Universidade Federal Fluminense
Escola de Engenharia
Curso de Graduação em Engenharia de Telecomunicações

Matheus D'Amato Campos

# Exportação e manutenção de bancos de dados relacionais em nuvem utilizando serviços Amazon: RDS, Redshift, S3 e EC2

Niterói – RJ

2019

Matheus D'Amato Campos

# Exportação e manutenção de bancos de dados relacionais em nuvem utilizando serviços Amazon: RDS, Redshift, S3 e EC2

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Telecomunicações da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Engenheiro de Telecomunicações.

Orientador: Prof. Dr. Alexandre Santos de la Vega

Niterói – RJ

2019

.

C198e Campos, Matheus D'Amato
Exportação e manutenção de bancos de dados relacionais em nuvem utilizando serviços Amazon: RDS, Redshift, S3 e EC2 / Matheus D'Amato Campos ; Alexandre Santos de la Vega, orientador. Niterói, 2019.
41 f.

Trabalho de Conclusão de Curso (Graduação em Engenharia de Telecomunicações)-Universidade Federal Fluminense, Escola de Engenharia, Niterói, 2019.

1. Banco de dados. 2. Banco de dados relacionais. 3. Manutenção de banco de dados. 4. Exportação de banco de dados. 5. Produção intelectual. I. Santos de la Vega, Alexandre, orientador. II. Universidade Federal Fluminense. Escola de Engenharia. III. Título.

CDD -

.

Matheus D'Amato Campos

Exportação e manutenção de bancos de dados relacionais em nuvem utilizando serviços Amazon: RDS, Redshift, S3 e EC2

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Telecomunicações da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Engenheiro de Telecomunicações.

Aprovada em 13 de dezembro de 2019.

BANCA EXAMINADORA

---

Prof. Dr. Alexandre Santos de la Vega - Orientador
Universidade Federal Fluminense - UFF

---

Profa. Dra. Fernanda Gonçalves de Oliveira Passos
Universidade Federal Fluminense - UFF

---

Profa. Dra. Maria Clicia Stelling de Castro
Universidade do Estado do Rio de Janeiro - UERJ

---

Prof. Dr. Carlos Eduardo Pantoja
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca - Cefet-RJ

Niterói – RJ

2019

# Resumo

Este trabalho apresenta a elaboração e a realização de um projeto de exportação de Banco de Dados em MySQL para um sistema de armazenamento de dados sem tecnologia de leitura dos dados escritos no sistema de armazenamento, implicando na necessidade do uso de outra tecnologia para realizar a leitura dos dados exportados. Na execução do projeto, foram utilizados serviços da *Amazon Web Services*, conhecida como AWS, que é uma plataforma de serviços de computação em nuvem. A motivação para o desenvolvimento deste projeto foi a necessidade de liberar espaço em disco do principal banco de dados da empresa Mobi2buy, pois desta maneira seria possível não aumentar o espaço em disco da instância de banco de dados contratada - o que implicaria diretamente em um não aumento dos custos operacionais da empresa. Tal banco de dados é uma instância RDS, utilizando linguagem MySQL, em São Paulo. A melhor maneira encontrada para solucionar este problema foi utilizar uma série de serviços da AWS, garantindo que as informações antigas estivessem disponíveis, não mais diretamente no banco de dados de produção, mas, ainda sim acessíveis. A ideia da exportação do banco de dados ficou definida de forma que todas as tabelas do banco seriam exportadas para dois destinos. O primeiro deles, *Amazon Redshift*, que é um grupo de banco de dados. O segundo destino é o S3 (*Simple Storage Service*), que funciona, em termos leigos, como uma espécie de armazenador de dados. A segunda estratégia de exportação faz surgir questionamentos que serão respondidos no decorrer deste documento: qual a necessidade da utilização de dois destinos diferentes? Como os dados exportados para o S3 serão consultados? Qual forma de tarifação dos serviços do S3? E qual serviço será utilizado para consultar os dados armazenados neste destino?

Palavras-chave: Banco de Dados Relacional. RDS. *Redshift*. *Redshift Spectrum*. S3.

# Abstract

This work presents the elaboration and the accomplishment of a project of export of Database in MySQL language for a system of storage of data without reading capability of the data written on it, implying in the use of another technology to read the exported data. For the execution of the project reported above, it will be necessary to use some services of Amazon Web Services (AWS), which is a cloud computing services platform. The main motivation for the development of this project was the need to free up disk space from Mobi2buy's main database, because in this way it would be possible not to increase the disk space - which would directly imply a non-increase in the company's operating costs. This database is a RDS instance, using MySQL language, in São Paulo. The best way to solve this problem was to use some of AWS services so that the old information was made available, no longer in the production database, but still accessible. The idea of exporting the database was defined as follows, all tables in the database would be exported to two destinations. The first one, called Amazon Redshift, is a database cluster. The second destination is S3 (Simple Storage Service), which functions, in layman's terms, as a kind of data store. This second storage strategy raises some questions that will be answered in the course of this document: what is the need to use two different destinations? How does the conversion of a MySQL database to a PostgreSQL database work? How the data saved in S3 will be consulted? What's the way of charging for S3 services? And what service will be used to query the data stored in this destination? On the progress of this work, the relevant issues regarding RDS exports to S3 will be dealt with in detail, however, as it escapes from the proposed scope, exporting RDS to Redshift will only be cited as motivation or explanation of export for S3, if it's necessary.

Keywords: Relational Database. RDS. Redshift. Redshift Spectrum. S3.

Dedico este trabalho a todos que contribuíram para o meu sucesso. Em especial, a minha mãe, Maria Lucia, ao meu pai, Carlos Eduardo, ao meu professor orientador, Alexandre, e ao meu CTO, Thiago Taranto.

# Agradecimentos

Gostaria de agradecer, em primeiro lugar, a meus pais, Maria Lucia D'Amato e Carlos Eduardo Campos, pois me apoiaram em todas as etapas da minha vida, sempre contribuindo de forma positiva para meu crescimento profissional e intelectual.

A minha noiva, Camila Pereira, pela compreensão em todos os momentos nos quais precisei me dedicar a este projeto, pelas risadas e por todos os momentos de descontração que me ajudaram a recuperar energias para voltar ao desenvolvimento deste documento.

Aos meus amigos da UFF, pela amizade, carinho e risadas, que contribuíram de forma a tornar a faculdade um caminho mais divertido, em especial ao Carlos Eduardo Bonon, Juni Ventura e Bruna de Mello, por todos os ensinamentos, trabalhos em equipe e suporte.

A Mobi2buy, pelos amigos e pelo projeto, que é tema deste documento, em especial a Thiago Taranto, Ricardo Villaça, Paulo Márcio Matias, Thiago Borges e Vitor Roxo, por muito terem colaborado no desenvolvimento do projeto responsável pela manutenção do nosso banco de dados.

Ao meu professor e orientador, Alexandre Santos de la Vega, e ao meu CTO, Thiago Taranto, pela oportunidade de transformar meu projeto da Mobi2buy neste documento. Novamente ao meu professor e orientador, por todos os ensinamentos a mim passados, pelo suporte, pelas cobranças, pela amizade e principalmente pela paciência.

Enfim, agradeço a todos que de alguma forma me ajudaram a concluir este projeto.

