Visualizador hierárquico de arquivos através de metadados

Ricardo Leardini Lobo

Outubro / 2020

Dissertação de Mestrado em Ciência da Computação

Visualizador hierárquico de arquivos através de metadados

Esse documento corresponde a Dissertação apresentada à Banca Examinadora para qualificação no curso de Mestrado em Ciência da Computação do UNIFACCAMP – Centro Universitário Campo Limpo Paulista.

Campo Limpo Paulista, 30 de outubro de 2020.

Ricardo Leardini Lobo

Eduardo Javier Huerta Yero (Orientador)
Marta Ines Velazco Fontova (Orientadora)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Ficha catalográfica elaborada pela

Biblioteca Central da Unifaccamp

L785v

Lobo, Ricardo Leardini

Visualizador hierárquico de arquivos através de metadados / Ricardo Leardini Lobo. Campo Limpo Paulista, SP: Unifaccamp, 2020.

Orientadora: Profª. Drª. Marta Ines Velazco Fontova

Coorientador: Profº Dr. Eduardo Javier Huerta Yero

Dissertação (Programa de Mestrado Profissional em Ciência da Computação) – Centro Universitário Campo Limpo Paulista – Unifaccamp.

1. Avaliação de dados. 2. Visualização hierárquica de dados. 3. Armazenamento de arquivos. 4. Árvores de grafos. I. Fontova, Marta Ines Velazco. II. Yero, Eduardo Javier Huerta. III. Centro Universitário Campo Limpo Paulista. IV. Título.

CDD-005.1

Dedicatória

Dedico este trabalho aos meus amados pais Sandra Regina Leardini Lobo e Nivaldo Mendes Lobo, e a minha esposa, Juliana Lobo, que contribuíram ativamente para o desenvolvimento deste através do grande incentivo e apoio incondicional.

Agradecimentos

A Deus, por iluminar o meu caminho.

Aos meus orientadores, professor Eduardo Javier Huerta Yero e professora Marta Ines Velazco Fontova, por sua paciência e ensinamentos que enriqueceram este trabalho.

À minha família, pela paciência nos momentos de ausência e apoio incondicional.

Resumo. *A quantidade de dados eletrônicos disponíveis para análise cresce a cada dia. É fato que esses dados encontram-se dispersos em diversas plataformas e formatos, como por exemplo, a Internet, onde é possível obter acesso a milhões de documentos. Usuários comuns de desktops podem ter milhares de arquivos e documentos armazenados em seus sistemas e deve-se considerar ainda os grandes servidores de armazenamento de dados utilizados por grandes corporações onde se encontram bilhões ou até mesmo trilhões de arquivos armazenados. Em um esforço para lidar com a dificuldade em entender o que há de fato armazenado em um sistema de arquivos hierárquico e realizar tomadas de decisão, mais e mais usuários e corporações estão se voltando para a construção de ferramentas de análise e visualização de maneira didática de informações através de metadados. A determinação de quais arquivos tem-se armazenados e de como está estruturado hierarquicamente o sistema de armazenamento é de vital importância para instituições empresariais ou acadêmicas. Entender a natureza dos dados armazenados e como eles são usados permite traçar estratégias para adquirir novos sistemas de armazenamento quando eles forem necessários. É possível avaliar e descartar ou armazenar em meios de armazenamentos menos custosos arquivos que não são usados frequentemente, detectar duplicação de dados, atender a requisitos de compliance, dentre outros. Esta dissertação propõe como resultado a construção de uma ferramenta de visualização de arquivos através de metadados organizados em uma estrutura hierárquica na forma de árvore, de modo a permitir de maneira simplificada tomadas de decisão sobre arquivos alocados em sistemas de armazenamento de larga escala. Os testes realizados com a aplicação proposta nesta dissertação demonstram que a visualização de forma fácil e didática de grandes quantidades de arquivos facilita a localização, tomada de decisão e manipulação desses arquivos.*

Palavras-chave: *Avaliação de dados, Visualização Hierárquica de Dados, Armazenamento de Arquivos, Árvores de Grafos.*

***Abstract**: The amount of electronic data available for analysis grows every day. It is a fact that this data is dispersed in several platforms and formats, such as the Internet, where it is possible to obtain access to millions of documents. Ordinary desktops can have thousands of files and documents stored on their systems, and one must also consider the large data storage servers used by large corporations where hundreds of millions of files are stored. In an effort to deal with the difficulty of understanding what is actually stored in a hierarchical file system and making decision-making, more and more users and corporations are turning to building analysis and visualization tools in a didactic way information through metadata. Determining which files have been stored and how hierarchically structured the storage system is of vital importance for business or academic institutions. Understanding the nature of stored data and how it is used allows you to devise strategies for acquiring new storage systems when they are needed. It is possible to evaluate and discard or store in less expensive storage media files that are not used frequently, detect duplication of data, meet compliance requirements, among others. This dissertation proposes the construction of a tool for visualizing files through metadata organized in a hierarchical structure in the form of a graph tree, in order to simplify decision making on files allocated to large scale storage systems. The tests performed with the application proposed in this dissertation demonstrate that the easy and didactic visualization of large amounts of files facilitates the localization, decision making and manipulation of such files.*

***Keywords:** Data Assessment, Data Hierarchical Visualization, File Storage, Tree Graph.*

Sumário

Glossário

BD	Banco de Dados
BDG	Banco de Dados de Grafos
NoSQL	*Not only SQL*
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*

Lista de Tabelas

Lista de Figuras

1. Introdução

Nos últimos anos, diversas mudanças e inovações foram vistas nos sistemas de computação e armazenamento. O número crescente de objetos armazenados exige sistemas de armazenamento com uma escalabilidade cada vez maior. Uma das claras tendências é que novas fontes de dados - como, por exemplo, a *web*, sequenciadores de genes e sensores sem fio - continuarão produzindo quantidades cada vez maiores de informações. Vários sistemas de armazenamento em nuvem, como o Microsoft Azure (Calder, 2012) e o Amazon AWS já ultrapassaram os trilhões de objetos (Barr, 2013).

As soluções de armazenamento ao longo dos últimos anos migraram naturalmente de um armazenamento local e *offline* para armazenamentos *on-line*, devido ao surgimento e crescimento da computação em nuvem. Novos caminhos foram abertos na área de tecnologia da informação, como por exemplo: *Big Data*, Mobilidade Corporativa e soluções para armazenamento de grandes quantidades de arquivos e dados em servidores remotos. Esse fato colaborou e colabora para que os ecossistemas de dados estejam cada vez mais interconectados, gerando uma demanda cada vez maior por capacidade de armazenamento digital (Seagate, 2019). Em 2017, segundo Reuters (2018), o tamanho global do mercado de serviços em nuvem foi de 291 milhões de dólares e espera-se que atinja 983 milhões de dólares até o final de 2025, com uma taxa de crescimento anual de 16,4% durante o período compreendido entre 2018 e 2025. Já o relatório *Cloud Storage Market*, publicado por Markets (2018) estima que o mercado de armazenamento em nuvem deva crescer 23,7% ao ano, atingindo até 2022 um valor agregado de 88,91 bilhões de dólares. Esta demanda crescente por armazenamento em nuvem é impulsionada por muitos fatores, dentre eles a crescente adoção do armazenamento em nuvem híbrida – estratégia de armazenamento na qual uma organização utiliza uma combinação de armazenamento entre uma nuvem privada e um ou mais serviços de nuvem públicos. À medida que a tecnologia e a demanda por capacidade de armazenamento aumentam, as organizações prezam cada vez mais por soluções de armazenamento que tenham uma combinação de opções como alto desempenho, economia e alta disponibilidade.

As tecnologias de computação para armazenamento em massa não oferecem apenas benefícios de custo e disponibilidade, mas também apresentam problemas ainda não solucionados e que podem interferir diretamente nos custos, segurança e eficiência na

busca de informações e arquivos. A informação pode ser considerada a força vital não apenas dos negócios modernos, mas também da vida moderna. As grandes corporações e organizações atualmente lidam com *petabytes* de informações, quantidade que cresce ano a ano (Markets, 2018). Atualmente os sistemas de computador armazenam grandes quantidades de dados a todo instante. Segundo (Markets, 2018), nos próximos três anos, mais dados e arquivos serão gerados do que durante toda a história humana anterior. Muitas informações são capturadas e armazenadas automaticamente por meio de sensores e sistemas de monitoramento. Muitas das transações simples que agora fazem parte de nossas vidas cotidianas, como pagar por alimentos e roupas com cartão de crédito ou usar o telefone, são normalmente registradas para referência futura por computadores. A geração exacerbada de dados e arquivos hoje faz com que sistemas de armazenamento em massa sejam contratados apenas para cuidarem do armazenamento e preservação desses arquivos e dados. Porém, após algum tempo, com uma grande massa de arquivos e dados armazenados, encontrar uma informação valiosa é difícil. A descoberta de arquivos é um processo de trabalho intenso, que envolve a coleta e análise de centenas ou milhares de dados de informações armazenadas. Mais do que isso, saber exatamente o que se tem armazenado e otimizar os processos de tomada de decisão e gastos com armazenamento torna-se algo quase impossível sem uma ferramenta de visualização de dados de maneira gráfica, visto que a maioria dos sistemas de arquivos e armazenamento de dados atuais não permitem visualizar arquivos e metadados (informações associadas a cada arquivo) de forma simplificada e eficaz.

Questões como a análise de qualidade dos dados armazenados, duplicidade e integridade de arquivos são apenas alguns dos problemas que contribuem para que o armazenamento de grandes quantidades de arquivos e dados em servidores externos torne-se um problema para as organizações. O uso de métodos automatizados pode ajudar a identificar algumas anomalias, mas a determinação do que constitui um erro depende do contexto de cada arquivo, requerendo, portanto, uma avaliação humana sobre o conteúdo armazenado (Kandel et al, 2012). Por esta razão, analisar e entender os arquivos e informações que se tem armazenado tem se tornado uma tarefa cada vez mais difícil. Embora existam algumas ferramentas de visualização para facilitar a análise de arquivos e dados, implementando políticas de *data assessment*, os analistas frequentemente precisam construir de forma manual as visões necessárias, o que demanda tempo e um

nível de conhecimento significativo das tecnologias empregadas. Descobrir e corrigir problemas de qualidade de dados também pode ser uma tarefa com um custo financeiro elevado: estima-se que a avaliação e limpeza de dados duplicados ou inconsistentes sejam responsáveis por até 80% do custo de projetos de hospedagem de dados (Kandel. et al, 2012).

Em uma era na qual conceitos como *Big Data* se fazem cada vez mais presentes no cotidiano, o gerenciamento eficaz de objetos armazenados através de metadados se torna essencial na manutenção dos sistemas de armazenamento.

A análise de informações através de metadados – processo que envolve a indexação de metadados de arquivos – utilizando uma ferramenta de visualização gráfica pode ajudar a resolver alguns problemas presentes em sistemas com grandes quantidades de arquivos armazenados. É possível visualizar e responder a questionamentos tais como "quais arquivos do usuário estão consumindo mais espaço?" ou “quais arquivos não são acessados há mais de um ano?” de maneira rápida, permitindo uma análise e tomada de decisão correta (Leung. et al, 2009). Assim, consegue-se eliminar arquivos não utilizados que consomem armazenamento, reduzir custos com espaço em disco, remover arquivos que comprometam a integridade organizacional ou ainda encontrar arquivos de forma facilitada.

Uma variedade de novos paradigmas e *frameworks* de visualização foi desenvolvida nos últimos anos. No entanto, conseguir visualizações flexíveis de informações contidas em grandes espaços de armazenamento, isto é, pré-processar grandes quantidades de informação, exibir o contexto da informação e suportar uma variedade de filtros de exploração são considerados problemas recorrentes na área de pesquisa em computação (Keim, 2001).

Uma representação visual fornece um grau de facilidade na análise das informações nela contida muito maior do que em representações dos tipos numérica ou textual. Este fato leva a uma forte demanda por técnicas e ferramentas de exploração visual e as tornam indispensáveis em conjunto com técnicas de exploração automáticas (Keim, 2001). Segundo Few (2009) pode se conceituar o termo “visualização” de maneira geral como a representação visual da informação, sendo que este termo pode ser diretamente associado a outros três termos com significados um pouco diferentes, sendo eles:

Visualização de Dados, Visualização da Informação e Visualização Científica. De acordo com (Few, 2009), o termo Visualização de Dados pode ser utilizado de modo a cobrir os variados tipos de representações visuais que suportam a exploração e análise de dados. Os termos Visualização da Informação e Visualização Científica podem ser considerados termos derivados do termo Visualização de Dados, sendo que ambos fazem referência a tipos específicos de representações (Few, 2009).

A visualização de estruturas de informação hierárquica é um tópico importante nas pesquisas sobre visualização de arquivos e metadados (Keim, 2001). A maior parte das pesquisas nessa área concentra-se no desafio de exibir grandes hierarquias de uma forma compreensível. Uma hierarquia de arquivos pode ser representada como uma árvore. Muitas vezes é necessário navegar pela hierarquia de arquivos para encontrar um arquivo específico. Qualquer um que tenha feito isso provavelmente experimentou alguns dos problemas envolvidos na visualização de gráficos: "Onde estou?" Onde está o arquivo que estou procurando? (Bundt, 2018).

A exibição de informações em estruturas organizadas de forma hierárquica, como árvores genealógicas, organogramas, sistemas de arquivos e diagramas de classe geralmente podem ser representadas por estruturas compostas por arestas e nós. Porém, além da exibição de dados, é necessário que os usuários e analistas explorem a visualização dos dados para conseguir extrair relações entre as informações. Segundo (Tukey, 1977) e (Bertin, 1981), a análise de dados exploratória e a necessidade de aplicações gráficas que deem suporte a este processo levaram ao desenvolvimento de muitas técnicas de visualização, nas quais o *design* da visualização possui um papel tão relevante na análise de dados quanto os mecanismos de interação fornecidos aos analistas. Ou seja, uma visualização didática dos dados é tão importante para um analista quanto as técnicas de análise empregadas no levantamento de informações (Cava, Luzzardi e Freitas, 2003).

