

Universidade Federal do Pampa

Uillian Luiz Ludwig

**Desenvolvimento de um Sistema para
Integração e Consulta de Rotas de Ônibus
Intermunicipais**

Alegrete

2015

Uillian Luiz Ludwig

Desenvolvimento de um Sistema para Integração e Consulta de Rotas de Ônibus Intermunicipais

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Claudio Schepke

Alegrete

2015

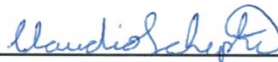
Uillian Luiz Ludwig

Desenvolvimento de um Sistema para Integração e Consulta de Rotas de Ônibus Intermunicipais

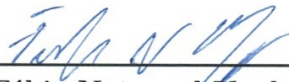
Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 2 de dezembro de 2015

Banca examinadora:



Claudio Schepke
Orientador



Fábio Natanael Kepler
UNIPAMPA



Jean Felipe Cheiran
UNIPAMPA

Resumo

Este trabalho relata a criação de um sistema para a consulta de rotas de ônibus intermunicipais do estado do Rio Grande do Sul. Ele foi motivado pela falta de sistemas que proveem este serviço de forma eficiente. Para o desenvolvimento deste projeto, decidiu-se pela criação de uma base de dados que foi construída a partir da extração de dados das fontes oficiais, como sites de rodoviárias. Além da criação de métodos para consultas simples aos dados, algoritmos de busca de caminhos foram aplicados nos dados para encontrar as melhores rotas entre uma cidade de origem e uma cidade de destino. Estes algoritmos foram disponibilizados aos usuários por meio de interfaces *web* e *mobile*. Durante este trabalho, a criação da base de dados teve os maiores desafios, onde pode-se aplicar técnicas para extrair, validar e expandir as informações sobre rotas. Além disso, durante a implementação dos algoritmos de busca, pode-se analisar e executar técnicas para a melhora do desempenho e qualidade dos resultados. O resultado final foi um sistema chamado ÔnibusRS, que pode ser acessado pelo endereço www.onibusrs.com.

Palavras-chave: Extração de Dados. Busca de Caminhos. Programação Web e Mobile.

Abstract

This project describes the creation of a system that provides a way to verify intercity bus services of the Brazilian state Rio Grande do Sul. It was motivated by the lack of systems that provide this service in an efficient way. In the development of this project, a database was built through data extraction from the official sources, such as the bus stations websites. In addition to the creation of a simple way to view the data, a path search algorithm was applied to the data in order to find complex routes between two cities. This algorithm was made available using a web and a mobile interface. During this project, the creation of the database had the most challenges, where techniques to extract, validate and expand could be applied. In addition, the path search algorithm was analysed and updated in order to achieve a better performance and result qualities. The results of this project was a system called ÔnibusRS, which can be viewed on www.onibusrs.com.

Key-words: Data Extraction. Path Finding. Web and Mobile Programming.

Lista de ilustrações

Figura 1 – Parte do arquivo robots.txt do Twitter.	20
Figura 2 – Exemplo de um arquivo XML.	21
Figura 3 – Diagrama de um sistema de banco de dados.	22
Figura 4 – Diagrama da visão lógica da arquitetura do MySQL.	24
Figura 5 – Diagrama da arquitetura do PostgreSQL.	25
Figura 6 – Grafo com relacionamento ama.	27
Figura 7 – Diagrama de execução do protocolo HTTP.	32
Figura 8 – Diagrama simplificado da pilha do Android.	34
Figura 9 – Diagrama de camadas do iOS.	35
Figura 10 – Procedimento da criação de uma aplicação móvel híbrida.	36
Figura 11 – Conjunto de passos a serem executados durante este projeto.	41
Figura 12 – Representação simplificada do modelo E/R do banco de dados.	45
Figura 13 – Exemplo do uso de distância para o tratamento dos nomes.	45
Figura 14 – Todos destinos de uma rota saindo de Alegrete.	46
Figura 15 – Rotas extras obtidas a partir da rota de Alegrete.	46
Figura 16 – Retorno da consulta utilizando o comando <code>allShortestPaths</code>	48
Figura 17 – Retorno da consulta com rotas ordenadas pelo preço.	49
Figura 18 – Impacto na utilização de índices em banco de dados no tempo de execução de três diferentes consultas.	52
Figura 19 – Formulário do sistema web para busca de uma rota.	53
Figura 20 – Resultado da busca entre Alegrete e Barra do Quaraí.	54
Figura 21 – Formulário para a visualização e o envio de comentários sobre uma rota.	56
Figura 22 – Formulário para cadastro de novas rotas.	56
Figura 23 – Tela de busca e resultado no aplicativo Android	59
Figura 24 – Tela de busca e resultado no aplicativo iOS	60

Lista de tabelas

Tabela 1 – Comparação de velocidade de consultas em SGBDs	26
Tabela 2 – Comparação entre serviços	38

Lista de siglas

ACID Atomicidade, Consistência, Isolamento e Durabilidade

API *Application Program Interface*

APK *Android Package File*

CSS *Cascading Style Sheets*

CSV *Comma-separated values*

CWP *Centos Web Panel*

DEX *Dalvik Executable Format*

DNS *Domain Name System*

HTML *HyperText Markup Language*

HTTP *Hypertext Transfer Protocol*

JSON *JavaScript Object Notation*

JVM *Java Virtual Machine*

SGBD Sistema Gerenciador de Banco de Dados

SQL *Structured Query Language*

VPS *Virtual Private Server*

XML *EXtensible Markup Language*

Sumário

1	INTRODUÇÃO	17
2	FUNDAMENTAÇÃO	19
2.1	Extração de Dados	19
2.2	Banco de Dados	21
2.2.1	Modelo Relacional	22
2.2.2	NoSQL	26
2.3	Algoritmos de Busca de Caminhos	28
2.3.1	Busca Não Informada	29
2.3.2	Busca Informada	30
2.4	Programação Web	31
2.5	Programação para Dispositivos móveis	33
3	PROJETOS RELACIONADOS	37
4	METODOLOGIA	41
5	DESENVOLVIMENTO	43
5.1	Extração de Dados	43
5.2	Testes com Bancos de dados NoSQL	47
5.3	Algoritmos de Busca	49
5.4	Programação Web	53
5.5	Programação para Dispositivos Móveis	57
5.5.1	Android	57
5.5.2	iOS	58
6	CONCLUSÃO	61
	REFERÊNCIAS	63

1 Introdução

A internet é uma grande fonte de informações disponíveis para usuários de todo o mundo. Tais informações são apresentadas para os usuários em formato de páginas de internet. O propósito destas páginas são os mais variados, podendo-se encontrar páginas com foco para entretenimento, informação, educação, entre outros.

No Brasil são mais de 100 milhões de usuários na internet (IBGE, 2014), número que tem aumentado significativamente nos últimos anos. Para atender as necessidades de todos usuários, os criadores de conteúdo para internet precisam analisar e entender as exigências de cada um, fato que muitas vezes não ocorre. Uma significativa parte dos sites existem apenas pela necessidade da empresa ter seu nome divulgado na internet, o que torna o seu uso frustrante (GIBSON, 2015). Em tais casos, ao invés do site contribuir para o usuário, ele acaba não trazendo as informações que o usuário procura ou as traz de forma ineficiente. Nessas situações uma análise e remodelação das páginas se faz necessária.

Para ilustrar este caso pode ser utilizado como exemplo empresas de transporte de passageiros. Tais empresas normalmente possuem em seu site uma ferramenta para listar as rotas que seus ônibus percorrem. O mesmo ocorre para muitas estações rodoviárias, que listam todos os ônibus que de lá partem. Apesar de muitos destes sites serem gerenciados por uma mesma empresa, ou empresas parceiras, eles não possuem uma ligação entre si, e o dado pertencente a um não pode ser visto ou utilizado pelo outro, fato que limita e dificulta o processo de checagem das informações por parte dos usuários. Existem alguns sistemas que tentam integrar tais dados, porém todos possuem limitações. Dentre elas pode ser citado o fato de não combinar rotas (obrigando o usuário a realizar múltiplas consultas) e também o fato de possuir informações incompletas ou desatualizadas. Além disso, em muitos casos, estes sistemas possuem uma segmentação em seu conteúdo, onde os dados, em sua maior parte, são de regiões específicas do Brasil ou do mundo.

Para resolver esse problema, este trabalho propõe um sistema para integração de rotas de ônibus, possibilitando a obtenção de informações completas em apenas uma consulta. Este trabalho também irá seguir a ideia de segmentação, que terá como objetivo principal a cobertura de todos os municípios do estado do Rio Grande do Sul. Tal fato, porém, não irá limitar a expansão dos dados, que será possível apenas com o aprendizado de novas rotas. Além disso, as ferramentas que serão desenvolvidas poderão ser utilizadas para a extração de outros dados, com a adaptação do código para o sistema desejado.

Organização

Este trabalho será organizado com a seguinte divisão de capítulos:

- O [Capítulo 2](#) discutirá os principais conceitos que cercam este trabalho. Dentre eles pode ser citado a extração de dados, os algoritmos de busca de caminhos e a programação web.
- O [Capítulo 3](#) focará nos projetos relacionados. Nesta seção será discutido o que já existe de projetos com propósitos similares, analisando os pontos positivos e negativos de cada um e o que um novo sistema deverá fazer diferente.
- O [Capítulo 4](#) mostrará a metodologia que será empregada neste trabalho, mostrando os passos de desenvolvimento do sistema, desde a criação do banco de dados até a programação dos aplicativos para os usuários.
- O [Capítulo 5](#) mostrará detalhadamente cada passo do desenvolvimento deste trabalho. Listando os problemas enfrentados e as soluções encontradas para cada um deles.
- O [Capítulo 6](#) finalizará o trabalho, discutindo as metas e objetivos alcançados. Além disso, foram planejadas possíveis melhorias para o futuro.

2 Fundamentação

2.1 Extração de Dados

A web possui um imenso conjunto de dados com informações prontas para serem coletadas e analisadas. Dados de clima, preços de produtos e tópicos em alta de redes sociais podem ser retirados de páginas da internet. Tais dados vão gerar informações que podem ser utilizadas de diversas maneiras. Por exemplo, preço de produtos coletados de vários sites podem ser utilizados para encontrar lojas com melhores preços e dados das redes sociais podem ser utilizados para descobrir o gosto dos consumidores.

Essa coleta de dados pode ser feita manualmente, copiando os dados da fonte e colando em uma tabela como a de um banco de dados. Esse trabalho manual é demorado, e muitas vezes erros de integridade acontecem. Para isso existe *web scraping*, termo em inglês que nomeia o ato de extrair informações da internet de forma automatizada (HANRETTY, 2013).

Motores de busca como o do Google utilizam esse conceito para navegar por páginas da internet, coletando informações sobre cada uma para montar um índice de busca. Esse tipo de *scraping* tem como propósito indexar um grande número de informações. Para isso ele utiliza um modelo que percorre todas as páginas referenciadas a partir de uma página inicial. Esse conceito é chamado de *web spider* ou *web crawler* (HANRETTY, 2013).

Web scraping é utilizado muitas vezes para prover um serviço mais completo do que o atualmente disponibilizado. Por exemplo, no Brasil é comum em épocas de eleições grandes empresas de mídia disponibilizarem formas para a consulta da apuração das eleições. Sem a utilização de uma API (*Application Program Interface*) para consultas direto ao banco de dados da Justiça Eleitoral tem-se a necessidade do uso do *web scraping*. As empresas coletam os dados do site oficial e podem então gerar uma forma de interação para seus usuários utilizando gráficos e tabelas.

Essa coleta de informações é feita por um robô que percorre as páginas e coleta os dados necessários. É comum em sites o uso de um arquivo chamado `robots.txt` que informa ao robô quais páginas ele pode ou não pode acessar. Isso evita que dados privados sejam coletados ou disponibilizados em motores de busca. Um exemplo deste arquivo pode ser visto na Figura 1. Ele deixa claro quais páginas a empresa gostaria ou não que fossem acessadas por um robô. Isso, porém, não garante que todos robôs vão respeitar o que está definido no arquivo. Por questões legais os robôs devem seguir tais indicações.

Figura 1 – Parte do arquivo robots.txt do Twitter.

```
# Every bot that might possibly read and respect this file.
User-agent: *
Allow: /*?lang=
Allow: /hashtag/*?src=
Allow: /search?q=%23
Disallow: /search/realtime
Disallow: /search/users
Disallow: /search/*/grid
```

Fonte: [Twitter \(2015\)](#)

Para coletar dados de redes sociais, como o Twitter, *scraping* provavelmente não é a melhor opção, pois ele disponibiliza [APIs](#) para a realização de requisições. Desse modo, consegue-se criar mecanismos para coletar e enviar informações utilizando recursos disponibilizados pela própria empresa. Porém, nem todos os sites possuem [APIs](#) e o *web scraping* torna-se um método importante para a coleta de dados.

Outro fator para levar em consideração ao utilizar esta técnica é a quantidade de requisições a serem feitas em um site. Muitos acessos podem causar problemas como negação de serviço e limite de banda utilizada no servidor. Desse modo, quando testes estiverem sendo feitos é preferível a execução local, ou com um número limitado de comunicações com o servidor. Evitando-se assim que o tráfego não necessário atrapalhe ambas as partes.

As técnicas utilizadas para *web scraping* incluem a utilização de expressões regulares e *parsers HTML* (*HyperText Markup Language*) ([ROEBUCK, 2012](#)). Com utilização de expressões regulares tem-se a necessidade de analisar o conteúdo [HTML](#) de cada página e criar expressões para cada parte do conteúdo que se deseja coletar. Esse método, porém, torna-se complicado quando tem-se muitos dados para extrair e quando o [HTML](#) não possui uma boa formatação. Por outro lado, os *parsers HTML* conseguem ler páginas [HTML](#) mal formatadas e possibilitam que consultas sejam realizadas. Estes *parsers* utilizam uma forma de consulta semelhante ao XPath ([ROEBUCK, 2012](#)).

XPath é um linguagem de consulta para documentos [XML](#) (*EXtensible Markup Language*) que possibilita a navegação entre elementos e atributos em um arquivo [XML](#) ([W3SCHOOLS, 2015](#)). Exemplos de expressões XPath para o arquivo da [Figura 2](#) podem ser: `bookstore/book` que irá selecionar todos elementos `book` que são filhos de `bookstore`; `/bookstore/book[last()]` que irá selecionar apenas o último filho; e `/bookstore/book[price>35.00]/title` que irá selecionar o título dos livros que possuem preço maior que \$35,00.