# Sumário

# Capítulo 1

# Introdução

Este documento trata da exportação do conteúdo das principais tabelas de um banco de dados relacional em nuvem para o serviço da *Amazon* chamado S3. Desta forma possibilita-se a construção de um *data lake* - a construção do *data lake* e os serviços utilizados serão explanados posteriormente. O cenário do principal banco de dados da empresa Mobi2buy, nesse momento, é explanado a seguir.

O principal banco de dados era uma instância RDS (do inglês *Relational Database Service*. A empresa estava contratando um acréscimo de disco rígido para o banco de dados para viabilizar que a instância em questão fosse capaz de inserir todos os dados solicitados pelas aplicações da empresa. Desta forma, surgiu a necessidade de exportar os dados contidos no disco do banco de dados para um armazenamento mais barato, possibilitando assim a remoção dos dados do disco rígido do banco de dados.

## 1.1 Organização do Documento

O texto está organizado da seguinte forma. No Capítulo 2 está o cenário inicial, onde são explicados todos os serviços *Amazon* utilizados e suas formas de tarifação. O Capítulo 3 se refere à exportação das tabelas, explicitando as problemáticas que surgem neste momento do projeto. O Capítulo 4 discorre da leitura das tabelas, as chaves de partição, as tabelas externas e a tarifação implícita no procedimento de consulta das tabelas externas, enquanto o Capítulo 5 se refere à validação das tabelas exportas, visando garantir que os dados presentes nas tabelas externas são idênticos aos dados do banco de dados de origem. O Capítulo 6 se refere a remoção dos dados exportados, garantindo que

parte do disco rígido do banco de dados seja liberado. Já o Capítulo 7 tem em seu corpo a conclusão e as sugestões para trabalhos futuros.

## 1.2 Banco de dados

Os bancos de dados são conjuntos de arquivos relacionados entre si com registros. São coleções organizadas de dados que se relacionam de forma a criar algum sentido - informação - e dar mais eficiência durante uma pesquisa ou estudo.

Os bancos de dados são operados pelos Sistemas Gerenciadores de Bancos de Dados (SGBD), que surgiram na década de 70. Antes destes, as aplicações usavam sistemas de arquivos do sistema operacional para armazenar suas informações. Na década de 80, a tecnologia de SGBD relacional passou a dominar o mercado, e atualmente utiliza-se bastante esta vertente, embora esteja perdendo força nos ambientes mais modernos. Outro tipo notável é o SGBD Orientado a Objetos, para quando sua estrutura ou as aplicações que o utilizam mudam constantemente.

A principal aplicação de Banco de Dados é o controle de operações empresariais. Outra aplicação também importante é o gerenciamento de informações de estudos, como fazem os Bancos de Dados Geográficos, que unem informações convencionais com espaciais [22].

### 1.2.1 Banco de dados relacional

Um banco de dados relacional é um mecanismo de armazenamento que permite a persistência de dados e opcionalmente implementar funcionalidades, como armazenar, modificar e extrair informações de um banco de dados.

Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDRs) são usados para armazenar a informação requerida por aplicações construídas usando tecnologias procedurais, tais como COBOL ou FORTRAN, tecnologias orientadas a objeto tais como Java e C# e tecnologias baseadas em componentes como *Visual Basic*.

Tabelas são objetos que contêm alguns dos dados de um banco de dados. Nas tabelas os dados são organizados de maneira lógica em uma estrutura de linha e coluna. Cada linha representa um registro e cada coluna representa um campo do registro.

Tabelas tipicamente possuem chaves, uma ou mais colunas que unicamente iden-

tificam uma linha na tabela. Para melhorar o tempo de acesso aos dados de uma tabela, são definidos índices. Um índice provê uma forma rápida para buscar dados em uma ou mais colunas em uma tabela, da mesma forma que o índice de um livro permite que nós encontremos uma informação específica rapidamente.

**Funcionalidades básicas de SGBDRs**

O uso mais comum de SGBDRs é para implementar funcionalidades simples do tipo CRUD (do inglês *Create*, *Read*, *Update* e *Delete* – que significa as operações de Inserção, Leitura, Atualização e Exclusão de dados). Por exemplo, uma aplicação pode criar uma nova compra e inserí-la no banco de dados. Ela pode ler uma compra, trabalhar com seus dados e, então, atualizar o banco de dados com a nova informação. Pode ainda optar por excluir uma compra existente. A grande maioria das interações com um banco de dados provavelmente implementará as funcionalidades básicas de CRUD [19]..

A forma mais fácil de manipular um banco de dados relacional é submeter declarações escritas na linguagem SQL (*Structured Query Language*) a ele [18]. Para criar uma nova linha em uma tabela, deve-se criar uma declaração de *INSERT*. Similarmente, para ler uma linha da tabela usa-se a declaração *SELECT*, para atualizar uma linha existente usa-se a declaração do tipo *UPDATE* e, finalmente, para excluir uma linha da tabela usa-se a declaração *DELETE* [19].

## 1.3 Exportação

A exportação do banco de dados utiliza, majoritariamente, uma biblioteca de *Python* chamada Pandas, pois esta é a biblioteca de análise e manipulação de dados mais consolidada no *Python* atualmente, garantindo que os dados exportados estejam na melhor forma possível para serem lidos posteriormente, pelo *data lake* que recebe os dados exportados do banco de dados relacional. São utilizadas algumas estratégias para garantir o bom desempenho e a menor tarifação possível pela utilização do repositório de dados: arquivos em formato *parquet*, compressão "gzip" e utilização de chaves de partição. Ao decorrer deste documento são explanados de forma mais explícita o formato *parquet* e a compressão "gzip". Note que todos os termos supracitados serão explanados no corpo deste documento.

## 1.4 Validação

A validação dos dados exportados consiste em comparar, de forma otimizada e inteligente, os dados do banco de dados de origem e do repositório de dados de destino, que é tratado ao longo do corpo do documento, por vezes, como banco de dados de destino ou *Redshift Spectrum*, garantindo assim que, nos dois sistemas, os mesmos dados estão disponíveis.

### 1.4.1 *Redshift Spectrum*

*Amazon Redshift Spectrum* é uma tecnologia do serviço *Redshift* que permite a análise simples ou complexa de dados armazenados no S3 [11]. Para maiores detalhes sobre o funcionamento do *Redshift*, consultar a Seção 2.2. Para o S3, consultar a Seção 2.3.

## 1.5 Exclusão

A exclusão das linhas das tabelas exportadas do banco de dados de origem garante liberação do armazenamento em disco. No entanto, este procedimento é o mais crítico dentre todos, posto que caso o dado exportado não tenha sido corretamente armazenado no repositório de dados, o mesmo não estará mais disponível para ser consultado, causando perda de informação.

# Capítulo 2

# Cenário inicial

O presente capítulo tem como intuito relatar e motivar as decisões que compõem o projeto de manutenção das tabelas de um banco de dados relacional.

Para tal, descreve-se os principais serviços da *Amazon Web Services* (AWS) que são utilizados, assim como as formas de tarifação de cada um deles, com o intuito de melhor justificar as decisões tomadas.

Além disso, toda abordagem relativa a linguagem de programação e bibliotecas utilizadas também são encontradas neste capítulo.

