na memória. O segundo problema, é um suporte mais degradado para índices secundários, uma vez que não há uma ordenação global. A solução fornecida por alguns sistemas consiste em duplicar os dados em uma ordenação diferente, resultando em uso exagerado de memória secundária.

Nesse sentido, é necessário buscar alternativas para contornar estes problemas.

2.3 PARTICIONAMENTO PARA SÉRIES TEMPORAIS

Se voltarmos aos bancos de dados relacionais podemos notar que, em sua maioria, estes são empregados para transações OLTP, isto é, operações que realizam atualizações nos registros, os quais podem estar em qualquer lugar das tabelas (MOLINA, 2000, p. 613). Operações desse tipo, tipicamente envolvem alterar porções muito pequenas de registros. Entretanto, a natureza de dados temporais, como os que são tratados no projeto, costuma envolver apenas operações de escrita, i.e., inserções nas tabelas. No caso concreto, leituras e remoções também serão executados. Entretanto, operações de atualizações em registros serão praticamente inexistentes. Em geral, a atividade consiste unicamente na inserção de registros baseados em uma marcação temporal.

Assim, é possível contrastar os dois tipos apontados por Molina na Tabela 1:

Tabela 1 - Principais diferenças entre Bancos de Dados OLTP e Timestamp

Escritas OLTP	**Escritas baseadas em Timestamp**
Dominadas por Atualizações de registros	Dominadas por Inserções de registros
Recebem ids de modo randômico	Recebem ids de intervalos recentes
Transações sobre múltiplas chaves primárias	Ids associados tanto com chaves primárias quanto com timestamps

Fonte: Elaborado pelo autor

Essas características abrem uma oportunidade para que possamos obter vantagens a fim de solucionar os problemas de escalabilidade. Quando as abordagens anteriores tentaram evitar escritas no disco, estas estavam tentando eliminar o problema de atualizações de registros de forma randômica, característica intrínseca dos sistemas OLTP. Entretanto, como estabelecido acima, séries de dados temporais funcionam de modo diferente: escritas são primariamente inserções, e não atualizações, e costumam ser sequenciais ao invés de randômicas. Em outras palavras, séries temporais apenas adicionam dados ao final dos conjuntos de dados, possibilitando uma ordenação temporal dos mesmos.

Prima faccie, os benefícios seriam os mesmos de simplesmente adicionar um índice baseado no tempo, entretanto, uma vez que quiséssemos outro índice (por exemplo, por id do carro controle), ou mesmo um índice secundário, esta abordagem nos levaria de volta ao problema de inserções randômicas em árvores do tipo B para este novo índice.

A saída encontrada pelos sistemas TSDB consiste em, ao invés de apenas indexar os dados por tempo, construir tabelas particionando os dados em duas dimensões: um intervalo de tempo e uma chave primária. Estas duas partições são conhecidas como *Chunks*, e são armazenadas em forma de tabelas. Desta forma, na etapa de planejamento de consultas, é possível imediatamente indicar quais Chunks devem ser acionadas, dado que o tamanho destas são conhecidas pelo analisador. Isso vale tanto para inserções quanto para remoções e consultas.

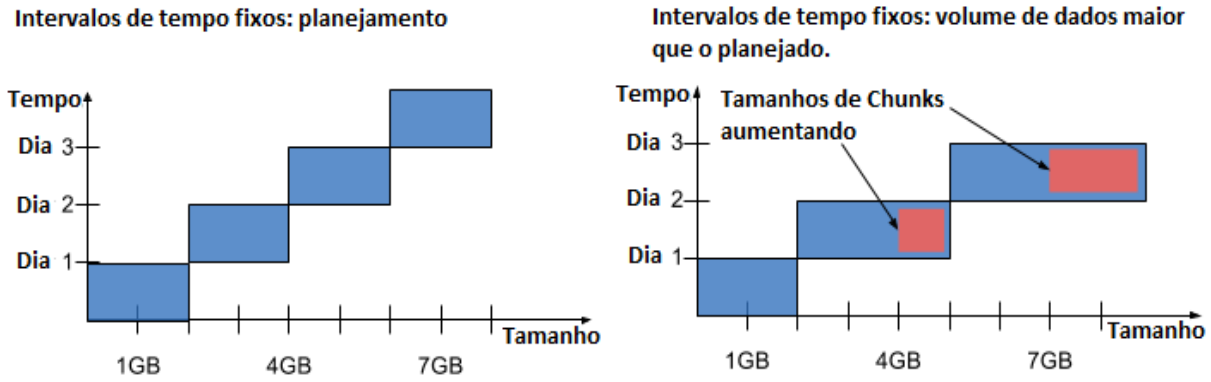
O benefício principal é que ao invés de construir índices referentes a toda tabela, estes podem ser construídos sobre os *Chunks*, que são muito menores, possibilitando que, caso sejam dimensionados adequadamente, estes cabem inteiramente na memória principal, evitando o *Swap* além de manter o suporte à índices secundários.

Basicamente, existem três maneiras de dimensionar os *Chunks*.

A primeira delas consiste em estabelecer *Chunks* com durações de intervalo fixo. Nesta abordagem todos os Chunks irão armazenar registros contidos em um único intervalo preestabelecido, por exemplo, 1 dia. Isso funciona muito bem quando o volume de dados coletados por intervalo não muda. Contudo, na medida que novos geradores de dados (tipicamente sensores) são inseridos, mais dados são direcionados para um mesmo *Chunk*, o que pode levar ao ressurgimento do *Swap*, conforme ilustrado na Figura 3. Por outro lado, escolher intervalos muito reduzidos

também pode ser ruim, uma vez que haverá tabelas em demasia para serem analisadas durante uma consulta.

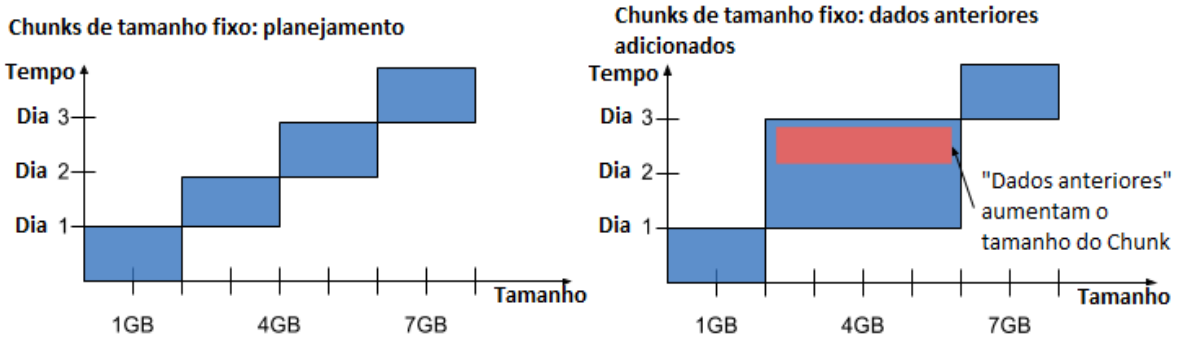
Figura 3 - Aumento do tamanho dos Chunks devido ao aumento do volume de dados no período.



Fonte: adaptado de FREDMAN, 2017

Na segunda abordagem, todos os *Chunks* possuem um tamanho preestabelecido, por exemplo 1GB. Assim, cada *Chunk* receberá dados até que sua cota esteja preenchida, momento no qual seus intervalos de tempo são selados. Nesta abordagem, o principal problema está relacionado com sincronização, pois se os dados chegarem fora da ordem temporal (devido a uma conexão intermitente, por exemplo), serão escritos em diferentes *Chunks*. Uma solução sugerida foi a de relaxar o tamanho originalmente estabelecido. Contudo, no caso dos dados iniciais de um *Chunk* serem de um período distante, dados anteriores que seriam subsequentes serão escritos num mesmo *Chunk*, aumentando seu tamanho demasiadamente, conforme ilustração da Figura 4:

Figura 4 - Aumento do tamanho dos *Chunks* devido a adição de informação anterior.



Fonte: adaptado de FREDMAN, 2017

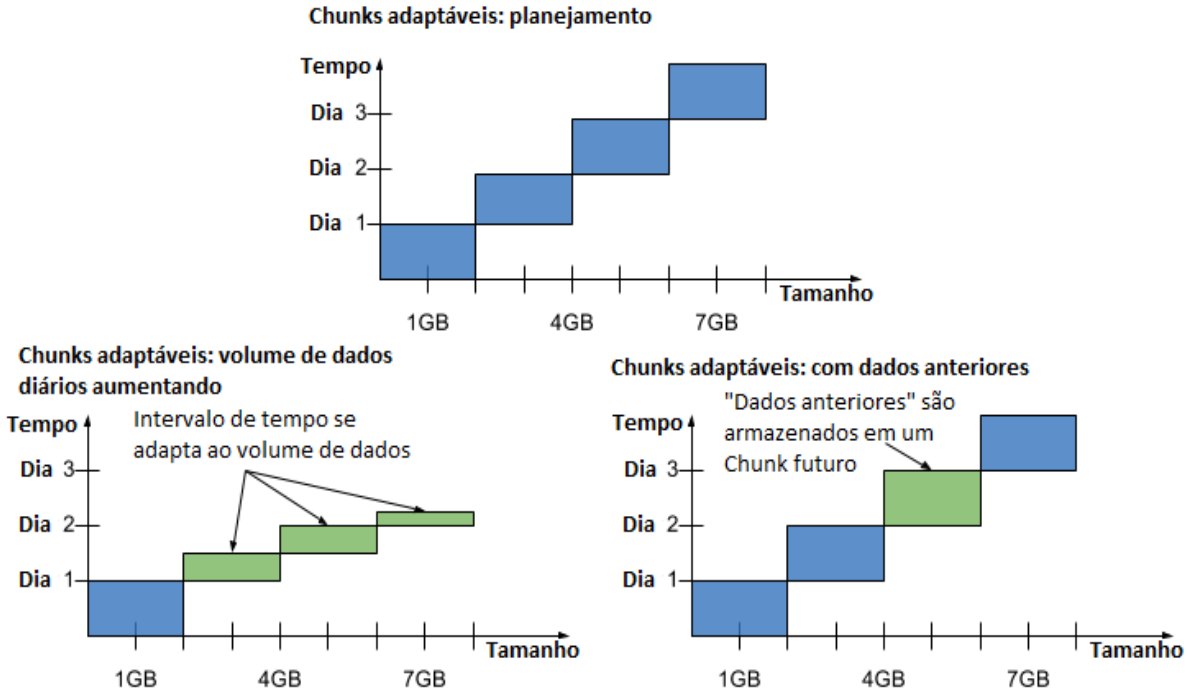
Assim, a terceira alternativa geralmente é adotada: *Chunks* adaptáveis. Nessa abordagem, os *Chunks* são criados com um intervalo fixo, mas esse intervalo se adapta de *Chunk* para *Chunk* baseado nas mudanças de volumes de dados.