Figura 2 – Exemplo de um arquivo XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>
<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>
</bookstore>
```

Fonte: [W3Schools \(2015\)](#)

Além destes conceitos aqui introduzidos, outro termo que é bastante utilizado neste contexto é mineração de dados. Mineração de dados descreve o ato de adquirir conhecimento sobre um grande conjunto de dados ([HAN, 2005](#)). Por exemplo, com a análise de postagens em redes sociais pode-se encontrar padrões que demonstram o interesse de pessoas por determinada marca ou produto. Não existe uma técnica específica para realizar mineração de dados, mas sim um modelo de análise específico para cada problema.

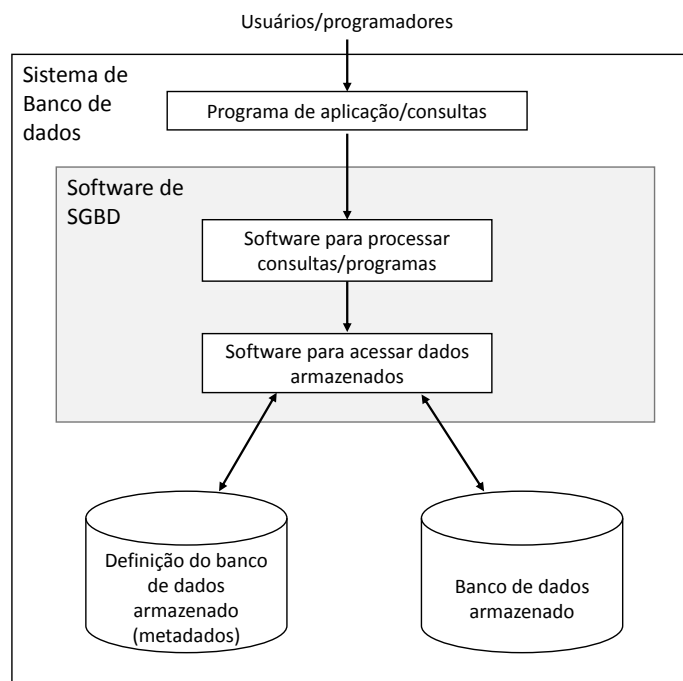
2.2 Banco de Dados

Em ciência da computação, bancos de dados possuem uma grande importância por serem a base de muitos sistemas computacionais. O banco de dados gerencia as informações que o sistema produz, provendo formas de salvá-las, recuperá-las e modificá-las de acordo com a necessidade do sistema. Esse gerenciamento é feito com a utilização de um Sistema Gerenciador de Banco de Dados ([SGBD](#)), que possui como objetivo principal prover uma forma de salvar e recuperar dados de maneira conveniente e eficiente ([SILBERSCHATZ; KORTH; SUDARSHAN, 2011](#)).

Antes da utilização de [SGBDs](#), programadores criavam suas próprias ferramentas para o controle dos dados. Nestes casos os dados eram salvos em algum formato previamente definido, um arquivo de texto, por exemplo, e uma aplicação era criada para fazer a leitura e escrita dos dados. Este método de controle de dados é uma boa solução quando o volume de dados não for grande. Porém, caso a aplicação tenha de trabalhar com um volume alto de dados, o gerenciamento destas informações torna-se difícil. Alguns problemas enfrentados são: redundância, inconsistência, integridade e segurança. Estes problemas são resolvidos com a utilização de um [SGBD](#) ([SILBERSCHATZ; KORTH; SUDARSHAN, 2011](#)).

SGBD é uma coleção de programas capazes de criar e manter um banco de dados sem a necessidade do usuário se preocupar em como ele faz isso (ELMASRI; NAVATHE, 2005). O **SGBD** provê formas de definição, construção, manipulação e compartilhamento de um banco de dados. A fase de definição possibilita ao usuário descrever uma determinada estrutura para o banco de dados, que envolve tipos, tamanhos e restrições. Tal estrutura é chamada de metadados e é utilizada para controlar a consistência e integridade do banco de dados. Na fase de construção, o banco de dados recebe informações e as armazena. A fase de manipulação permite que o usuário faça consultas e obtenha informações específicas sobre os dados. Na fase de compartilhamento, outras aplicações acessam o banco de dados, realizando consultas e modificações (ELMASRI; NAVATHE, 2005). A Figura 3 é uma representação de como o banco de dados executa cada uma das fases mencionadas.

Figura 3 – Diagrama de um sistema de banco de dados.



Fonte: Elmasri e Navathe (2005, p. 4)

2.2.1 Modelo Relacional

Um banco de dados relacional consiste de uma ou mais tabelas. Estas tabelas são chamadas de relações e recebem um identificador único. Uma relação deve possuir atributos, que são as colunas da tabela. Estes atributos normalmente possuem um tipo associado; CHAR para caracteres, INTEGER para números inteiros, DATE para data, por exemplo. Quando uma relação é preenchida com dados, tem-se várias linhas em uma

tabela; tais linhas são chamadas de tuplas (SILBERSCHATZ; KORTH; SUDARSHAN, 2011).

Uma relação pode possuir uma chave, que será um ou mais atributos da relação que irá diferenciar cada uma das tuplas. Estas chaves facilitam a organização do arquivo, pois o SGBD pode as usar como índices para recuperar os dados de forma eficiente. Para realizar consultas em um banco de dados relacional tem-se a utilização de linguagens de consultas (*query language*). A álgebra relacional define um conjunto de operadores que podem ser executados em uma relação. Entre eles estão o operador de seleção que retorna todas as tuplas que satisfazem uma condição informada e projeção que retorna os atributos desejados.

Para a execução em um SGBD a linguagem mais utilizada é o SQL (*Structured Query Language*). Ela não apenas provê formas de consulta, mas também possibilita formas de criação e modificação de relações, inserção e modificação de dados e também alteração de configurações do banco de dados. SQL é uma linguagem bem consolidada e é utilizada pela maioria dos SGBDs disponíveis atualmente, dentre eles MySQL, PostgreSQL e SQLite.

Comandos SQL são normalmente executados dentro de uma transação. Uma transação pode ser definida como um conjunto de uma ou mais operações que juntas constituem uma operação lógica. Usualmente, uma transação é definida por delimitadores de início e fim e todos comandos internos aos delimitadores constituem uma transação. Banco de dados relacionais normalmente executam transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Uma transação é atômica se todas as operações em uma transação são completas com sucesso ou nenhuma é. Se um comando durante uma transação falhar, toda transação falha. Para consistência, cada transação leva o banco de dados de um estado válido para outro estado válido, preservando sua consistência e integridade. O isolamento garante que apesar de existir mais de uma transação sendo executada em paralelo, nenhuma irá interferir na outra. Para durabilidade tem-se que depois de uma transação ser completada com sucesso, ela continuará salva mesmo com falhas no sistema (SILBERSCHATZ; KORTH; SUDARSHAN, 2011)

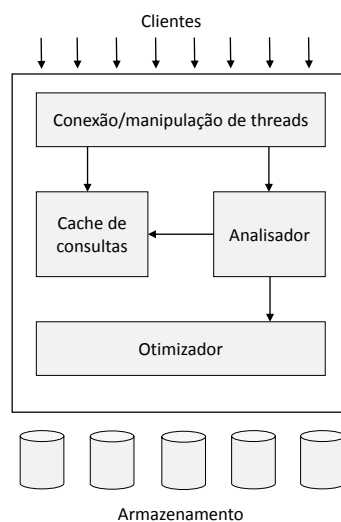
MySQL

MySQL é o banco de dados de código livre mais popular no mundo, e é utilizado por grandes empresas como Facebook, Twitter e Yahoo! (MYSQL, 2015). MySQL é disponibilizado pela Oracle em basicamente duas versões, uma gratuita e outra comercial. Em sua versão gratuita ele é comumente disponibilizado junto com Apache e PHP em um pacote chamado LAMP, fato que o torna uma fácil opção para criação de conteúdos para web.

A arquitetura de um servidor MySQL pode ser dividida em três camadas, como

pode ser visto na [Figura 4](#). Na primeira camada estão os serviços básicos de conexão, autenticação e criação de *threads*. Na segunda camada está o núcleo do MySQL, onde encontra-se a análise, otimização e *cache* das consultas, as funções nativas do MySQL e as funções criadas pelos usuários. Na terceira camada está o armazenamento, que irá diferir de acordo com o sistema de arquivos em que o ambiente está configurado. Cada disco de armazenamento, se utilizado mais de um, não se comunica com outros discos, ele apenas responde às requisições do servidor ([ZAITSEV; TKACHENKO, 2012](#)).

Figura 4 – Diagrama da visão lógica da arquitetura do MySQL.



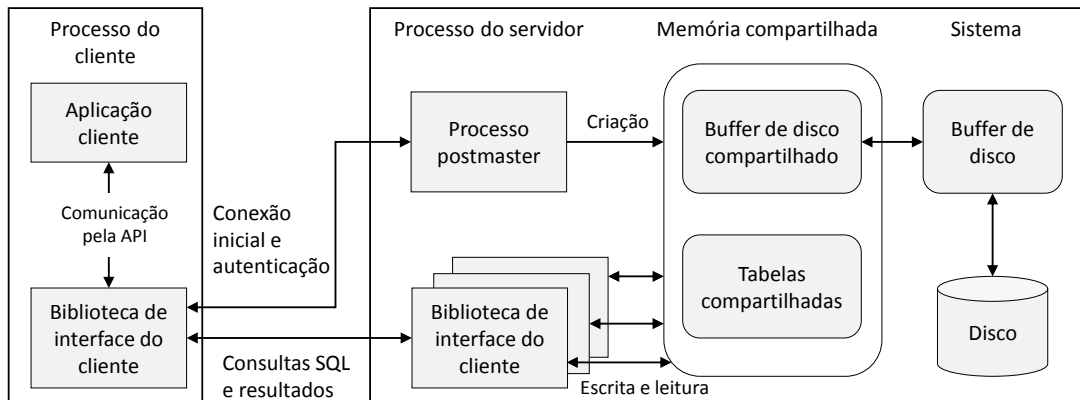
Traduzido de: [Zaitsev e Tkachenko \(2012, p. 2\)](#)

PostgreSQL

PostgreSQL é um [SGBD](#) de código fonte aberto que está a mais de 15 anos no mercado. Ele executa consultas [SQL](#) e tem suporte para os diversos tipos de dados, além de suportar o armazenamento de arquivos binários, como fotos e vídeos. PostgreSQL tem suporte nativo para várias linguagens de programação, incluindo C/C++, Java e Python ([POSTGRESQL, 2015](#)).

PostgreSQL necessita da criação de um servidor, que irá aceitar e tratar conexões dos clientes. Ele possui um processo chamado *postmaster* que aceita a conexão inicial, autentica e cria um processo para cada cliente. Após, cada cliente comunica-se diretamente com o processo criado, executando consultas e recebendo as respostas. Esse modelo, como visto na [Figura 3](#), é chamado de *process-per-connection* ([SILBERSCHATZ; KORTH; SUDARSHAN, 2011](#)).

Figura 5 – Diagrama da arquitetura do PostgreSQL.



Traduzido de: [Silberschatz, Korth e Sudarshan \(2011, p. 1154\)](#)

SQLite

O SQLite é um [SGBD](#) que é amplamente utilizado em dispositivos móveis e em pequenas aplicações para computadores. SQLite difere dos demais por não exigir a execução de um servidor. Um banco de dados SQLite é salvo em um arquivo com qualquer extensão (normalmente `.bd` ou `.sqlite`), e com a utilização de bibliotecas, permite a execução de consultas [SQL](#). Este fato pode ser uma desvantagem ao se comparar com os demais [SGBDs](#), uma vez que um usuário possui acesso ao arquivo ele poderá realizar quaisquer modificações no banco de dados.

Como o SQLite salva os dados em um único arquivo, ele não possibilita que múltiplos usuários escrevam ao mesmo tempo. O arquivo é bloqueado enquanto um usuário estiver escrevendo, e os outros usuários ficam em uma fila. Já para leituras não se tem este problema, pois múltiplos usuários podem realizar leituras simultâneas.

Um teste de comparação de velocidade dos bancos de dados foi feito pelo `sqlite.com` ([SQLITE, 2015b](#)). Este teste comparou, entre outros fatores, a velocidade de inserção e seleção de dados. Os bancos de dados comparados foram: PostgreSQL, MySQL e SQLite. O SQLite teve dois casos de testes, um em que após cada consulta ele executa uma função de sincronização esperando o término das operações de entrada e saída e a outra sem esta função. Foram executados os seguintes testes: 1000 inserções (Inserção I), 25000 inserções dentro de uma transação (Inserção II), 100 seleções com comparação de strings (Seleção) e 3 exclusões de tabelas (Deleção). Os resultados podem ser visualizados na [Tabela 1](#), onde os tempos estão registrados em segundos.

Pode ser verificado que o SQLite teve um bom resultado nos testes. Porém é bom salientar que os testes foram realizados com apenas um usuário, e que com a necessidade da utilização de múltiplos usuários o tempo provavelmente seria maior. Também, o tempo

Tabela 1 – Comparação de velocidade de SGBDs.

SGBD	Inserção I	Inserção II	Seleção	Deleção
PostgreSQL	4,373	4,900	13,409	0,135
MySQL	0,114	2,184	4,649	0,015
SQLite com sincronização	13,061	0,914	3,372	0,939
SQLite sem sincronização	0,233	0,757	3,362	0,254

para inserção com a função de sincronização, que exige mais do disco, foi significativamente maior que o sem sincronização; então, tem-se de analisar a necessidade de garantir que os dados foram escritos antes de realizar a próxima consulta. O próprio `sqlite.com` provê uma lista com dicas para escolher o melhor banco de dados. Entre as dicas estão as ocasiões em que o SQLite não é a melhor alternativa: se o banco de dados está separado da aplicação via rede, se a aplicação possuir muitas escritas concorrentes ou se forem muitos dados; em outros casos, o SQLite funcionará bem (SQLITE, 2015a).

2.2.2 NoSQL

NoSQL é uma combinação de duas palavras: No (não, em inglês) e SQL. No começo NoSQL tinha intenção de não ser um banco de dados relacional e não usar SQL. Porém, agora o termo NoSQL é definido como *Not only SQL* (não apenas SQL), e enquadra todos os bancos de dados que de certa forma não seguem o modelo tradicional de banco de dados relacionais. NoSQL não é um produto, mas sim uma definição para uma classe de produtos (TIWARI, 2011).

Em contraste aos bancos de dados relacionais tradicionais que são caracterizados por transações ACID, NoSQL é caracterizado pelo acrônimo BASE (*Basically Available, Soft state, Eventually consistent*). *Basically Available* (basicamente disponível) significa o uso de replicações dos dados em diferentes servidores para reduzir a possibilidade de o sistema ficar indisponível. *Soft state* significa que o sistema permite que os dados estejam inconsistentes e cabe ao programador o controle deste fator. *Eventually consistent* (Eventualmente consistente) significa que, apesar de em um certo momento o banco de dados esteja inconsistente, NoSQL garante que em futuro próximo os dados estarão consistentes (ORACLE, 2012).

Empresas como Google e Amazon possuem terabytes de dados e com milhares de usuários realizando operações de leitura e escrita sobre eles. Levando isso em consideração, torna-se custoso manter servidores distribuídos executando transações ACID. NoSQL, por outro lado, além de prover uma maior performance, também provê uma alta disponibilidade, atributo essencial para aplicações web (HECHT; JABLONSKI, 2011).

Levando em conta o modelo dos dados, bancos de dados NoSQL podem ser divididos em quatro categorias: chave-valor, orientado a documentos, orientado a colunas e

grafos. Chave-valor é basicamente um dicionário, onde tem-se para cada chave única um valor atribuído a ela. No modelo orientado a documentos tem-se a chave-valor armazenada em documentos [JSON](#) (*JavaScript Object Notation*). Cada documento possui um identificador único que é utilizado para distinguir um documento dos demais. O modelo orientado a colunas possui uma representação similar ao modelo relacional, porém, os dados de uma linha não estão armazenados de forma contígua, mas sim os dados de uma coluna. Esse fator facilita algumas consultas, onde busca-se apenas uma coluna de uma tabela. O modelo de grafos é especializado no gerenciamento eficiente de dados conectados e tem sua maior utilização para problemas baseados em localização, busca de caminhos e sistemas de recomendações ([HECHT; JABLONSKI, 2011](#)).

Neo4j

Neo4j é um banco de dados baseado em grafos que possui transações [ACID](#), alta disponibilidade e escalabilidade para bilhões de nodos e relacionamentos como características principais. Os dados que são armazenados são nós e relacionamentos. Cada nó pode possuir propriedades de tipo numérico, *string* ou booleano. Tem-se um caminho quando um ou mais nós estão conectados por meio de relacionamentos.

Neo4j usa sua própria linguagem para consultas chamada Chyper. Ela possibilita a descrição do que se deseja alterar, inserir ou deletar de um grafo sem requerer a descrição de como fazer isso ([NEO4J, 2015](#)). Considerando o grafo da [Figura 6](#) onde tem-se dois nodos com o relacionamento ama. Em Cypher este grafo é escrito da seguinte maneira: (a) -[:AMA]-> (b). A escrita dos nodos é feita entre parenteses, dessa forma tem-se uma representação visual como se fosse um círculo (nodo). Uma consulta com retorno de propriedades de um nodo pode ser executado com o seguinte código Chyper: `MATCH (nodo1)-[:AMA]->(nodo2) RETURN nodo2.nome`; isso irá retornar todos os nomes de quem é amado.

Figura 6 – Grafo com relacionamento ama.



Alterado: de [Neo4j \(2015\)](#)

Neo4j possui funções semelhantes ao [SQL](#) que podem ser utilizadas em uma consulta Cypher. Funções como `COALESCE` para retornar valores não nulos, `LENGTH` para retornar o tamanho de um valor, e outras funções matemáticas e de tratamento de strings. Além disso, Neo4j disponibiliza uma [API](#) para Java que possui várias funcionalidades, entre elas algoritmos para busca de caminhos em grafos, como A^* e Dijkstra.

OrientedDB

OrientDB é um banco de dados orientado a grafos e a documentos. OrientDB usa uma versão modificada do [SQL](#), suporta transações [ACID](#) e utiliza conceitos de orientação a objetos. Ele suporta diversos tipos de dados, entre eles Boolean, Integer, Float, String e Date ([ORIENTDB, 2015](#)).

Para a representação do grafo da [Figura 6](#) em OrientDB é necessário a criação de duas classes (visto no exemplo em: a, b), uma que estende V (vértice) e outra E (aresta). Após, deve-se inserir cada um dos nodos (c, d); e então cria-se uma aresta utilizando os nodos previamente criados (e). Diferente do Neo4j, OrientDB não possui uma representação visual, mas opta por manter uma sintaxe semelhante ao [SQL](#), evitando o custo de aprender uma linguagem completamente diferente.

- (a) `CREATE CLASS Nodo EXTENDS V`
- (b) `CREATE CLASS Ama EXTENDS E`
- (c) `CREATE VERTEX Nodo SET nome = 'a'`
- (d) `CREATE VERTEX Nodo SET nome = 'b'`
- (e) `CREATE EDGE Ama FROM (SELECT FROM Nodo WHERE nome = 'a')
TO (SELECT FROM Nodo WHERE nome = 'b')`

OrientDB possui um grande número de funções que executam em consultas [SQL](#), entre elas funções para grafos, coleções e cálculos matemáticos. Para funções de grafos dá-se destaque para a `shortestPath`, que retorna o caminho mais curto entre dois nodos, e `dijkstra` que retorna o caminho mais curto em um grafo valorado. Outra função interessante é a `distance`, que retorna a distância entre duas coordenadas utilizando o algoritmo de Haversine.

2.3 Algoritmos de Busca de Caminhos

Muitos problemas computacionais utilizam técnicas de busca de melhores caminhos para encontrar soluções. Em jogos de tabuleiro, por exemplo, uma solução pode ser o caminho do início do jogo até o final, buscando a vitória do jogador. Em jogos 3D, utiliza-se busca de caminho para encontrar a melhor forma de um jogador encontrar seu adversário. Em mapas, a solução encontrada será um caminho entre uma cidade e outra, sendo ele preferencialmente o mais curto. Em todos os casos tem-se a definição de problema, com seus estados, ações e custos.

Para exemplificar, [Russell e Norvig \(2009\)](#) definem o problema de busca de caminhos em um ambiente de viagens aéreas. Estados são todos os aeroportos e horário atual de cada um. Estado inicial é o aeroporto de partida informado pelo usuário. Uma ação é o ato de voar de um local para outro. Modelo de transição é o estado gerado ao executar

uma ação. Estado objetivo é o destino final informado pelo usuário, que será testado a cada ação. Custo do caminho pode incluir preço, tempo, entre outros.

Tendo conhecimento do problema, pode-se então construir métodos para a busca de um caminho. Essa busca irá retornar um conjunto de ações, que ao serem executadas vão levar ao estado objetivo. As estratégias comumente utilizadas para este tipo de problema são: busca não informada e busca informada.

2.3.1 Busca Não Informada