Ao longo das últimas décadas, várias técnicas de visualização foram desenvolvidas para permitir a navegação e análise em conjuntos de informações exibidas de forma hierárquica. Dentre as formas de visualização podemos destacar: Mapa de Árvore (treemap) (Johnson e Shneiderman, 1991), (Shneiderman, 1992), Árvores de Cone (cone trees) (Robertson, Mackinlay e Card, 1991) e o navegador Hiperbólico (Lamping, Rao e

Pirolli, 1995). Pode-se citar ainda o uso de grafos, representados por arestas e nós para representar um sistema de arquivos ou a estrutura de páginas de um site (Lamping, Rao e Pirolli, 1995), (Munzner, 1997). Essas técnicas de visualização transmitem uma melhor percepção de alguns atributos ou fornecem recursos de interação adicionais na avaliação de dados, pois fornecem uma representação visual de todo o espaço de informações, bem como uma visão detalhada de algum item ou região de interesse selecionado. Cada uma das diversas técnicas de visualização voltadas para a exploração de informações dispostas de maneira hierárquica existentes tenta incorporar recursos extras para melhorar a percepção do usuário, visando apoiar o processo de *data assessment* ou tomada de decisões (Cava, Luzzardi e Freitas, 2003).

Uma das visualizações de árvores mais conhecidas é a árvore enraizada (Holten, 2006). A árvore com raiz, conforme pode ser observada na Figura 1, é um exemplo de uma visualização em árvore baseada na representação intuitiva dos nós: o relacionamento entre os nós pai e filho é representado por meio de linhas que os interconectam. O *layout*, visualizado de cima para baixo, posiciona nós filhos abaixo de seu respectivo nó pai, sendo este o *layout* de árvore com raiz mais comum.

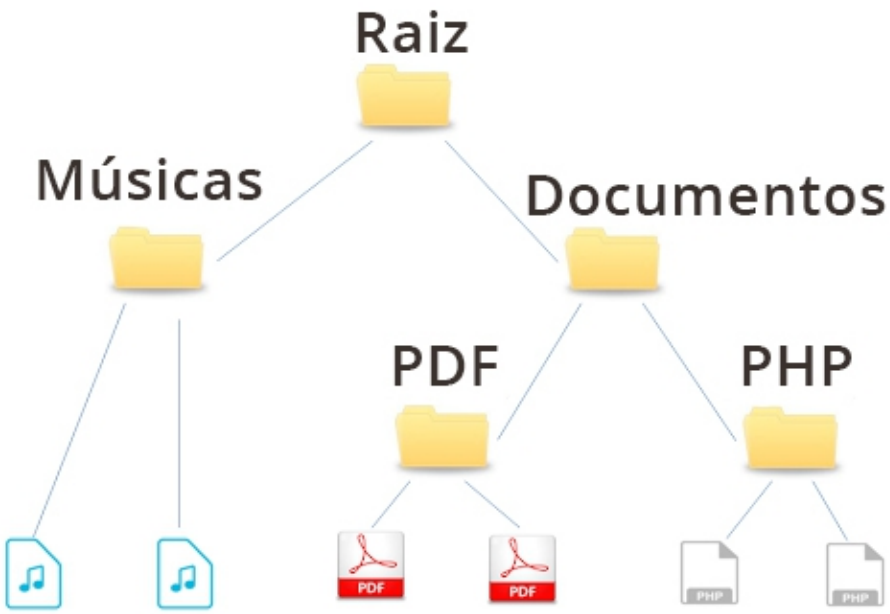

Figura 1. Árvore com nó raiz.

O *layout* de mapa de árvore, conforme mostrado na Figura 2, é uma técnica de *layout* que exibe uma estrutura de árvore através do preenchimento de espaços, o que a torna uma técnica ideal para exibir árvores grandes (Shneiderman, 1992). No exemplo

exposto na Figura 2, cada retângulo representa um arquivo. A cor do retângulo indica o tipo de arquivo e o tamanho indica a quantidade de espaço utilizado em disco pelo arquivo que está sendo representado. Uma desvantagem do uso de mapas de árvore é que se torna mais difícil para os usuários perceberem a relação hierárquica entre os nós (Holten, 2006).

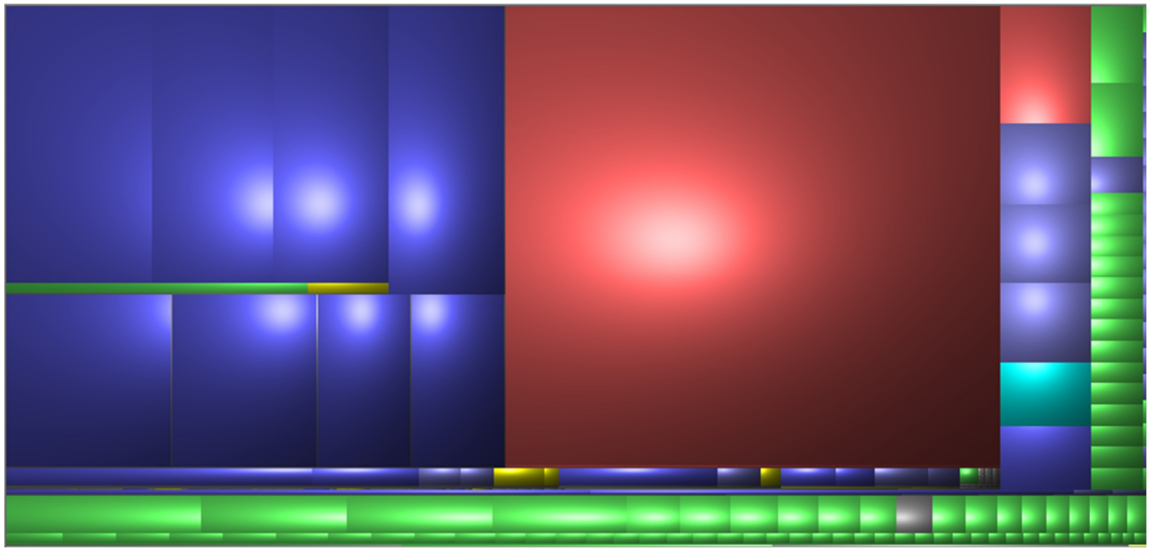

Figura 2. Mapa de Árvore.

Nesta dissertação é proposta uma solução para percorrer um sistema de armazenamento hierárquico (como por exemplo, um sistema de arquivos, um sistema de armazenamento de objetos, dentre outros) e gerar representações visuais simples e intuitivas, baseadas nos metadados disponíveis e nas pesquisas realizadas sobre usabilidade, sendo uma alternativa visual as demais técnicas pesquisadas.

Enquanto os objetivos específicos dessa dissertação são:

- Realizar o estudo e análise a respeito da visualização de dados e arquivos, bem como das técnicas e ferramentas existentes e das características desejadas e usabilidade para ferramentas de visualização.
- Oferecer a análise dos principais conceitos de um sistema de arquivos e seus metadados.
- Propor um protótipo de *software* modular para leitura e armazenamento de metadados de dados e arquivos;
- Propor um módulo de visualização didático de arquivos armazenados de maneira hierárquica, através de seus metadados, agregando umm conjunto de conceitos de usabilidade a uma técnica de visualização não encontrados nos demais softwares

estudados, e que atue como um facilitador para tomadas de decisão baseadas na visualização de arquivos como, por exemplo: identificação de arquivos pela sua extensão através de ícones coloridos, identificação de diretórios que ocupam mais espaço em disco, dentre outros, sendo uma alternativa aos *softwares* gratuitos já existentes e as ferramentas e técnicas de visualização existentes;

Essa dissertação está organizada da seguinte maneira:

O Capítulo 2 apresenta a metodologia de pesquisa bem como a pesquisa de referencial bibliográfico com aplicações e trabalhos correlatos.

O Capítulo 3 apresenta a pesquisa bibliográfica com as definições de visualização de dados bem como seus conceitos e ideais de usabilidade. O capítulo apresenta também a definição e descrição das técnicas e sistemas de visualização de dados.

O Capítulo 4 trata das principais definições referentes a sistema de arquivos e metadados de arquivos. A seção apresenta também como os diretórios podem ser estruturados em um sistema de arquivos.

O Capítulo 5 trata da solução de *software* proposta nessa dissertação. São discutidos aspectos como a arquitetura e modelo de dados utilizado no desenvolvimento do protótipo do projeto. O capítulo disserta também a respeito dos tipos de sistemas de bancos de dados NoSQL, fundamentais para o entendimento e escolha da base de dados a ser utilizada no protótipo de software obtido nesta dissertação. Por fim, o capítulo apresenta também as justificativas e motivos das escolhas tecnológicas para o desenvolvimento do protótipo de *software* proposto nessa dissertação.

O Capítulo 6 apresenta e detalha o funcionamento da aplicação obtida como resultado deste projeto de dissertação e o comparativo do resultado obtido com as demais ferramentas correlatas pesquisadas.

O Capítulo 7 apresenta as considerações finais desta dissertação, junto às contribuições identificadas e trabalhos futuros para o tema desta pesquisa.

2. Revisão Bibliográfica

Este capítulo tem como objetivo descrever aplicações correlatas ao protótipo proposto neste projeto de dissertação. Será abordado de maneira geral o conceito de visualização de dados e suas diversas formas de apresentação, com um enfoque maior para o modelo de visualização em árvore. Serão apresentados também exemplos de aplicações correlatas desenvolvidas tanto com fins acadêmicos quanto com fins comerciais.

2.1. Metodologia de Pesquisa

Para a obtenção de materiais visando obter conteúdo consistente para a fundamentação teórica desta dissertação, foram realizadas buscas nas seguintes bases de artigos acadêmicos: Google Acadêmico, IEEE Xplore, ACM DL e Springer Link.

As buscas nas bases foram realizadas utilizando as seguintes palavras-chave no idioma inglês: *data assessment, metadata search*, *large-scale storage systems*, *data quality assessmen*t, *hierarchical view*, *data visualization* e *hierarchical data view*. No idioma português, as palavras-chave utilizadas foram: avaliação de dados, visualização hierárquica de dados, busca de metadados.

Também foram realizadas buscas por artigos no buscador Google, com foco em encontrar soluções comerciais relacionadas ao protótipo proposto nesta dissertação. Para a busca de soluções comerciais foram utilizadas as seguintes palavras-chave: *data assessment* e *data visualization*.

O processo de seleção de artigos foi realizado seguindo os seguintes passos:

1) Pré-seleção de artigos baseada na análise dos títulos e leitura dos resumos.
2) Leitura integral dos artigos pré-selecionados. Neste ponto, os artigos cujo conteúdo era pouco relevante ao assunto principal da pesquisa (baseado na palavra-chave utilizada para pesquisa e obtenção do artigo) foram descartados.
3) Sintetização e avaliação dos resultados obtidos com a leitura dos artigos com contribuições consideradas relevantes para a fundamentação teórica desta dissertação (cujos resultados obtidos contribuíam para esta dissertação de mestrado, seja no seu desenvolvimento prático ou em sua pesquisa a respeito das técnicas de visualização).

4) Pesquisa e avaliação das técnicas de visualização e dos *softwares open source* e gratuitos encontrados para identificação de pontos de alternativa e melhoria.

2.2 Aplicações Correlatas

Há uma série de ferramentas de visualização gráfica semelhantes que compartilham o foco comum de visualização de informações do sistema de arquivos. A seguir está uma lista de alguns dos *softwares* semelhantes, juntamente com breves descrições de cada produto.

O *software Autopsy* é baseado no *software Sleuth Kit* sendo este uma coleção de ferramentas operadas através de linhas de comando, voltado para examinar imagens de disco. As funções do *software* são expansíveis através de módulos. O *Sleuth Kit* varre o sistema de arquivos do sistema operacional para localizar arquivos ocultos ou excluídos, permitindo examinar o *layout* do disco e extrair partições. É capaz de exibir todos os metadados e atributos de arquivo, hashes e detalhes da estrutura de metadados, porém de forma textual (The Sleuth Kit, 2019).

O aplicativo *Autopsy* é uma ferramenta gratuita para análise de arquivos e metadados utilizado para análise forense. Sua principal finalidade é combater o crime cibernético, mas pode ser usado para recuperação de dados pessoais. Permite compilar relatórios de uso sobre quando determinados eventos ocorreram no computador, localizar arquivos corrompidos, ocultos ou perigosos e extrair metadados de imagens e procurar ameaças. O *Autopsy* é a interface gráfica do *The Sleuth Kit* e exibe os dados coletados dos comandos do TSK (Autopsy, 2019). A Figura 3 mostra um exemplo da interface do *Autopsy*.

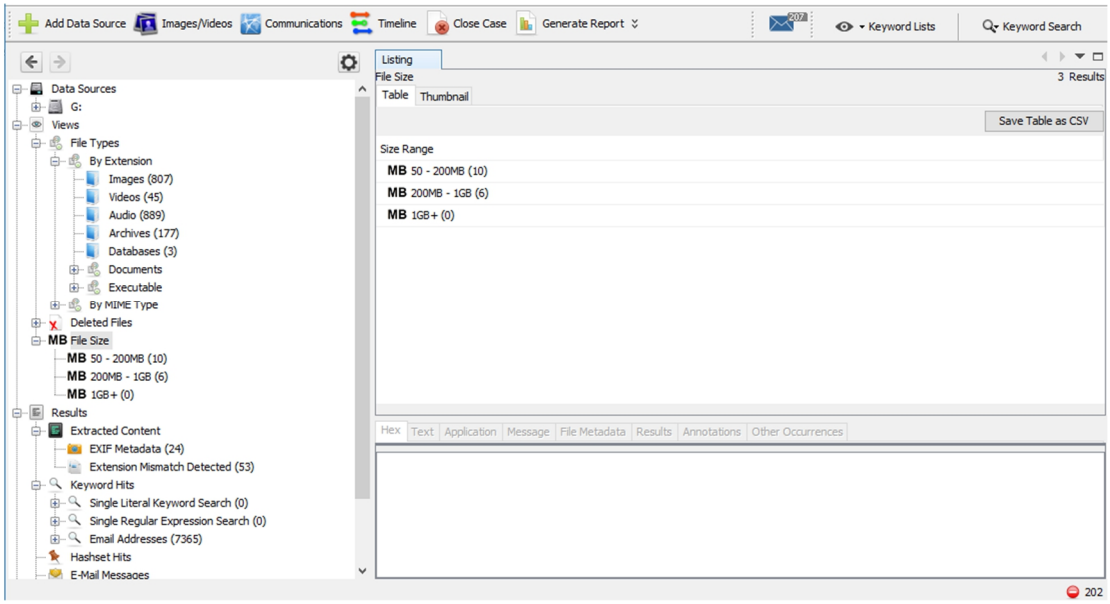

Figura 3. Interface do *software Autopsy*.