Ao evitar intervalos fixos, assegura-se que os dados iniciais que chegam não irão criar intervalos muito longos que futuramente levarão à *Chunks* muito grandes. Isso também contribui de modo significativo para operações de exclusão de registros, uma vez que basta fazer uma operação de exclusão sobre o *Chunk* (que em realidade são tabelas, como dito acima). Isso significa que arquivos que estão sob o sistema de arquivos podem ser apagados, ao invés de excluir tuplas individualmente, o que iria requerer que porções do arquivo fossem apagadas ou mesmo invalidadas em operações de atualizações de registros. Assim, evita-se a fragmentação de arquivos do banco de dados, o que elimina a necessidade de operações de manutenção conhecidas por “vacuum”, as quais costumam ter um custo especialmente elevado em tabelas de grande porte.

Essa abordagem garante que os *Chunks* sejam dimensionados apropriadamente, de modo que sempre possam estar contidos na memória, mesmo que o volume dados mude. O particionamento através de chave primária então pega cada intervalo de tempo posterior e o divide em um número de *Chunks* menores, os quais dividem o mesmo intervalo de tempo mas são localizados em termos de chave identificadora. Além disso, ainda se obtém a vantagem de uma paralelização aprimorada, tanto em servidores com múltiplos discos (para inserção e seleção) bem como em múltiplos servidores. A ilustração das *Chunks* adaptáveis pode ser vista na Figura 5.

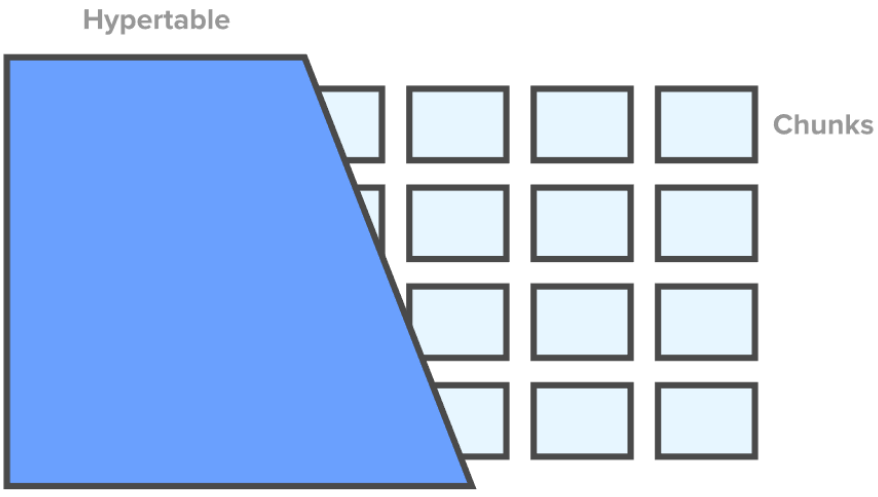
Esta solução é adotada, por exemplo, pelo TimeScaleDB, um banco de dados otimizado para séries temporais. Ele fornece uma interface de uma tabela única e contínua, que atua sobre todas as séries temporais, mesmo que estas estejam divididas em vários *Chunks*. Esta tabela é conhecida por “Hypertable”. Assim, é possível tratar esta tabela como uma tabela padrão do PostgreSQL, tanto para consultas quanto para inserções, remoções e mesmo atualizações de registros. Assim, quando um usuário insere uma nova tupla, o TimeScaleDB se assegure que esta seja inserida no *Chunk* adequado, de modo transparente ao usuário. Da mesma forma quando há consultas: o TimeScaleDB irá se assegurar de que a menor quantidade possível de *Chunks* sejam trazidos à memória. A Figura 6 ilustra esse mecanismo.

Figura 5 - Chunks adaptáveis respondem ao crescimento do volume de dados e à adição de “dados anteriores”.



Fonte: adaptado de FREDMAN, 2017

Figura 6 - *Hipertable* gerada pelo TimeScale



Fonte: TIMESCALEDB, 2018.

Em um mesmo banco de dados pode haver várias Hypertables. Em cada qual pode haver uma ou mais dimensões, e sua há a promessa de um grande ganho de desempenho:

> “TimescaleDB aprimora o desempenho e armazenamento ao particionar de Hypertables de forma transparente através de multiplas dimensões: através de um intervalo de tempo e, opcionalmente, por uma ou mais colunas de seus dados (por ex., identificadores de sensores, localizações, clientes, usuários), utilizando ambos os intervalos e particionamento de hash. Cada partição é chamada de chunk, que é automaticamente criada pelo sistema sem interações administrativas. O sistema pode escalar para vários desses chunks – facilmente lidando com 10000 chunks em um único nódulo – e esses chunks podem estar espalhados de forma transparentes em vários discos, aumentando a capacidade”. (TIMESCALEDB, 2018, tradução nossa[9]).

Outro aspecto relevante está na forma com que consultas são geradas. O TimeScaleDB se integra diretamente com o planejador de consultas do PostgreSQL bem como no mecanismo de execução, promovendo otimizações que incluem a exclusão de constraints tanto nas dimensões temporais quanto espaciais, visando minimizar o número de *Chunks* acessadas, além de introduzir otimizações especiais para as clausulas “GROUP BY”, “ORDER BY” e “LIMIT”. Além disso, se iguala a maioria dos bancos de dados SQL disponíveis, no sentido de suportar JOINS, índices secundários, subconsultas, agregações e ordenações.

2.4 PARTICIONAMENTO COM POSTGRESQL 10

A ideia de particionar tabela de dados é útil para dividirmos uma grande tabela em partições (ou tabelas) menores, de forma a tornar as consultas de geração de relatório e estatísticas menos onerosas para o banco de dados.

Até a versão 10, particionar tabelas no PostgreSQL era um pouco mais complexo e trabalhoso que em SGBDs como SQL Server e Oracle, pois não havia um recurso nativo do banco de dados e sim uma adaptação de recursos disponíveis para esse fim, através da utilização de heranças e gatilhos. Entretanto, com a versão 10, foi acrescentado um recurso nativo para este fim (ANVESH, 2017).

---

[9] TimescaleDB scales performance and storage by transparently partitioning hypertables across multiple dimensions: by a time interval, and optionally by one or more additional columns in your data (e.g., device identifiers for sensor data, symbols in tick data, locations, customers, or users), using both interval and hash partitioning. Each such partition is called a chunk, which is automatically created by the system without administrative interaction. The system scales to many such chunks – easily handling 10,000s of chunks on a single node at scale – and such chunks can be transparently spread across many disks to scale up capacity.”

Nesta versão, é oferecido suporte à dois tipos de particionamento horizontal: intervalo e lista. O particionamento de intervalo literalmente divide as tabelas em intervalos, definidos por uma ou mais colunas-chave, sem sobreposição dos intervalos designados para cada partição. Já no particionamento por lista a tabela é dividida pela listagem explícita das chaves que aparecerão em cada partição. Há ainda métodos alternativos que funcionam através do uso de heranças em conjunto com views “UNION ALL”, porém com perdas de desempenho e de benefícios trazidos pelos métodos declarativos implementados.

Para criar um particionamento no PostgreSQL 10, inicialmente é necessário criar uma tabela particionada e junto com ela definir o método (por exemplo, intervalo) e uma lista de colunas que serão utilizadas como as chaves das partições. Após, é necessário criar as partições propriamente ditas e, por fim, os índices.

A tabela particionada funciona de forma análoga a uma tabela “master” no esquema de herança de tabelas, porém de forma melhorada, servindo como uma interface mais poderosa para as tabelas de partição filhas. Por exemplo, comandos como os de *truncate* e *copy* são propagados para as tabelas filhas através da execução na tabela de partição. Além disso, usuário podem inserir dados nas tabelas filhas a partir da tabela particionada, uma vez que estes são automaticamente alocados nas partições correspondentes durante uma inserção de registro, de modo que se dispense os gatilhos (i.e., *triggers*) necessários antes da versão 10, quando também se utilizava o esquema de herança.

Como resultado, todo o processo fica mais direto e automatizado para o usuário.

Entretanto, o particionamento declarativo é apenas o primeiro passo dessa nova abordagem do PostgreSQL. Versões futuras podem incluir, por exemplo, a possibilidade de inserir dados nas próprias tabelas “pais”. Até o momento, trata-se basicamente de uma melhoria de interface e sintaxe para definir as partições.

2.5 VIRTUALIZAÇÃO

Tanto os testes realizados quanto a aplicação em si executam sobre máquinas virtuais. A principal vantagem da virtualização é a portabilidade que ela oferece, de modo que seja possível executar o mesmo programa em ambientes de hardware

bastante distintos. Essa característica advém da própria definição de máquina virtual: “uma duplicata eficiente e isolada de uma máquina real” (LAUREANO, 2006, p. 17). A virtualização de servidores também traz diversas vantagens para empresas, entre as quais podemos destacar: (a) a redução de downtime: eliminação de paradas de ambiente de produção; Redução do tempo de downtime com virtualização de servidores; Prevenção de perda de dados; Prevenção de downtimes não planejados. (b) Automação e gerenciamento: Sistemas de gerenciamento centralizado de máquinas virtuais com interface amigável e intuitiva; Gerenciamento de ambiente de produção e homologação; A Virtualização de Servidores proporciona Gerenciamento de implantação; Gerenciamento de atualização de versões de softwares e firmwares. (c) Otimização da Infraestrutura: Pesquisas apontam que a utilização de servidores convencionais é em torno de 5 – 20%, através da virtualização essa taxa fica em torno de 65% - 90%; Maior ROI; Redução de até 40% de custo operacional; Com Virtualização de Servidores é obtido menor TCO (*Total Cost Ownership*)[10] de servidores; Melhor Gerenciamento; Otimização de infraestrutura, espaço físico e maximização da utilização de recursos; Redundância em caso de falha de Hardware, Virtualização de Servidores, o ambiente virtualizado migra as máquinas virtuais para os demais servidores virtualizados.