## 2.1 *Relational Database Service*

O *Relational Database Service* (RDS) é um serviço de Banco de Dados oferecido pela AWS que consiste em disponibilizar um conjunto de instâncias otimizadas em prol de atender a diferentes necessidades de uso de banco de dados relacionais. Os tipos de instâncias variam com diversas combinações de CPUs, memória RAM, armazenamento interno e capacidade de rede, oferecendo assim uma grande flexibilidade de recursos disponíveis para os mais variados modelos de negócio dos usuários. As instâncias RDS variam de 1 CPU e 1 GB RAM, chamada de db.t3.micro, até a db.m5.24xlarge, com 96 CPUs e 384 GB RAM [7].

A forma de tarifação é igual para qualquer instância RDS - determinado valor por hora de servidor disponível. No entanto, a localização física do servidor que hospeda o serviço contratado faz variar o valor do aluguel da instância, assim como a capacidade de desempenho da instância - número de CPUs, quantidade de memória RAM e afins.

A menor instância contratável, chamada de db.t3.micro, custa US$0,017 para instância hospedadas na Virgínia do Norte e US$0,035 para instâncias hospedadas em São Paulo, enquanto a instância db.m5.24xlarge custa US$8,208 para instâncias hospedadas na Virgínia do Norte e US$11,088 para instâncias hospedadas em São Paulo [8]. Note que a distância da instância contratada em relação a localização do usuário faz aumentar ou diminuir a latência, caso o usuário esteja mais distante ou mais próximo da unidade contratada, respectivamente.

## 2.2 *Redshift Data Warehouse*

*Redshift* é um armazém de dados rápido e escalável que permite analisar todos os dados do armazém e de um repositórios de dados com simplicidade e economia [9]. O *Redshift* oferece desempenho dez vezes maior que qualquer outro usando aprendizado de máquina, execução massivamente paralela de consultas e armazenamento colunar em discos de alto desempenho [9]. É possível configurar e implantar um novo armazém de dados em alguns minutos para executar consultas em *petabytes* de dados em um armazém de dados do *Redshift* e em *exabytes* de dados em um repositório de dados no *Amazon S3* [9]. A menor instância *Redshift*, chamada de dc2.large, possui 2 CPUs e 15GB de armazenamento, custa apenas US$0,25 por hora na Virgínia do Norte e US$0,40 por hora em São Paulo. É possível escalá-la até US$250 por *terabyte* por ano, menos de um décimo do custo de outras soluções, para instâncias hospedadas na Virgínia do Norte [10].

Armazéns de dados e repositórios de dados são amplamente usados para armazenar grande volume de dados. Um repositório de dados é um vasto conjunto de dados brutos, cuja finalidade ainda não está definida. Um armazém de dados é um conjunto de dados filtrados e estruturados que já foram processados para uma finalidade específica [9].

## 2.3 *Simple Storage Service*

O *Simple Storage Service*, também chamado de *Amazon S3*, é um serviço de armazenamento de objetos que oferece a melhor escalabilidade do mercado, disponibilidade de dados, segurança e desempenho. Isso significa que usuários de todos os portes e setores podem usá-lo para armazenar qualquer volume de dados em uma grande variedade de casos de uso, como *websites*, aplicativos para dispositivos móveis, *backup* e restauração,

arquivamento, aplicativos empresariais, dispositivos *IoT* e análises de grande volume de dados [15]. O *Amazon S3* fornece recursos de gerenciamento fáceis de usar, de maneira que é possível organizar os dados e configurar os controles de acesso refinados para atender a requisitos específicos comerciais, organizacionais e de conformidade. O *Amazon S3* foi projetado para 99,999999999% (também conhecido como "os 11-noves") de durabilidade [15].

O S3 oferece uma variedade de classes de armazenamento específicas para cada caso de uso. Essa variedade inclui o S3 *Standard* para propósitos gerais de armazenamento de dados armazenados com frequência, o S3 *Standard-Infrequent Access*, ou S3 *Standard-IA*, e S3 *One Zone-Infrequent Access*, ou S3 *One Zone-IA* para dados de longa duração, mas com menor frequência de acessos e o S3 *Glacier* e S3 *Glacier Deep Archive* para dados que exigem longa preservação digital e são raramente acessados. O *Amazon S3* também oferece recursos para gerenciar seus dados em todo o seu ciclo de vida. Vale salientar que todos os diferentes serviços citados acima diferem entre si no tempo de despendido para consultar os arquivos, o que influencia diretamente no custo pago para manter os arquivos armazenados. Quanto maior o tempo para consultar um arquivo armazenado, mais barato é o mantenimento do mesmo. Depois que uma política de ciclo de vida do S3 for definida, os dados são automaticamente transferidos para uma classe de armazenamento diferente, sem qualquer alteração em seu aplicativo [16].

A tarifação do *Amazon* S3 *Standard*, para *buckets* na Virgínia do Norte, para os primeiros 50 TB/mês é de US$0,023 por GB, os próximos 450 TB/mês custam US$0,022 por *gigabyte* e acima de 500 TB/mês, US$0,021 por GB. Para *buckets* mantidos em São Paulo, os primeiros 50 TB/mês custam US$0,0405 por *gigabyte*, os próximos 450 TB/mês são US$0,039 por *gigabyte* e acima de 500 TB/mês, US$0,037 por *gigabyte*. Note que balde é um tipo de pasta para objetos, e pode ter uma série de configurações que permitem distinguir usuários e chaves de acesso, tal como a permissão de cada usuário, como leitura, escrita e administração de arquivos no *bucket*. Objetos são quaisquer arquivos que contenham dados e metadados que descrevam este arquivo [17].

## 2.4 *Elastic Compute Cloud*

Conhecido como EC2, é um serviço que disponibiliza capacidade computacional segura e redimensionável na nuvem. Este serviço foi projetado para facilitar a computação em nuvem para desenvolvedores. O serviço de EC2 oferece um controle completo de seus recursos computacionais e permite que o usuário trabalhe no ambiente computacional da *Amazon*. A escalabilidade deste serviço permite que apenas alguns minutos sejam suficientes para se obter e inicializar novas instâncias de servidor, permitindo que seja dimensionado a capacidade de processamento rapidamente para mais e para menos, à medida que seus requisitos de computação mudarem [3].

Este serviço possui quatro subdivisões:

- Instâncias sob demanda: O preço varia de acordo com a quantidade de usuários que contratando os serviços EC2 naquele momento, podendo ser maior ou menor do que o valor dos servidores dedicados, que é o serviço mais comum.

- Instâncias reservadas: Instâncias são contratadas por longos períodos, um ou três anos, e o preço das instâncias varia de acordo com o período contratado. Quanto maior o tempo de reserva de uma instância, menor o custo por minuto dela;

- Instâncias *spot*: Instâncias são criadas para executar determinada função, preferencialmente durando um curto período de tempo, e posteriormente são destruídas;

- Instâncias dedicadas: Instâncias são contratadas e somente destruídas quando solicitado. [4]

Posto que somente os serviços de instâncias reservadas e servidores dedicados serão utilizados neste projeto, não são abordadas as formas de tarifação dos outros tipos de serviço. Vale ainda notar que a vantagem do serviço de instâncias reservadas é o preço pago por cada instância, e a desvantagem é que a instância contratada fica limitada ao tamanho de instância contratado, e, sendo ou não utilizado, será cobrado. Enquanto o serviço de servidores dedicados possui alta flexibilidade na escalabilidade das instâncias, sendo pago somente pelo que é consumido, no entanto, com valor por minuto mais alto do que o valor cobrado pelas instâncias reservadas.