O *software File System Visualizer* é voltado para visualização dos arquivos contidos em diretórios em um mapa de árvore 3D. É visualizado onde os arquivos estão armazenados na memória, exibindo os diretórios como retângulos e os arquivos como blocos internos aos diretórios. Os blocos são proporcionais em tamanho ao espaço de armazenamento utilizado pelo arquivo. Há também uma interface na forma de árvore hierárquica 2D, exibindo os diretórios e seus subdiretórios (3D File System Visualizer, 2019). A Figura 4 mostra a interface de visualização do *software File System Visualizer.*

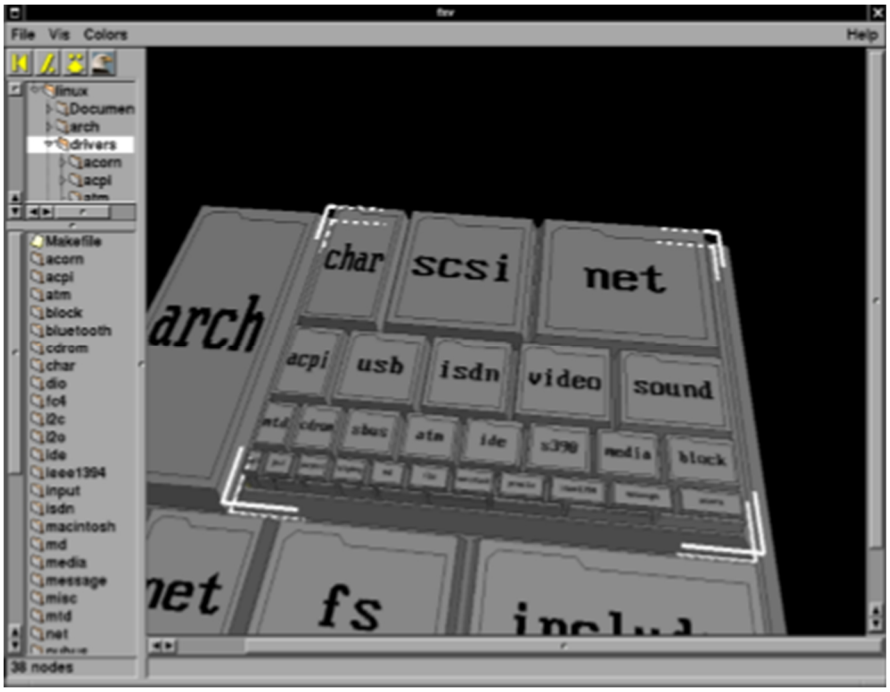

Figura 4. Interface do *software File System Visualizer*.

Windows Directory Statistics (WDS), também conhecido por Win DirStat, é um programa que varre o sistema de arquivos do Windows criando uma lista de diretórios e subdiretórios em formato de mapa de árvore e uma lista de extensões. Ele exibe uma representação visual do uso da memória. A lista de diretórios e o mapa de árvore exibem visualmente os arquivos e diretórios, enquanto a lista de extensões atua como um descritor para mostrar detalhes sobre o sistema de arquivos e seus componentes. A Figura 5 mostra como os dados são exibidos em forma visual. Cada retângulo representa um arquivo diferente, sendo proporcional ao espaço de armazenamento utilizado pelo arquivo. Da mesma forma, os diretórios são formados a partir desses retângulos e incluem sub-diretórios e outros arquivos. As cores representam o tipo de arquivo e o brilho interno de cada retângulo indica a estrutura de diretórios ao qual o mesmo pertence (WinDirStat, 2019).

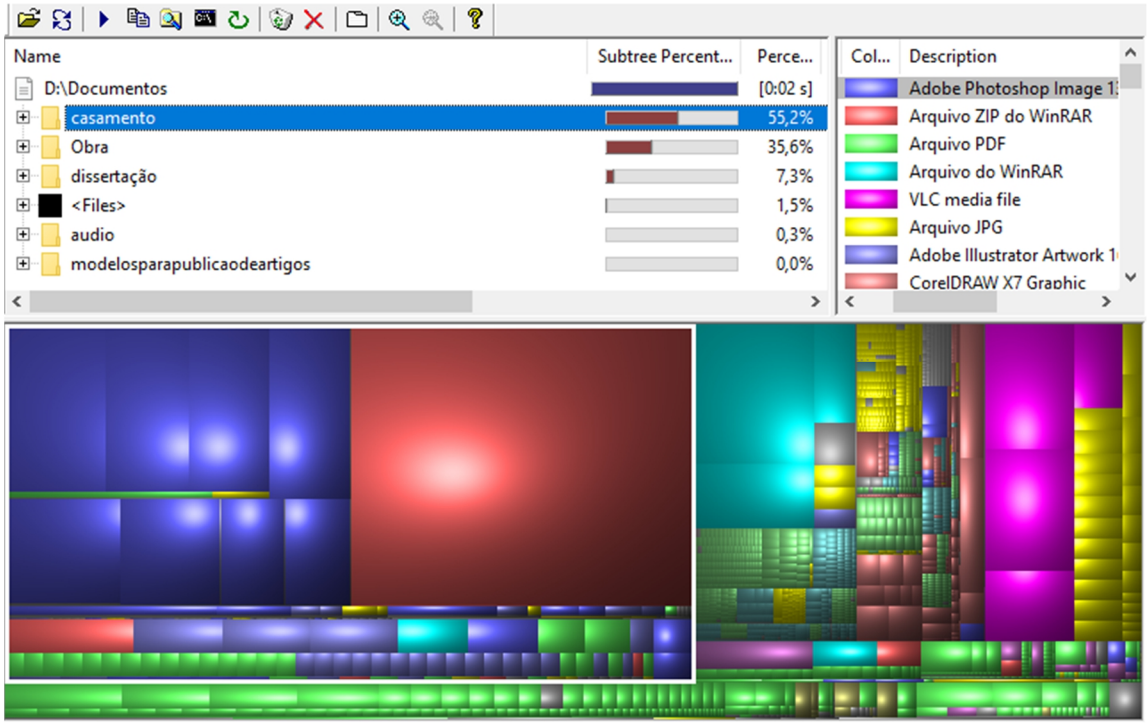

Figura 5. Interface do *software* Win DirStat.

O *software* comercial *Data Insight* foi desenvolvido e licenciado pela empresa *Veritas* e possui como objetivo principal aumentar a governança sobre dados armazenados, reduzindo assim custos de armazenamento. A ferramenta faz a indexação dos dados armazenados em um disco através de seus metadados, possibilitando ao usuário visualizar as informações dos arquivos armazenados através de filtros como data de criação do arquivo, permissões de acesso, data da última modificação, dentre outros. Uma versão de demonstração do *software* pode ser acessada pelo link https://riskanalyzer.apps.veritas.com/#!/admin/analyze. O *software Veritas Data Insight* possui o custo de licença de aproximadamente US$ 30,00 por máquina. A Figura 6 apresenta a interface do *software* exibindo os resultados da análise de um conjunto de diretórios.

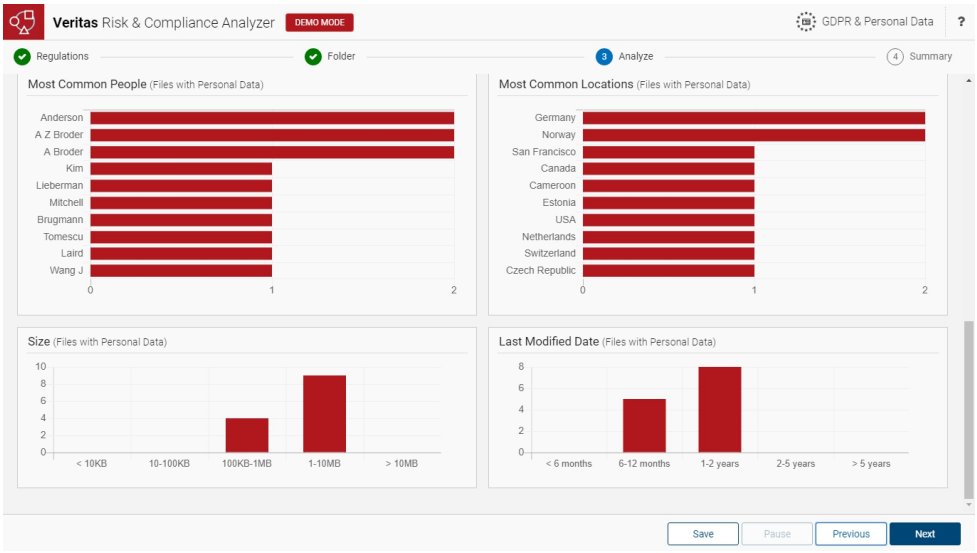

Figura 6. Interface do *software* Veritas Data Insight.

2.3 Síntese do Capítulo

As diversas técnicas existentes para visualização hierárquica de informações visam reduzir o tempo gasto na busca e facilitar a tomada de decisões sobre um conjunto de informações. Dentre os diversos modelos de visualização de informações destaca-se a visualização através de árvores enraizadas, onde as relações hierárquicas entre as informações ficam destacadas.

Todas as soluções existentes apresentadas neste capítulo possuem uma curva de aprendizado e utilização consideravelmente elevada para um usuário sem maiores conhecimentos em sistemas computacionais. A ferramenta *Sleuth Kit*, por exemplo, sequer apresenta ambiente gráfico. Já a ferramenta *Autopsy* apresenta seus resultados através de textos, deixando assim de possuir um entendimento visual rápido das informações e hierárquia de diretórios. As ferramentas Win DirStat e *File System Visualizer* apresentam seus resultados na forme de *mapa de árvores*, com informações detalhadas de cada diretório, porém que demandam um maior conhecimento técnico para serem analisadas. Os pontos apresentados acima destacam-se por serem lacunas não exploradas nos *softwares* já existentes.

Nas seções seguintes serão discutidos conceitos fundamentais para o desenvolvimento de uma ferramenta de visualização de metadados de forma gráfica.

3. Visualização de Informações

Na atualidade, é cada vez mais comum conjuntos de dados complexos e de grande tamanho, para os quais os métodos tradicionais e já existentes de visualização e análise não são tão eficazes. Conforme o tempo passar a ter um papel cada vez mais crítico e importante no processo de análise e tomada de decisão, sendo um dos fatores preponderantes para o sucesso de uma ação sobre um conjunto de dados, propor e encontrar alternativas de visualização e análise de dados e informações torna-se cada vez mais urgente.

Através do sistema visual humano (composto pelos olhos, nervos e as estruturas acessórias), a percepção desempenha um papel fundamental no que diz respeito a área da visualização, sendo auxiliar aos processos cognitivos. Portanto, considerar fatores da percepção visual humana no projeto e desenvolvimento de ferramentas de visualização de dados tem papel de extrema importância para o meio científico e para a comunidade de modo geral. Estudos que focam nestas áreas são atualmente bem vindos e necessários, dado que existem inúmeras possibilidades de pesquisa das ferramentas de visualização de dados relacionadas aos processos cognitivos humanos, sendo estas áreas que se complementam (Luzzardi, 2003).

Sendo assim, ainda de acordo com Luzzardi (2003) o elevado número de dados e informações mostrados pelas ferramentas e técnicas de visualização é uma das principais preocupações dos estudos e pesquisas. As atuais abordagens para selecionar e separar resultados relevantes nas ferramentas de visualização partem do princípio de se abstrair de alguma forma detalhes do conjunto de dados e informações e proporcionar alguma forma de organização desse conjunto de acordo com um ou mais critérios.

Quando as ferramentas e sistemas de visualização exibem dados e informações associadas e objetos físicos, fenômenos ou itens de um domínio espacial, é comum referir-se a esse conjunto de técnicas como visualização de dados científicos. Por outro lado, tem-se a visualização de informações, obtida quando se exibe dados abstratos, como relacionamentos entre dados e informações inferidas a partir de um conjunto de dados mensurados (Dulclerci e Tavares, 2007).

Utilizando-se de uma combinação de aspectos de interface homem-computador, mineração de dados, design e computação gráfica, a visualização de informações possibilita a apresentação do conjunto de dados e informações de maneira gráfica, permitindo que o usuário possa utilizar sua percepção visual para analisar e compreender as informações (Solo e Gupta, 2000).

De forma resumida, as técnicas e ferramentas de visualização de informações buscam representar de maneira gráfica dados de um determinado domínio de informações, de modo que a representação gráfica visual obtida explore as capacidades de percepção e compreensão humanas a partir das informações apresentadas, possibilitando assim a análise e dedução de novos conhecimentos (Dastani, 2002). As representações gráficas geradas podem considerar duas, três ou quatro dimensões. A dimensão de um dado em uma visualização, indica na realidade a dimensão do chamado domínio (espaço) onde este dado encontra-se. Um dado pode estar definido em um espaço unidimensional (1D), bidimensional (2D), tridimensional (3D) ou até mesmo n-dimensional (nD). Como exemplos de domínios de dados unidimensionais tem-se as distâncias medidos a partir de um ponto ou as características de algo observado durante um período de tempo. Domínios de dados bidimensionais e tridimensionais são geralmente valores observados em áreas geográficas num plano ou espaço tridimensional. Como exemplo para domínio de dados n-dimensionais tem-se as aplicações que geram dados multivariados, como dados de sensoriamento remoto ou dados populacionais. A dimensão de um atributo permite a representação de informações espaciais, temporais, espectrais ou multidimensionais, de acordo com a interpretação dos mesmos (Luzzardi, 2003).

Informações vindas de dados relacionados de uma maneira geral correspondem a um grafo representativo de uma relação entre entidades. No caso de os relacionamentos serem hierárquicos, a estrutura é implicitamente em formato de árvore.

Relativo à estrutura dos dados que são representados através das ferramentas de visualização, os mesmos encontram-se organizados em algumas das formas:

- **Listas e tabelas:** Representam um conjunto de dados relacionados de maneira linear que podem conter um dado primitivo ou estruturado como: textos, imagens, dados relacionais, dentre outros.