O nome busca não informada, também chamado de busca cega, é dado pelo fato de que não se tem nenhuma informação extra, além dos dados definidos no problema. Este método gera sucessores e verifica para cada sucessor gerado se ele é o estado objetivo. As estratégias mais utilizadas são busca em largura e busca em profundidade (RUSSELL; NORVIG, 2009).

Busca em Largura

Busca em largura é uma estratégia em que se expande todos os nós a partir da raiz, depois todos os nós sucessores são expandidos, e assim sucessivamente. Todos os nós de uma mesma profundidade são expandidos antes do próximo nível. Este algoritmo utiliza uma fila, onde o primeiro nó a entrar será o primeiro a ser analisado e expandido.

Se um caminho existir, a busca em largura sempre irá encontrá-lo. Porém, esse caminho nem sempre será o mais curto; isso porque a busca em largura considera que todas as distâncias são iguais. Sendo assim, ele pode ser considerado um algoritmo ótimo se o custo desejado é o número de ações; caso contrário, ele não é considerado ótimo. Ele é completo para ambos os casos, pois sempre encontrará um caminho, se existente (RUSSELL; NORVIG, 2009).

Considerando o tempo e espaço a busca em largura não possui bons resultados para problemas grandes. Considerando b o número de filhos gerados por cada nó, e d a profundidade, tem-se um custo $O(b^d)$ tanto para tempo quanto para espaço. Para um problema com $b = 10$ e $d = 4$ tem-se um tempo aproximado de 11 milissegundos e uso de 10,6 megabytes de memória. Por outro lado, para $b = 10$ e $d = 12$ tem-se um tempo de 13 dias com o uso de 1 petabyte de memória. Desse modo, pode ser visto que tanto para memória quanto para espaço a busca em largura não possui um bom desempenho para problemas grandes (RUSSELL; NORVIG, 2009).

Busca em Profundidade

A busca em profundidade irá expandir o nó mais profundo em uma árvore de busca. Esse método utiliza uma pilha, onde o último nó a entrar será o primeiro a sair.

O nó do topo da pilha é analisado, se ele não é o estado objetivo então ele é expandido e seus filhos são adicionados no topo da pilha. Quando o nó do topo não possuir mais filhos e não for o nó objetivo ele é descartado e expande-se o próximo elemento do topo. Esse procedimento acaba quando o nó objetivo é encontrado ou quando não existe mais elementos na pilha (RUSSELL; NORVIG, 2009).

Existem duas versões da busca em profundidade, uma que evita estados repetidos e outra que não evita. Quando não se evita estados repetidos o algoritmo pode entrar em um loop infinito, o que o torna não completo. No outro caso, com estados finitos e evitando repetições, ele é um algoritmo completo. Para ambos os casos ele não retorna um caminho ótimo. A complexidade temporal é $O(b^m)$ e espacial $O(bm)$, onde b é o número de filhos gerados por cada nó e m é a profundidade máxima da árvore de busca (RUSSELL; NORVIG, 2009).

Uma variação deste algoritmo é a busca em profundidade limitada. Nesta variação existe um limite na profundidade que será avaliada. Quando o limite é atingido, o nó não é expandido, apenas é removido da pilha. Outra variação é a busca em profundidade progressiva, em que o limite da profundidade começa em 1 e aumenta até o caminho ser encontrado ou todos os nós serem avaliados. A busca em profundidade limitada possui complexidade temporal de $O(b^l)$ e espacial de $O(bl)$, onde l é o limite da profundidade. Já a busca em profundidade progressiva possui complexidade temporal de $O(b^d)$ e espacial de $O(bd)$, onde d é a profundidade da solução (RUSSELL; NORVIG, 2009).

2.3.2 Busca Informada

Busca informada é um tipo de busca que considera informações geradas durante a execução do algoritmo. O método mais utilizado é o melhor-primeiro (*best-first*). Ele seleciona o nó a ser expandido com base em uma função de avaliação $f(n)$. Essa função tem como base o custo estimado, e o nó com o menor custo é expandido primeiro. Os algoritmos desse grupo utilizam uma função heurística $h(n)$ que retorna o custo estimado do nó n até o nó objetivo (RUSSELL; NORVIG, 2009).

Busca Gulosa

A busca gulosa expande o nó que está mais perto do nó objetivo. A função f é dada apenas utilizando a heurística: $f(n) = h(n)$. A busca gulosa encontra o caminho com o menor número de passos, porém ela não garante que o caminho com menor custo será encontrado. Além disso, em alguns casos ela nenhum caminho será retornado. Isso ocorre pelo fato de que em alguns casos é necessário uma ação que leva a um estado mais distante, para sim chegar ao estado objetivo. A busca gulosa, porém, considera apenas a função heurística e não desfaz uma ação previamente tomada (RUSSELL; NORVIG, 2009).

A complexidade temporal e espacial é $O(b^n)$, onde b é o número de filhos gerados por cada nó e m a profundidade máxima da busca. Com uma heurística $h(n)$ otimizada a complexidade pode ser reduzida. Porém isso depende de vários fatores como o tipo problema e a qualidade da heurística (RUSSELL; NORVIG, 2009).

A*

A* é o mais conhecido e utilizado algoritmo para busca informada. Sua função $f(n)$ considera não apenas a heurística $h(n)$ mas também o custo do nó inicial até o nó atual, dada pela função $g(n)$. Tem-se então $f(n) = g(n) + h(n)$, que resultará no custo estimado da solução passando pelo nó n (RUSSELL; NORVIG, 2009).

Com uma função heurística boa, tem-se um algoritmo não apenas completo mas também ótimo. Uma função heurística é dita boa se ela é admissível e consistente. Ela é admissível se ela nunca superestima o custo de atingir o objetivo. A função tem de ser sempre otimista. Por exemplo, a distância em linha reta sempre será um caminho otimista para a distância entre duas cidades. A função é considerada consistente se a cada ação tomada, o novo o custo de chegar ao nó objetivo não é maior que o custo de chegar ao nó objetivo antes da ação ser tomada (RUSSELL; NORVIG, 2009).

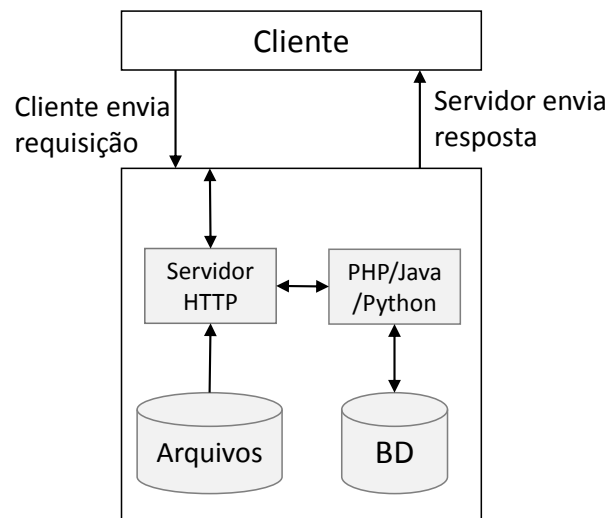
2.4 Programação Web

A programação para web vem em constante ascensão no decorrer dos últimos anos. São mais de 900 milhões de sites ativos, e a cada segundo um novo site é criado (Internet Live Stats, 2015). Esse fato pode ser explicado pela facilidade da criação e compartilhamento de um site.

Um site pode ser algo bem simples, desde uma página criada apenas com [HTML](#) (*HyperText Markup Language*) e conteúdo estático, até páginas com conteúdo dinâmico e funções JavaScript complexas. Um projeto para web pode ser dividido em duas partes: cliente e servidor. Na parte do cliente ocorre a execução de scripts JavaScript e apresentação da página [HTML](#) com a aplicação do [CSS](#) (*Cascading Style Sheets*) nela. Na parte do servidor tem-se principalmente o acesso ao banco de dados para a geração dinâmica das páginas [HTML](#).

Para troca de arquivos entre o cliente e o servidor tem-se a utilização do protocolo [HTTP](#) (*Hypertext Transfer Protocol*). O funcionamento do [HTTP](#) pode ser visto na [Figura 7](#). Nela está ilustrado o cliente, que por meio de um navegador de internet irá fazer uma requisição à um servidor, normalmente na porta padrão 80. O servidor [HTTP](#) vai então responder esta requisição com: a) um arquivo [HTML](#) que será gerado pelo processamento de uma linguagem de programação; b) um arquivo, se a requisição for para um arquivo estático.

Figura 7 – Diagrama de execução do protocolo HTTP.



Adaptado de: [Morris \(2015\)](#)

Se uma requisição a um arquivo `.php` ocorrer, ele não pode ser simplesmente enviado para o cliente. Se isso ocorrer o cliente irá receber um arquivo fonte em vez do conteúdo `HTML`. Isso não ocorre porque o servidor `HTTP` identifica quais arquivos têm de ser processados antes de serem enviados. Os arquivos que não exigem processamento, como imagens e vídeos, são simplesmente enviados para o cliente.

Uma conexão `HTTP` é *stateless*. Isso significa que para cada requisição uma nova conexão tem que ser criada ([MORRIS, 2015](#)). Isso pode ser um fator importante na performance de um sistema web. Se uma página possui muitas requisições para imagens, arquivos JavaScript ou `CSS`, esta página tende a demorar mais para carregar; além de sobrecarregar o servidor com muitas requisições. Desse modo, uma boa prática de programação é sempre que possível reduzir requisições `HTTP`.

Cliente

A programação da parte do cliente é feita normalmente utilizando `HTML`, `CSS` e JavaScript. O `HTML` irá definir a estrutura da página e o conteúdo que será mostrado. O `CSS` descreve como a página tem de ser mostrada no navegador, incluindo fontes, cores e tamanhos. O JavaScript é utilizado para prover dinamicidade na página, provendo meios de modificá-la mesmo após o carregamento.

Para prover uma navegação otimizada, tem-se normalmente a execução de requisições assíncronas ao servidor. Esta técnica de requisições assíncronas é chamada de AJAX. Com o uso destas requisições, pode-se prover conteúdos para o usuário sem a necessidade

dele requisitar toda a página novamente, pois apenas a parte necessária é requisitada. Uma biblioteca JavaScript que facilita a realizações de requisições assíncronas é o jQuery.

jQuery não apenas provê uma forma de realizar requisições AJAX, mas também provê muitas outras funcionalidades como manipulação de eventos, animações e documentos (JQUERY, 2015). Além disso, existem dezenas de *plugins* que proveem funcionalidades extras. Dentre tais funcionalidades, pode-se dar destaque aos *plugins* de interface com o usuário, como calendário, tabelas e barras de progresso.

Servidor

O servidor web é a parte do software que irá receber requisições dos clientes, processá-las e respondê-las. Em um servidor web tem-se normalmente um servidor HTTP rodando na porta 80, e ele irá tratar as requisições dos clientes. O servidor HTTP mais utilizado é o Apache (APACHE, 2015).

Em um site estático não existe a necessidade da utilização de nenhuma linguagem de programação, onde pode-se apenas utilizar a linguagem de marcação HTML. Essa forma porém, é apenas utilizada para páginas simples, em que o conteúdo é estático. Além disso, para realizar modificações na página é necessário modificar diretamente o arquivo, isso pois não existe uma interface que simplifica tais modificações.

Por outro lado, sites normalmente não são estáticos e são desenvolvidos utilizando linguagens de programação como ASP, PHP, Python e Ruby. Tais linguagens analisam parâmetros passados pelo usuário, fazem consultas ao banco de dados e retornam uma página com conteúdo HTML. Independentemente da linguagem utilizada ela irá realizar o processamento necessário e retornar uma página com conteúdo HTML.

2.5 Programação para Dispositivos móveis

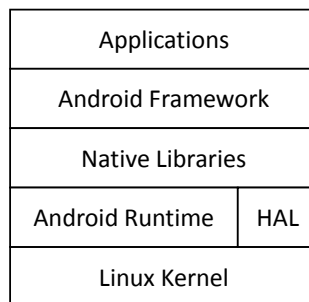
Android

AndroidTM é um sistema operacional da Google que é utilizado por um grande número de celulares atualmente. Seu código fonte é aberto e isso possibilita a customização e otimização das versões do Android para cada dispositivo de uma empresa. Além disso, o Android não funciona apenas em celulares. Atualmente é crescente o uso em relógios e televisores inteligentes (ANDROID, 2015).

Para a implementação de aplicações para Android utiliza-se a linguagem de programação Java. O Android, porém, não possui uma *Java Virtual Machine (JVM)*, mas sim uma máquina virtual chamada Dalvik. Essa máquina virtual é otimizada para a execução de aplicações em dispositivos móveis (LECHETA, 2012).

Para uma aplicação ser executada na máquina virtual Dalvik, ela primeiramente tem de ser escrita em Java, convertida para bytecode (.class) e então convertida para **DEX** (*Dalvik Executable Format*). Depois disso todos os arquivos **DEXs** e os demais recursos, como imagens e sons, são empacotados em um arquivo **APK** (*Android Package File*). Esse arquivo **APK** é a aplicação final que pode ser instalada em dispositivos Android ([LECHETA, 2012](#)).

Figura 8 – Diagrama simplificado da pilha do Android.



Adaptado de: [Android \(2015\)](#)

Na [Figura 8](#) pode ser visualizado a pilha de camadas em que o Android funciona. A camada mais do topo é a camada das aplicações, onde tem-se a inclusão tanto das aplicações nativas como Câmera, Contatos, Discador e Mensagens, quanto as aplicações de terceiros, como Facebook, Twitter e Whatsapp. A segunda camada é a do *framework* do Android, nela tem-se serviços disponíveis para as aplicações, como localização, notificações e dados. A terceira camada é a camada das bibliotecas, nela tem-se o gerenciamento de áudio, OpenGL e SQLite. A quarta camada pode ser dividida em duas; tem-se a parte de execução das aplicações pela máquina virtual Dalvik, e também a parte de gerenciamento do hardware. Na última camada tem-se o kernel do Linux que inclui todos os *drivers* necessários.

iOS

[Apple \(2015b\)](#) define iOS como o mais avançado sistema operacional para dispositivos móveis. Definição que possui relevância, pois o iOS é desenvolvido exclusivamente para produtos da Apple, desse modo ele consegue ser perfeitamente otimizado para trabalhar diretamente com o hardware que ela produz. Em outro lado, Android é um sistema genérico, que pode ser utilizado em qualquer hardware, fato que o torna não tão otimizado em muitos casos.

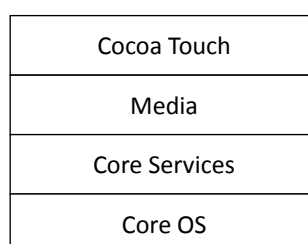
Para desenvolvimento de aplicações para iOS necessita-se de um Mac rodando o OS X 10.9.4 ou superior. Além disso, Apple disponibiliza uma interface de desen-

volvimento chamada Xcode e um pacote de desenvolvimento que possui ferramentas, frameworks e compiladores. Para a programação dos aplicativos, utiliza-se Objective-C, que pode ser definido como um C com modificações influenciadas pelo Smalltalk (APPLE, 2015a). Além do Objective-C, uma nova linguagem, que está ganhando destaque, é o Swift. Comparando-a com Objective-C ela é mais segura, tem melhor controle de memória, é mais rápida e é mais fácil de se escrever e ler códigos (SOLT, 2015).

Para um aplicativo ser aceito na loja da Apple ele necessita passar por uma completa revisão. Aplicativos de teste ou de demonstração não são aceitos. Se um aplicativo não possui nenhum diferencial ou já existe outro aplicativo com a mesma funcionalidade ele não será aceito. Se um aplicativo possui qualquer tipo de problema ele não será aceito (App Store, 2015). Estas revisões, portanto, tem o intuito de manter o alto nível de qualidade dos aplicativos da loja da Apple.

A Figura 9 tem a descrição da arquitetura do iOS em camadas. Na camada mais ao topo tem-se o Cocoa Touch. Ele é um *framework* para a construção de aplicativos que define como a aparência da tela e a sua interação será dada. A camada de Media contém os recursos de áudio e vídeo que são utilizados para implementar o aplicativo. Na camada de Core Services tem-se o acesso a serviços essenciais para a aplicação, como acesso a localização, rede e banco de dados. Na última camada, Core OS, tem-se a implementação de baixo nível das outras camadas. Nessa camada também tem-se a implementação de funcionalidades de segurança (Apple Inc., 2014).

Figura 9 – Diagrama de camadas do iOS.



Adaptado de: Apple Inc. (2014, p. 9)

Programação Híbrida

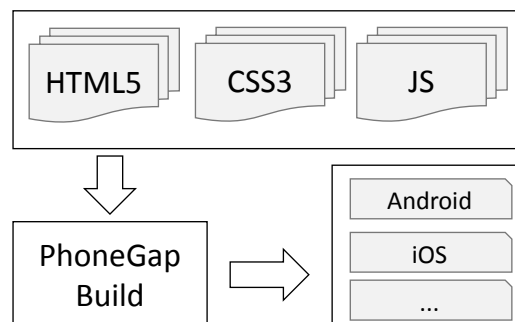
Um novo método de desenvolvimento para dispositivos móveis que está em destaque é o modelo híbrido. Neste modelo não se desenvolve um código nativo para cada plataforma. Por exemplo, não existe a necessidade de escrever um código Java para Android e outro Objective-C/Swift para iOS. Ao invés disso, escreve-se um código em uma única linguagem e ele será transformado em um aplicativo para cada uma das plataformas.

O modelo híbrido é comumente utilizado porque ele diminui consideravelmente o custo de desenvolvimento. Escreve-se um único código, normalmente utilizando [HTML](#) e JavaScript. Esse código irá gerar um arquivo [HTML](#) que pode ser acessado normalmente em um navegador de internet. Nativamente tais linguagens não possuem acesso a recursos dos dispositivos móveis como sensores, câmeras e GPS. Para a utilização de tais recursos existe a necessidade da utilização de *frameworks* que vão prover o acesso por meio de códigos JavaScript. Independentemente da plataforma utilizada o código JavaScript será o mesmo e o *framework* irá converter e executar o código para a plataforma em que ele está sendo executado.

Os *frameworks*, porém, não proveem acesso a todos os recursos disponíveis no sistema móvel. Além disso, os recursos providos podem não funcionar como os de uma aplicação nativa. Desse modo é necessário analisar as vantagens e desvantagens da utilização do modelo híbrido. Por exemplo, em aplicações que necessitam de um maior controle do hardware, alto desempenho ou acesso a diversos recursos do sistema, a programação nativa irá proporcionar um melhor resultado. Por outro lado, em aplicações que não exigem alto processamento ou que proveem um serviço que também é provido em um meio web, o modelo híbrido provavelmente será uma boa solução.

O *framework* mais popular neste contexto é o Cordova. Ele provê o acesso a diversas funcionalidades nativas de diversos sistemas operacionais móveis como Android, iOS e Windows Phone. Dentre as funcionalidades providas está o acesso a câmera, acelerômetro e armazenamento. Estas funcionalidades podem ser acessadas por meio do navegador em uma página web ([CORDOVA, 2015](#)). O Cordova pode gerar aplicativos para a instalação em dispositivos móveis. Porém, é comum a utilização junto com outros programas, como Adobe PhoneGap e Intel XDK, que provem recursos extras. Um exemplo do processo de criação de um aplicativo utilizando códigos [HTML](#) e PhoneGap está ilustrado na [Figura 10](#).

Figura 10 – Procedimento da criação de uma aplicação móvel híbrida.



Adaptado de: [Adobe \(2015\)](#)

3 Projetos Relacionados