Contudo, também há desvantagens. Primeiramente, podemos citar o grande uso de espaço em disco, já que é preciso de todos os arquivos para cada sistema operacional instalado em cada máquina virtual. A seguir, há uma dificuldade caso o software necessite acesso direto a hardware. Além disso, também é constatado um aumento na utilização de memória RAM e uma dificuldade um pouco maior no dimensionamento gerenciamento de recursos.

Dentre as várias opções de virtualização disponíveis no mercado, podemos citar três em especial: VirtualBox, VMWare e Docker. As duas primeiras oferecem a possibilidade de virtualizar todo o sistema operacional, enquanto o Docker nos permite separar as aplicações da infraestrutura, de modo a possibilitar uma entrega de software acelerada. Assim, ele também consegue ser consideravelmente mais leve, já que não necessita fornecer todos as funções de um sistema operacional completo. Por esse motivo, foi o escolhido para rodar no servidor do carro controle, e sobre ele, o Import.

[10] O Custo total de (*hardware* + *software*) de um servidor.

De um modo geral, podemos dizer que o docker nos fornece a possibilidade de empacotar e executar uma aplicação em um ambiente isolado, conhecido como “container”. O isolamento e segurança nos permitem executar vários containers simultaneamente em um determinado host, uma vez que são muito mais leves que as alternativas e rodam diretamente no *kernel* do sistema hospedeiro. Além disso, o Docker ainda proporciona uma plataforma para gerenciar o ciclo de vida dos containers: é possível desenvolver a aplicação utilizando containers, e quando esta estiver concluída, basta implementá-la no ambiente de produção como o próprio container.

Docker funciona como uma aplicação do tipo “cliente-servidor”, e tem como principais componentes:

- Um servidor que comanda os dockers (daemon process);
- Uma API do tipo REST que especifica uma interface, a qual os programas utilizam para se comunicar com o daemon[11];
- Um interface de linha de comando (CLI) do cliente, i.e., do docker.

Assim, o principal ganho para o projeto se dará por conta da portabilidade e facilidade de implementação, uma vez que para qualquer correção de erro ou nova funcionalidade bastará substituir o container existente pelo container com a correção ou funcionalidade.

[11] A Representational State Transfer (REST), em português Transferência de Estado Representacional, é um estilo de arquitetura que define um conjunto de restrições e propriedades baseados em HTTP. Web Services que obedecem ao estilo arquitetural REST, ou web services RESTful, fornecem interoperabilidade entre sistemas de computadores na Internet. Os web services compatíveis com REST permitem que os sistemas solicitantes acessem e manipulem representações textuais de recursos da Web usando um conjunto uniforme e predefinido de operações sem estado. Outros tipos de web services, como web services SOAP, expõem seus próprios conjuntos arbitrários de operações.

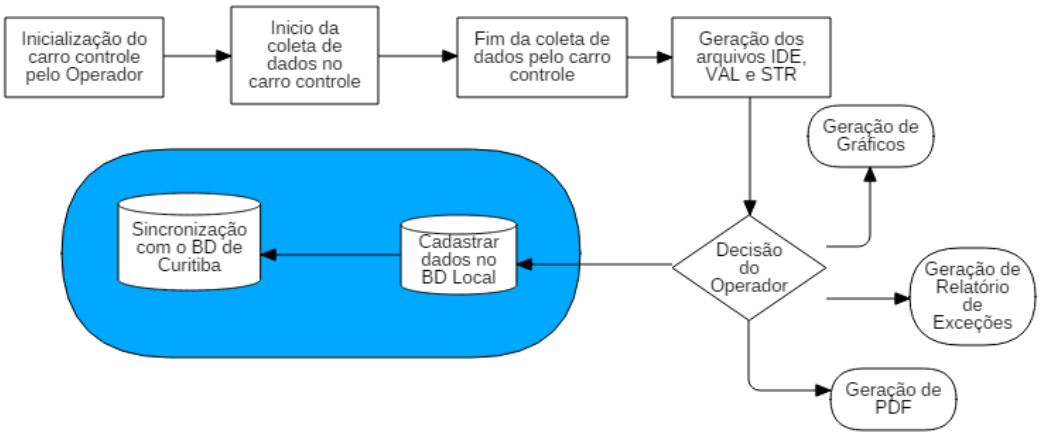
3 IMPLEMENTAÇÃO DO IMPORT

Neste capítulo trará um resumo do que foi a implementação do Import. No contexto do projeto, este software em específico se revela importante na medida em que os demais dependem deste, uma vez que buscam informações do banco de dados. Na realidade, é possível pensar em soluções que não envolvam bancos de dados, ou mesmo bancos de dados NoSQL, como os vistos no Capítulo 2. Entretanto, estas não são soluções elegantes e de acordo com as tecnologias disponíveis atualmente.

Para inserir os dados no Import, o usuário logado acessa a plataforma principal enviando os arquivos. A seguir a aplicação principal chama o Import em segundo plano, passando os parâmetros necessários. Ao usuário, é fornecida uma mensagem indicando que um relatório lhe será enviado ao final do processamento dos arquivos. Ao final da execução é fornecido um relatório para o usuário, contendo a quantidade total de registros enviados e a quantidade de registros não inseridos e inseridos.

Entretanto, como este processo de carga de dados para o servidor não é automatizado, e o volume de dados é considerável, é desejável que um usuário efetue a carga dos mesmos no servidor de forma automatizada, e é exatamente este o programa alvo do presente estudo. Um fluxograma com cada uma das etapas descritas pode ser visto na Figura 7, onde a parte em azul compreende tanto o resultado da implantação do “Import”, quanto a sincronização com o banco de dados remoto, ainda a ser desenvolvida.

Figura 7 - Fluxograma das etapas do processo de geração e carga de dados



Fonte: Elaborado pelo autor.

Para implementar esta automatização, criou-se o “Import”, uma aplicação que lê os arquivos gerados pelos sensores e os armazena em um banco de dados no servidor do carro controle. Estes arquivos são gerados após um processamento (cálculo e ajuste) dos dados brutos oriundos dos sensores e podem possuir uma das seguintes extensões: IDE, VAL ou STR. Assim, é trabalho do operador inserir os dados contidos nesses arquivos para o banco de dados, informando ao Import em qual diretório os mesmos estão contidos.

Nos arquivos IDE, mapeados para a tabela “medicao”, são encontrados dados de caráter mais geral das aferições, como os referentes à linha, hora e equipe responsável, dando origem às seguintes colunas: local de origem, local de destino, marco inicial, distância inicial, marco final, distância final, data da medição, hora da medição, sequência da medição, classe, equipe (até quatro pessoas), observação (até 4 itens)[12]. Além destes, durante a modelagem também foram consideradas as colunas “usuário” – tendo em vista que o sistema também guardará o nome de quem realizou a importação para o banco – e “id medicao”, a qual será referenciada pela tabela “valor”.

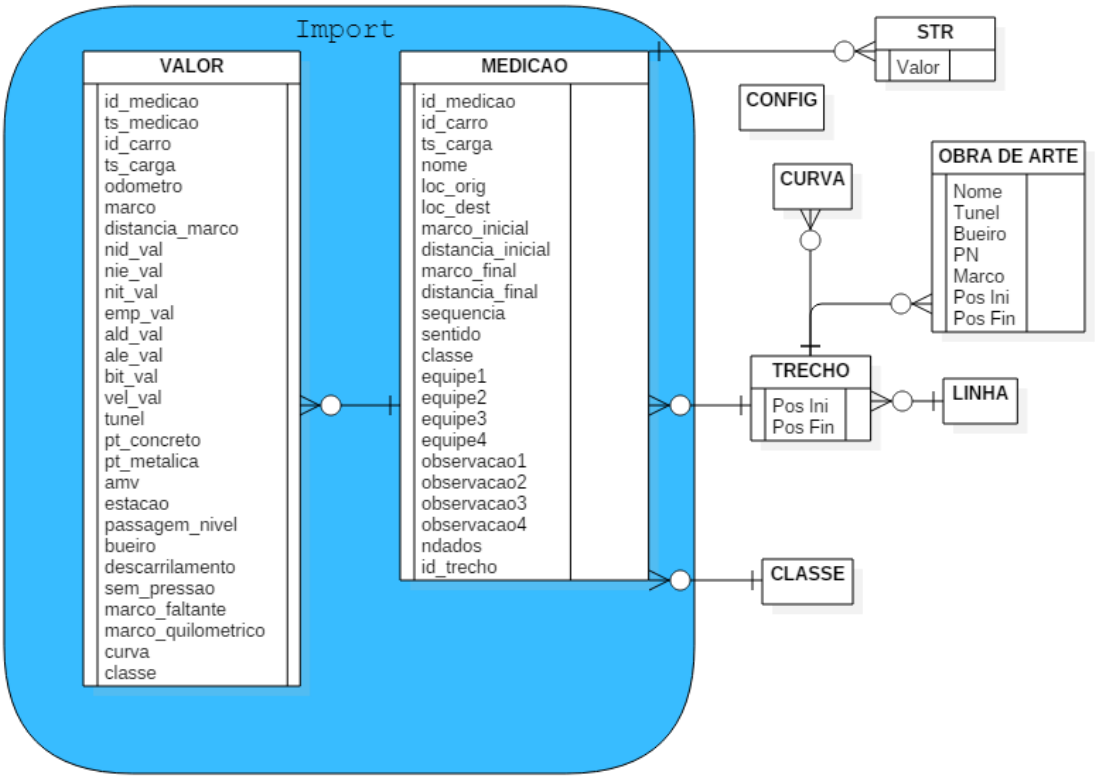
Já nos arquivos VAL estão os dados que contém os valores das marcações propriamente ditas, e foram mapeados para a tabela “valor” e odômetro, marco, distância do marco, NID (Nivelamento Longitudinal Direito), NIE (Nivelamento Longitudinal Esquerdo), empeno, alinhamento (direito e esquerdo), bitola (distância entre os trilhos, velocidade, túnel (registro de presença de), ponte de concreto, ponte metálica, AMV (Aparelho de Mudança de Via), estação, passagem de nível, bueiro, descarrilamento, sem pressão (Indica se houve alívio da pressão no rodado de medição para, por exemplo, passar por um AMV), marco quilométrico, marco faltante (local onde deveria existir um marco quilométrico) e curva (se o carro está percorrendo uma curva ou não). Esta tabela se encontra em um relacionamento do tipo “1 para N” com a tabela “medicao”, uma vez que para cada arquivo de medição pode haver diversos registros no arquivo de valores.