- **Árvores:** Representam um conjunto de dados que possuem uma relação hierárquica, ou seja, dados que possuem uma subordinação em relação a outro dado. Pode-se citar como exemplos: uma estrutura de diretórios e arquivos, árvores genealógicas, dentre outros.
- **Grafos:** Representam de modo geral um conjunto de relações entre dados, podendo ser visualizados como um conjunto de nós interligados através de arestas. Como exemplos de representações de grafos pode-se citar: o mapa de páginas e estrutura de um site, os relacionamentos entre entidades, dispositivos redes de computadores, dentre outros.

Desta forma, quando no desenvolvimento de novas técnicas ou ferramentas de visualização, os desenvolvedores devem considerar paralelamente a melhor maneira de representar graficamente um conjunto de dados ou informações de uma forma que permita a interpretação das mesmas de maneira fácil pelos usuários e também devem fornecer formas que permitam limitar a quantidade de informações visualizadas. Ou seja, a ferramenta de visualização deve apresentar uma elevada usabilidade. É também primordial que se ofereça formas de manipulação do conjunto de dados exibidos, como por exemplo rotações e zoom ou até mesmo a redução/expansão do conjunto de dados exibidos de acordo com algum critério. Deve-se ter também especial atenção na escolha da técnica de visualização que será empregada de acordo com a aplicação da ferramenta de visualização, da origem dos dados e das tarefas que são objetivos dos usuários (Luzzardi, 2003).

3.1 Usabilidade nas Interfaces de Visualização

A norma ISO 9241 (International Organisation for Standardisation) define usabilidade como a capacidade de um sistema interativo ser operado de forma eficaz, eficiente e agradável para a realização de tarefas por parte dos usuários. A usabilidade pode ser entendida ainda como uma intersecção entre as características cognitivas dos usuários e as funções de operações do sistema durante a realização de tarefas. Segundo Abowd, Coutaz e Nigay (1992) a usabilidade contém três fatores essenciais:

- **Facilidade de aprendizado:** característica da interface que permite a um usuário inexperiente e sem maiores conhecimentos no sistema compreender e utilizar a interface, alcançado em pouco tempo um maior nível de performance na operação.
- **Flexibilidade de Interação:** contém as características que permitem a troca de informações entre o usuário e o sistema.
- **Robustez de Interação:** contém as características de interação do sistema que visam a realização e avaliação de objetivos.

Já (Nielsen, 1993) afirma que a usabilidade possui múltiplos componentes com sua definição associada a cinco características, sendo elas: aprendizado, eficiência de uso, satisfação subjetiva do usuário, erros do usuário e memorização.

A capacidade de aprendizado é relativa a facilidade do usuário aprender a usar uma interface gráfica, levando em consideração suas características cognitivas. A eficiência de uso faz referência ao tempo necessário que um usuário necessita para tornar-se eficiente no uso das funções e no desenvolvimento das tarefas durante o uso de uma interface. A satisfação subjetiva tem relação direta com a satisfação do usuário com a interface gráfica do sistema de visualização, ou seja, como o usuário sente-se diante da interface. Erros do usuário faz referência a quantidade e frequência com que os usuários comentem erros ao manipular e interagir com a interface gráfica do sistema de visualização. Memorização compreende a capacidade da interface gráfica ser fácil de entender, não exigindo do usuário treinamento para executar as funções disponíveis na ferramenta.

Já Shneiderman (1996) conceituou as tarefas do usuário e a usabilidade das aplicações de visualização em sete ações: visão geral, zooming, filtragem, detalhes por demanda ou visão geral + detalhe, relações, histórico e extração.

Visão Geral: Segundo Schneiderman os usuários precisam visualizar de uma maneira geral todos os dados representados. Esta visão deve mostrar como os dados estão relacionados com outros objetos. A visão geral do conjunto total de dados é útil pois reduz o tempo de busca e auxilia os usuários na localização dos itens buscados e na escola dos próximos passos na aplicação de visualização. Os atributos gráficos mais utilizados para esta visualização geral são: localização, cores, tipos e tamanhos dos símbolos

Zooming: permite um aumento limitado nos detalhes dos dados e informações representados. Uma aplicação de visualização deve possibilitar aos usuários o zooming por dois motivos principais: focalizar um subconjunto de dados ou visualizar detalhes.

Detalhes por demanda: permitem ao usuário visualizar detalhes de um dado em particular enquanto exploram o conjunto total de informações na aplicação de visualização. Normalmente isto é implementado em seleções de um dado, exibindo os detalhes em uma janela ou *pop-up* auxiliar.

Visão Geral + Detalhes: é o conceito de oferecer múltiplas formas de visualização ao usuário na aplicação, como por exemplo uma visão geral do conjunto de dados e a possibilidade de se detalhar ou focalizar um dado em específico.

Detalhes por demanda: permitem ao usuário visualizar detalhes de um dado em particular enquanto exploram o conjunto total de informações na aplicação de visualização. Normalmente isto é implementado em seleções de um dado, exibindo os detalhes em uma janela ou *pop-up* auxiliar.

Relações: é importante ao usuário em determinadas situações conhecer as relações de uma informação quando focaliza a mesma ou no âmbito da visão geral. Estas relações podem estar vinculadas a outras informações ou atributos.

É importante salientar que as técnicas de visualização são destinadas a dar suporte a exploração e análise de um conjunto de dados representados de maneira gráfica, portanto a utilização de ações de interação é de vital importância para qualquer aplicação deste tipo.

3.2 Técnicas de Visualização

As diversas técnicas de visualização de dados e informações existentes utilizam modelos de representação gráfica para exibir visualmente dados que normalmente não possuem nenhum tipo de representação visual de forma direta e natural. Nas diversas técnicas existentes, frequentemente, os desenvolvedores buscaram aproximar a visualização dos dados com objetos do mundo real (ou ainda objetos geométricos). As técnicas e modelos podem utilizar representações visuais do conjunto de dados em uma dimensão, duas dimensões ou três dimensões, porém as dimensões de visualização não necessariamente precisam estar de acordo com a dimensão do espaço de informações.

Segundo Jerding e Stasko (1998) a técnica de visualização denominada como Information Mural mostra em um espaço de duas dimensões um conjunto de informações, em miniatura e utilizando pontos espaçados e cores. Diversos tipos de informações podem ser representados pelo Information Mural, como por exemplo: dados geográficos e documentos de textos. O Information Mural foi desenvolvido com o objetivo de manter o máximo possível o padrão das informações originais, minimizando as perdas por compressão. A Figura 7 apresenta um exemplo da visualização do Information Mural.

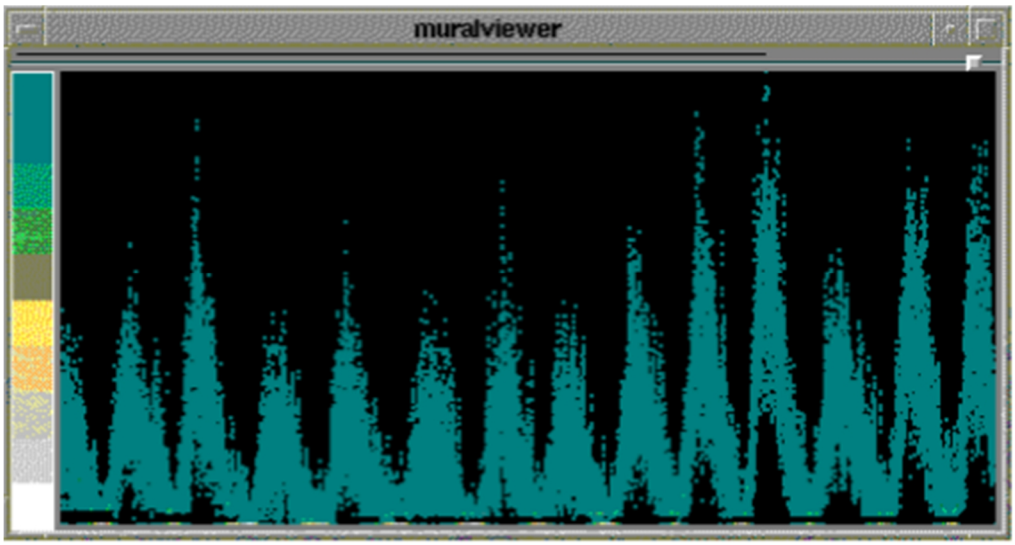

Figura 7. Information Mural (Jerding e Stasko, 1998).

A técnica Bifocal Display, proposta por Spence e Apperley (1982) apresenta documentos, figuras, gráficos, dentro outros itens de informação em três áreas separadas, sendo a área central destinada a exibir a informação em foco e as áreas laterais destinadas a exibir as demais informações do espaço de informação visualizado. Devido ao fato da área central exibir em foco a informação destacada, a aplicação de visualização apresenta uma distorção nas áreas laterais. A Figura 8 exibe um exemplo da técnica Bifocal Display.

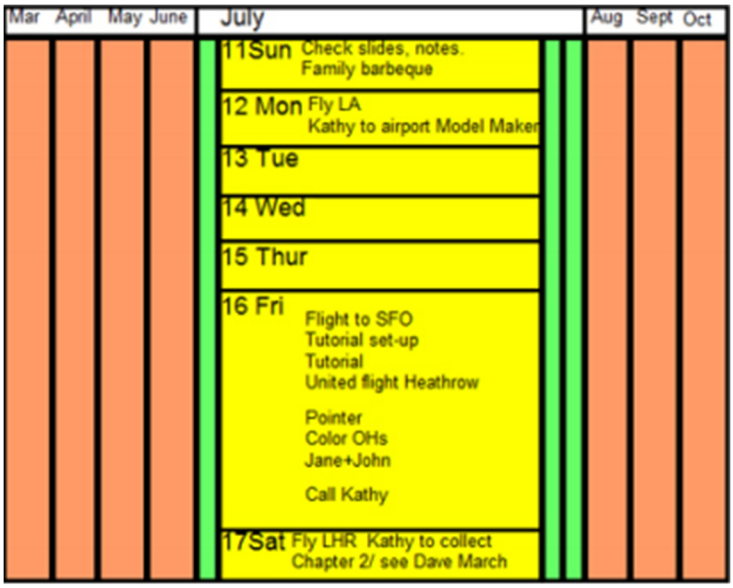

Figura 8. Bifocal Display (Spence e Apperley, 1982).

Holmquist (1997) propôs a técnica de visualização denominada como Flip Zooming. Nesta técnica, utiliza-se de ações envolvendo foco e contexto para visualizar conjuntos hierárquicos de informações, sendo estes representados por objetos distintos em uma ordem sequencial. Cada um desses objetos é identificado como uma área retangular com o foco em um desses objetos. O foco é colocado no centro da área destinada a visualização com os demais retângulos dispostos ao redor da área com foco. A técnica Flip Zooming divide a informação exibida em pequenas partes que são visualizadas dentro da área reservada a um retângulo. A Figura 9 exibe um exemplo da técnica Flip Zooming.

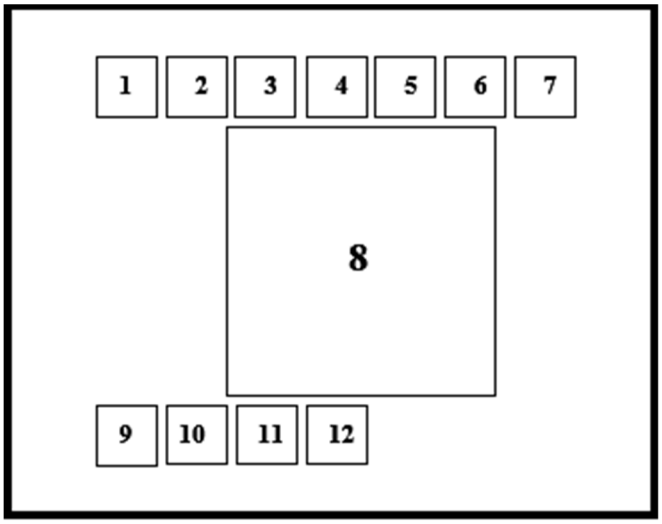

Figura 9. Flip Zooming (Holmquist, 1997).

A técnica Perspective Wall proposta por Mackinlay, Robertson e Card (1991) inicialmente voltada para a visualização de estruturas temporárias de informação como por exemplo sistemas de arquivos, baseia-se na visualização linear de espaços de informação através de visões detalhadas e de contexto geral. O layout da Perspective Wall possuir uma transformação de 2D para 3D, integrando regiões de visualização geral com regiões para visualização em detalhes de determinada informação. Uma área pode ser vista em primeiro plano enquanto as demais informações representadas ficam em segundo plano. Assim, a Perspective Wall consegue fornecer uma utilização eficiente do espaço de visualização. A Figura 10 apresenta um exemplo da técnica Perspective Wall.

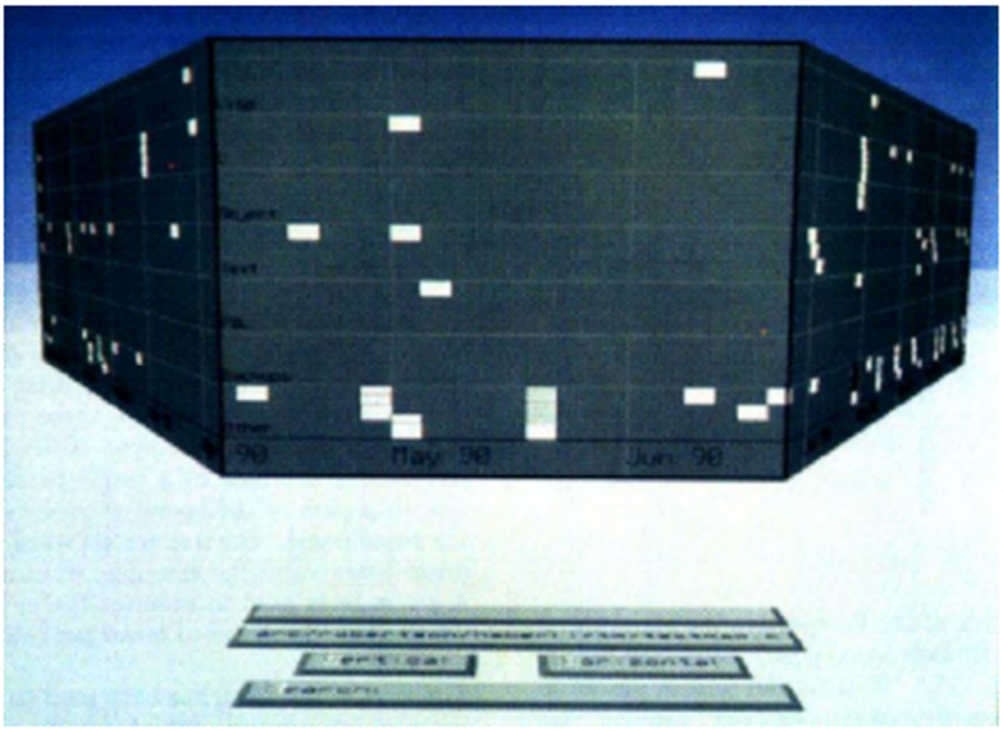

Figura 10. Perspective Wall (Mackinlay, Robertson e Card, 1991).