Dentre os sistemas que proveem um serviço semelhante ao proposto neste trabalho o Decolar pode ser visto como um dos mais importantes no Brasil. Decolar.com é um site brasileiro para busca e compra de passagens aéreas (DECOLAR, 2015a). Ele é uma filial da empresa argentina Despegar.com, que disponibiliza este serviço para países da América Latina e EUA. Apesar do foco desta empresa ser passagens aéreas, ela possui opções para busca de hotéis e alugueis de carros. Decolar, como uma empresa capitalista, possui seu modelo voltado ao lucro. Tal fato faz com que informações que usuários geram durante o uso do sistema, como preferências, buscas recentes e dados pessoais, sejam disponibilizados para seus parceiros, como empresas de marketing e publicidade online (DECOLAR, 2015b). O sucesso desta empresa se deve ao compromisso de entregar o menor preço em passagens aéreas para seus clientes. Onde, em caso de outra empresa possuir um preço menor, Decolar promete cobrir a diferença.

Neste mesmo contexto de passagens aéreas, Hipmunk pode ser considerado um dos destaques internacionais. Hipmunk é um site com base nos Estados Unidos da América, e tem como seu co-fundador o Steve Huffman, criador da popular rede social Reddit (HIPMUNK, 2015). Diferentemente do Decolar, Hipmunk não vende as passagens diretamente. Ele busca as melhores passagens e lista onde o usuário pode as encontrar. Em 2012 Hipmunk foi intitulado o melhor site de viagens pela Forbes (FORBES, 2012). Um dos motivos por tal título foi o fato de que Hipmunk não anuncia publicidade para seus usuários, visando que eles usem o site pelo menor tempo possível, com buscas rápidas e inteligentes. Como Hipmunk não vende as passagens ou publicidades, sua forma de geração de renda é por meio de comissões que são pagas pelas empresas que vendem as passagens. Tal comissão não deve passar de alguns dólares, mas com o grande número de usuários que Hipmunk possui, cerca de 2 milhões visitantes únicos em fevereiro de 2015 (COMPETE, 2015), a torna uma empresa multimilionária.

Buscando ferramentas que possuem uma semelhança maior com este trabalho, no caso de viagens de ônibus, um sistema que possui grande qualidade, apesar de não ser tão popular no Brasil, é o Rome2rio (ROME2RIO, 2015). Rome2rio é uma empresa com sede na Austrália que tem como foco integrar e organizar dados de transporte de todo o mundo. Tais dados incluem rotas de ônibus, aviões e trens. Quando uma rota por um meio de transporte público não está disponível, uma rota alternativa via carro é mostrada. As pesquisas levam em conta preço, distância e tempo. O sistema não possui uma forma de compra integrada, porém em alguns casos ele disponibiliza um link externo para a aquisição da passagem.

Tratando-se de serviços brasileiros para rotas de ônibus, podem ser citados o Busca Ônibus (Busca Ônibus, 2015), Quero Passagem (Quero Passagem, 2015) e Rodoviária Online (RODOVIARIAONLINE, 2015). Todos eles possuem rotas de ônibus de todo o Brasil. Quero Passagem e Rodoviária Online possuem um sistema de vendas integrado, onde o passageiro tem a possibilidade escolher a sua poltrona e fazer o pagamento online. Por outro lado, Busca Ônibus apenas lista os ônibus disponíveis. Nenhum deles possibilita a busca de rotas com conexões.

Análise dos Serviços

Como visto até agora, existem diversas ferramentas que proveem um sistema semelhante ao proposto neste trabalho. Sabendo disso, tem-se em mente que este trabalho não busca fazer algo totalmente inovador, mas sim melhorar as ferramentas existentes e fazer um serviço completo e mais eficiente. Para identificar onde e o que melhorar, analisou-se os sistemas encontrados, com o foco da análise para o objetivo de integrar dados de rotas de ônibus intermunicipais do estado do Rio Grande do Sul, possibilitando consultas rápidas e eficientes.

Os critérios para comparação que foram utilizado são: a) Se o serviço possui informações sobre rotas de ônibus do estado do Rio Grande do Sul. b) Se o serviço não apenas une os dados, mas possibilita consultas complexas, com conexões de rotas; por exemplo, para uma consulta de origem Alegrete e destino Jari não existe uma única rota, neste caso o sistema deverá mostrar a combinação de duas ou mais rotas. c) Se o serviço possui uma forma de avaliação das rotas para facilitar a detecção de rotas com atrasos frequentes. d) Se o serviço possui um sistema de venda de passagens integrado; e) Se o serviço possui um aplicativo para dispositivos móveis.

Tabela 2 – Comparação entre serviços.

Serviço	Rotas Gaúchas	Conexões	Feedback	Venda	Mobile
Decolar	–	Sim	Sim	Sim	Sim
Hipmunk	–	Sim	Sim	Não	Sim
rome2rio	Boa parte	Sim	Não	Não	Não
Busca Ônibus	Incompleto	Não	Não	Não	Não
Quero Passagem	Não	Não	Não	Sim	Não
Rodoviária Online	Não	Não	Não	Sim	Não
Este trabalho	Busca-se cobrir 100%	Sim	Sim	Não	Sim

Com a análise da Tabela 2 pode ser percebido que, uma vez atingido os objetivos deste trabalho, este serviço tem potencial para ajudar a população do Rio Grande do Sul a encontrar as rotas de ônibus desejadas. Dentre os serviços analisados podemos dar destaque aos dois que proveem dados para passagens aéreas, por possuírem a maior parte das funcionalidades desejadas. Para as passagens de ônibus o destaque é o rome2rio que

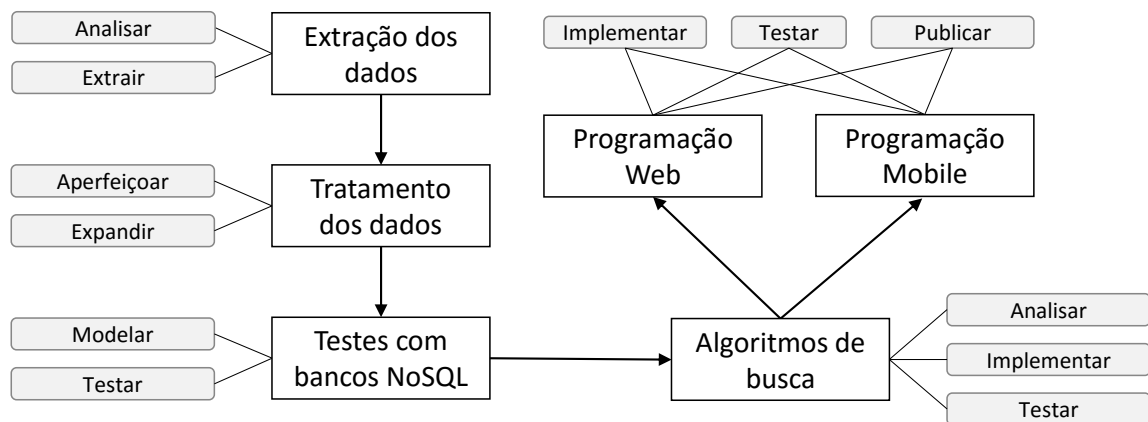
possui uma grande quantidade de rotas por todo o mundo, mas não possibilita a consulta em aplicativos de dispositivos móveis, ou envio de avaliações sobre as rotas. Quero Passagem e Rodoviária Online funcionam basicamente como uma rodoviária virtual, onde vendem passagens para grandes centros como Rio de Janeiro e São Paulo.

Neste trabalho será buscado cobrir o maior número de rotas possíveis para o estado do Rio Grande do Sul. Uma parte importante do trabalho é o fato de possibilitar conexões, ou seja, se uma cidade não possui uma rota direta até outra, duas ou mais rotas poderão ser unidas com diferentes critérios como preço, duração e distância. Para ter uma boa qualidade nos dados é importante que o usuário dê sua avaliação para cada rota; se um rota tiver constantes atrasos o usuário poderá ser alertado para evitá-la.

4 Metodologia

Este trabalho planejou a criação de um sistema para consulta de rotas de ônibus intermunicipais. Para atingir tal objetivo, modelou-se um plano de ação, com os passos que deveriam ser executados. Este plano de ação pode ser visualizado na [Figura 11](#).

Figura 11 – Conjunto de passos a serem executados durante este projeto.



A primeira etapa do processo foi a extração dos dados. Nela analisou-se quais dados deveriam ser coletados e como esta coleta seria realizada. Após esta análise, os dados foram salvos em um banco de dados local. Estes dados não sofreram nenhum tipo de tratamento e podem ser vistos como um conjunto de dados bruto. Uma das principais técnicas utilizadas nesta etapa foi o *web scraping*.

A próxima etapa, tratamento dos dados, moldou os dados brutos em um banco de dados relacional. Os dados passaram a possuir integridades referenciais, onde, por exemplo, uma rota é relacionada com uma cidade de origem e uma cidade de destino. Além deste tratamento, os dados passaram por uma expansão, onde dados extras foram obtidos a partir dos dados já existentes.

Na etapa de testes com bancos NoSQL buscou-se ter uma visão geral das capacidades de utilização de bancos de dados NoSQL orientados a grafos. Para isso, converteu-se a base de dados relacional para uma base de dados NoSQL e realizou-se testes para descobrir a capacidade de execução de algoritmos de busca diretamente nas consultas.

Para a implementação dos algoritmos de busca, analisou-se o problema, a fim de descobrir os melhores algoritmos a serem utilizados. Após, implementou-se os algoritmos e os testou. Com esta fase completa, obteve-se um algoritmo, que utilizando a base de

dados, seria integrado com as interfaces *web* e *mobile* geradas nas etapas seguintes.

Para a programação web e mobile, seguiu-se o mesmo modelo de passos, onde, primeiramente, implementou-se as interfaces e suas interações. Após, testou-se sua utilização a fim de encontrar qualquer tipo de problemas. Por fim, publicou-se os sistemas para uso da comunidade em geral.

5 Desenvolvimento

5.1 Extração de Dados

A extração dos dados pode ser vista como o pilar principal do desenvolvimento deste projeto. Nessa fase inicial planejou-se a criação de um banco de dados relacional, com informações sobre cidades e rotas de ônibus intermunicipais. Os dados das rotas de ônibus foram coletados de um principal site de rodoviárias do Rio Grande do Sul, controlado pela RodoSoft e do site da rodoviária de Porto Alegre Veppo. Além destes dados, informações sobre cidades foram obtidas do site do IBGE.

Em uma abordagem inicial a extração seria realizada em um único processamento. Esse processamento englobaria a obtenção do conteúdo [HTML](#) das páginas, a extração dos dados e a validação dos mesmos junto ao banco de dados. Essa abordagem, porém, tornou-se inviável pelas seguintes razões: o tempo de execução era alto, onde identificou-se que o tempo de transferência dos dados pela rede era o principal gargalo, elevando assim o tempo total do processamento; se, por algum motivo, um dado importante deixasse de ser obtido, todo o processamento teria de ser executado novamente (o que inclui mais transferências); testes nas fases de obtenção dos dados e validação não poderiam ser realizados, pelo mesmo motivo do excesso de comunicação pela rede. Dessa forma, optou-se pela divisão em três módulos de processamento: download, extração dos dados e tratamento dos dados. Estes módulos foram desenvolvidos em Java com a utilização de um *parser* [HTML](#) chamado JSoup.

O módulo de download possui a finalidade de realizar a transferência de todos os dados necessários nos módulos seguintes. Assim, identificou-se as páginas necessárias e obteve-se o conteúdo [HTML](#) de cada uma delas. Os dados foram salvos em arquivos locais, onde mais de 2GB de conteúdo foi obtido. Esse tamanho dos dados é consideravelmente alto; fazendo esse módulo necessário para evitar inúmeras transferências desnecessárias. Considerando o lado do servidor, esse número representa menos de 0,1% do limite de tráfego mensal em um servidor comum. Sendo assim, é um valor aceitável para as transferências, desde que não sejam frequentes. Dessa forma, acredita-se que atualizações na base de dados possam ser realizadas uma vez por mês.

O módulo de extração dos dados trabalha sobre os arquivos obtidos pelo módulo de download. Esse módulo é dividido em duas partes: cidades e rotas. Para as cidades, ele percorre as páginas do IBGE, que é organizada por estados, e obtém dados como nome, estado e coordenadas das cidades. Estes dados então são salvos no banco de dados, gerando um identificador sequencial para cada cidade. Para as rotas de ônibus, muda a

forma que as páginas são percorridas e quais dados são obtidos. As páginas iniciais das rodoviárias possuem um elemento `HTML` chamado *select*. Esse elemento é uma lista de opções, onde cada opção possui um valor associado. Nele estão presentes os nomes das cidades e os identificadores associados a cada cidade. Para cada cidade presente na lista, uma nova página é gerada, onde o identificador é informado como parâmetro. Essa nova página contém os dados das rotas de ônibus, os quais devem ser coletadas. Os dados retirados são, entre outros, a cidade de origem e destino, horários de partida e chegada, preço, distância e empresa. Estes dados não passam por nenhum tipo de tratamento, se pensarmos na estrutura do banco de dados, eles podem ser vistos apenas como uma sequência de caracteres.

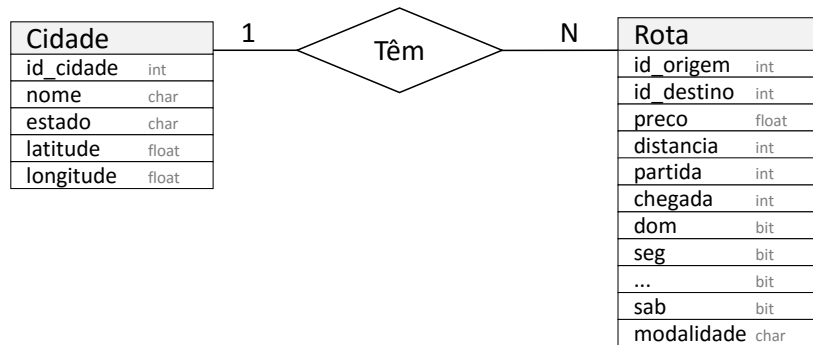
Para exemplificar esse processo, vamos utilizar a rodoviária de Alegrete como exemplo. A página inicial da rodoviária de Alegrete é acessada utilizando o seguinte arquivo `InCdOrigem=6` (que se refere aos parâmetros passados na URL no site original). Este arquivo possui um elemento *select* com todas as cidades com rotas partindo de Alegrete. Para Santa Maria, por exemplo, a opção é a seguinte `<option value='1'>Santa Maria-RS</option>`. Assim, para obter todas rotas de Alegrete para Santa Maria, o arquivo `codPro=6&tipIdx=1` deve ser acessado. Esse arquivo contém uma tabela com as informações de cada rota de Alegrete para Santa Maria. Essas informações são coletadas e salvas no banco de dados. Todo esse processo de obtenção dos valores, como identificadores nos elementos *select* ou as informações nas tabelas, são feitos com a utilização do XPath.

Um importante fator a ser ressaltado, é que todos sites controlados pela RodoSoft possuem uma mesma estrutura. Os dados, porém, são independentes para cada rodoviária. Dessa forma, a lista de identificadores presentes nos *selects* são sempre diferentes, assim, nenhum padrão pode ser obtido. Porém, o algoritmo funciona independente dos dados, obtendo os identificadores em cada processamento.

Para realizar operações sobre os dados, o campo preço, por exemplo, não deve ser visto como uma *string*, mas sim como um decimal. Além disso, as informações das rotas devem se conectar com as informações das cidades, ou seja, em uma rota, a origem deve se conectar a uma cidade e o destino deve se conectar a outra. Assim, o módulo de tratamento de dados se faz necessário, realizando tais conversões. O banco de dados do modelo anterior é modificado, gerando um modelo específico para as necessidades do sistema. O modelo entidade relacionamento pode ser visualizado na [Figura 12](#).

Os principais problemas enfrentados no tratamento dos dados foram as inconsistências nos nomes das cidades. São diversos casos em que nomes das cidades estão escritos de forma errada, de forma reduzida ou em alguns casos os nomes das cidades são utilizados referindo-se a localidades. Cada rota possui o nome da cidade de origem e destino e para esse nome ser associado a uma cidade foram utilizadas duas técnicas: comparação

Figura 12 – Representação simplificada do modelo E/R do banco de dados.

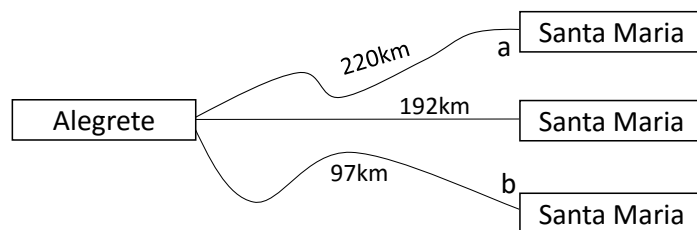


dos nomes e distância entre cidades.

Primeiramente associou-se as cidades de origem a um identificador. Essa tarefa foi simples, pois todas utilizavam o nome idêntico às informações já coletadas e todas as informações eram sobre cidades e não localidades. Assim, o campo `id_origem` da tabela `rota` foi associado a um identificador da tabela `cidade`.

Como em cidades de destino existem muitas localidades incluídas, apenas a utilização do nome da cidade seria inadequado. Isso porque em alguns casos um nome de cidade também é utilizado para referir a uma localidade. Para descobrir se o nome refere-se uma cidade ou a uma localidade utilizou-se a distancia em linha reta e a distância obtida na rota.

Figura 13 – Exemplo do uso de distância para o tratamento dos nomes.

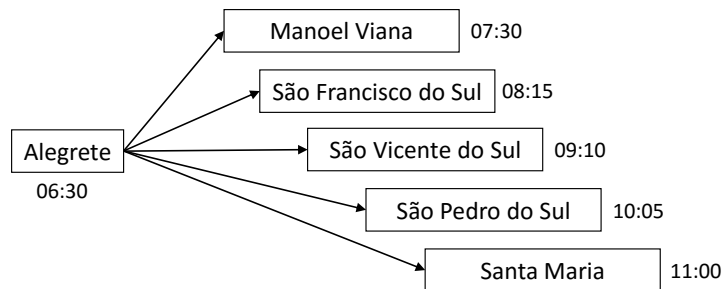


Dada a [Figura 13](#), onde tem-se duas rotas entre Alegrete e Santa Maria. A rota *a* tem distância de 220km e a rota *b* 97km. A distância em linha reta entre Alegrete e Santa Maria é de 191km. A distância pela rota, que é feita por uma estrada, deve ser maior que a distância em linha reta; nunca menor. Dessa forma, o nome Santa Maria na rota *b* não pode se referir a cidade, mas sim a uma localidade. Para o limite superior não encontrou-se nenhum padrão, pois a distância pela rota pode ser até 150% mais longa que a distância em linha reta. Dessa forma, apenas eliminou-se aquelas que possuíam uma distância inferior.

Atualmente, este é um problema enfrentado pelos usuários, pois se existe uma

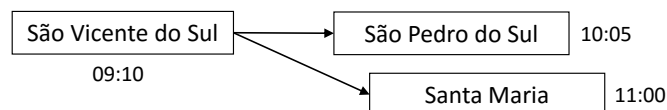
rota para uma localidade com o mesmo nome de uma cidade, isso irá causar confusão. Por exemplo, na rodoviária de Alegrete existe uma rota até Toropi. Com a utilização das técnicas acima, ela foi considerada uma rota para a cidade de Toropi. Porém, por conhecimento prévio, sabe-se que essa rota não vai até lá. Este nome Toropi refere-se a uma localidade no interior de uma cidade vizinha, o que faz com que a distância seja compatível. Problemas como esse só puderam ser corrigidos se identificados manualmente. Nenhuma forma automatizada foi identificada para tal solução.

Figura 14 – Todos destinos de uma rota saindo de Alegrete.



Como não existem sites de todas as rodoviárias do estado muitas cidades ficam sem nenhuma rota partindo delas. Para obter outras rotas a partir das rotas conhecidas foi implementado uma técnica para a identificação de rotas existentes. Dado um exemplo na [Figura 14](#), pode ser visualizado que existe uma rota que parte as 06:30 de Alegrete com diversos destinos. A base de dados não possui nenhuma informação sobre rotas partindo de São Vicente do Sul, assim pode ser assumido que existe uma rota de São Vicente do Sul partindo as 09:10 com destino São Pedro do Sul e Santa Maria, como visto na [Figura 15](#). Tais rotas foram marcadas no banco de dados como suposições com o propósito de indicar ao usuário que elas foram geradas pelo sistema, e os horários podem não ser exatos. Uma versão simplificada do algoritmo utilizado na realização deste processo pode ser visualizado no [Algoritmo 1](#).

Figura 15 – Rotas extras obtidas a partir da rota de Alegrete.



Com a extração inicial dos dados obteve-se mais de 100 mil rotas. Deste total, cerca de 60% foi detectado como rotas com destino para localidades. Assim, após o tratamento dos dados manteve-se cerca de 45 mil rotas. Além disso, com a aplicação da técnica de obtenção de rotas extras, obteve-se cerca de 4 mil rotas de cidades que anteriormente não possuíam nenhuma. No total, estas rotas partem de 415 cidades diferentes, levando a 465 destinos. Isso gera um total de cerca de 83% das cidades gaúchas cobertas.