Por fim, os arquivos STR trazem informações a respeito de exceções ocorridas na leitura dos dados, os quais serão representados por uma coluna text na tabela de medição. O esquema Entidade-Relacionamento gerado por esta configuração de

[12] A definição de alguns destes itens pode ser vista no Anexo A, ao final do trabalho.

arquivos representa o mapeamento destes dados nas tabelas do banco de dados, e pode ser visto na Figura 8:

Figura 8 - Esquema ER do programa “Import”.



Fonte: Elaborado pelo autor.

Inicialmente, o programa define várias opções de configuração, como pode ser visto na Figura 9:

Figura 9 - Opções de configuração pré-definidas

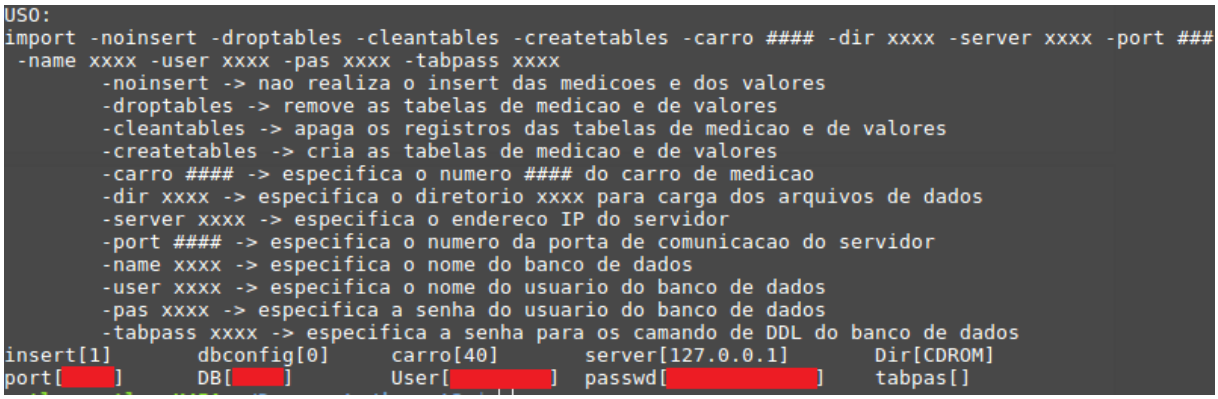
```
#define PORT          26543
#define DIRSIZE       8192
#define MSG_SIZE      8192
#define SQL_SIZE      8192
#define DIRETORIO     "TESTE"       // Diretório Padrão, se não for fornecido como argumento
#define DBSERVER      "127.0.0.1"   // Endereço do baco de dados
#define DBPORT        5432          // Porta do banco de dados
#define DBNAME        "TESTE"       // nome do banco de dados
#define DBUSER        "TESTE"       // nome do operador
#define DBPASSWD      "TESTE"       // senha do operador
```

```
#define DBTABPASS   "TESTE"         // senha usada como parâmetro para verificar os comando
de tabelas
#define DBCARRO       40            // carro
```

Fonte: Elaborado pelo autor

Alguns destes, podem ser passados via parâmetro. As opções, nesse caso, são exibidas quando executamos a aplicação sem passar parâmetro algum, como mostra a Figura 10:

Figura 10 - Parâmetros possíveis do Import



Fonte: Elaborado pelo autor.

A seguir, caso sejam passados argumentos para alteração no banco de dados, tais como criação ou esvaziamento de tabelas, estas são processadas. Ocorre uma tentativa de conexão com o banco e, caso esta não seja bem sucedida, é retornada uma mensagem ao usuário informando o motivo. Do contrário, as alterações são processadas e o programa segue seu fluxo.

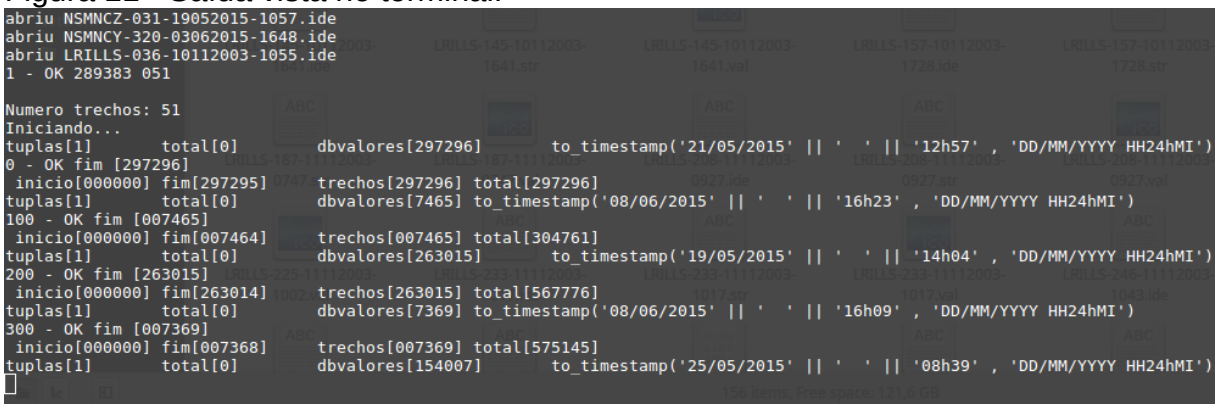
Prosseguindo, o diretório informado é varrido por uma função de busca dos arquivos VAL, IDE e STR. Caso nenhum diretório tenha sido informado, o diretório padrão é vasculhado e há tentativas de criação de trechos para cada arquivo. Caso seja um trecho válido, este é armazenado em outra estrutura que constitui o retorno da função.

Finalizada esta fase, é iniciada a fase de carregamento dos trechos do conjunto propriamente dita. Nesta função, inicialmente é feita a verificação se a tupla já está presente na tabela de medição, e caso esteja, a operação é abortada. Tanto a verificação quanto a inserção são realizadas com o autocommit desligado. Caso não tenham ocorrido falhas, é retornado o sinal positivo.

Caso a função de inserção de medição tenha sido bem sucedida, é chamada a função de inserção dos valores. Nela, inicialmente é checada a integridade dos dados referentes às posições dos trechos. Caso haja problemas, a operação é abortada e as posições com problemas são informadas ao usuário. Caso não haja, é verificado, inicialmente, se já existem valores para a medição corrente e caso estes já existam, é possível que tenha havido alguma falha em uma importação anterior, de modo que estes valores são apagados, para evitar problemas de chaves duplicadas. Por fim, é criada e executada a consulta que realiza a inserção na tabela de valores. Nessa operação, é utilizado um *commit* por arquivo. Por fim, é incrementado um contador de valores inseridos, e transmitido ao usuário uma mensagem contendo um pequeno relatório das inserções do trecho.

Ao final, a conexão com o banco de dados é fechada e a estrutura contendo o conjunto de trechos é removida da memória. Um exemplo da saída fornecida pode ser visto na Figura 11.

Figura 11 - Saída vista no terminal.



Fonte: elaborado pelo autor.

Nesta imagem, é possível ver o total de trechos, e o total de valores inseridos, bem como os arquivos que foram abertos com sucesso e os que geraram algum tipo de erro. Salienta-se que a aplicação é executada em segundo plano, de modo que o usuário não se encontra impedido de realizar outras ações durante o período de processamento. Ao final do upload dos arquivos VAL, IDE e STR, um email é enviado ao usuário contendo um relatório de envio.

Já ao final do processamento dos arquivos, outro email é enviado contendo os gráficos gerados a partir dos arquivos. Um exemplo destes gráficos pode ser visto no

Anexo II, enquanto um exemplo tanto do relatório de envio, quanto da interface de envio pode ser visto no Anexo III.

Uma vez que se trata de uma aplicação de uso intensivo, é necessário adequá-la às necessidades ou mesmo desenvolver funcionalidades ainda não pensadas, a fim de tornar a sua utilização a mais eficaz possível. Pensando nisso, elaborou-se um questionário que será fornecido aos usuários do Import, tendo em vista a validação da aplicação e a consequente melhoria da mesma. As questões podem ser vistas a seguir:

1 - Numa escala de 0 a 10, onde 0 significa nada seguro e 10 significa totalmente seguro, marque o quanto você considera o software seguro:
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

2 - Numa escala de 0 a 10, onde 0 significa muito difícil e 10 significa muito fácil, marque o quanto você considera o software fácil de utilizar:
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

3 - Se você pudesse realizar mudanças para facilitar sua utilização, quais mudanças você faria?

4 TESTES

Para averiguar a viabilidade do TimeScaleDB no caso do Import, alguns testes foram conduzidos e serão descritos neste capítulo.

A máquina utilizada para os testes consistiu em um processador Intel Core i7, modelo 8700, que conta com 6 núcleos funcionando a 4.3 Ghz, 16 GB de memória RAM do tipo DDR 4, funcionando a 2400 Mhz, e um SSD Kingston com capacidade de 120 GB, modelo UV400, conectados a uma placa-mãe MSI modelo H310M. Sobre esta configuração, foi instalado o sistema operacional Windows 10, na versão LTSB, e sem qualquer alteração no mesmo, foi instalado o aplicativo de virtualização VirtualBox. Nesse aplicativo, foi criada uma máquina virtual (VM), para a qual foram alocados 8 GB de memória e 4 núcleos de processamento, e instalado o sistema operacional Debian, na versão 9.4. Esta máquina foi replicada três vezes, tendo em vista a intenção da realização de testes com diferentes versões do bancos de dados Postgres (versão 10 e 9.6) além da própria ferramenta TimescaleDB, com ambos as versões do bancos de dados.

Inicialmente, nas VMs contendo o TimescaleDB, criou-se um banco de dados tal como indicado no próprio site da ferramenta, i.e., um banco de dados com uma única tabela, chamada de "conditions", conforme as Figuras 12 e 13:

Figura 12 - Criação do Banco de Dados.

```
-- Create the database, let's call it 'tutorial'
CREATE database tutorial;

-- Connect to the database
\c tutorial

-- Extend the database with TimescaleDB
CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
```

Fonte: Projeto TimescaleDB, site GitHub[13].

[13] Disponível em: <https://github.com/timescale/timescaledb/blob/master/README.md>. Acesso em: 23 de Junho de 2018.