A menor instância EC2 da AWS se chama t3.nano:

- Usa distribuições Linux;

- Na Virgínia do Norte custa US$0,002 por hora, enquanto em São Paulo o valor é US$0,005/h;
- 2 CPUs Intel escaláveis de 2,5GHz;
- 512MB de memória RAM;
- armazenamento EBS (*Elastic Block Store*, que é um serviço de armazenamento de alta performance, projetado para trabalhos com altas taxas de transferência).

A maior instância EC2 se chama p3dn.24xlarge:

- Usa distribuições Linux;
- Na Virgínia do Norte custa US$19,215 por hora e indisponível no Brasil;
- Otimizada para HPC (*High-Perfomance Computing*) e aprendizado de máquina;
- 96 CPUs Intel *Xeon* escaláveis;
- 8 GPUs (*Graphical Processing Unit*) NVIDIA V100 *Tensor Core* com 32GB de RAM cada um;
- 1,8TB de armazenamento SSD (*Solid State Drive*) em NVMe (*Non-Volatile Memory Express*).

A maior instância disponível no Brasil é a 1x.32xlarge, que pode ser alugada por US$16,029/h [5].

## 2.5 Ambiente de desenvolvimento

Tendo em vista a importância do projeto, é fundamental que qualquer desenvolvedor seja capaz de realizar correções no mesmo, caso necessário. Por conseguinte, Python foi escolhida como principal linguagem de programação para desenvolvimento do projeto, posto que tal linguagem tem a legibilidade como uma dos suas maiores vantagens. A sintaxe do Python permite que códigos sejam escritos em menos etapas, quando comparado com Java ou C++. Além disso, outras vantagens do Python em relação às outras linguagens são o extensivo suporte de suas bibliotecas e os recursos de integração e produtividade aumentada do programador (os *designs* limpos, orientados a objeto, aumentam

de duas a dez vezes a produtividade do programador quando comparado a linguagens como Java, VB, Perl, C, C++ e C#) [27]. Como desvantagem na utilização do Python, cita-se o desempenho da linguagem, entretanto, este fator não é relevante para este projeto posto que o tempo de processamento realizado pelos servidores que utilizam Python é muito menor que o tempo de processamento feito diretamente no banco de dados. Em média, experimentalmente, avaliou-se que o tempo de processamento no banco de dados é cerca de 300 vezes maior do que o tempo de processamento nos computadores *cloud*. Desta forma, alterar a linguagem de programação não seria, considerando desempenho como ponto de vista, eficiente. Vale notar também que como o Python não faz tipagem de variáveis, a utilização de memória RAM é maior quando se compara com a utilização de memória RAM feita por outras linguagens, entretanto, como será explanado posteriormente, a biblioteca Pandas força a tipagem das variáveis, reduzindo o consumo de memória.

Posto que Python foi a linguagem de programação escolhida, seguem as bibliotecas utilizadas:

- MySQL *Connector*, para estabelecer conexão com a instância RDS (*Relational Database Service*) de MySQL; [21];
- Psycopg2, para estabelecer conexão com a instância *Redshift* [26];
- Boto3, para estabelecer conexão com o S3 [25];
- *Pandas*, para tratar os dados do MySQL, permitindo leitura dos dados quando armazenados no S3 e para comprimir os arquivos que geram as tabelas exportadas (ver Seção 4.2) [23];
- *PyArrow*, para converter os dados para uma extensão chamada *parquet* (ver Seção 4.2) [24];
- Outras bibliotecas como *Decouple*, *Datetime* e *Botocore*, para controlar variáveis de ambiente, tempo de execução de código e gerenciamento de arquivos no servidor, respectivamente.

## 2.6 Infraestrutura do ambiente de desenvolvimento

Considerando os valores das instâncias EC2 explanados na Seção 2.4 e que o Pandas escreve em memória RAM do servidor o conteúdo das tabelas do banco de dados. Posteriormente, avaliou-se, tendo custo como principal fator e desempenho logo em seguida, a melhor instância EC2 para realizar estes procedimentos.

Durante testes de exportação (rotinas de exportação com acompanhamento que simulam uma exportação real e automatizada), avaliando o comportamento das instâncias EC2, definiu-se que instância com 4GB de RAM eram suficientes para tratar uma pequena parte de uma tabela. Esta decisão foi tomada considerando principalmente dois fatores. Caso seja reduzida a quantidade de memória RAM disponível no servidor, será necessário que a tabela exportada tenha seu procedimento de exportação mais limitado (menos linhas seriam exportadas a cada rodada de exportação). Caso seja aumentada a quantidade de memória RAM disponível, as consultas para extrair os dados das tabelas seriam mais demoradas, causando aumento do processamento do banco de dados, o que não é interessante tendo em visto que o processo de exportação não deve prejudicar outras aplicações que necessitam do banco de dados para funcionamento correto. Instâncias com mais memória RAM tratariam mais dados por vez, mas isso não necessariamente implica em melhora de desempenho, posto que o tempo de tratamento aumenta com a quantidade de dados tratados. O segundo fator de avaliação durante os testes foi a capacidade de processamento dos servidores, que influencia diretamente no tempo de processamento de cada intervalo da tabela. Servidores com capacidade de processamento reduzida são mais baratos e levam mais tempo para tratar os dados, o inverso é recíproco. Quando em comparação com a capacidade do Banco de Dados de fornecer os dados solicitados para exportação, instâncias c5.large (4GB RAM e 2 núcleos de processamento) e c5.xlarge (8 GB RAM e 4 núcleos de processamento) foram eleitas [3]. Estas instâncias da família C5 possuem processador Intel *Xeon Scalable Cascade Lake* (segunda geração) ou processador Intel *Xeon Platinum* série 8000 Skylake-SP (primeira geração) com uma frequência turbo sustentada de 3,4 GHz em todos os núcleos e uma frequência máxima turbo de 3,5 GHz em um único núcleo com a Intel *Turbo Boost Technology*. Isso indica que as instâncias são otimizadas para executar cargas de trabalho avançadas, com intenso uso de computação [3].

Durante testes de exclusão (situações de remoção dos dados das tabelas em mo-

mento de baixa utilização do banco de dados), observou-se que toda carga de trabalho fica diretamente vinculada ao banco de dados, sendo o EC2 responsável somente por controlar o volume de requisições de exclusão feitas ao banco. Portanto, um servidor EC2 nano é suficiente para desempenhar esta função.

# Capítulo 3

# A exportação das tabelas

Neste capítulo, são tratados todos os assuntos pertinentes às exportações das tabelas, assim como os principais complicadores e suas respectivas soluções.