```
rotas_iniciais = todas rotas onde destino não possui nenhuma rota partindo de lá;  
for rota_inicial in rotas_iniciais do  
  rotas_encontradas = todas rotas;  
  remove os elementos de rotas_encontradas que não:  
    - partem da mesma origem que rota_inicial;  
    - possuem o mesmo horário de partida que rota_inicial;  
    - possuem o mesmo número do serviço que rota_inicial;  
    - possuem horário de chegada superior que rota_inicial;  
  for rota_encontrada in rotas_encontradas do  
    nova_rota = rota_encontrada;  
    nova_rota.origem = rota_inicial.origem;  
    adiciona(nova_rota);  
  end  
end
```

Algoritmo 1: Algoritmo simplificado para a obtenção de novas rotas.

5.2 Testes com Bancos de dados NoSQL

Durante o momento de estudo de tecnologias que poderiam se adequar na implementação desse projeto, encontrou-se os bancos de dados NoSQL orientados a grafos. Pelo fato de os dados deste trabalho serem perfeitamente representados em formas de grafos, decidiu-se então testar um destes bancos para ter-se uma ideia de suas capacidades.

Utilizou-se o [SGBD Neo4j](#) para os testes. Ele disponibiliza um software para a criação de um servidor em um diretório local. O banco de dados pode ser acessado por uma interface web, onde por meio dela consultas podem ser feitas. Os comandos, em Cypher, quando executados, retornam sempre que possível uma representação gráfica do grafo com seus nodos e relacionamentos.

Os dados foram exportados do banco de dados do modelo relacional em formato [CSV](#) (*Comma-separated values*). No Neo4j os dados dos estados e cidades foram importados e uma relação entre eles foi criada. Então, as rotas foram importadas, criando-se uma relação chamada *VaiPara* da cidade de origem até a cidade de destino, onde os atributos desta relação eram valores como preço, horário de partida e horário de chegada.

Neo4j, por ser um banco de dados orientado a grafos, possui algumas funções especiais como a `allShortestPaths`. Ela retorna o caminho mais curto, em relação ao número de ações necessárias, entre um nodo e outro. Por exemplo, se o comando `allShortestPaths((origem)-[r:VaiPara*]->(destino))` for executado, ele irá retornar todos os caminhos mais curtos entre o nodo `origem` e o nodo `destino`. O exemplo completo para busca de um caminho mais curto entre um nodo e outro pode ser visualizado abaixo. Neste exemplo, busca-se todos os caminhos mais curtos entre *Alegrete* e *Erechim*.

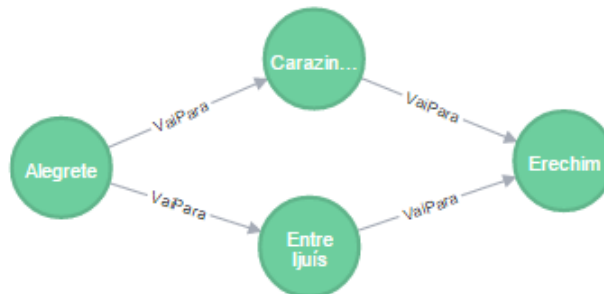
```