Figura 13 - Criação da tabela.

```
-- We start by creating a regular SQL table

CREATE TABLE conditions (
  time        TIMESTAMPTZ       NOT NULL,
  location    TEXT              NOT NULL,
  temperature DOUBLE PRECISION  NULL,
  humidity    DOUBLE PRECISION  NULL
);
```

Fonte: Projeto TimescaleDB, site GitHub[14].

A seguir, a *Hypertable* foi gerada, conforme a Figura 14:

Figura 14 - Criação da *Hypertable*

```
-- This creates a hypertable that is partitioned by time
--   using the values in the `time` column.

SELECT create_hypertable('conditions', 'time');

-- OR you can additionally partition the data on another
--   dimension (what we call 'space partitioning').
-- E.g., to partition `location` into 4 partitions:

SELECT create_hypertable('conditions', 'time', 'location', 4);
```

Fonte: Projeto TimescaleDB, site GitHub[15].

Continuando, buscou-se inserir, medindo tempo para realizar 500.000 operações de inserção, iguais a operação sugerida pelo tutorial, conforme a Figura 15:

Figura 15 - Comando utilizado para inserir os valores de teste

```
DO
$do$
BEGIN
   FOR i IN 1..500000 LOOP
      INSERT INTO conditions(time, location, temperature, humidity)
      VALUES (NOW(), 'office', 70.0, 50.0);
   END LOOP;
END
$do$;
```

Fonte: elaborado pelo autor

[14] Disponível em: <https://github.com/timescale/timescaledb/blob/master/README.md>. Acesso em: 23 de Junho de 2018.
[15] *Idem, Ibidem.*

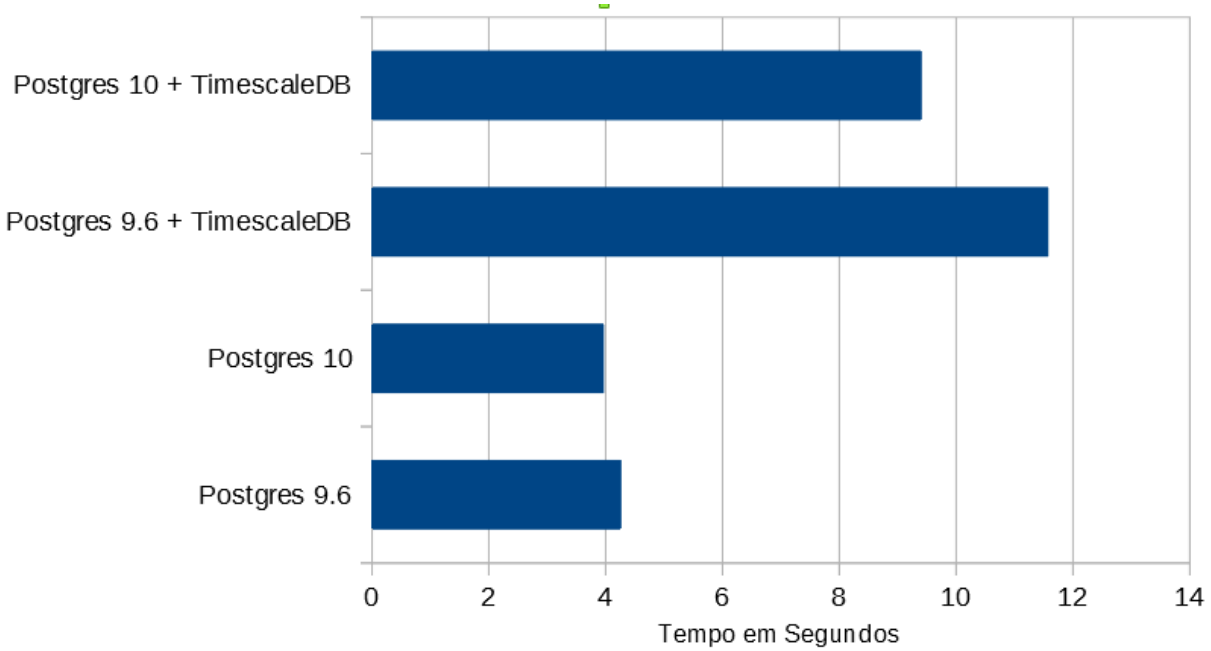
Em outro teste, foram criadas VMs com as mesmas tabelas e a mesma inserção, contendo apenas as versões do PostgresSQL, a fim de que se possa fazer o comparativo de desempenho. Os testes foram realizados três vezes em cada VM, e os resultados que podem ser vistos na Tabela 2, são uma média aritmética simples dos resultados de cada VM.

Tabela 2 - Média dos resultados das inserções nas ferramentas de teste.

Ferramenta	Tempo Médio (em segundos)
Postgres 9.6	4,256
Postgres 10	3,958
Postgres 9.6 + TimescaleDB	11,566
Postgres 10 + TimescaleDB	9,400

Fonte: elaborado pelo autor.

Figura 16 - Gráfico dos tempos registrados no primeiro experimento.



Fonte: elaborado pelo autor.

Findados estes testes, passou se para uma segunda bateria de testes, desta vez utilizando o próprio Import e com diferentes cargas de dados. Assim, foram criadas *Hypertables* tanto para a tabela de medições quanto para a de valores, e nestas, foram inseridos 2 e 51 trechos, totalizando 581.103 e 6.813.359 valores respectivamente,

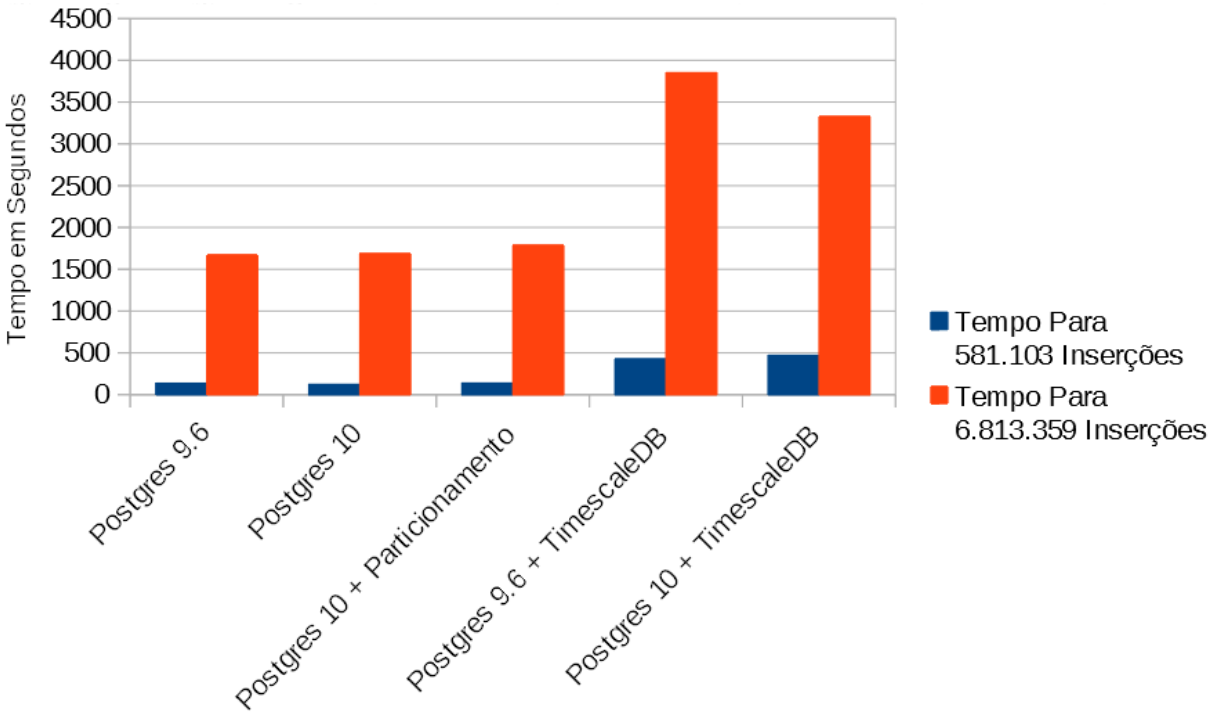
obtidos a partir de arquivos IDE e VAL de exemplo, em dois testes para cada VM. Além disso, neste teste foi incluída uma quinta VM agora explorando o particionamento horizontal do Postgres 10. A Tabela 3 mostra a média aritmética simples dos testes executados em cada VM:

Tabela 3 - Média de tempo dos testes utilizando o Import.

	Tempo Para 581.103 Inserções	**Tempo Para 6.813.359 Inserções**
Postgres 9.6	2m 17s	27m 55s
Postgres 10	2m 03s	28m 11s
Postgres 10 + Particionamento	2m 19s	29m 51s
Postgres 9.6 + TimescaleDB	7m 10s	64m 13s
Postgres 10 + TimescaleDB	7m 51s	55m 32s

Fonte: elaborado pelo autor.

Figura 17 - Gráfico dos tempos registrados no segundo experimento.



Fonte: elaborado pelo autor.

Também foram realizados testes para selecionar e apagar os registros, estes sobre o banco de dados contendo ainda os 6.813.359 registros. No teste de

recuperação de registros, foi medido o tempo de retorno da consulta “select * from valor_cc where id_medicao = '2002-11-13 15:05:00'”, a qual retorna 287.786 linhas de uma parte intermediária da tabela - isto é, existem registros acima e abaixo dos valores selecionados - em três testes. Já para o teste de remoção, foi executado o comando “delete from valor_cc where id_medicao = '2002-11-13 15:05:00'”.

Tabela 4 - Média de tempo dos testes de recuperação e remoção de registros.