## 3.1 Grande volume de dados

O comando de consulta (*query*) a seguir calcula, em *megabytes*, o tamanho de todas as tabelas presentes em um banco de dados MySQL, ordenando das maiores para as menores:

```
SELECT
     table_schema as ‘Database‘,
     table_name as ‘Table‘,
     round(((data_length + index_length) / 1024 / 1024), 2)
FROM information_schema.TABLES
ORDER BY (data_length + index_length) DESC;
```

Portanto, a tabela na primeira posição do resultado da consulta é a tabela que mais ocupa espaço no disco rígido do banco de dados. Desta forma, foi definida a primeira tabela a ser exportada, que, de agora em diante, é chamada de Tabela_A.

A consulta abaixo retorna a quantidade de linhas da Tabela_A:

```
SELECT COUNT(1)
FROM Tabela_A;
```

Em posse do resultado da *query* acima (cerca de 11 bilhões de linhas) e após análise da Tabela_A, notou-se a necessidade de realizar a exportação da tabela em pequenas partes, posto que seria impossível guardar toda a tabela em memória RAM nos servidores disponíveis para tratamento.

Cada parte exportada contempla um milhão de linhas da Tabela_A, que possui 18 colunas, com seus tipos variando entre *varchar*, *bigint*, *int*, *smallint*, *float* e *datetime*.

Como explanado previamente na Seção 2.6, a biblioteca Pandas foi utilizada para converter os dados do banco de dados. Neste momento, alguns problemas surgiram.

### 3.1.1 Colunas de tipos inteiros

Todas as colunas de tipo inteiro apresentam problemas para conversão para *parquet*, quando havia ao menos um elemento nulo em seu conteúdo (este erro se justifica pela incapacidade da biblioteca *PyArrow* de realizar a conversão dos arquivos para *parquet*):

- Tipo *Bigint* com linhas nulas;
- Tipo *Int* com linhas nulas;
- Tipo *Smallint* com linhas nulas.

Note que os outros tipos de inteiro (*tinyint* e *mediumint*) não estão disponíveis no sistema de leitura utilizado.

Como solução, foi preciso inserir um valor nas células nulas das colunas numéricas. Como a informação contida nas tabelas não pode ser alterada, foi necessário um estudo da tabela para que os números inseridos na tabela não fizessem sentido na sua interpretação. Por exemplo, caso a coluna contivesse números de telefone e fossem inseridos 8 algarismos nas células nulas, a informação inserida para correção deste problema poderia se confundir com a informação previamente presente na Tabela_A. Na maioria dos casos "-99" ou "0" foram inseridos como substitutos para as células nulas.

### 3.1.2 Colunas de tipos decimais

Assim como relatado na Subseção 3.1.1, todas as colunas que contém números decimais apresentam problemas para conversão para *parquet*, quando possuem ao menos um elemento nulo em seu conteúdo:

- Tipo *float* com linhas nulas;
- Tipo *decimal* com linhas nulas;
- Tipo *double precision* com linhas nulas.

https://www.overleaf.com/project/5c130ead6eb16b722f50f8ac

A solução segue exatamente o mesmo princípio explanado na Subseção 3.1.1.

### 3.1.3 Colunas de tipo data e hora

Em se tratando de colunas que representam datas e horas, o padrão adotado foi o *timestamp*, que é representado por "YYYY-MM-DD HH:MM:SS.sss". Nesse caso, primeiro tem-se o ano, com 4 dígitos, separado do mês por um hífen, com 2 dígitos, separado do dia do mês por um hífen, também com dois dígitos. Um espaço separa a data do horário, que tem sua representação dada por dois dígitos para hora, dois dígitos para minuto, dois dígitos para segundo e 3 dígitos para microssegundo, sendo todos separados por dois pontos, salvo o segundo do microssegundo, que é separado por ponto.

Neste ponto do tratamento de dados também surgiu o problema na conversão para quando havia ao menos um elemento nulo na coluna, mas a solução é mais simples do que nos casos anteriores. As colunas que apresentaram problemas foram as seguintes:

- Tipo *date* com linhas nulas;
- Tipo *datetime* com linhas nulas;
- Tipo *timestamp* com linhas nulas;
- Tipo *time* com linhas nulas.

Em todas as situações, pensando em desconsiderar qualquer questão relativa a fuso-horário, o uso do tipo *datetime* foi descartado [20]. O tipo *time* não é utilizado nas colunas tratadas, entretanto a solução para o problema dos elementos nulos dos outros tipos de coluna também se aplica a este tipo.

A solução, desta vez, encontra-se diretamente contida na conversão do dado para o formato de data. A configuração do parâmetro *errors* do Pandas como *coerce* traduz, forçadamente, as datas nulas para NaT (*Not-a-Time*).

Vale notar que, implicitamente contido no Pandas, o NumPy é a biblioteca Python efetivamente responsável por gerar arquivos *parquet*, enquanto o Pandas somente faz a leitura e o tratamento dos dados. O NaT, a representação do NumPy para valores vazios em colunas *timestamp*, é um valor sentinela pseudo-nativo que apresenta compatibilidade com o NaN (*Not-a-Number*) adotado pelo Pandas. O NaN representa valores vazios para as colunas de inteiro (não compatível com o formato *parquet*, como explanado previamente na Subseção 3.1.1). Portanto, elementos NaT são suportados em tabelas externas.

### 3.1.4 Colunas com letras e símbolos

A conversão de colunas de texto para *parquet* é a mais simples dentre os tipos de coluna possíveis. O NumPy consegue fazer a conversão de *strings* nulas para *parquet*, então não é necessário alterar os valores dos elementos vazios das colunas de texto.

Neste momento, vale citar que durante a leitura das tabelas notou-se que as colunas de texto com elementos nulos não apareciam como elementos vazios. O NumPy preenche o elemento *null* escrevendo, literalmente, *"null"* no conteúdo das células. Desta forma, para que as tabelas exportadas não contivessem repetidas vezes a palavra *"null"*, foi adotada uma forma de evitar tal metodologia do NumPy.

Os elementos *"null"* foram substituídos por um espaço vazio. Desta forma, as tabelas exportadas assemelharam-se com maior grau de precisão, às tabelas do RDS MySQL. Note que esta solução é válida para qualquer tipo de coluna de texto, *varchar*, *char*, ou *text*, assim como suas respectivas variações de tamanho.

## 3.2 Arquivos *parquet*

Esta seção trata de todos os assuntos pertinentes ao *parquet*, que foi a extensão escolhida para manter os arquivos das tabelas exportadas, posto que arquivos *parquet* são construídos para serem lidos de forma colunar, não de forma linear, como padrão. Vale salientar que o *Redshift* realiza consultas de forma colunar. A combinação tanto dos dois fatores supracitados aumenta consideravelmente o desempenho do Redshift.

### 3.2.1 A estrutura dos dados e a construção do modelo

O *parquet* é construído a partir do zero, tendo em mente estruturas complexas de dados aninhados, usando um algoritmo de fragmentação e montagem de registros [1]. Acredita-se que essa abordagem é superior ao achatamento simples de espaços de nome aninhados [1].

O *parquet* é construído para suportar esquemas de compressão e codificação muito eficientes. Vários projetos demonstraram o impacto no desempenho da aplicação do esquema de compactação e a codificação correta dos dados. Tal extensão permite que os esquemas de compactação sejam especificados em um nível por coluna e é preparado para que no futuro possa permitir adicionar mais codificações, à medida que forem propostas e implementadas [1].