A técnica Information Cube é voltada para a visualização de estruturas de informação hierárquicas em um espaço 3D (Rekimoto e Green, 1993). As informações são exibidas em cubos de fundo transparente encaixados internamente entre si, onde o cubo mais externo representa o nó raiz da hierarquia. O segundo nível da hierarquia é representando por cubos que estão contidos internamente ao cubo mais externo, o terceiro nível fica contido dentro dos cubos que representam o segundo nível e assim sucessivamente. Os títulos indicativos da informação representada ficam dispostos na

superfície dos cubos. A técnica fornece os mecanismos de zooming, rotação e movimentação da área de visualização.

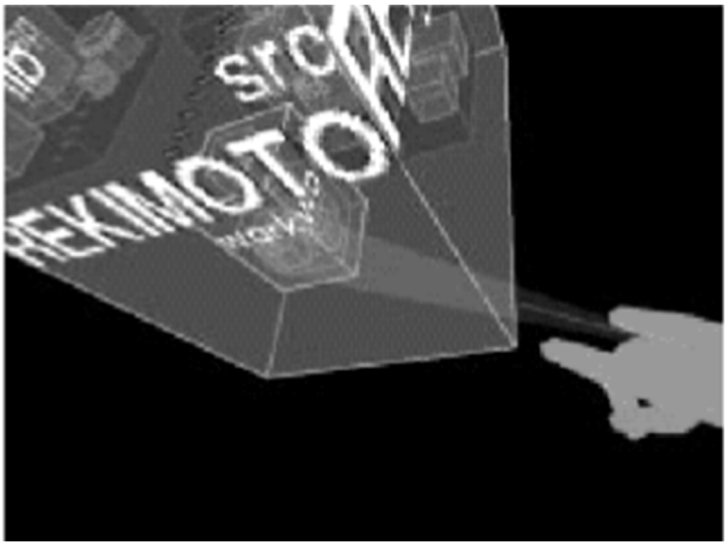

Figura 11. Information Cube (Rekimoto e Green, 1993).

A técnica de visualização CityScape, proposta por Gershon e Eick (1997) realiza a generalização de gráficos de barras em três dimensões como forma de representar dados de uma estrutura hierárquica. As barras, ou também denominadas "construções" contém informações que são visualmente diferenciadas pelo seu tamanho, cor e estilo gráfico. A Figura 12 demonstra um exemplo de visualização com a técnica CityScape.

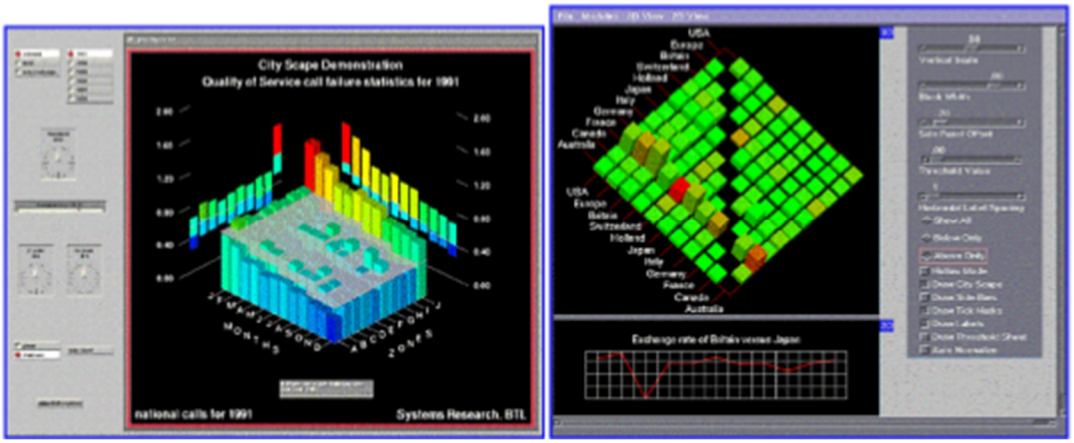

Figura 12. City Scape (Gershon e Eick, 1997).

Robertson, Card e Mackinley (1991) desenvolveram uma técnica de visualização na qual uma representação tridimensional de informações hierárquicas é visualizada no interior de um cone translucido. O nó raiz da hierarquia é representando por um retângulo e fica disposto no ápice do cone enquanto todos os nós filhos são dispostos na base do

cone. Esta técnica foi denominada como Cone Tree, sendo seu objetivo apresentar grandes quantidades de dados estruturados de forma hierárquica sem a necessidade do usuário realizar scrolling para visualizar a estrutura inteira. A técnica conta com recursos de zooming, rotação e animação (Robertson, Card e Mackinlay, 1991).

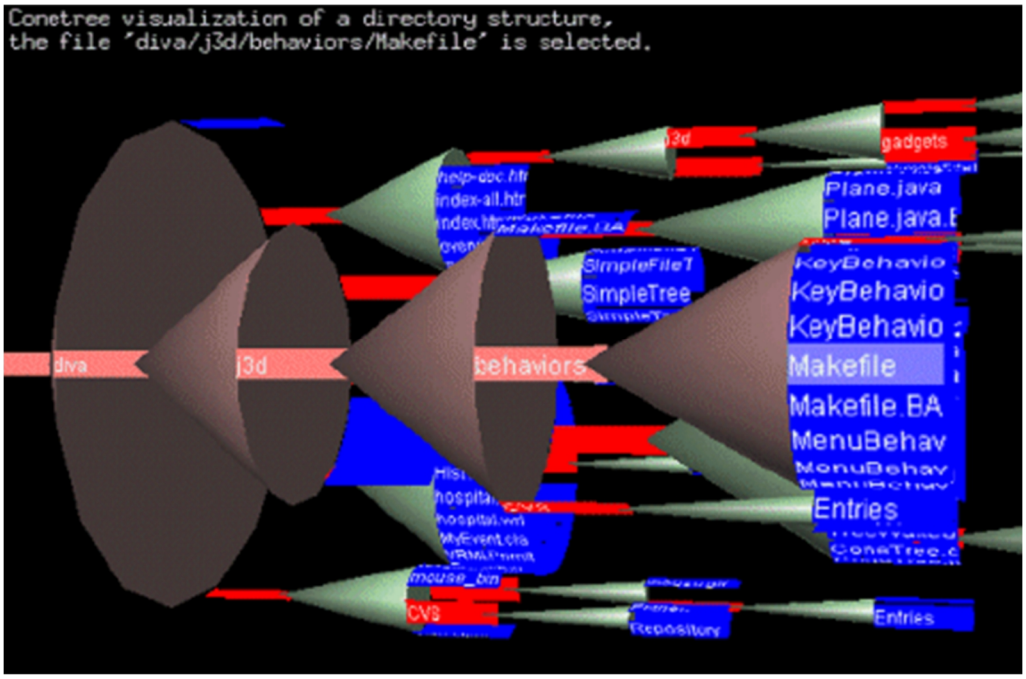

Figura 13. Cone Tree (Schonhage, van Ballegooij e Eliëns, 2000).

Johnson e Shneiderman (1991) propuseram uma técnica de visualização que se utiliza de todo o espaço da tela para representar elementos de informação. Esta técnica foi denominada de Tree Maps. Nela, uma estrutura hierárquica é representada em retângulos divididos sucessivamente no espaço todo da tela. Cada sub-espaço de um retângulo representa um diretório na hierarquia e suas subdivisões representam os arquivos e subdiretórios que o compõem. A técnica de Tree Maps é utilizada por exemplo para representar a estrutura hierárquica de diretórios de um sistema de arquivos.

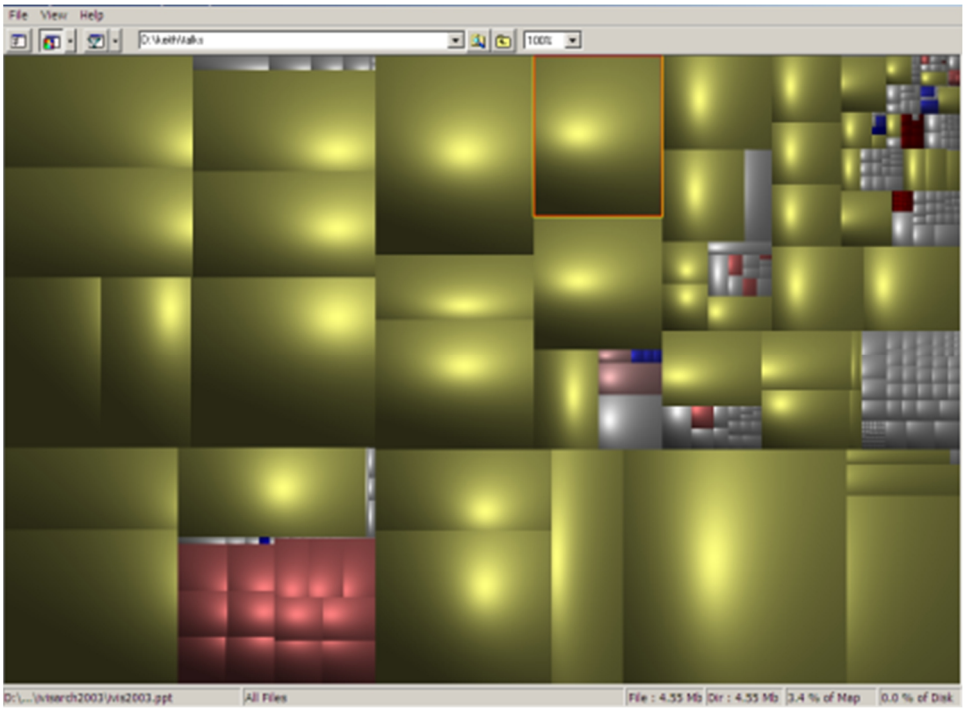

Figura 14. Tree Maps.

A técnica denominada de Time Tube, proposta por Chi et al. (1998) e demonstrada na Figura 15, foi parte integrante de um sistema chamado WEEV (Web Ecology and Evolution Visualization), que possuía como objetivo auxiliar no entendimento dos relacionamentos existentes entre os conteúdos presentes na Web. Uma visualização em Time Tube é constituída por uma ou mais Disk Trees que representam a estrutura de hiperlinks presentes em um site.

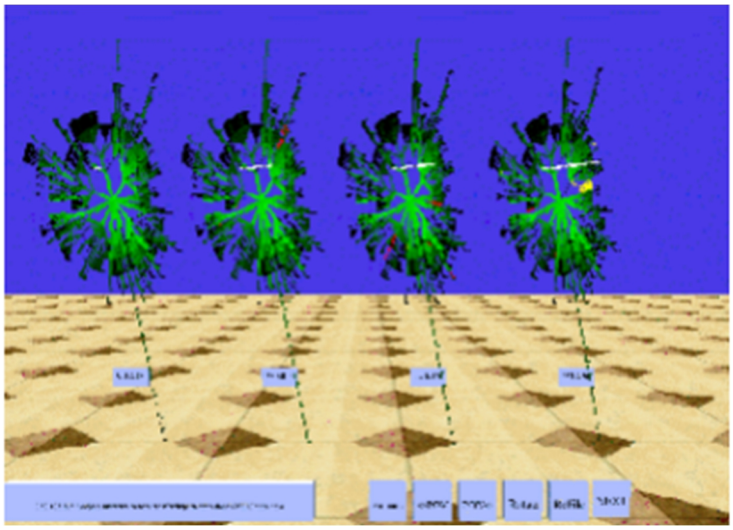

Figura 15. Time Tube (Chi, 1998).

A técnica denominada Hyperbolic Tree representa um conjunto de hierarquias definidas no plano hiperbólico e depois mapeado em duas dimensões. Utiliza-se do efeito Fisheye juntamente de um modo de navegação simples, onde a partir de um nó selecionado, que é exibido no centro da representação em detalhes, o restante da hierarquia é mantido pela exibição do restante do diagrama em nós em tamanhos menores ao redor do nó selecionado. A técnica Hyperbolic Tree é empregada atualmente no navegador comercializado pela empresa Inxight Co (Luzzardi, 2003).

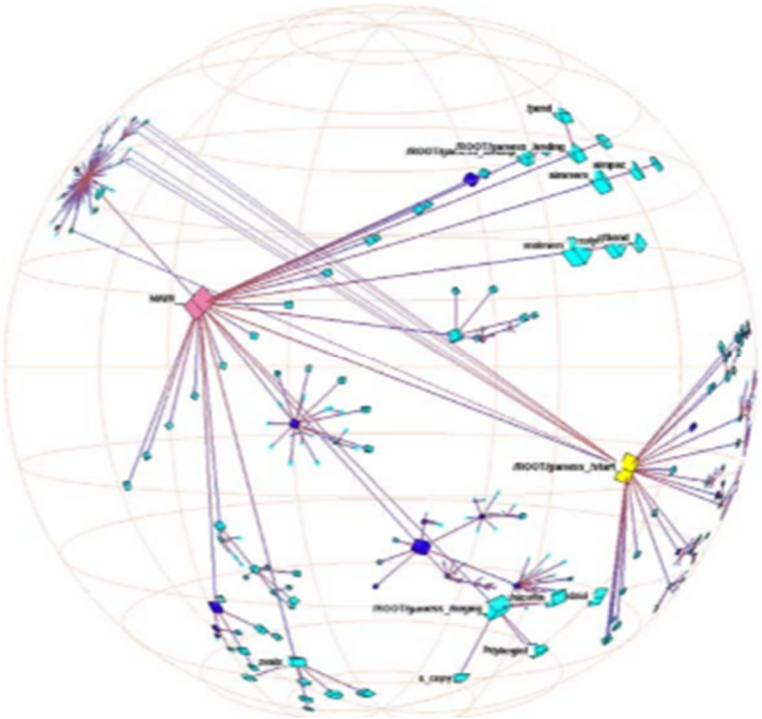

Figura 16. Hyperbolic Tree (Luzzardi, 2003).