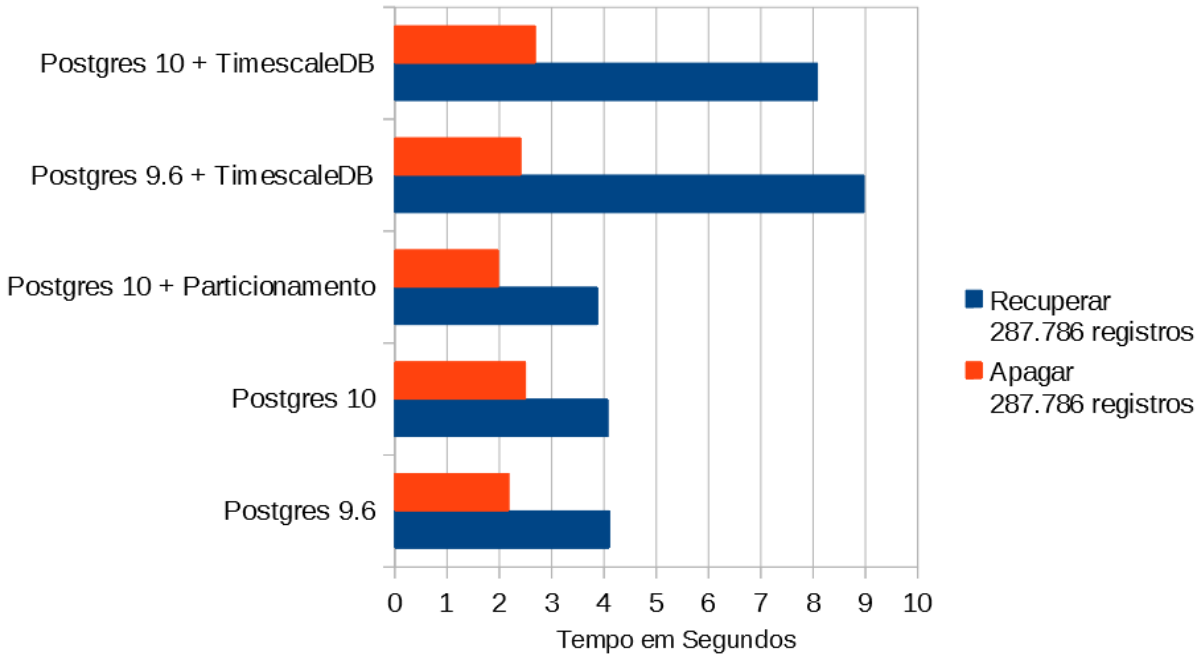
	Recuperar 287.786 registros	**Apagar 287.786 registros**
Postgres 9.6	4,108 segundos	2,183 segundos
Postgres 10	4,079 segundos	2,497 segundos
Postgres 10 + Particionamento	3,880 segundos	1,982 segundos
Postgres 9.6 + TimescaleDB	8,975 segundos	2,412 segundos
Postgres 10 + TimescaleDB	8,080 segundos	2,689 segundos

Fonte: elaborado pelo autor.

Em uma representação gráfica destes dados é possível verificar uma ligeira vantagem do Postgres 10 com o particionamento. Isto pode ser observado na Figura 16.

Com base nesses testes, é possível observar que o desempenho do PostgreSQL é, na verdade, negativamente afetado pela adição do TimescaleDB, e a opção do postgres 10 se mostrou a mais eficiente. Assim, a opção natural é a não utilização da ferramenta TimescaleDB na aplicação.

Figura 18 - Média de tempo dos testes de recuperação e remoção de registros.



Fonte: elaborado pelo autor.

5 CONCLUSÃO

Em geral diversas soluções podem ser pensadas para um determinado problema no mundo da computação, cada qual oferecendo seu próprio desempenho e adequação aos problemas enfrentados. Embora o processo costuma ser parecido, i.e., o levantamento de requisitos, planejamento e a conceituação seguida de um modelo que então é implementado, por vezes o desempenho pretendido não é alcançado. Algo semelhante transcorreu no curso deste trabalho: esperava-se que a ferramenta de banco de dados TimescaleDB oferecesse um ganho significativo de performance, conforme anunciado. Entretanto, como apontam os testes, não apenas a ferramenta não favoreceu o desempenho como também se revelou uma opção consideravelmente pior. Contudo, as melhorias do PostgreSQL 10 trouxeram um resultado interessante nos quesitos “recuperação” e “eliminação” de dados, embora mostrasse um custo um pouco mais elevado para inserir dados do que a mesma ferramenta sem o particionamento.

De modo geral, é possível dizer que o objetivo principal, foi alcançado, a saber, a concepção e implementação de um software capaz de dar conta da massa de dados gerados pelos carros controle, de modo que a mesma cumpre todas as funcionalidades requeridas.

Além disso, salienta-se também a consolidação dos objetivos secundários, isto é, o contato com conceitos como os de banco de dados para séries temporais, virtualização, e mesmo a própria implementação da solução, que levou o conhecimento teórico das salas de aula de encontro à prática, fornecendo uma experiência valorosa que deve contribuir para a superação dos desafios enfrentados enquanto egresso do curso.

6 REFERÊNCIAS

AGUIAR, Lucas T. **Inspeção De Via Permanente: Um Fator Determinante No Processo De Direcionamento Da Manutenção Ferroviária.** Universidade Federal de Juiz de Fora. Trabalho de Graduação: Juiz de Fora, 2011.

DEVEL SISTEMAS. **Virtualização de servidores – Vantagens e desvantagens.** Disponível em: <http://www.develsistemas.com.br/virtualizacao-de-servidores-vantagens-e-desvantagens/>Acesso em: 10 de Junho de 2018.

FREDMAN, Mike. **Time-series data: Why (and how) to use a relational database instead of NoSQL**. TimeScale Blog: 2017. Disponível em: <https://blog.timescale.com/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c> Acesso: 14 de Junho de 2018.

GALVÃO, Junior. **Entendo como funciona o Particionamento Horizontal e Vertical de Tabelas**. 2011. Disponível em: <https://pedrogalvaojunior.wordpress.com/2011/09/06/entendo-como-funciona-o-particionamento-horizontal-e-vertical-de-tabelas/>. Acesso em: 21 de Junho de 2018.

GARCIA-MOLINA, Hector; ULLMAN, Jefrey; WIDOM, Jennifer. **Database System Implementation**. New Jersey: Prentice Hall, 2000.

IBM. **Solving Business Problems with Informix TimeSeries.** 2012. Disponível em: <http://www.redbooks.ibm.com/redbooks/pdfs/sg248021.pdf>. Acesso: 20 de Junho de 2018.

JOSHI, Nishes. **Interoperability in Monitoring and Reporting Systems**. Oslo: Network and System Administration University of Oslo, 2012.

KORTH, Henry F., Sistema de Banco de Dados, tradução [da 2ª ed.] Maurício Heihachiro Galvan Abe. São Paulo: Makron Books, 1995.

LAUREANO, Marcos. **Máquinas Virtuais e Emuladores: Conceitos, Técnicas e Aplicações**. São Paulo: Novatec, 2006.

LINODE. **When and Why to Use Docker**. Disponível em: <https://www.linode.com/docs/applications/containers/when-and-why-to-use-docker/> Acesso em: 13 de Junho de 2018.

PATEL, Anvesh. **PostgreSQL 10: Introduced Native Table Partitioning.** Database Search And Development: 2017. Disponível em: <https://www.dbrnd.com/2017/12/postgresql-10-introduced-native-table-partitioning/> Acesso em: 14 de Junho 2018.

POSTGRESQL. **PostgreSQL 9.6.9 Documentation.** Disponível em: <https://www.postgresql.org/docs/9.6/static/ddl-partitioning.html>. Acesso em: 05 de Junho de 2018.

SOLHEIM, John E. **Object relations in a NoSQL database**. Disponível em: <https://restdb.io/blog/object-relations-in-a-nosql-database>. Acesso em: 13 de Junho de 2018.

7 ANEXOS

ANEXO I

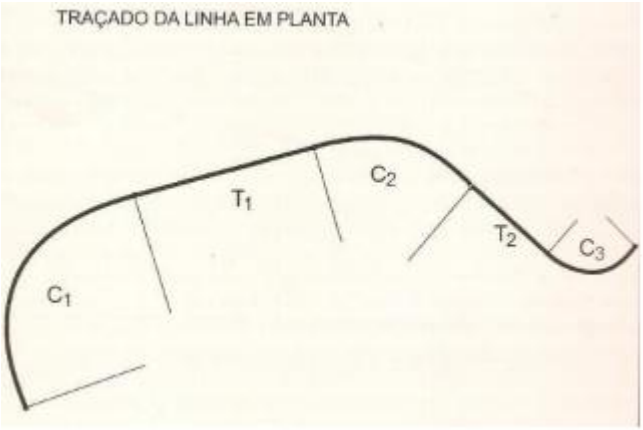
O monitoramento dos parâmetros geométricos da linha férrea exige que se conheçam estes termos. Assim, o presente anexo visa ilustrar alguns termos associados ao trabalho, tendo como base o trabalho de AGUIAR, 2017.

1. Tangentes e Curvas

Tangentes são segmentos de reta que unem duas curvas, tangenciando-as em projeção horizontal. Elas são consideradas fatores críticos no que diz respeito ao traçado da linha que formam, visto que sua variação anormal pode causar prejuízos maiores. As curvas, por sua vez, são grandes responsáveis pelas restrições impostas à circulação de trens, dadas suas características geométricas e os efeitos físicos gerados pela passagem de composições.

Uma ilustração de curva pode ser vista na Figura A1:

Figura A1 - Ilustração de uma curva:



Fonte: AGUIAR, 2017

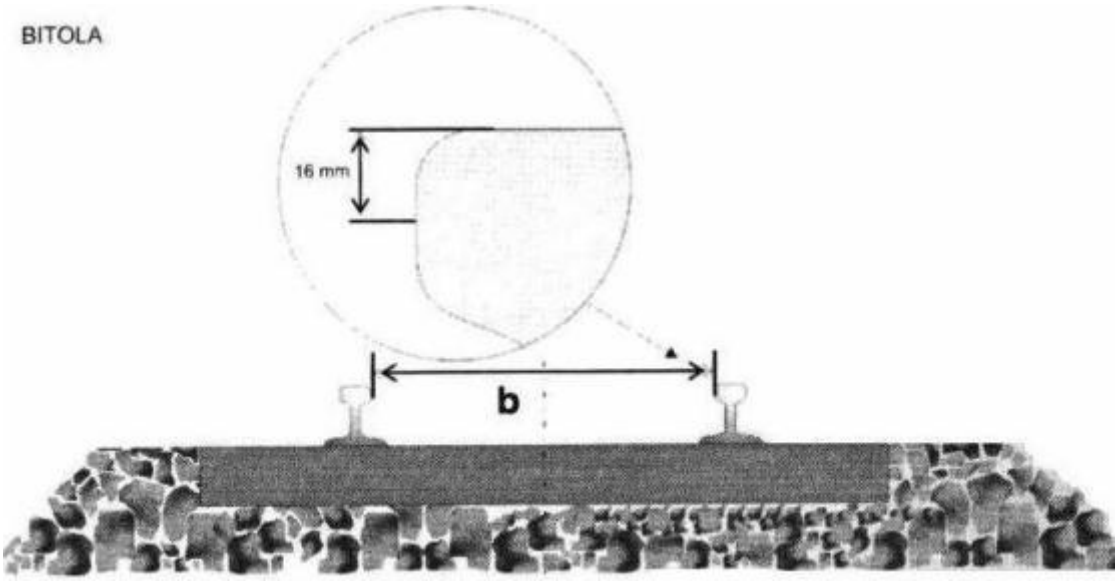
2. Bitola

Por definição, bitola é a menor distância interna entre os boletos das duas filas do trilho. Para que a linha se mantenha segura é necessária a manutenção da

distância entre os trilhos, que, de acordo com cada caso, possui limites mínimos e máximos de comprimento.