### 3.2.2 A conversão

A biblioteca Pandas oferece um método específico para realizar a conversão de quadros de dados em memória para arquivos *parquet*: *to_parquet*. Como parâmetros para este método, tem-se, dentre os utilizados:

- *fname*, que recebe o nome do arquivo que é gerado;
- *engine*, que recebe a biblioteca responsável pela forma como o quadro de dados é convertido para *parquet*;
- *compression*, que recebe o tipo de compressão que é utilizado.

Como o armazenamento no S3 segue o princípio de chaves, arquivos com mesmo nome são sobrepostos. Por isso, *fname* recebe o nome da tabela que gerou o arquivo, concatenado com o primeiro *id* do quadro de dados. Como nas tabelas em MySQL o *id* é auto-incremental, eles não se repetem, garantindo assim que nunca dois arquivos diferentes terão o mesmo nome - o que faria com que eles fossem sobrepostos.

A *engine* utilizada é sempre a *PyArrow*, pois somente duas *engines* estão disponíveis para o procedimento supracitado. A outra *engine* é a *FastParquet*, que ainda está em desenvolvimento [24].

O tipo de compactação utilizado é o "gzip" (baseado em *DEFLATE*, também é utilizado no envio de HTTP comprimido e no formato de arquivo de imagens png). Os

outros algoritmos disponíveis são o “snappy” (desenvolvido pela *Google*, escrito em C++, baseado em LZ77 e de código aberto) e o “brotli” (também baseado no LZ77, usa codificação de Huffman e modelagem de contexto de segunda ordem). Vale ressaltar que o algoritmo *DEFLATE* também utiliza codificação de Huffman e se baseia no LZ77, que é o algoritmo de compressão de dados, sem perdas, desenvolvido por Abraham Lempel e Jacob Ziv, em 1977.

Vale salientar que embora compressão “snappy”, experimentalmente, resulte em um arquivo menor, o arquivo gerado não é divisível. A compressão “gzip” comprime menos do que a “snappy”, mas gera arquivos divisíveis. Como o *Redshift Spectrum* usa 20 frentes de leitura para realizar varredura dos dados solicitados, arquivos divisíveis são divididos em 20 partes, fazendo com que cada frente de leitura leia 1/20 do total dos arquivos varridos. Caso arquivos não divisíveis sejam lidos, todas as frentes de leitura lerão o arquivo inteiro, reduzindo em 20 vezes a eficiência do procedimento [12].

# Capítulo 4

# A leitura das tabelas exportadas

## 4.1 A criação das tabelas

O recurso de Tabelas Externas do *Redshift Spectrum* permite pesquisar dados em arquivos de extensão *parquet* - além de outros - armazenados no S3. As principais limitações para este tipo de tabela se encontram na impossibilidade da execução de comandos como *update*, *delete* ou *insert*. A explicação para essa impossibilidade se dá no fato de que o conteúdo da tabela não se encontra no disco rígido do banco de dados, mas sim em arquivos de texto. Como a tabela externa é um recurso somente para leitura, não é possível realizar comandos que alterem os dados da tabela.

### 4.1.1 A DDL da tabela externa

As DDL's, *Data Definition Language*, das tabelas externas possuem alguns detalhes importantes para sua criação e utilização:

- O *CREATE TABLE* habitualmente usado para tabelas internas deve ser alterado para *CREATE EXTERNAL TABLE*, tendo em vista que está sendo criada uma tabela externa, não uma tabela convencional;
- O formato do arquivo de texto que será lido deve ser explicitado após a definição dos nomes e respectivos tipos das colunas. No caso dos arquivos de extensão *parquet*, a DDL recebe a linha *STORED AS PARQUET*;
- A tabela deve apontar para a pasta onde os arquivos que comporão a tabela estão armazenados, como em *LOCATION 's3://bucket_name/path_up_to_the_folder_which*

*_contains_the_parquet_files/'* [6].

Segue um exemplo de DDL de criação de uma tabela externa:

```
CREATE EXTERNAL TABLE external_schema.external_table
(
COLUMN1 TYPE,
... ...,
... ...,
LAST_COLUMN TYPE
)
STORED AS PARQUET
LOCATION 's3://bucket/key';
```

## 4.2 A tarifação da leitura das tabelas

Com o *Redshift Spectrum*, são cobrados US$ 5,00 por *terabyte* de dados examinados, arredondados para cima para o *megabyte* mais próximo, com um mínimo de 10 MB por consulta. Por exemplo, se você verificar 10 *gigabytes* de dados, a cobrança será de US$ 0,05. Se você verificar 1 *terabyte* de dados, a cobrança será de US$ 5,00 [13].

Como o *Redshift Spectrum* lê as tabelas de forma colunar e os dados que as compõem também estão estruturados de forma colunar, consultas que selecionem poucas colunas, que tenham *where* bem definidos em colunas específicas e afins, reduzem o custo das consultas. Já consultas que selecionem todas as colunas da tabela, com *join* e afins, mesmo que usem *group by* ou *limit*, serão mais dispendiosas, pois não é avaliado a quantidade de informação trazida ao usuário pela *query*, mas sim a quantidade de dados que são consultados para selecionar a informação requerida [13].

## 4.3 Chave de partição

Ao dividir os dados exportados em partições, pode-se restringir a quantidade de varreduras de dados do *Redshift Spectrum* filtrando pela chave de partição.

Uma prática comum é dividir os dados com base no tempo. Por exemplo, pode-se escolher a partição por ano, mês, data e hora. Se houver dados vindos de várias origens,

pode-se dividi-los em partições por um identificador de fonte de dados e por data.

Para dividir seus dados em partições:

- Armazene os dados em pastas no *Amazon* S3 de acordo com sua chave de partição;
- Crie uma pasta para cada valor de partição e nomeie a pasta com a chave e o valor da partição. Por exemplo, para dividir-se por data, deve-se organizar os dados em pastas denominadas nome_da_coluna=AAAA-MM-DD_0, nome_da_coluna=AAAA-MM-DD_1, e assim por diante. O *Redshift Spectrum* faz uma varredura dos arquivos na pasta da partição e em todas as subpastas, ignorando os arquivos ocultos e os arquivos que começam com um ponto, um sublinhado ou uma cerquilha ou terminam com um til;
- Crie uma tabela externa e especifique a chave de partição na cláusula *PARTITIONED BY*;
- A chave de partição não pode ser o nome de uma coluna da tabela. O tipo de dados pode ser qualquer tipo padrão do *Amazon Redshift*, exceto *timestamptz*;
- Adicione as partições, como abaixo:

```
ALTER TABLE nome_do_schema.nome_da_tabela
ADD PARTITION(nome_da_chave=valor_da_chave)
LOCATION caminho_da_chave_no_S3
```

Adicione cada partição especificando a coluna e o valor da chave de partição, além do local da pasta de partição no *Amazon* S3. Você pode adicionar várias partições em um único comando [14].

# Capítulo 5

# A validação das tabelas

A validação dos dados ocorre de forma simples, entretanto não deve-se diminuir a importância desta etapa, pois é ela que garante que a exportação ocorreu corretamente e que os dados exportados podem ser removidos do banco de dados de origem. São somente duas etapas: avaliação quantitativa e avaliação qualitativa.