A técnica Bifocal Tree reúne as características de duas outras abordagens: Hyperbolic Tree e a técnica de preenchimento de espaços. Na técnica Bifocal Tree, uma hierarquia é exibida através de um diagrama composto por duas áreas conexas, uma área de contexto e uma área de detalhes. A raiz da hierarquia inicialmente é exibida em foco. Quando um outro nó da hierarquia é selecionado, o mesmo passa a ter o foco e seu nó pai passa a estar na área de contexto (Cava, Luzzardi e Freitas, 2003). A Figura 17 demonstra uma Bifocal Tree

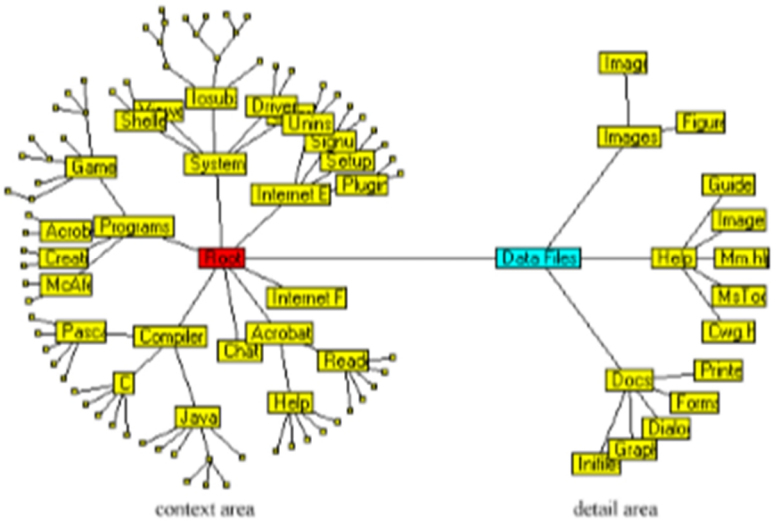

Figura 17. Bifocal Tree (Cava, Luzzardi e Freitas, 2003).

3.3 Síntese do Capítulo

Os projetos de ferramentas e as técnicas de visualização de dados devem levar em consideração durante o seu desenvolvimento as características do sistema visual humano e da capacidade de percepção oriunda deste sistema, que são fatores fundamentais para auxiliar nos processos cognitivos. Dado que o papel das técnicas e ferramentas de visualização é prover uma visão abrangente e objetiva sobre um conjunto de dados e informações permitindo que o usuário possa ter sucesso e agilidade em uma tomada de ação sobre o conjunto visualizado é possível observar que as técnicas discutidas neste capítulo buscaram, cada uma a sua maneira, interagir com o sistema visual e despertar a percepção e atenção do usuário através de seus processos cognitivos. É possível constatar, porém, que as técnicas de visualização são apresentadas com estudos e testes de usabilidade baseados geralmente em um grupo pequeno de usuários, sendo possível ressaltar que os dados dos estudos em sua maior parte são apresentados de maneira informal e frequentemente se resumem a uma simples avaliação subjetiva dos usuários sobre a ferramenta ou técnica avaliada.

4. Sistema de Arquivos

Para um correto entendimento sobre o termo sistema de arquivos, faz-se necessário observar de antemão o aspecto histórico do termo arquivo. O conceito de arquivo está diretamente relacionado com a história, sendo o termo derivado da palavra na palavra grega *archeîon*, a qual compreende os conjuntos de documentos, o lugar de armazenamento e o funcionário responsável pela custódia dos mesmos (Silva, 2017). As bibliotecas são excelentes exemplos da aplicação do termo arquivos. Em uma biblioteca tradicionalmente são armazenados diversos itens contendo informações, os quais podem ter diferentes formatos (impresso, sonoro, mapas, etc.). No contexto computacional, arquivos podem ser definidos de uma maneira geral como uma sequência de bits (0 e 1) armazenados e que representam alguma informação (texto, áudio, imagem, dentre outras). Uma unidade de armazenamento pode conter milhares ou milhões deles, cada um independente dos outros e representando uma informação específica (Tanenbaum e Woodhul, 2009). Neste capítulo, serão apresentados os principais conceitos referentes ao armazenamento de arquivos e seus principais metadados.

4.1 Armazenamento de arquivos

Todos os sistemas e aplicações de computadores precisam armazenar e recuperar informações. Para realizar tais funções, processos podem realizar a leitura de arquivos existentes e se necessário criar novos arquivos. As informações armazenadas em arquivos devem ser persistentes, isto é, não devem ser afetadas pela criação e término de um processo. Um arquivo deve desaparecer apenas quando o seu proprietário o excluir (Tanenbaum e Woodhul, 2009).

É papel do sistema operacional controlar as permissões, proteções, acessos, armazenamento e recuperação de informações de um arquivo. Assim, é denominado de sistema de arquivos a parte do sistema operacional que lida com os mesmos (Fraga e Sallum, 2016).

Um sistema de arquivo pode estruturar um arquivo em uma unidade de armazenamento de diversas maneiras diferentes (Saliba Júnior, 2017). A Figura 18 apresenta o armazenamento através de uma sequência de bytes. Neste modelo, o sistema operacional trata o arquivo apenas como uma sequência de bytes, sem importar-se com

detalhes. Tal maneira de armazenamento oferece uma grande flexibilidade, visto que programas de usuário podem armazenar qualquer conteúdo em seus arquivos. Todas as versões do UNIX (incluindo Linux e OS X) e o Windows usam esse modelo de arquivos (Tanenbaum e Woodhul, 2009).

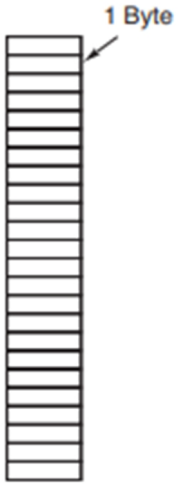

Figura 18. Estruturação de armazenamento de arquivos como sequência de bytes (Tanenbaum e Woodhul, 2009).

A Figura 19 abaixo apresenta o armazenamento estruturado de arquivos através de uma sequência de registros. Neste modelo, um arquivo é tratado como uma sequência de registros de tamanho fixo na unidade de armazenamento. Cada registro possui uma estrutura interna com as informações que compõem o arquivo. Nenhum sistema operacional atual usa esse modelo de armazenamento em seu sistema de arquivos (Tanenbaum e Woodhul, 2009).

Figura 19. Estruturação de armazenamento de arquivos como sequência de registros (Tanenbaum e Woodhul, 2009).

A Figura 20 apresenta um modelo de armazenamento de arquivos estruturado através do conceito de árvore de registros. Nesse tipo de organização, um arquivo é formado por uma árvore de registros. Os registros presentes na árvore não necessitam serem todos do mesmo tamanho, sendo que cada um deve conter um campo chave em uma posição fixa (Tanenbaum e Woodhul, 2009). A árvore é ordenada através do campo chave, permitindo uma busca rápida por um registro que contenha um valor de chave específico. Para a adição de novos arquivos, o sistema operacional é quem decide onde será realizado o armazenamento, e não o usuário (Saliba Júnior, 2017). Este modelo de armazenamento de arquivos é utilizado apenas por sistemas de arquivos de computadores de grande porte para o processamento de grandes quantidades de dados (Tanenbaum e Woodhul, 2009).

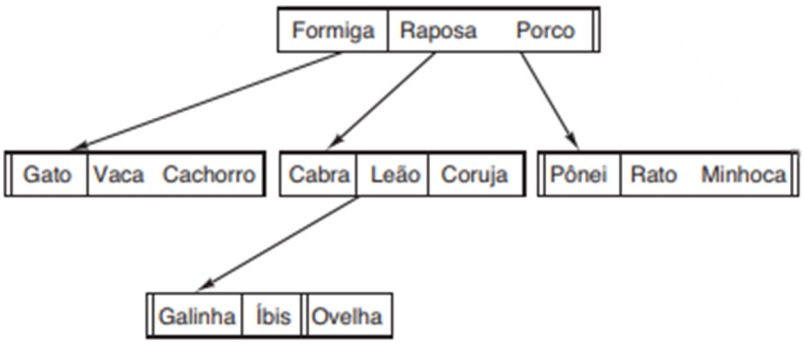

Figura 20. Estruturação de armazenamento de arquivos como árvore (Tanenbaum e Woodhul, 2009).

4.2 Metadados de arquivos

Todos os sistemas operacionais ao realizarem o armazenamento de um arquivo associam diversas informações a ele. Essas informações associadas são comumente denominadas de atributos ou metadados. Alguns metadados de arquivos armazenados são, por exemplo, a data e o horário de criação do arquivo, data e horário da última modificação do arquivo, bem como o tamanho do arquivo. A lista de metadados associados a cada arquivo varia de acordo com o sistema operacional e o sistema de arquivos utilizado (Silva, 2018). A Tabela 1 mostra os principais metadados de arquivos existentes a seus

respectivos significados. Ressalta-se que nenhum sistema operacional trabalha com todos os principais metadados existentes (Tanenbaum e Woodhul, 2009).

Tabela 1: Possíveis metadados de arquivos (Tanenbaum e Woodhul, 2009).

Atributo	Significado
Proteção	Quem tem acesso ao arquivo e de que modo
Senha	Necessidade de senha para acesso ao arquivo
Criador	ID do criador do arquivo
Proprietário	Proprietário atual
Flag de somente leitura	0 para leitura/escrita; 1 para somente leitura
Flag de oculto	0 para normal; 1 para não exibir o arquivo
Flag de sistema	0 para arquivos normais; 1 para arquivos de sistema
Flag de arquivamento	0 para arquivos com backup; 1 para arquivos sem backup
Flag de ASCII/binário	0 para arquivos ASCII; 1 para arquivos binários
Flag de acesso aleatório	0 para acesso somente sequencial; 1 para acesso aleatório
Flag de temporário	0 para normal; 1 para apagar o arquivo ao sair do processo
Flag de travamento	0 para destravados; diferente de 0 para travados
Tamanho do registro	Número de bytes em um registro
Posição da chave	Posição da chave em cada registro
Tamanho da chave	Número de bytes na chave
Momento de criação	Data e hora de criação do arquivo
Momento do último acesso	Data e hora do último acesso do arquivo
Momento da última alteração	Data e hora da última modificação do arquivo
Tamanho atual	Número de bytes no arquivo
Tamanho máximo	Número máximo de bytes no arquivo

4.3 Estrutura de diretórios de um sistema de arquivos

Para organizar o armazenamento dos arquivos os sistemas de arquivos implementam o conceito de diretórios ou pastas. Um diretório pode ser considerado como uma subdivisão lógica que permite o agrupamento de arquivos relacionados entre si. A subdivisão e agrupamento de arquivos não causa necessariamente uma organização física das informações relativas a um diretório em uma unidade de armazenamento (Fraga e Sallum, 2016).

Os diretórios presentes em um sistema de arquivos podem ser também organizados de diferentes maneiras, sendo:

- Sistemas de diretórios de nível único: o sistema de arquivos implementa um único diretório, comumente chamado de diretório raiz, na unidade de armazenamento. Desta maneira, todos os arquivos se concentram armazenados neste único diretório. A Figura 21 abaixo exemplifica um sistema de diretório único, contendo quatro arquivos (Tanenbaum e Woodhul, 2009).

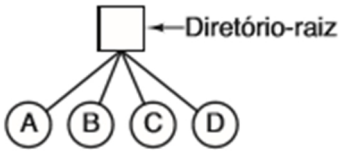

Figura 21. Sistema de diretório único (Tanenbaum e Woodhul, 2009).

- Sistema de diretórios hierárquicos: nesta abordagem, é possível, a partir do diretório raiz, criar outros subdiretórios. Desta maneira, os próprios diretórios também são organizados através de estrutura hierárquica, facilitando a recuperação de diretórios por usuários. Na atualidade, quase todos os sistemas de arquivos são organizados dessa maneira. A Figura 22 apresenta um exemplo de organização hierárquica de diretórios a partir de um diretório raiz (Tanenbaum e Woodhul, 2009).

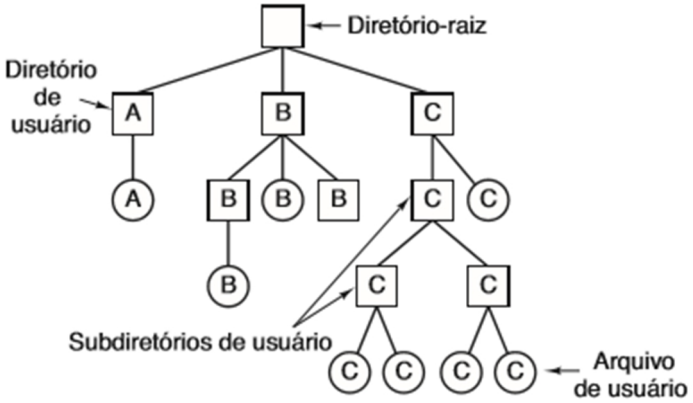

Figura 22. Sistema hierárquico de diretórios (Tanenbaum e Woodhul, 2009).

4.4 Síntese do Capítulo

Os sistemas de arquivos possuem a responsabilidade de fornecer meios para o armazenamento de dados, sendo que alguns dos sistemas organizam os dados em uma

estrutura de diretórios. Este capítulo abordou os principais aspectos dos sistemas de arquivos bem como as diversas formas de organização da estrutura de diretórios. Também foi abordado o conjunto de metadados que podem ser obtidos dos arquivos e diretórios presentes em um sistema de arquivos e que constituem o foco do presente projeto de pesquisa.

5. Arquitetura da Solução Proposta e Escolhas Tecnológicas

Neste capítulo será discutida a arquitetura do protótipo obtido como resultado desta dissertação; um visualizador hierárquico de metadados. Adicionalmente, será apresentado o respectivo modelo conceitual de dados utilizado. Por fim, a seção explora e apresenta as tecnologias, bem como suas respectivas características que justificam as escolhas realizadas para o desenvolvimento do protótipo de *software*.