Um desenho ilustrativo pode ser visto na Figura A2:

Figura A2 - Ilustração de bitola



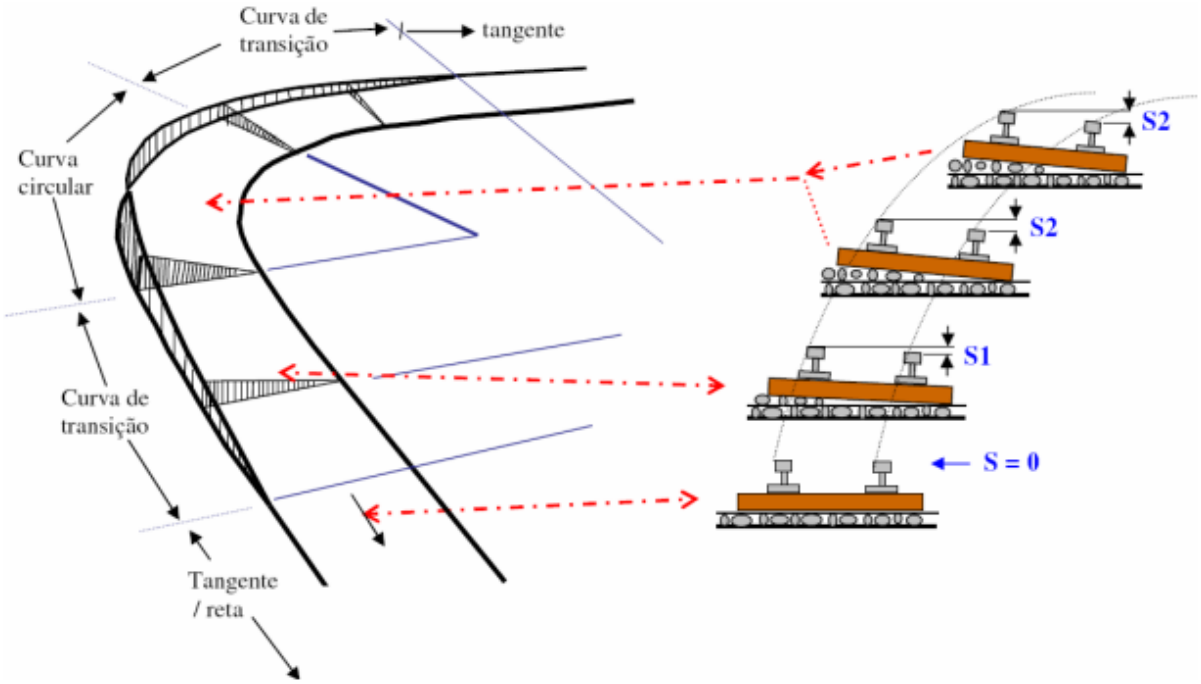
Fonte: AGUIAR, 2017

3. Superelevação

Superelevação é o incremento de altura que se dá à fila externa do trilho em curvas, para que seja possível compensar a ação da força centrífuga". Em uma via ferroviária estabelecida em um plano horizontal, a força centrífuga desloca o veículo no sentido do trilho externo, provocando neste um forte atrito através dos frisos das rodas, podendo ocorrer um tombamento do veículo caso a grandeza da força exceda certo limite. Por isso é realizado o processo de inclinação de um dos lados da via, com finalidade de contrabalancear o efeito nocivo da força.

Uma ilustração de superelevação pode ser vista na Figura A3:

Figura A3 - Ilustração do conceito de superelevação



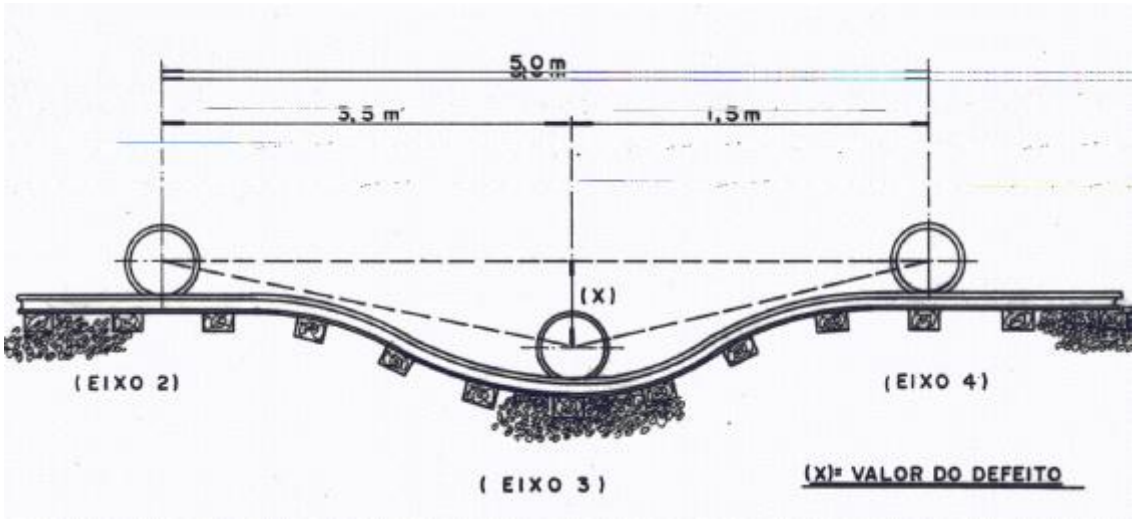
Fonte: AGUIAR, 2017

4. Nivelamento Longitudinal

É a medida no plano longitudinal do nível da linha com relação a dois pontos na superfície do boleto de um trilho. A flecha gerada por uma corda de 10 ou 20 metros estendida no plano longitudinal indica o tamanho do desnivelamento no ponto determinado.

Um esquema ilustrativo de nivelamento longitudinal pode ser visto na Figura A4:

Figura A4 - Ilustração de nivelamento longitudinal

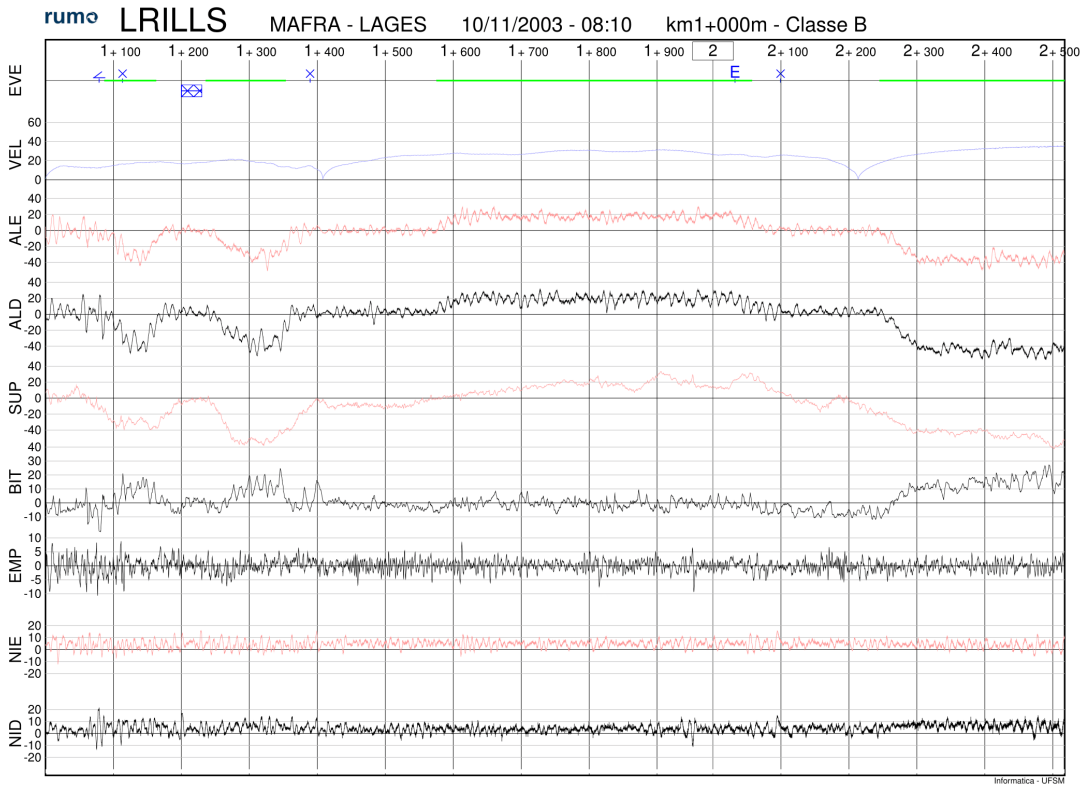


Fonte: AGUIAR, 2017

ANEXO II

Este anexo exemplifica os gráficos produzidos pelo software utilizado pelo operador.