## 5.1 Avaliação quantitativa dos dados exportados

É realizada uma contagem dos dados exportados, tanto no banco de dados de origem quanto no banco de dados externo (o banco de dados de destino). Caso a quantidade de linhas exportadas do banco de origem seja igual a quantidade de linhas inseridas no banco de destino, a contagem está validada.

Abaixo estão dois exemplo de consulta para validação quantitativa, a primeira no banco de origem e a segunda no banco de destino, utilizando chave de partição em data no formato AAAA-MM-DD:

```
SELECT COUNT(1) FROM tabela_A
WHERE coluna_de_controle_de_data
    BETWEEN 'AAAA-MM-DD 00:00:00' AND 'AAAA-MM-DD 23:59:59';

SELECT COUNT(1) FROM esquema_externo.tabela_A
WHERE chave_de_partição='AAAA-MM-DD';
```

Caso a contagem dos dados nos dois bancos de dados em questão não represente a mesma quantidade de linhas nos dois bancos de dados, então, o processo de exportação deve ser refeito.

## 5.2 Avaliação qualitativa dos dados exportados

Após obter sucesso na avaliação quantitativa, é necessário realizar a avaliação qualitativa. Ela consiste em ler a tabela externa criada para garantir que as informações contidas na tabela de origem foram corretamente exportadas.

É comum receber o erro de "fetch" nesse momento da validação. Este erro revela que ao menos uma coluna da tabela não está sendo lida da forma como foi tratada. Por exemplo, no código em Python, converteu-se o conteúdo de determinada coluna para *int*, enquanto a tabela externa foi criada com uma DDL que descrevia esta coluna como *bigint* - note que qualquer outro tipo de coluna diferente do especificado no código em Python causaria este erro.

Obtendo sucesso nesta etapa de avaliação qualitativa, conclui-se a validação da tabela exportada - ou do intervalo exportado, caso os testes sejam realizados em somente um intervalo, o que é comum devido ao tamanho das tabelas.

Posto que o momento da validação qualitativa dos dados exportados é o primeiro momento de leitura dos dados da tabela externa gerada, é importante saber especificidades do funcionamento de uma tabela externa com e sem chaves de partição:

- Tabelas externas com ao menos uma chave de partição:
  - Consultas de contagem com chave de partição especificada leem somente os dados que estiverem dentro da pasta da partição determinada;
  - Consultas de contagem sem chave de partição especificada leem todos os dados da tabela;
  - Consultas do tipo *SELECT* com chave de partição especificada leem somente os dados que estiverem dentro da pasta da partição determinada;
  - Consultas do tipo *SELECT* sem chave de partição especificada não leem e não retornam dados.
- Tabelas externas sem chave de partição:
  - Consultas de contagem com chave de partição especificada consideram a chave como um filtro de um *WHERE*, e por isso precisam ler todos os dados da tabela;

- Consultas de contagem sem chave de partição especificada leem todos os dados da tabela;
- Consultas do tipo *SELECT* com chave de partição especificada usam o mesmo procedimento do caso das consultas de contagem com chave de partição em tabelas sem chave de partição;
- Consultas do tipo *SELECT* sem chave de partição especificada leem a tabela inteira.

# Capítulo 6

# A exclusão das linhas exportadas

Este é o momento mais crítico de toda a operação, mas concomitantemente é o mais simples.

Pode-se perder dados importantes caso a exportação não tenha sido de forma correta e tenha sido erroneamente validada.

Outro fator importante é que o comando *delete* é um dos mais complexos, computacionalmente, para o banco de dados realizar, causando demora na realização do processo e podendo até mesmo bloquear a tabela que está sendo deletada para realização de novas consultas por determinado período. É importante que o procedimento de remoção das linhas exportadas de determinada tabela seja feito no momento de menor uso tanto da tabela quanto do banco de dados no geral.

## 6.1 Boas práticas para exclusão de dados

Como forma de evitar a sobrecarga tanto do banco de dados quanto da tabela no processo de exclusão das linhas da tabela exportada, cita-se dois erros comuns:

- Erro 1390 - A requisição contém muitos espaços reservados

  Quando realiza-se uma requisição de exclusão que contemple mais de $2^{16} - 1$ linhas, retorna-se o erro 1390 - *Prepared statement contains too many placeholders*. Isto acontece porque supera-se o limite operacional do banco de dados. Note que não é recomendado que seja alterado o valor definido neste limite. Entretanto, o item explanado abaixo salienta a não-necessidade da preocupação com este erro.

- Bloqueio por exceder o tempo limite de resposta

  Quando muitas consultas são executadas em um banco de dados, a carga de trabalho (*workload*) do banco de dados aumenta. Quando a carga de trabalho supera 1.0 (entende-se que 1.0 é quantidade de carga de trabalho que o processador de um dispositivo consegue processar simultaneamente. Cargas superiores a 1.0 geram filas de processamento, enquanto cargas inferiores a 1.0 não consomes toda capacidade computacional do dispositivo em questão), gera-se uma fila de requisições - o que é normal para um banco de dados de produção. O problema surge quando na fila de consultas há uma requisição que provoque uma grande atualização do conteúdo da tabela, exigindo um longo tempo de processamento. As requisições que somente retornam dados ficam paradas aguardando o fim da atualização da tabela para somente depois serem efetuadas, posto que a tabela está sendo alterada e isto pode influenciar nos dados retornados. Portanto, como forma de evitar que outras aplicações fiquem paradas esperando a resposta da tabela, é recomendado que *commits* no banco de dados sejam feitos de pequenas em pequenas porções. Habitualmente, no projeto deste documento, exclui-se linhas em grupos de 1000 - o valor de 1000 foi um parâmetro considerado ideal para a infraestrutura vigente no momento dos testes de exclusão. Maior quantidade de linhas excluídas simultaneamente aumenta a probabilidade de *lock timeout* da tabela, enquanto menor quantidade de linhas excluídas simultaneamente diminui a eficiência das exclusões.

  Note que o erro de bloqueio por exceder o tempo limite de resposta ocorre quando uma requisição estoura o tempo limite na fila de consultas.

O trecho de código abaixo é composto por uma sugestão de como garantir que os dois problemas supracitados não ocorrerem:

```
query = dao.read_query('delete_tabela_A')

#  a linha abaixo tem como única funcionalidade adentrar ao ``while''
rowcount = 1000
while rowcount == 1000:
    dao.mysql.cursor.execute(query)
    rowcount = dao.mysql.connection.rowcount
```

Note que dao é o objeto de uma classe de acesso a dados, do inglês *Data Object Access*. O método *read_query* lê o arquivo com nome passado como parâmetro, em extensão ".sql", o qual contém a consulta de exclusão de linhas. A classe de acesso a dados também controla a conexão com os dois bancos de dados do projeto, como exemplificado acima no parâmetro "dao.mysql": este parâmetro possui parâmetros da biblioteca *MySQL Connector* que gerenciam a execução das consultas (*cursor*) e que gerenciam a conexão com o banco de dados (*connection*).

Idealmente, a *query* de remoção de linhas limita por si própria a quantidade de linhas afetadas, como a seguir:

```
DELETE FROM tabela_A
WHERE coluna_controle_exportada = conteúdo_da_coluna
LIMIT 1000;
```

Obtendo sucesso neste procedimento, o espaço em disco do banco de dados aumenta.