5.1. Arquitetura da Aplicação

A aplicação está estruturada em uma arquitetura cliente-servidor, baseada em um navegador *web* para realizar a requisição e exibir a resposta do servidor da aplicação. O servidor da aplicação é também o responsável por manipular o sistema de arquivos e o banco de dados onde serão armazenados os metadados dos arquivos encontrados. A arquitetura cliente-servidor implementada no projeto pode ser entendida em três níveis ou módulos distintos, sendo eles:

1. Interface Gráfica.
2. Servidor de Aplicação.
3. Servidor de Banco de Dados e Sistema de Arquivos.

Em um primeiro nível encontra-se a interface gráfica da aplicação. É através da interface gráfica que o usuário da aplicação irá interagir com a mesma e visualizar todas as informações. No nível de interface gráfica o usuário poderá escolher em qual unidade de armazenamento do computador o programa irá fazer uma indexação dos metadados de diretórios e arquivos e ainda é neste nível que o usuário visualizará os resultados desta indexação.

No segundo nível há o servidor de aplicação (também denominado *middleware*). Este recebe as solicitações vindas da interface gráfica e realiza o processamento necessário, retornando para a interface os recursos solicitados. Para tal, o servidor de aplicação realiza todas as interações necessárias com o sistema de arquivos, fornecendo os dados necessários ao servidor de banco de dados.

Em um terceiro nível da aplicação encontram-se o servidor de banco de dados e o sistema de arquivos da máquina hospedeira da aplicação. Através do servidor de aplicação

o sistema de arquivos é manipulado e as informações referentes à hierarquia e metadados dos arquivos e diretórios são armazenados no servidor de banco de dados, para posterior consulta e visualização.

A Figura 23 apresenta um modelo da arquitetura cliente-servidor, em três níveis, utilizado pela aplicação.

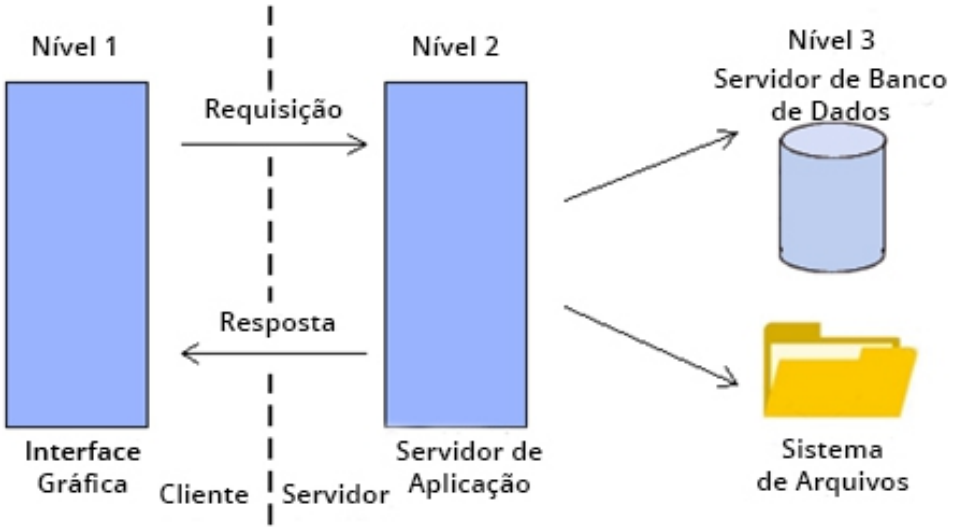

Figura 23. Diagrama da arquitetura da aplicação.

5.2 Modelo de Dados

O modelo de dados adotado no projeto corresponde a um grafo em forma de árvore enraizada, sendo este um grafo conexo (existe caminho entre quaisquer dois de seus nós) e acíclico (não possui ciclos). O nó raiz da árvore de grafo gerada é constituído pela unidade lógica do sistema de arquivos onde se realizou a busca. Cada nó subsequente representa um subdiretório ou um arquivo. Considera-se como profundidade de um nó a distância do mesmo até o nó raiz. Um conjunto de nós com a mesma profundidade é denominado de nível da árvore. A maior profundidade de um nó é a altura da árvore. Cada nível da árvore possui como nós os subdiretórios encontrados internamente nos diretórios representados pelos nós que estão um nível acima. A disposição hierárquica em forma de árvore dos diretórios e arquivos no sistema operacional onde o projeto foi concebido e testado (Microsoft Windows) e o fato desta forma de organização ser a mais comumente utilizada pelos sistemas operacionais comerciais segundo Tanenbaum e Woodhul (2009) foram determinantes na escolha do modelo de dados. A Figura 24 apresenta a estrutura utilizada como modelo de dados.

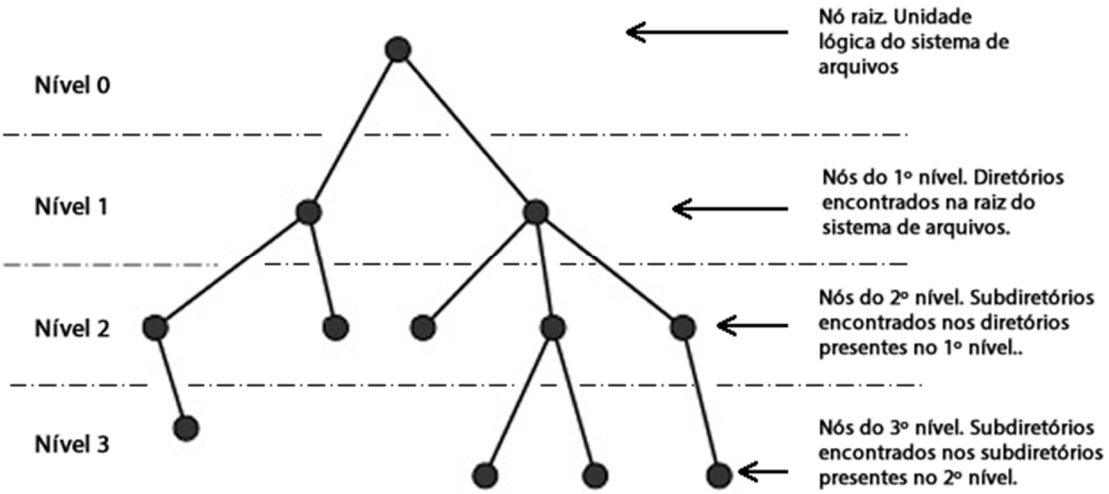

Figura 24. Modelo de dados da aplicação.

5.3 Node.JS

De acordo com (Tilkov e Vinoski, 2010), o Node.js é um interpretador de código Javascript mantido pela Node.js Foundation. O mesmo é totalmente construído com base no compilador JIT (Just In Time) Javascript denominado Chrome V8 desenvolvido pelo Google. O Node.js é voltado para a construção de aplicações *web* escaláveis e de alto desempenho, possuindo versões para diversos sistemas operacionais. A plataforma possibilita o desenvolvimento de aplicações *web* denominadas de aplicações em tempo real, termo utilizado para aplicações que mantém através do protocolo *websockets* uma conexão bi-direcional aberta entre cliente e servidor. Através do protocolo *websockets* é possível que grande quantidade de dados seja trocado entre o lado cliente e o servidor de maneira eficiente utilizando a programação orientada a eventos e operações de entrada e saída não bloqueantes; uma das principais características da plataforma. Nativamente, o Node.js também oferece funcionalidades que permitem a construção de um servidor *web* com base no protocolo HTTP (Tilkov e Vinoski, 2010).

As características tecnológicas que permitem a utilização em projetos do protocolo *websockets* e o desempenho superior a outras tecnologias *web* ajudam a justificar escolha para uso no protótipo desenvolvido. Foram consideradas inicialmente três tecnologias para o desenvolvimento da aplicação resultado desta dissertação, sendo elas: linguagem PHP com suporte do servidor Apache, linguagem PHP com suporte do servidor Nginx e o

Node.js. Apesar da linguagem PHP apresentar sintaxe mais simplificada quando comparada a sintaxe utilizada no Node.js, a mesma apresentou desvantagens consideráveis, como a realização de operações de entrada e saída bloqueantes e desvantagem de desempenho. A subseção a seguir apresenta um comparativo de desempenho entre as tecnologias acima citadas. Os resultados destes testes foram fundamentais na escolha tecnológica para o desenvolvimento da aplicação. Questões históricas e maiores detalhes da plataforma Node.js, que justificam seu desempenho, são apresentados no Apêndice A desta dissertação.

5.3.1 Performance Node.JS

Em estudos conduzidos por (Chaniotis, Kyriakou e Tselikas, 2014) foram executados testes com o objetivo de comparar o desempenho dos principais servidores de *backend web*: Apache, Nginx e Node.JS. Foram realizados testes comparativos de desempenho para operações de entrada e saída (E/S), consumo de memória, consumo de CPU e eficiência no processamento de algoritmos *hash*.

No teste de requisições em operações de entrada e saída após a realização de uma entrada, cada servidor testado respondeu à requisição com uma saída do tipo *string "hello world"* (Chaniotis, Kyriakou e Tselikas, 2014). Conforme pode ser observado na Figura 25, que mostra o gráfico comparativo entre as requisições por segundo em conexões concorrentes, o Nginx é um pouco mais de 2,5 vezes mais rápido em operações de entrada e saída quando comparado com o servidor Apache. Pode-se observar também que o Node.js é mais rápido do que ambos. Outro ponto que pode ser observado é que tanto o Apache quanto o Nginx estão usando o PHP-FPM, ficando evidente que o Apache é o fator limitante de desempenho e não a linguagem PHP.

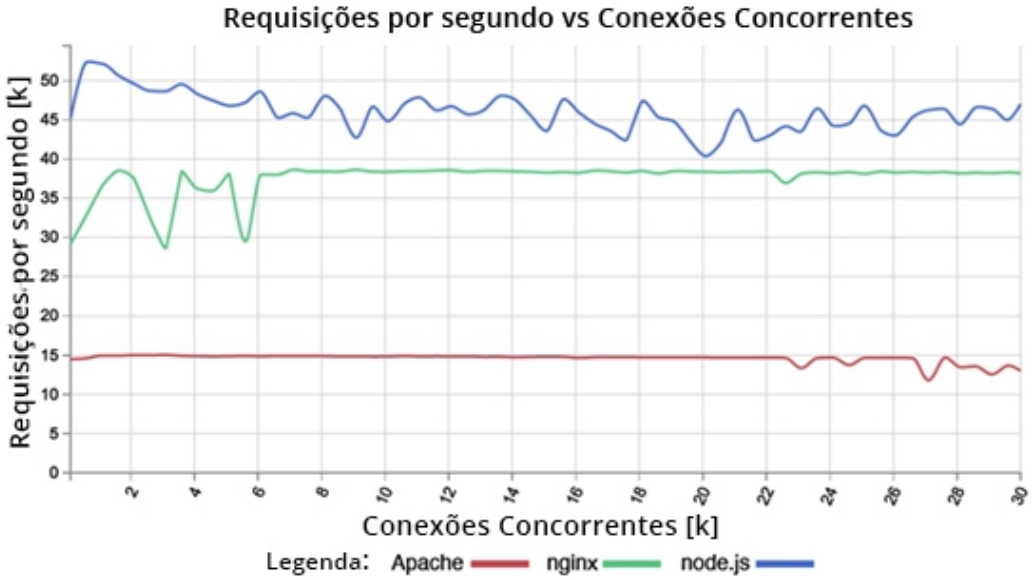

Figura 25. Comparativo de requisições por segundo em conexões concorrentes (Chaniotis, Kyriakou e Tselikas, 2014).

A Figura 26 mostra o comparativo de desempenho no uso da CPU em conexões concorrentes. Tanto o Apache quanto o Nginx mostram um desempenho similar no uso da CPU, embora o Apache consiga lidar com menos requisições por segundo. Assim, o Apache é mais ineficiente no uso de CPU durante a realização de operações de entrada e saída. De maneira comparativa ao Apache e ao Nginx, nota-se que o Node.js possui uma utilização de cerca de 1,5 vezes maior da CPU.

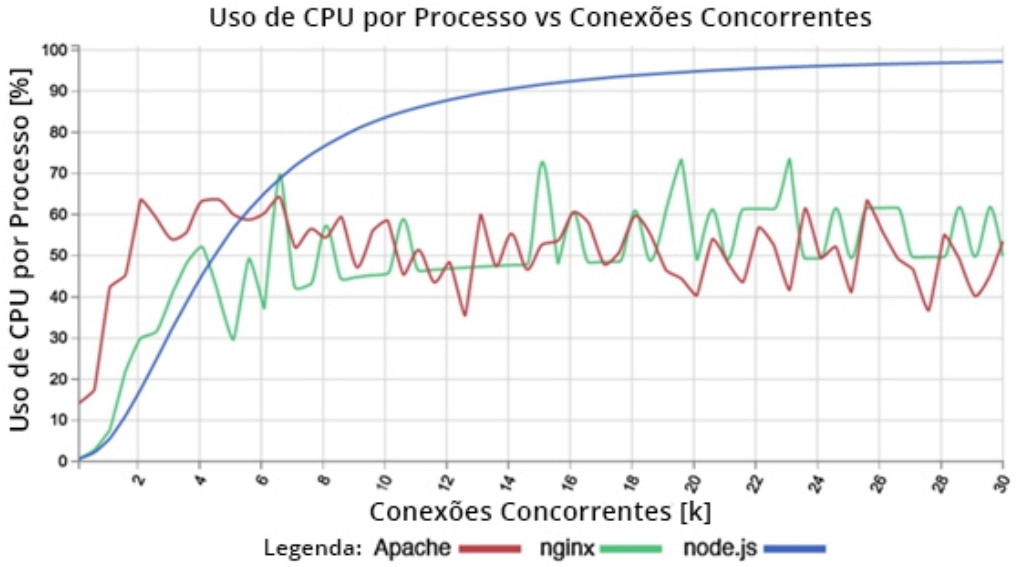

Figura 26. Comparativo de desempenho no uso da CPU em conexões concorrentes (Chaniotis, Kyriakou e Tselikas, 2014).

Pode-se observar na Figura 27 o comparativo de uso da memória primária da máquina de acordo com o número de conexões concorrentes em operações de entrada e saída. Embora os mesmos recursos estivessem disponíveis durante o teste, tanto para o Apache quanto para o Nginx, o Apache não conseguiu aproveitá-los de maneira satisfatória devido ao gargalo de operações de entrada e saída bloqueantes. Por outro lado, o Node.js mostra excelente utilização de memória para todo o intervalo de conexões concorrentes utilizado no teste.