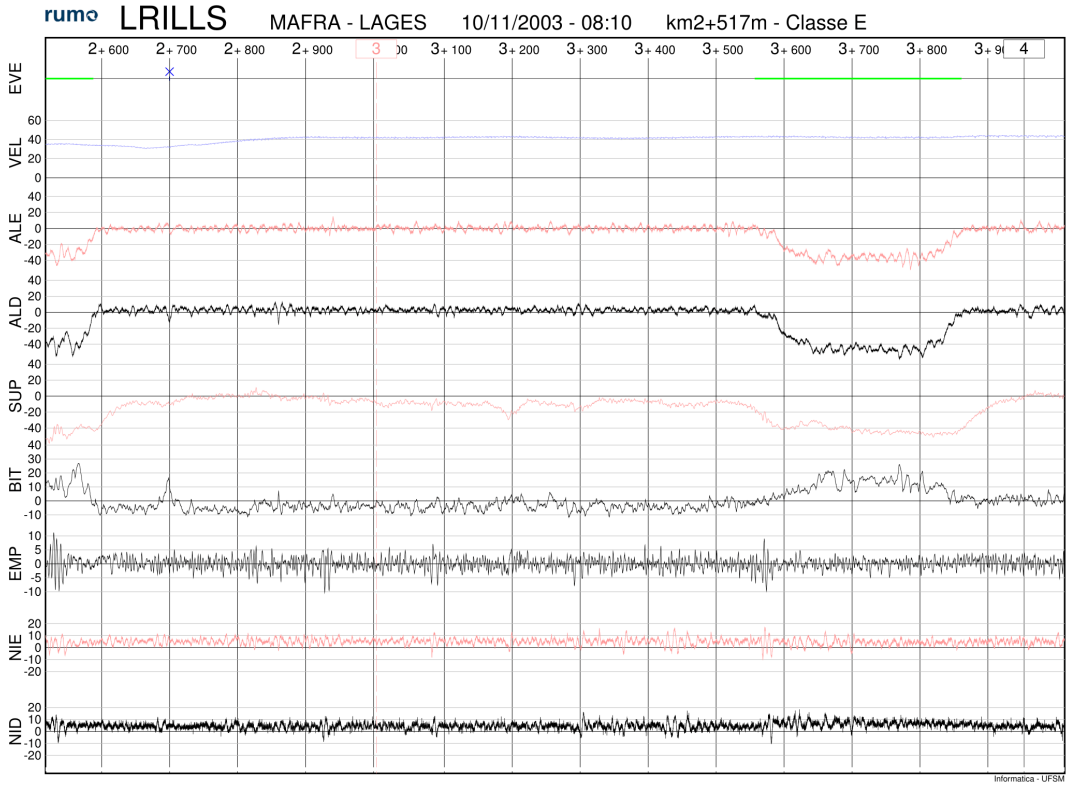
Figura A5 - Gráfico produzido pelo software.



Fonte: elaborado pelo autor.

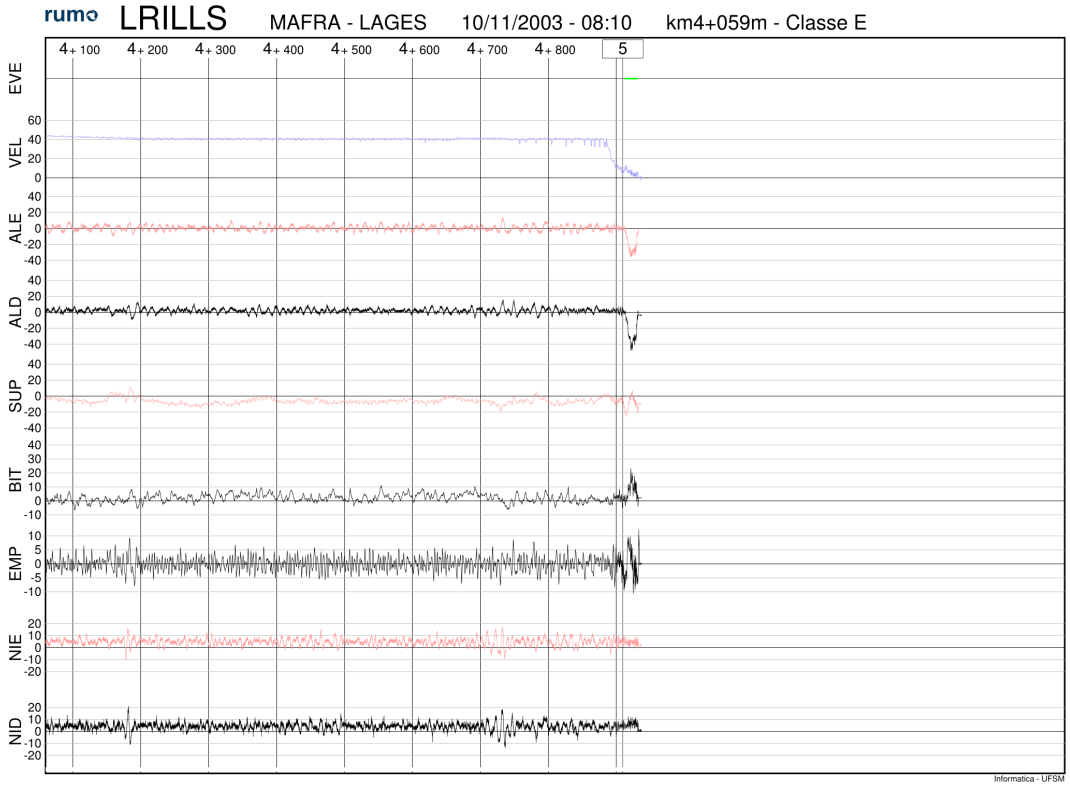
Na Figura A6, é possível verificar, por exemplo, uma pequena curva indicada pela barra verde e, nos indicadores ALE e ALD que se direcionam para o mesmo lado, de modo que curva é para a direita.

Figura A6 - Gráficos produzidos pelo software.



Fonte: elaborado pelo autor.

Figura A6 - Gráficos produzidos pelo software.

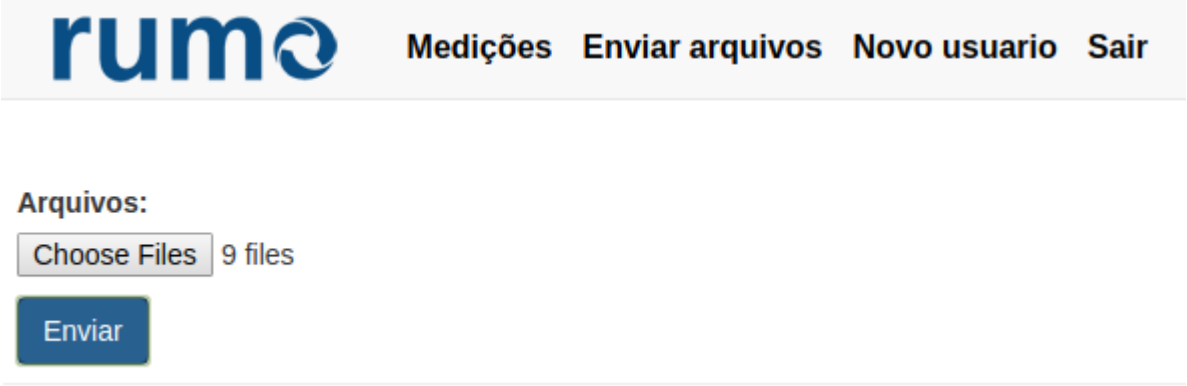


Fonte: elaborado pelo autor.

ANEXO III

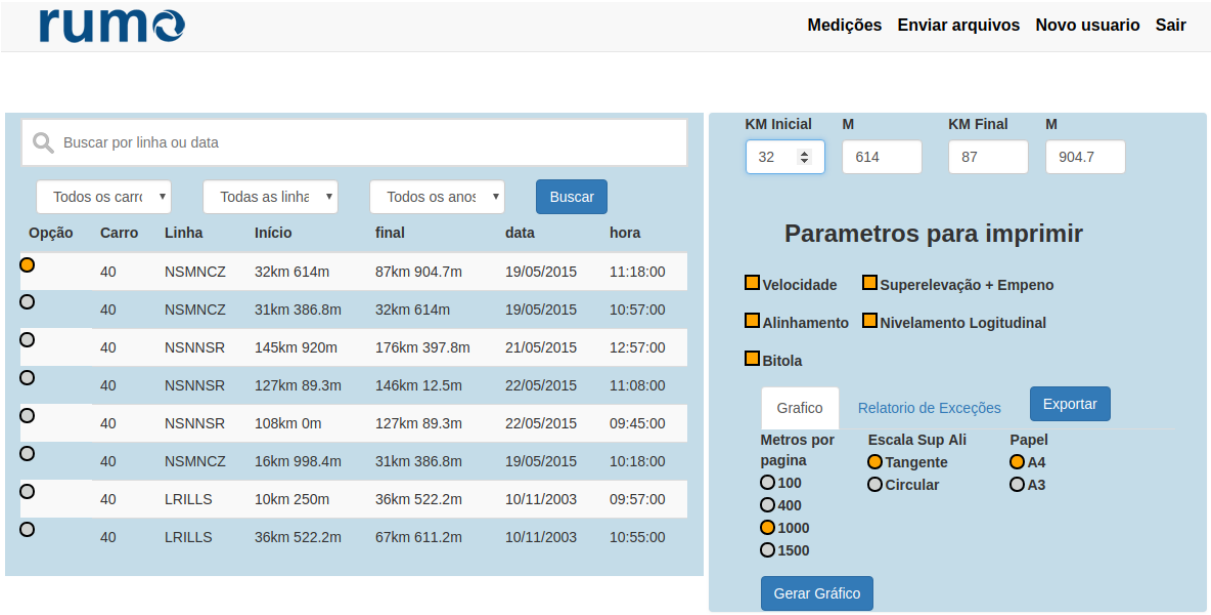
Este anexo traz algumas ilustrações do software que se utilizará do Import para a carga de dados, ou seja, a interface gráfica vista pelo usuário na hora de enviar os arquivos VAL, IDE e STR.

Figura A7 - Interface do usuário para carregamento dos dados.



Fonte: elaborado pelo autor.

Figura A8 - Opções que o usuário possui para gerar os gráficos.



Fonte: elaborado pelo autor.

Figura A9 - Email contendo o relatório de envio, recebido logo após o envio dos arquivos.

```
Arquivos as serem processados e numero de registros:
/src/arquivos/arq_str/LRILLS-010-10112003-0957    256642
/src/arquivos/arq_str/LRILLS-036-10112003-1055    305995
/src/arquivos/arq_str/LRILLS-001-10112003-0810     38777
Importando arquivos...[/src/arquivos/arquivos_ide_val_str/2018-07-07-12-18-15.655378]
Numero trechos: 3
Medicao: Carro 0040 LRILLS (MAFRA - LAGES) 10 km  10/11/2003 09h57      trechos[256642]
Medicao: Carro 0040 LRILLS (MAFRA - LAGES) 36 km  10/11/2003 10h55      trechos[305995]
Medicao: Carro 0040 LRILLS (MAFRA - LAGES) 1 km  10/11/2003 08h10       trechos[038777]
```

Fonte: elaborado pelo autor.