MATCH (origem: Cidade {nome_cidade:"Alegrete"}),
      (destino: Cidade {nome_cidade:"Erechim"}),
paths = allShortestPaths((origem)-[r:VaiPara*]->(destino))
RETURN paths

```

Com a execução deste comando 14 possíveis rotas são retornadas. Para questão de visualização limitou-se o retorno para 2 rotas, as quais podem ser visualizadas na [Figura 16](#), onde o tempo de consulta foi de 1386 ms.

Figura 16 – Retorno da consulta utilizando o comando `allShortestPaths`.



Estas rotas ainda não estão otimizadas, pois o único critério de seleção foi o menor número de ações. Conseguiu-se adicionar campos como distância e preço para ordenar os resultados. Dessa forma, obtêm-se os resultados ordenados de acordo com o campo escolhido. A consulta Cypher utilizada, onde ordenou-se os resultados pelo preço de cada rota, pode ser visualizada abaixo. O retorno da consulta pode ser visualizada na [Figura 17](#), onde o tempo de consulta foi de 1477 ms.

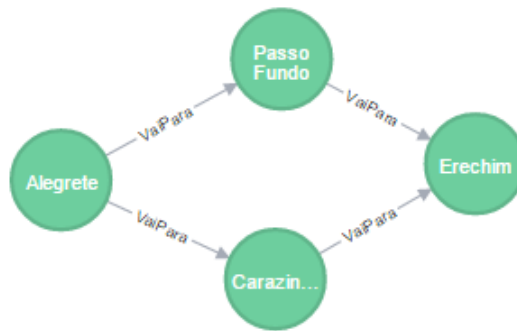
```