# Capítulo 7

# Conclusão e sugestões para trabalhos futuros

## 7.1 Conclusão

Este projeto utilizou diversos serviços da empresa *Amazon Web Services*, os principais foram RDS, *Redshift*, *Redshift Spectrum*, EC2 e S3.

*Softwares* e ferramentas foram desenvolvidos para possibilitar a exportação de tabelas de uma instância RDS para o S3, construindo um repositório de dados que pode ser lido se for apontado por uma tabela externa, utilizando o *Redshift Spectrum*.

Com a construção deste repositório de dados e a implementação de diversas tecnologias, como a extensão *parquet*, a compressão dos dados e a utilização de chaves de partição, ler os dados exportados tornou-se praticável, simples e barato.

Isto permite uma série de estudos relacionados aos dados, como predição de informações, aprendizado de máquina e afins. Entretanto, a principal vantagem da exportação dos dados é a discorrida no corpo deste documento: a exclusão dos dados do banco de dados de origem, liberando espaço em disco da instância RDS. Uma instância RDS que faz menos uso de disco pode custar menos do que uma instância RDS que faz muito uso de disco, caso tal uso ultrapasse o limite gratuito concedido na contratação da instância. Além disto, bancos de dados com tabelas pequenas têm desempenho melhor do que bancos de dados com tabelas grandes. Pense que um mesmo processador que procura 1.000 linhas de uma tabela em seu disco as encontra mais rapidamente do que o mesmo processador procurando 1.000.000 linhas de uma mesma tabela, no mesmo disco.

Este projeto foi realizado por meio de uma parceria entre a empresa Mobi2buy, onde o autor é desenvolvedor de softwares, e a empresa *Amazon*. Por mensagem enviada por e-mail a empresa, a *Amazon* classificou o projeto como **estado da arte**.

## 7.2 Sugestões para trabalhos futuros

Com base no trabalho desenvolvido, três vertentes de trabalhos futuros podem ser identificadas.

Utilização de chave de partição composta, pois esta garante sub-filtros que diminuem mais ainda a tarifação da leitura das tabelas externas, além de melhorar o desempenho das consultas na tabela.

Estudar maneira de extinguir a necessidade da substituição de inteiros nulos para geração de arquivos *parquet*.

Estudar a mesma aplicação utilizando *PySpark*, que é a uma *framework* para computação em *cluster*, feito para ser de fácil uso, rápido e para análise dados. A maior vantagem da utilização *PySpark* frente ao *Pandas* é que esta *framework* não salva os dados lidos em memória RAM, mas sim em disco, fazendo com que o tamanho máximo do quadro de dados lido seja muito maior do que com o *Pandas* [2].

# Referências Bibliográficas


[1] APACHE. **Apache Parquet**. Disponível em: <http://parquet.apache.org/documentation/latest/>. Online; Acesso em: 22 Nov. 2019.

[2] APACHE SPARK. **Spark**. Disponível em: <https://spark.apache.org/docs/latest/api/python/index.html>. Online; Acesso em: 28 Nov. 2019.

[3] AWS-EC2. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/ec2/?nc1=h_ls>. Online; Acesso em: 10 Dez. 2018.

[4] AWS EC2. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/ec2/instance-types/?trk=ec2_landing_def>. Online; Acesso em: 15 Jul. 2019.

[5] AWS EC2. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls>. Online; Acesso em: 20 Jul. 2019.

[6] AWS EXTERNAL TABLES. **Amazon Web Services**. Disponível em: <https://docs.aws.amazon.com/redshift/latest/dg/r_CREATE_EXTERNAL_TABLE.html>. Online; Acesso em: 24 Jul. 2019.

[7] AWS-RDS. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/rds/?nc1=h_ls>. Online; Acesso em: 02 Dez. 2018.

[8] AWS RDS. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/rds/mysql/pricing/?pg=pr&loc=2>. Online; Acesso em: 10 Jul. 2019.

[9] AWS-REDSHIFT. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/redshift/?nc1=h_ls>. Online; Acesso em: 03 Dez. 2018.

[10] AWS REDSHIFT. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/redshift/pricing/?nc1=h_ls>. Online; Acesso em: 10 Jul. 2019.

[11] AWS-REDSHIFT-SPECTRUM. **Amazon Web Services**. Disponível em: <https://docs.aws.amazon.com/en_pv/redshift/latest/dg/c-using-spectrum.html>. Online; Acesso em: 04 Dez. 2018.

[12] AWS REDSHIFT SPECTRUM. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/blogs/big-data/10-best-practices-for-amazon-redshift-spectrum/>. Online; Acesso em: 28 Nov. 2019.

[13] AWS REDSHIFT SPECTRUM. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/blogs/aws/amazon-redshift-spectrum-exabyte-scale-in-place-queries-of-s3-data/>. Online; Acesso em: 30 Jul. 2019.

[14] AWS REDSHIFT SPECTRUM. **Amazon Web Services**. Disponível em: <https://docs.aws.amazon.com/redshift/latest/dg/c-spectrum-external-tables.html>. Online; Acesso em: 02 Ago. 2019.

[15] AWS-S3. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/s3/?nc1=h_ls#>. Online; Acesso em: 03 Dez. 2019.

[16] AWS S3. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/s3/storage-classes/>. Online; Acesso em: 10 Jul. 2019.

[17] AWS S3. **Amazon Web Services**. Disponível em: <https://aws.amazon.com/s3/pricing/?nc1=h_ls>. Online; Acesso em: 10 Jul. 2019.

[18] BEN FORTA. **Sams Teach Yourself SQL in 10 Minutes**.

[19] DEVMEDIA. **Banco de Dados Relacionais**. Disponível em: <https://www.devmedia.com.br/bancos-de-dados-relacionais/20401>. Online; Acesso em: 24 Nov. 2019.

[20] MYSQL 8.0. **MySQL 8.0 Reference Manual**. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/datetime.html>. Online; Acesso em: 24 Jul. 2019.

[21] ORACLE. **MySQL Connector 8.0.16**. Disponível em: <https://dev.mysql.com/doc/connector-python/en/>. Online; Acesso em: 21 Jul. 2019.

[22] OSVALDO KOTARO TAKAI, ISABEL CRISTINA ITALIANO E JOÃO EDUARDO FERREIRA. **Banco de Dados**. Disponível em: <https://www.ime.usp.br/~jef/apostila.pdf>. Online; Acesso em: 24 Nov. 2019.

[23] PANDAS 0.25.3. **Pandas 0.25.3 Documentation**. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/>. Online; Acesso em: 10 Jan. 2019.

[24] PYARROW 0.15.1. **Pyarrow 0.15.1 Manual**. Disponível em: <https://pypi.org/project/pyarrow/>. Online; Acesso em: 10 Jan. 2019.

[25] PYPI. **Boto3 1.9.157**. Disponível em: <https://pypi.org/project/boto3/>. Online; Acesso em: 21 Jul. 2019.

[26] PYPI. **Psycopg2-Binary 2.8.4**. Disponível em: <https://pypi.org/project/psycopg2-binary/>. Online; Acesso em: 21 Jul. 2019.

[27] PYTHON ORG. **Python/C API**. Disponível em: <https://docs.python.org/3/c-api/index.html>. Online; Acesso em: 20 Jul. 2019.