MATCH (origem: Cidade {nome_cidade:"Alegrete"}),
      (destino: Cidade {nome_cidade:"Erechim"}),
paths = allShortestPaths((origem)-[r:VaiPara*]->(destino))
WITH REDUCE(p = 0, rel in rels(paths) | p + rel.preco) AS peso, paths
RETURN paths, peso
ORDER BY peso ASC

```

Os resultados retornados por estas consultas foram promissores. O problema neste caso foi o fato de que a escolha de uma rota depende da rota anterior. Por exemplo, uma rota deve ter tempo de partida superior ao tempo de chegada da rota anterior. Infelizmente, não encontrou-se nenhum meio de garantir tal fator utilizando o Cypher. Assim, neste momento decidiu-se pela não utilização deste modelo de banco de dados.

Figura 17 – Retorno da consulta com rotas ordenadas pelo preço.



5.3 Algoritmos de Busca

Durante a fase de análise dos algoritmos de busca identificou-se vários potenciais algoritmos. Um deles se encaixa perfeitamente na solução deste problema por sempre retornar a melhor rota baseado em pesos definidos. Este algoritmo é o A*. Os demais algoritmos, ou não retornam o melhor caminho em alguns casos ou este melhor caminho leva em considerações o menor número de ações, o que não é interessante para este problema. Assim, a escolha pelo A* para a utilização neste sistema foi dada por suas características.

Decidiu-se implementar o algoritmo primeiramente em PHP, por ser a linguagem utilizada nas fases iniciais de testes. Logo nos testes iniciais identificou-se o alto tempo de processamento para atividades básicas, como a iteração entre a lista de sucessores e a criação de objetos. Dessa forma, decidiu-se por não continuar com a implementação em PHP e seguir para as demais linguagens.

As linguagens escolhidas para a implementação foram Python e Java. Esta escolha deu-se pelo fato de ambas serem altamente utilizadas. Uma versão simplificada do algoritmo A* implementado pode ser visualizado no [Algoritmo 2](#). Nele foi ocultado a forma em que os elementos são ordenados na fila de prioridade e a função de busca de sucessores.

Os potenciais critérios para utilizar na ordenação dos elementos na fila de prioridade são: distância, preço, número de troca de ônibus, tempo de viagem e tempo de espera. Todos os valores foram transformados em uma escala de 0 até 1 e pesos foram atribuídos a cada um deles. Decidiu-se pela execução de três configurações de pesos. Cada uma delas favorece um tipo de consumidor. Na primeira, busca-se uma rota com a menor duração, incluindo menos troca de ônibus e o preço não é tão importante. Na segunda, considera-se o preço como o fator determinante. Por fim, na última configuração os pesos são igualmente distribuídos. Pensou-se em disponibilizar tal funcionalidade de atribuição

de pesos ao usuário, porém, em diferentes configurações, como quando todos valores são zerados e apenas um é favorecido, a execução tende a demorar muito mais que em uma configuração equilibrada, retornando resultados ruins.

A função `busca_sucessores`, vista no [Algoritmo 2](#), é realizada por meio da execução de uma consulta [SQL](#). Esta consulta filtra as rotas que: possuem o mesmo `id_origem` da cidade de origem, partem depois do horário estipulado (ou horário de chegada do ônibus anterior), partem no mesmo dia (ou próximo dia, se a chegada é próxima a meia-noite) e a rota leva a uma cidade mais próxima do destino.

Entrada: origem, destino, dia_semana, horario

Saída: melhor caminho entre origem e destino

inicializa `fila_prioridade`, `lista_pais`, `lista_fechados`, `melhor_atual`;

`sucessores = busca_sucessores(origem, destino);`

for *sucessor* **in** *sucessores* **do**

if *sucessor* < *melhor_atual* **then**

 | `melhor_atual = sucessor`

end

`fila_prioridade.add(sucessor);`

end

while *fila_prioridade não está vazia* **do**

`elemento = fila_prioridade.pop();`

if *elemento == destino* **then**

 | melhor rota encontrada;

 | `retorna obtem_caminho(lista_pais, elemento);`

else

`sucessores = busca_sucessores(elemento, destino);`

for *sucessor* **in** *sucessores* **do**

if *sucessor not in lista_fechados* **then**

 | **if** *sucessor* < *melhor_atual* **then**

 | `melhor_atual = sucessor`

 | **end**

 | `fila_prioridade.add(sucessor);`

 | `lista_pais[sucessor] = elemento;`

 | **end**

 | **end**

 | `lista_fechado.add(elemento);`

end

end

nenhum caminho encontrado;

`retorna obtem_caminho(lista_pais, melhor_atual);`

Algoritmo 2: Implementação simplificada do A*.

Um fator importante é que em alguns casos o sistema não possui os dados de rotas sobre algumas cidades. Estes dados podem estar faltando, por não existir informações na internet, ou realmente não existir nenhuma rota. Nestes casos, o algoritmo busca cidades mais próximas para realizar a busca. Quando a cidade de origem não possui rotas partindo

dela, a função `busca_sucessores` que é executada inicialmente irá utilizar como `origem` as cidades com as menores distâncias em linha reta até a origem escolhida pelo usuário. A rota entre a origem e a cidade escolhida é destacada como uma possível rota de carro ou táxi. O mesmo ocorre para quando não existe uma rota até a cidade de destino. Porém, neste caso utiliza-se a variável `melhor_atual`, que contém um caminho com o menor peso até a cidade de destino.

Com execuções iniciais, o algoritmo estava encontrando boas rotas, porém com um tempo elevado em muitos casos. Decidiu-se então analisar as duas partes principais do sistema: o banco de dados e o algoritmo de busca. Primeiramente, utilizou-se uma ferramenta para análise da execução de programas em Java, para identificar partes em que o tempo de execução era elevado. Após, buscou-se melhorar o desempenho do banco de dados, utilizando índices de busca.

Como decidiu-se pela execução de três configurações de pesos diferentes, planejou-se em executá-las em três processos separados. Isso porém, estava apresentado um problema, que a utilização de uma ferramenta para Java chamada JProfiler permitiu a descoberta. Esta ferramenta permite analisar a execução de um código, mostrando os recursos utilizados e os tempos de execução em cada área do programa (ej-technologies, 2015).

Durante as análises da execução com o JProfiler, identificou-se uma área do programa que possuía um tempo de computação elevado. Em um teste, a busca de rota com uma configuração de busca levou um total de 878ms. Desse tempo, 78.9% (692ms) foi apenas da realização inicial da conexão do programa com o banco de dados. O tempo restante inclui o algoritmo em si, com todas as consultas no banco de dados. Como executava-se três configurações em instâncias diferentes, este tempo de conexão era multiplicado por três. Este alto tempo de conexão pode ser explicado pelo fato dos bancos de dados criarem uma *thread* (MySQL) ou um processo (PostgreSQL) para cada conexão realizada pelo usuário, o que pode ser lento.

Para resolver este problema, realizou-se uma simples modificação, onde com apenas uma conexão executa-se todas as configurações de busca, conseguiu-se reduzir o tempo em mais de 1s. Esta técnica foi replicada ao código Python, onde o desempenho também foi melhorado. Para tentar reduzir ainda mais o tempo, comparou-se a execução em diferentes bancos de dados, como MySQL e PostgreSQL, porém ambos tiveram o mesmo comportamento.

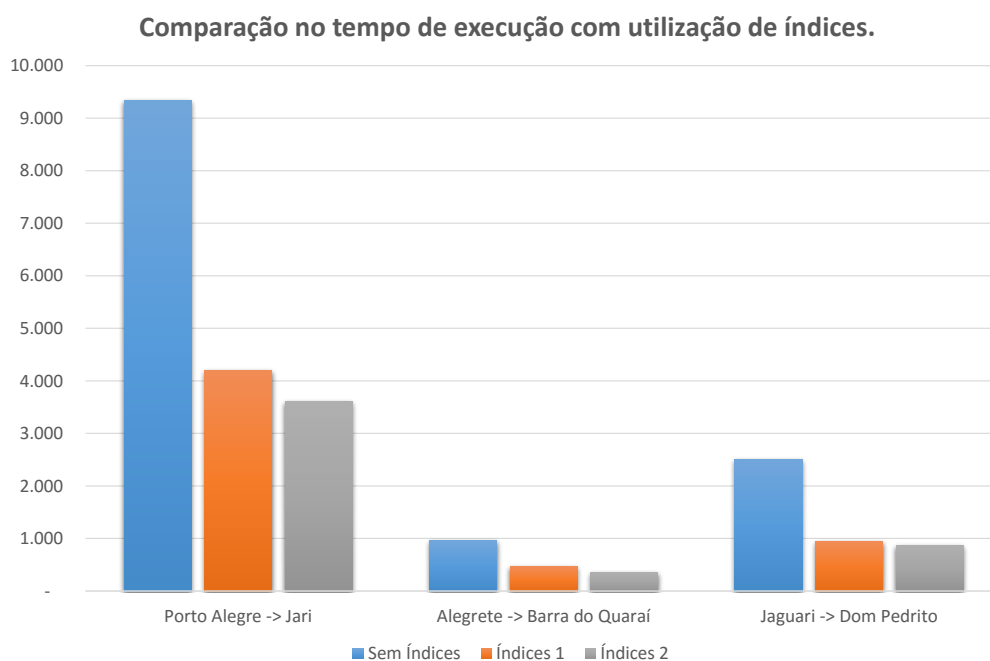
Além deste problema maior, outro problema de menor impacto foi identificado pelo JProfiler. Existia algumas chamadas desnecessárias ao banco de dados para calcular a distância entre duas cidades. Os dados necessários para tal cálculo, como latitude e longitude, já estavam carregados, assim o cálculo poderia ser realizado na memória. No mesmo exemplo anterior, o tempo total das consultas para obtenção das distâncias pelo banco de dados era 23ms, tempo que foi reduzido para menos de 1ms com a execução em

memória. Tempo que, não possui um impacto muito grande, mas ajuda a reduzir a carga de trabalho do banco de dados.

A utilização de índices no banco de dados apresentou um impacto altamente positivo, como pode ser visto no gráfico da [Figura 18](#). Inicialmente, nenhum índice estava sendo utilizado, assim, ainda estava-se com um tempo elevado na execução de diversos testes de busca. Ao adicionar os índices na tabela de rotas conseguiu-se reduzir consideravelmente o tempo de execução.

Neste exemplo da [Figura 18](#), comparou-se o tempo de execução em três casos: sem índices, onde nenhum índice estava sendo utilizado; índices 1, onde inclui-se índices nas colunas dos dias da semana (dom...sab) e partida; índices 2, onde inclui-se todos os índices do anterior mais índices sobre as colunas id_origem e id_destino. Pode-se constatar que a inclusão dos índices do conjunto 1 teve um alto impacto, reduzindo mais que a metade o tempo de execução. Este alto impacto pode ser explicado pela constante utilização de seleções que filtram os dados por dias da semana e horários de partida. A inclusão dos índices do conjunto 2 teve um impacto de menor significância, porém, ajudou a reduzir o tempo total de busca. No geral, estes resultados mostram a importância da utilização de índices para a busca eficiente em banco de dados.

Figura 18 – Impacto na utilização de índices em banco de dados no tempo de execução de três diferentes consultas.



Além do impacto sobre os índices, a análise da [Figura 18](#) mostra a alta diferença

no tempo de execução entre as três buscas. Um dos motivos encontrado é o fato que a rota Porto Alegre -> Jari encontra uma quantidade muito superior de sucessores que a rota Alegrete -> Barra do Quaraí e Jaguari -> Dom Pedrito. Uma possível solução é limitar o número de sucessores, porém não sabe-se qual o impacto no resultado final das buscas. Atualmente, limitou-se a busca de sucessores em 100, assim, as 100 melhores rotas de cada cidade são consideradas. Para o caso de Porto Alegre, onde o número de rotas partindo de lá é elevado, esta limitação ajudou a reduzir o tempo total e, nos testes realizados, não apresentou impacto na qualidade dos resultados.

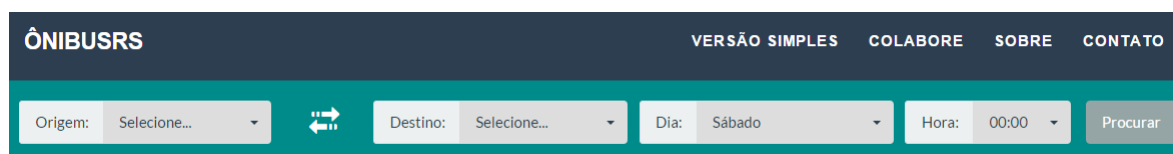
Após os testes realizados, notou-se que, independente das linguagens e ferramentas utilizadas, o tempo de execução e qualidade dos resultados eram os mesmos. A principal diferença é os métodos que foram aplicados para a solução do problema. Assim, decidiu-se pela utilização do banco de dados MySQL e a linguagem Python pelo seu bom desempenho e maior facilidade na configuração inicial junto a um novo ambiente, como em um servidor.

5.4 Programação Web

Na parte do *front-end*, decidiu-se por utilizar um *framework* chamado Bootstrap. Utilizou-se ele por facilitar o desenvolvimento e melhorar a qualidade das páginas geradas. Adicionou-se também um *plugin* ao Bootstrap chamado bootstrap-select. Ele provê uma lista de seleção personalizada, com opções de busca por texto e utilização da seleção nativa de seleção em dispositivos móveis. Além disso, utilizou-se JQuery para a realização de chamadas assíncronas ao servidor e demais interações das páginas.

Decidiu-se por manter os mesmos critérios de busca dos sites das rodoviárias com quatro listas de seleção. As listas adicionadas foram cidades de origem e destino, dia da semana e horário. A lista de cidades possui os nomes de todas as cidades do Rio Grande do Sul. Para o dia da semana não disponibilizou-se uma forma de busca em qualquer dia, pois dessa forma o algoritmo de busca deveria executar sete vezes, então o usuário deverá escolher o dia em que deseja viajar. Por padrão, o dia da semana selecionado é o dia atual. O horário funciona para determinar após qual hora o usuário deseja iniciar sua viagem. A tela com o formulário com as opções de busca pode ser vista na [Figura 19](#).

Figura 19 – Formulário do sistema web para busca de uma rota.



A imagem mostra a interface de usuário do sistema ÔNIBUSRS. No topo, há o logotipo "ÔNIBUSRS" e links para "VERSÃO SIMPLES", "COLABORE", "SOBRE" e "CONTATO". Abaixo, há um formulário de busca com os seguintes campos: "Origem:" com uma lista suspensa "Selecione...", um ícone de setas duplas, "Destino:" com uma lista suspensa "Selecione...", "Dia:" com o valor "Sábado" e uma lista suspensa, e "Hora:" com o valor "00:00" e uma lista suspensa. Um botão "Procurar" está à direita dos campos.

Ao selecionar a consulta desejada e clicar em procurar o sistema irá realizar uma requisição POST assíncrona a uma página PHP do servidor. Esta página irá realizar o

processamento, chamando o *script* de busca se necessário, e retornar as rotas. Quando a requisição terminar, o retorno é escrito na tela. O código de busca do arquivo PHP pode ser visualizado no Algoritmo 3, onde o `buscador_externo` é o algoritmo de busca A* em Python. O resultado de uma busca pode ser visualizada na Figura 20.

```

Entrada: configurações de busca
Saída: lista de rotas formatadas em HTML
if existe rota direta then
    | obtém todas rotas diretas;
    | formata e escreve as rotas na saída;
else if rota já foi procurada anteriormente then
    | obtém rotas no banco de dados;
    | formata e escreve as rotas na saída;
else
    | chama buscador_externo;
    | salva rotas no banco de dados;
    | formata e escreve as rotas na saída;
end

```

Algoritmo 3: Script de busca no servidor.

Figura 20 – Resultado da busca entre Alegrete e Barra do Quaraí.

ROTA 1					
Alegrete	→	Uruguaiana	Planalto	Semi Direto	🗨
16:40	149km	18:15	R\$31,65	0798	
Uruguaiana	→	Barra do Quaraí	Perini	Comum	🗨
19:00	75km	19:56	R\$13,35	1873	
Distância Total: 224		Duração: 03:16		Preço Total: R\$45,00	
ROTA 2					
Alegrete	→	Uruguaiana	Planalto	Semi Direto	🗨
00:45	149km	01:55	R\$31,50	0405	
Uruguaiana	→	Barra do Quaraí	Perini	Comum	🗨
06:30	75km	07:26	R\$13,35	1873	
Distância Total: 224		Duração: 06:41		Preço Total: R\$44,85	
ROTA 3					
Alegrete	→	Uruguaiana	Ouro e Prata	Comum	🗨
09:00	149km	10:52	R\$26,50	0568	
Uruguaiana	→	Barra do Quaraí	Perini	Comum	🗨
12:00	75km	12:56	R\$13,35	1873	
Distância Total: 224		Duração: 03:56		Preço Total: R\$39,85	

O algoritmo A* retorna uma lista de rotas para o PHP no seguinte formato: `XXX-XXX/YYY-YYY-YYY/ZZZ-ZZZ`, onde o resultado de cada configuração de busca é separada por um `/`, e os identificadores de cada rota são separados internamente por um `-`. Então, o código PHP obtém as informações sobre as rotas e as formata para o [HTML](#)

apropriado. Decidiu-se por não formatar os dados no *script* Python para evitar o envio de muitos dados entre os processos e pela maior facilidade de manipulação do HTML em um arquivo PHP. Além disso, as rotas neste formato podem ser salvas no banco de dados para evitar a execução do algoritmo de busca sobre os mesmos dados. Assim, se mais de um usuário realizar uma consulta com os mesmos parâmetros, o *script* PHP irá verificar se essa busca já foi realizada; caso ela tenha sido, ele obtém as rotas no banco de dados sem a necessidade da execução do algoritmo novamente.

Adicionou-se também uma forma de consulta simples. Esta forma é idêntica ao sites das rodoviárias. Porém, com todos dados das diversas rodoviárias em um único lugar. Assim, se o usuário não estiver satisfeito com a rota gerada pela busca avançada, ele poderá utilizar a busca simples. Esta busca funciona no mesmo formato da busca avançada, onde a única mudança é que a lista de destinos é preenchida dinamicamente ao selecionar uma cidade de origem, mostrando apenas cidades onde existe uma rota até ela.

Como pode ser visualizado na direita de cada rota de ônibus da [Figura 20](#), existe um ícone de comentário. Ao clicar neste ícone, uma janela irá aparecer para que o usuário possa visualizar comentários e avaliações existentes e submeter suas próprias avaliações. Esta janela pode ser visualizada na [Figura 21](#). Utilizou-se os seguintes critérios para avaliação: rodoviária, ônibus e pontualidade. A qualidade da rodoviária deverá determinar se a rodoviária é confortável para a espera de um ônibus, onde o usuário deverá considerar fatores como espaço interno, opções para comida, entre outros. A qualidade do ônibus deverá mostrar ao usuário a qualidade geral do ônibus, incluindo assentos e banheiros. Já a pontualidade servirá para o usuário demonstrar se uma rota de ônibus possui constantes atrasos. As notas para cada critério vão de uma escala de 1 a 5, e a média geral é mostrada, junto com a lista de comentários ao lado.

Os comentários servem apenas como detalhes informativo ao usuário. Nenhuma das informações são utilizadas pelo algoritmos de busca, assim servem apenas para uma ajuda extra ao usuário na hora de decidir qual ônibus ele deverá utilizar. Esta decisão de não utilizar tais dados pelo algoritmo é que eles podem ser maliciosamente manipulados, e em alguns casos excluir uma boa rota para a adição de uma ruim. Assim, caberá ao usuário a análise dos comentários.

Como atualmente não conseguiu-se 100% das rotas gaúchas, decidiu-se pela criação de uma forma do usuário contribuir com a adição de novas rotas. Assim, criou-se um formulário, onde os usuários podem adicionar novas rotas, com todas as informações detalhadas. Este formulário pode ser visualizado na [Figura 22](#), onde pode ser visto a adição de uma rota entre Jari e São Pedro do Sul.

Ainda não sabe-se como utilizar estes dados providos pelos usuários no sistema. Atualmente eles são salvos em uma tabela separada e devem ser movidos manualmente

Figura 21 – Formulário para a visualização e o envio de comentários sobre uma rota.

AVALIAÇÕES SOBRE ROTAS ×

Comentário:

Qualidade da Rodoviária:

1
2
3
4
5

Qualidade do Ônibus:

1
2
3
4
5

Pontualidade:

1
2
3
4
5

Enviar comentários

Enviar

Média geral	
Rodoviária	-
Ônibus	-
Pontualidade	-

Nenhum comentário encontrado.

Fechar

para a tabela principal. Como nem sempre possui-se capacidade para julgar se uma rota é válida, ainda pensa-se em uma melhor solução. Acredita-se que uma forma de moderação entre os usuários possa ser uma boa alternativa. Isso, porém, no momento não é possível pela inexistência de uma comunidade ativa no sistema. Sendo assim, se no futuro esta ferramenta venha ser utilizada, planeja-se a criação de um módulo adicional para moderação das rotas. Neste momento, mantém-se a moderação manual.

Figura 22 – Formulário para cadastro de novas rotas.

COLABORE COM INFORMAÇÕES ×

Origem:

Jari

Destino:

São Pedro do Sul

Dias da Semana:

Dom

Seg

Ter

Qua

Qui

Sex

Sáb

Partida:

12:00

Chegada:

13:30

Empresa:

GMT

Modalidade:

Comum

Preço:

10,50

Distância:

54

Cancelar

Salvar

Hospedagem e Publicação

Para a hospedagem e execução deste sistema poderia-se contratar um serviço de hospedagem comum. Nele diversos usuários compartilham um mesmo ambiente virtual,

onde nenhum usuário possui acesso *root* e pode apenas utilizar os recursos definidos pelo vendedor. Isso porém é um problema, pois os recursos na máquina virtual já são limitados. Adicionando-se mais usuários poderia tornar o sistema não utilizável. Além disso, precisa-se de um controle sobre a máquina para a utilização de todas as ferramentas necessárias para a execução do projeto, como a instalação do Python, que muitas vezes não são providas em hospedagens comuns. A solução encontrada foi a contratação de um servidor virtual privado.

Contratou-se então um **VPS** (*Virtual Private Server*) com as seguintes configurações: 2 cores do Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz, 2GB de memória RAM, 40GB de espaço em disco SSD, com 2TB de tráfego mensal e com um IP fixo (31.220.18.58). Instalou-se o sistema operacional Centos 6.7 em sua instalação mínima para posterior instalação do **CWP** (*Centos Web Panel*). A instalação desse painel é importante pois com ele diversos recursos necessários, como Apache, PHP e **DNS** (*Domain Name System*) server, são baixados, compilados e instalados. Dessa forma, a instalação torna-se mais limpa e segura que a instalação manual de cada recurso.

Em um VPS é comum a existência de múltiplos usuários, onde cada um possui um domínio diferente. O **CWP** possui meios de editar usuários do sistema. Além disso, cada usuário pode possuir um domínio diferente e o **CWP** prove um meio de gerenciar o **DNS** para que o domínio aponte para o respectivo conteúdo. Isso é feito de maneira transparente pelo painel, onde a única necessidade é o cadastro em um sistema *FreeDNS*, que é um servidor de **DNS** gratuito. O servidor **DNS** é responsável em traduzir os nomes (domínios) em endereços de IP ([ANDERSON, 2015](#)).

Contratou-se um domínio, onibusrs.com, que foi então configurado junto ao *FreeDNS* e apontado ao servidor. O **CWP** é responsável então por identificar o domínio e redirecionar ao diretório correto. Neste caso o diretório é a pasta *public_html* do usuário *onibusrs*. Assim, o domínio onibusrs.com se traduz ao seguinte endereço: <http://31.220.18.58/~onibusrs/>.

5.5 Programação para Dispositivos Móveis

5.5.1 Android

Em alguns testes iniciais utilizando o Intel XDK + Cordova, com uma interface simples, o **APK** gerado possuía tamanho superior a 20MB. Isso era devido ao fato de que uma *WebView* otimizada estava sendo empacotada junto com a aplicação. Apesar de ter-se descoberto formas de utilizar a *WebView* nativa, decidiu-se pela utilização da programação nativa. Essa decisão baseou-se pelo tamanho do **APK** gerado, que ficou em torno de 3MB e também pela maior qualidade do software gerado.

Em um primeiro momento planejou-se a implementação dos algoritmos de busca na própria aplicação. Porém, após análises, constatou-se que se a aplicação executasse algoritmos complexos tanto o desempenho quanto o uso da bateria do celular seria afetado consideravelmente. Assim, decidiu-se pela realização do processamento no servidor.

O aplicativo possui a base de dados SQLite embutida na aplicação. Assim, se existir uma rota direta entre o destino e a origem, nenhuma comunicação com o servidor será feita. Apenas uma consulta SQL é realizada, e o resultado é mostrado. Caso nenhuma rota direta existir, uma requisição é feita ao servidor com o envio dos seguintes parâmetros: origem, destino, dia da semana e horário. Como resposta, uma sequência de caracteres como a descrita na seção 5.4 é recebida e os dados são obtidos na base de dados local e mostrados ao usuário. Assim, consegue-se realizar a comunicação com o servidor com envio e recebimento de uma baixa quantidade de dados.

O código de busca utilizado segue o mesmo padrão do Algoritmo 3. Onde, o aplicativo realiza uma busca local, utilizando a base de dados SQLite, para verificar se existe uma rota direta. Se não existir, o algoritmo A* é chamado, passando os parâmetros necessários e recebendo as rotas no formato já mencionado. Assim, o aplicativo obtêm as informações de cada identificador recebido na base de dados local e exibe o resultado na tela.

A aplicação foi publicada na Google Play e pode ser visualizada pelo seguinte endereço <https://play.google.com/store/apps/details?id=com.unkapps.onibusr>. A tela de busca e seu retorno pode ser visualizado na Figura 23, onde pode ser visualizado a resposta de uma busca entre Alegrete e Jari.

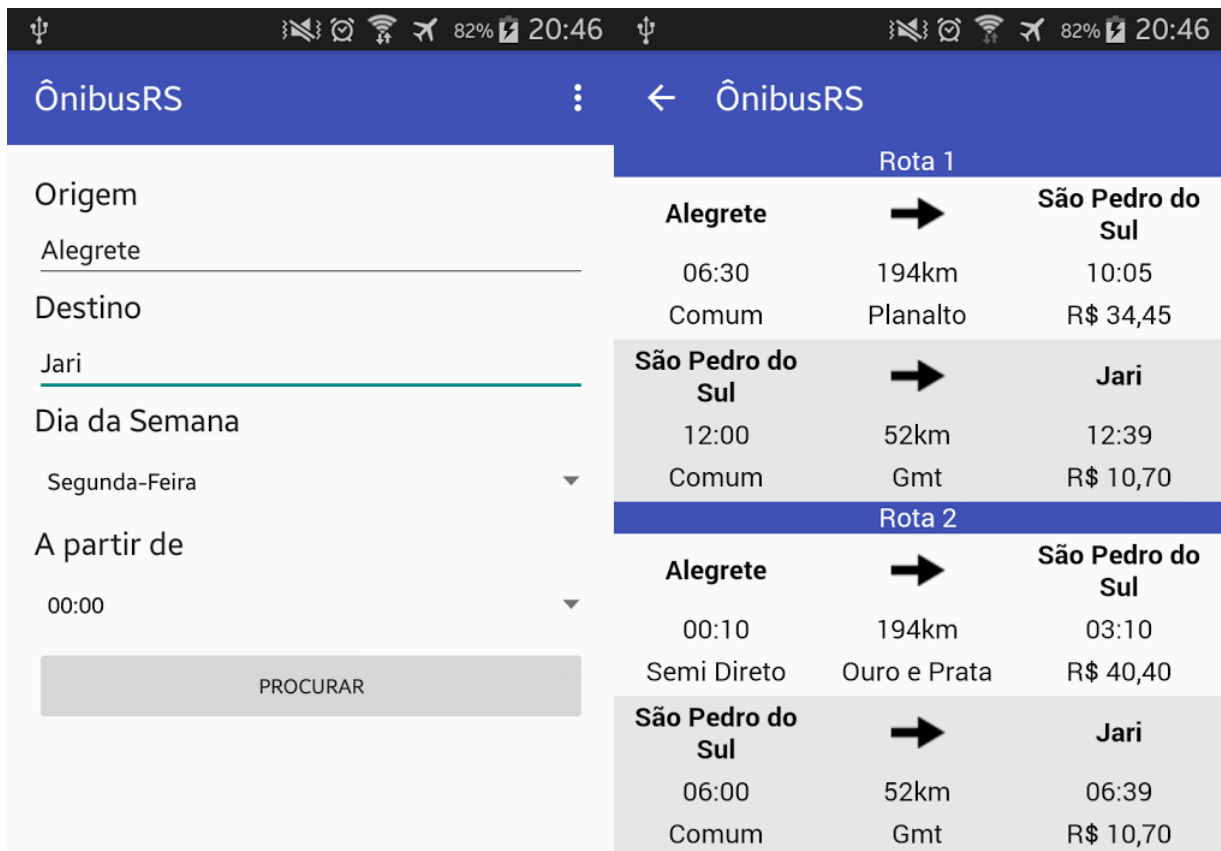
As funcionalidades extras vistas no sistema online como a visualização de comentários e envio de sugestões de rotas não foram adicionadas nesta versão. Possivelmente tais funcionalidades possam vir em alterações futuras. Porém, em um primeiro momento será analisado o engajamento do público com tais funções. Assim, elas serão adicionadas em caso de boa aceitação dos usuários.

5.5.2 iOS

O desenvolvimento para iOS seguiu os mesmos padrões da aplicação Android. A principal diferença foi a interface gráfica. Tentou-se seguir o mesmo padrão empregado na interface Android, porém, sem sucesso. Decidiu-se então utilizar um objeto de lista de seleção para que em cada tela uma opção seria mostrada. Desenvolveu-se então quatro telas principais, cidade de origem, cidade de destino, dia da semana e horário. Assim, ao clicar em procurar, na última tela uma nota interface com o resultado é mostrado. As telas podem ser visualizadas na Figura 24.

Aplicativos para iOS podem ser escritos tantos em Objective-C ou Swift. Em uma

Figura 23 – Tela de busca e resultado no aplicativo Android

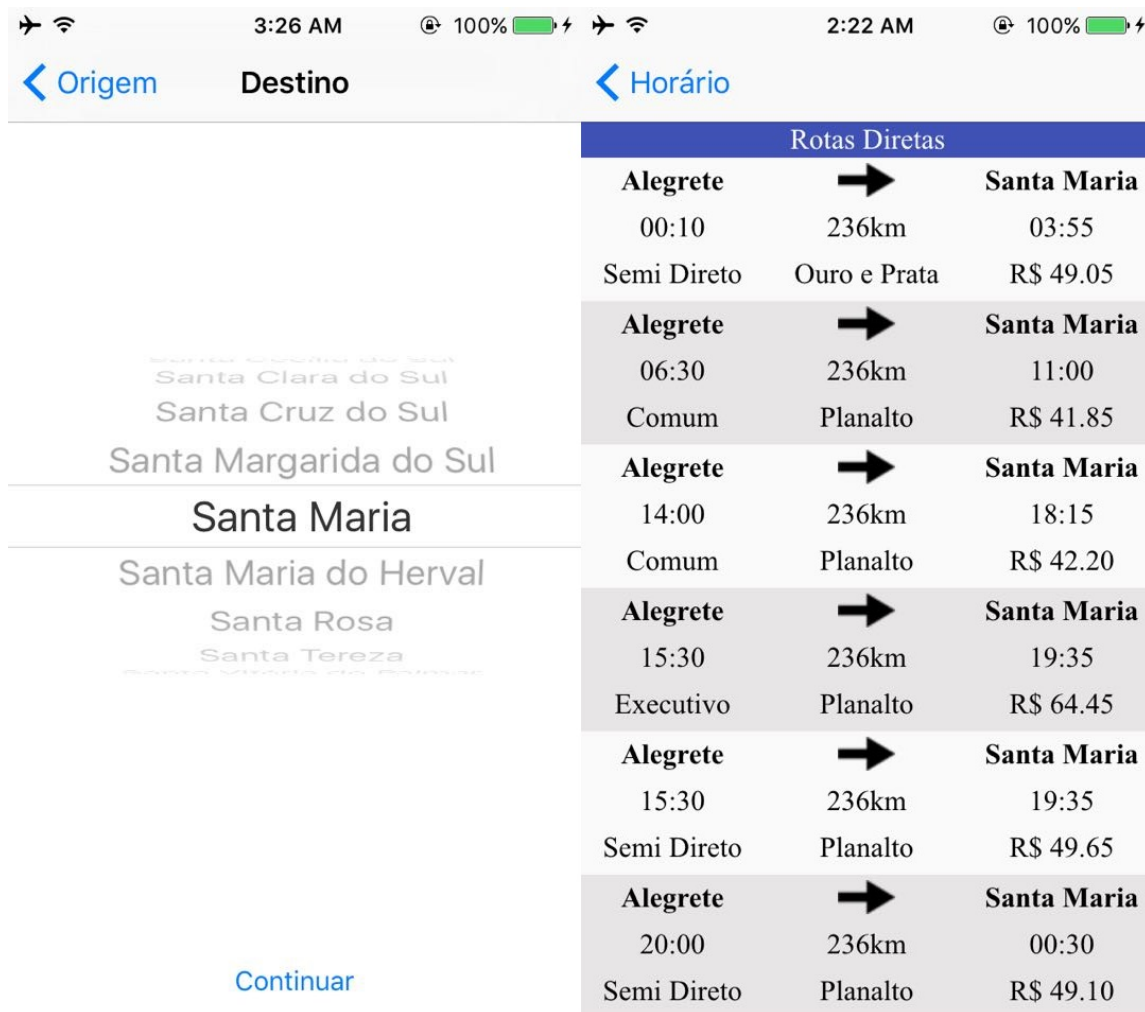


análise de ambas linguagens, constatou-se que o Swift era uma linguagem mais clara e mais próxima de linguagens que estava-se acostumado a utilizar, como Python. Assim, escolheu-se o Swift como linguagem a ser utilizada.

O alto custo de desenvolvimento foi um dos principais problemas para o desenvolvimento deste aplicativo para iOS. Enquanto para um aplicativo para Android ser testado e publicado o desenvolvedor pode utilizar qualquer sistema operacional, testar de graça e publicar na loja com pagamento único de 25 dólares, o processo de desenvolvimento de aplicativos para dispositivos Apple é diferente. Primeiramente necessita-se de um computador Apple e a conta de desenvolvedor custa 100 dólares anuais. Além disso, antes da versão 7 do Xcode apenas desenvolvedores que pagavam a licença poderiam executar e testar em dispositivos reais. Com esta atualização, lançada em setembro de 2015, desenvolvedores podem testar em dispositivos reais, porém a publicação na loja é liberada apenas mediante pagamento.

Decidiu-se por não investir em uma licença de desenvolvedor por seu alto custo. Assim, a aplicação desenvolvida pode ser apenas instalada no ambiente de testes. Por esse motivo, alguns detalhes tanto na interface, quanto nas funcionalidades não foram

Figura 24 – Tela de busca e resultado no aplicativo iOS



tratados. Como exemplo, o sistema atual apenas realiza consulta local, sem comunicação com o servidor. Tais aprimoramentos devem ser feitos antes de uma possível publicação na loja de aplicativos da Apple.

6 Conclusão

Durante a implementação deste projeto pode-se aplicar conhecimentos de banco de dados, inteligência artificial e programação para a criação de um sistema de busca de rotas de ônibus intermunicipais. As metas estabelecidas foram atingidas, o que inclui o sistema funcional *online*.

Além do sistema web e mobile terem potencial para contribuir com a sociedade de um modo geral, a criação dos mesmos contribuíram para a aquisição de conhecimentos durante a implementação das diversas partes deste projeto. Como exemplo, aprendeu-se técnicas para extração e tratamento de dados; aprimorou-se técnicas para gerenciamento de banco de dados; aprimorou-se técnicas para desenvolvimento web com integração de múltiplas linguagens de programação; por fim, aprendeu-se programação para dispositivos móveis, onde com a utilização das linguagens Java e Swift desenvolveu-se dois aplicativos.

No [Capítulo 3](#) analisou-se os diferentes projetos relacionados, fazendo um planejamento para o que seria atingido neste projeto em comparação aos outros. Para as rotas gaúchas, obteve-se cerca de 83% de cidades com rotas. Porém, este número não representa o total de rotas, que provavelmente deve ser menor. A busca de rotas com conexões foi implementada e adicionada ao sistema. Mesmo quando algumas cidades não possuem rotas de ônibus, uma sugestão até a cidade mais próxima é mostrada. Buscando-se assim, sempre ajudar o usuário a decidir o seu caminho. Além disso, a opção de *feedback* foi adicionada para ajudar o usuário a decidir quais as melhores rotas. Por fim, criou-se os aplicativos para dispositivos móveis Android, publicado na Google Play, e iOS, atualmente não publicado.

Os resultados obtidos, considerando os sistemas em geral, foram muito satisfatórios. Isso, porém, não implica que mudanças e melhorias não possam serem feitas. Dependendo da aceitação do aplicativo para Android, planeja-se a publicação do aplicativo na loja da Apple. Além disso, algumas melhorias no algoritmo de busca devem ser feitas para que tanto sua performance, quanto sua qualidade de resultados sejam melhoradas.

Referências

- ADOBE. *Adobe PhoneGap Build*. 2015. Disponível em: <<https://build.phonegap.com/>>. Citado na página 36.
- ANDERSON, J. *FreeDNS*. 2015. Disponível em: <<http://freedns.afraid.org/>>. Citado na página 57.
- ANDROID. *Android Open Source Project*. 2015. Disponível em: <<http://source.android.com/index.html>>. Citado 2 vezes nas páginas 33 e 34.
- APACHE. *Apache HTTP Server Project*. 2015. Disponível em: <<http://httpd.apache.org/>>. Citado na página 33.
- App Store. *App Store Review Guidelines*. 2015. Disponível em: <<https://developer.apple.com/app-store/review/guidelines/>>. Citado na página 35.
- APPLE. *Start Developing iOS Apps Today*. 2015. Disponível em: <<https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/>>. Citado na página 35.
- APPLE. *What is iOS*. 2015. Disponível em: <<https://www.apple.com/ios/what-is/>>. Citado na página 34.
- Apple Inc. *iOS Technology Overview*. Apple Inc., 2014. Disponível em: <<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iOSTechOverview.pdf>>. Citado na página 35.
- Busca Ônibus. *Busca Ônibus - Horários e passagens de ônibus*. 2015. Disponível em: <<http://www.buscaonibus.com.br/>>. Citado na página 38.
- COMPETE. *Digital Marketing Optimization Solutions | Compete*. 2015. Disponível em: <<https://siteanalytics.compete.com/hipmunk.com/>>. Citado na página 37.
- CORDOVA. *Apache Cordova*. 2015. Disponível em: <<https://cordova.apache.org/>>. Citado na página 36.
- DECOLAR. *Passagens Aéreas, hotéis e pacotes e muito mais! | Decolar.com*. 2015. Disponível em: <<http://www.decolar.com/>>. Citado na página 37.
- DECOLAR. *Política de Privacidade | Brasil*. 2015. Disponível em: <<http://comercial.decolar.com/br/confidentiality/>>. Citado na página 37.
- ej-technologies. *JProfiler*. 2015. Disponível em: <<https://www.ej-technologies.com/products/jprofiler/overview.html>>. Citado na página 51.
- ELMASRI, R.; NAVATHE, S. *Sistemas de banco de dados*. [S.l.]: Pearson Addison Wesley, 2005. ISBN 9788588639171. Citado na página 22.
- FORBES. *Why Hipmunk Is The World's Best Travel Site*. 2012. Disponível em: <<http://www.forbes.com/sites/bruceupbin/2012/06/29/why-hipmunk-is-the-worlds-best-travel-site/>>. Citado na página 37.

- GIBSON, J. *Why Your Company's Website Is Useless*. 2015. Disponível em: <http://www.huffingtonpost.com/james-c-gibson/why-your-companys-website_b_4311034.html>. Citado na página 17.
- HAN, J. *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 1558609016. Citado na página 21.
- HANRETTY, C. *Scraping the Web for Arts and Humanities*. University of East Anglia, 2013. ISBN 9781449332495. Disponível em: <https://www.essex.ac.uk/ldev/documents/going_digital/scraping_book.pdf>. Citado na página 19.
- HECHT, R.; JABLONSKI, S. Nosql evaluation - a use case oriented survey. *International Conference on Cloud and Service Computing*, IEEE Computer Society, Los Alamitos, CA, USA, p. 336–341, 2011. Citado 2 vezes nas páginas 26 e 27.
- HIPMUNK. *Cheap Flights, Airline Tickets, Flight Search | Hipmunk*. 2015. Disponível em: <<https://www.hipmunk.com>>. Citado na página 37.
- IBGE. *Acesso à Internet e posse de telefone móvel celular para uso pessoal*. 2014. Disponível em: <<http://www.ibge.gov.br/home/estatistica/populacao/acessointernet/>>. Citado na página 17.
- Internet Live Stats. *Internet Usage and Social Media Statistics*. 2015. Disponível em: <<http://www.internetlivestats.com/>>. Citado na página 31.
- JQUERY. *jQuery - write less, do more*. 2015. Disponível em: <<https://jquery.com/>>. Citado na página 33.
- LECHETA, R. R. *Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. [S.l.]: Novatec, 2012. ISBN 9788575222447. Citado 2 vezes nas páginas 33 e 34.
- MORRIS, J. *Python Software Foundation*. 2015. Disponível em: <[>](http://www.cgl.ucsf.edu/Outreach/bmi219/slides/web_client.html#/). Citado na página 32.
- MYSQL. *MySQL :: About MySQL*. 2015. Disponível em: <<http://www.mysql.com/about/>>. Citado na página 23.
- NEO4J. *Intro to Cypher*. 2015. Disponível em: <<http://neo4j.com/developer/cypher-query-language/>>. Citado na página 27.
- ORACLE. *Oracle NoSQL Database*. [s.n.], 2012. Disponível em: <<http://www.oracle.com/technetwork/database/NoSQLdb/learnmore/nosql-wp-1436762.pdf>>. Citado na página 26.
- ORIENTDB. *Why OrientDB?* 2015. Disponível em: <<http://orientdb.com/why-orientdb/>>. Citado na página 28.
- POSTGRESQL. *PostgreSQL: About*. 2015. Disponível em: <<http://www.postgresql.org/about/>>. Citado na página 24.
- Quero Passagem. *Passagens de ônibus sem sair de casa | Quero Passagem*. 2015. Disponível em: <<http://queropassagem.com.br>>. Citado na página 38.
- RODOVIARIAONLINE. *Rodoviariaonline - Seu Destino, Nosso Compromisso!* 2015. Disponível em: <<http://rodoviariaonline.com.br/>>. Citado na página 38.

ROEBUCK, K. *Mashups: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Emereo Publishing, 2012. ISBN 9781743444726. Disponível em: <<https://books.google.com.br/books?id=dFIPBwAAQBAJ>>. Citado na página 20.

ROME2RIO. *Rome2rio: discover how to get anywhere*. 2015. Disponível em: <<http://www.rome2rio.com/>>. Citado na página 37.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN 0136042597, 9780136042594. Citado 4 vezes nas páginas 28, 29, 30 e 31.

SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. *Database System Concepts, Sixth Edition*. [S.l.]: McGraw-Hill, 2011. ISBN 978-0-07-352332-3. Citado 4 vezes nas páginas 21, 23, 24 e 25.

SOLT, P. *Swift vs. Objective-C: 10 reasons the future favors Swift*. 2015. Disponível em: <<http://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html>>. Citado na página 35.

SQLITE. *Appropriate Uses For SQLite*. 2015. Disponível em: <<http://sqlite.com/whentouse.html>>. Citado na página 26.

SQLITE. *SQLite Database Speed Comparison*. 2015. Disponível em: <<http://sqlite.com/speed.html>>. Citado na página 25.

TIWARI, S. *Professional NoSQL*. Wiley, 2011. ISBN 9781118167809. Disponível em: <<http://books.google.com.br/books?id=tv5iO9MnObUC>>. Citado na página 26.

TWITTER. *Robots.txt - Twitter*. 2015. Disponível em: <<https://twitter.com/robots.txt>>. Citado na página 20.

W3SCHOOLS. *XPath Syntax*. 2015. Disponível em: <http://www.w3schools.com/xpath/xpath_syntax.asp>. Citado 2 vezes nas páginas 20 e 21.

ZAITSEV, P.; TKACHENKO, V. *High Performance MySQL: Optimization, Backups, and Replication*. O'Reilly Media, 2012. ISBN 9781449332495. Disponível em: <http://books.google.com.br/books?id=D0b_Xg3UeXEC>. Citado na página 24.