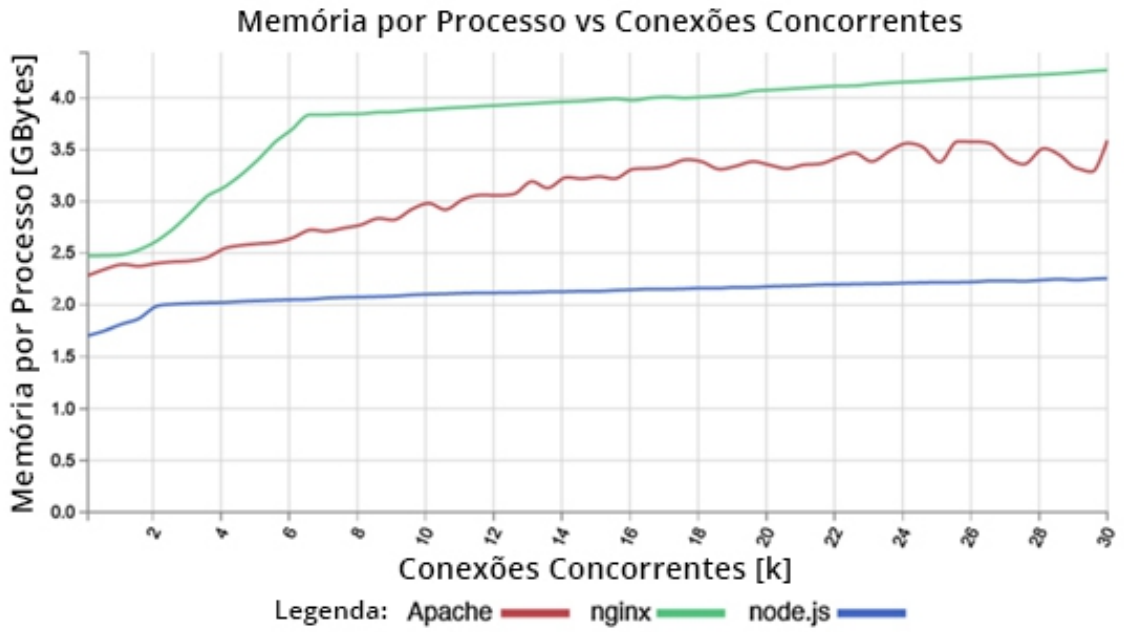

Figura 27. Comparativo do uso de memória primária em conexões concorrentes (Chaniotis, Kyriakou e Tselikas, 2014).

No comparativo de desempenho considerando o número de requisições com falha de acordo com o número de conexões concorrentes, o Nginx e o Node.js apresentaram grande estabilidade, não obtendo falhas. Em contrapartida, o Apache apresentou grande número de requisições com falha logo após o limiar de 19 mil conexões concorrentes, como observado na Figura 28.

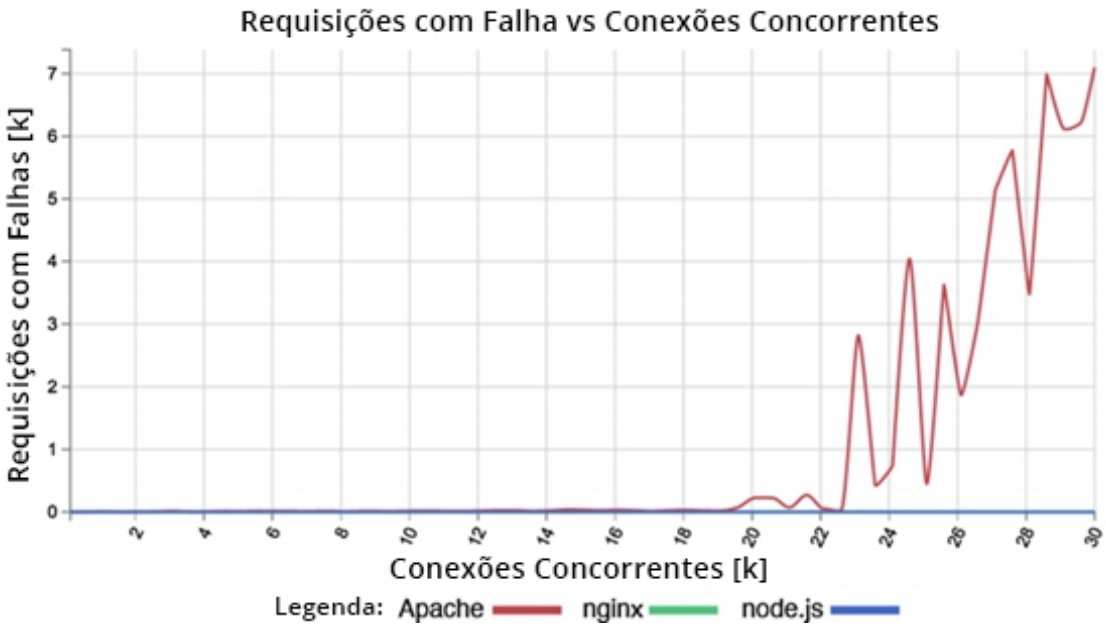

Figura 28. Comparativo do número de requisições com falha (Chaniotis, Kyriakou e Tselikas, 2014).

No teste de desempenho computacional no processamento de *hashing*, um conjunto fixo de coordenadas geográficas foi codificado em cada solicitação usando um algoritmo de *geohashing*. Como pode ser observado na Figura 29, o Nginx perde a maior parte da vantagem de desempenho demonstrada durante o teste de requisições de operações E/S. A diferença de desempenho em relação ao Apache foi minimizada, o que pode ser atribuído a ineficiências computacionais da linguagem PHP. O Node.js supera em mais de 2,5 vezes o desempenho dos servidores PHP. Nota-se também que o Node.js não apresenta perdas de desempenho significativas em relação ao teste de requisições de operações de entrada e saída, o que significa que possui maior capacidade e eficiência em lidar com o processamento de dados em relação aos servidores PHP.

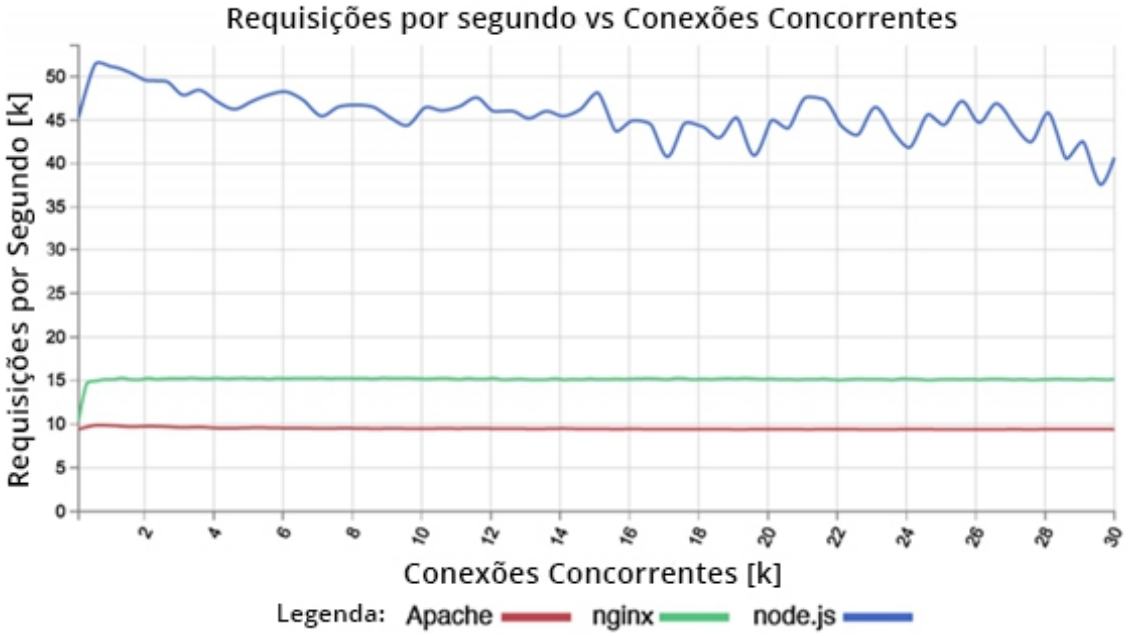

Figura 29. Comparativo desempenho computacional no processamento de *hashing* (Chaniotis, Kyriakou e Tselikas, 2014).

Observando o uso da memória primária em relação a conexões concorrentes para processamento de *hash*, conforme exposto na Figura 30, não se constatam diferenças significativas do desempenho das tecnologias examinadas, em comparação com o teste de uso de memória primária em requisições de entrada e saída.

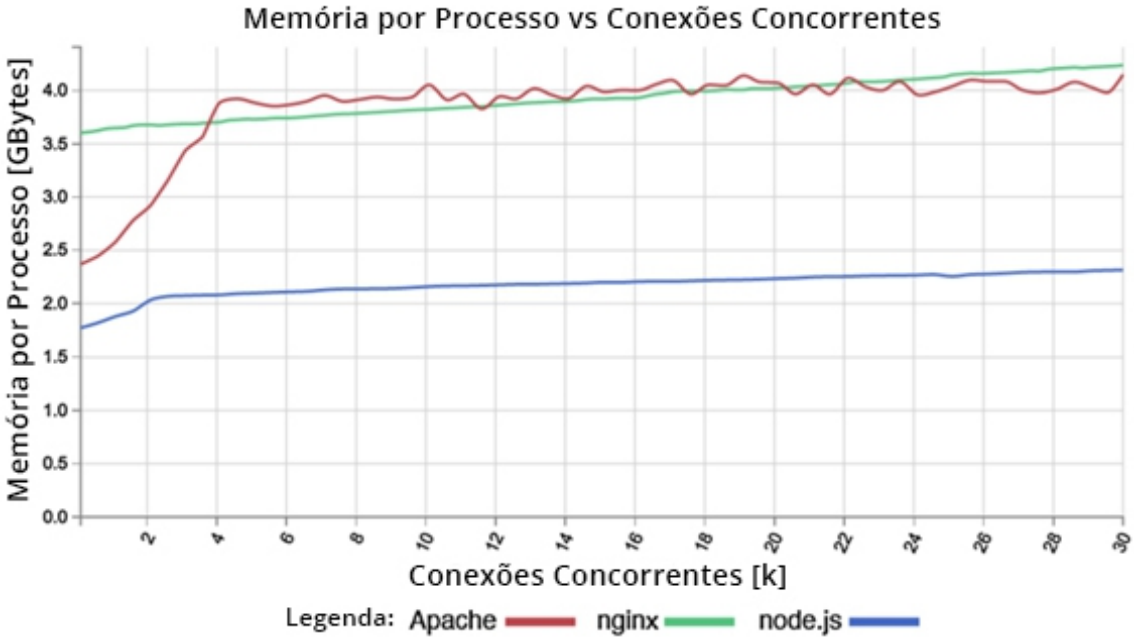

Figura 30. Comparativo do uso de memória primária no processamento de *hash* (Chaniotis, Kyriakou e Tselikas, 2014).

Com os testes de avaliação de desempenho realizados por Chaniotis, Kyriakou e Tselikas (2014), fica evidenciado que o servidor Apache é o servidor *web* com menor desempenho em todos os casos testados, sendo bastante ineficiente e exigindo muita memória, mas sem oferecer ganhos significativos de desempenho. O Nginx supera de maneira indiscutível ao Apache em todos os testes realizados. Por outro lado, o Node.js supera claramente ambos os servidores PHP no desempenho de processamento computacional e de operações de entrada e saída, sendo mais eficiente em termos de memória e utilizando todo o poder de processamento disponível (Chaniotis, Kyriakou e Tselikas, 2014).

5.4 NoSQL

Um dos grandes problemas enfrentados pela comunidade de desenvolvedores e pesquisadores da área de banco de dados tem sido a manipulação do grande volume de dados heterogêneos. Estes dados são gerados por diferentes aplicações web que possuem requisitos diferentes, como escalabilidade e alto grau de disponibilidade (Stonebraker, 2010). As redes sociais, por exemplo, geram diariamente um grande volume de dados heterogêneos provenientes de milhões de usuários do mundo todo. Diversas outras aplicações geram grandes volumes de dados, que podem ser semiestruturados, não estruturados ou ainda utilizar estruturas de dados não convencionais. Como tentativa de solucionar essa problemática, surgiram nos últimos anos diversos sistemas gerenciadores de banco de dados que não trabalham com base no modelo relacional, sendo todos estes novos sistemas gerenciadores classificados sob o termo NoSQL (Not only SQL) (Lóscio, Oliveira e Pontes, 2011).

O termo NoSQL surgiu em 1998, como denominação feita por Carol Strozzi a um sistema de banco de dados não relacional que não utilizava a linguagem SQL. Posteriormente em 2006, o artigo "BigTable: A Distributed Storage System for Structured Data" publicado pelo Google associou o termo ao gerenciamento e manipulação de grandes quantidades de dados não estruturados de maneira relacional (Oliveira, 2014).

Os Sistemas Gerenciadores de Banco de Dados (SGBDs) que utilizam o modelo relacional como base para suas operações surgiram na década de 70. Tais SGBDs tinham somente o propósito de realizar a persistência de dados de forma estruturada em tabelas, sendo necessário o uso de um esquema de dados. Um esquema de dados representa a

configuração lógica da totalidade ou de parte de uma base de dados relacional (Oliveira, 2014).

Segundo estudos realizados pela IBM (Cher, 2012) são gerados diariamente petabytes de dados advindos de aplicações web como redes sociais e serviços de streaming. Estima-se ainda que este volume de dados tenda a dobrar a cada período de 18 meses. Boa parte desses dados provém de aplicações que não são adequadas para os bancos de dados relacionais, visto que não possuem uma estrutura de dados predefinida (Muniz e Santos, 2018).

Os SGBDs denominados NoSQL não utilizam a álgebra relacional para estruturar suas operações e não possuem de forma nativa a linguagem de consulta SQL. Segundo Rockenbach. et al. (2014), os SGBDs NoSQL não possuem estrutura de relacionamentos, mas sim uma estrutura simplificada com suporte natural a replicação. São escaláveis, e graças às suas características de desempenho satisfatório com grande volume de dados, são recomendados para uso na web.

De acordo com Lóscio, Oliveira e Pontes, (2011) algumas características fundamentais dos SGBDs NoSQL são:

I) Escalabilidade horizontal (criação de novas threads no servidor ou distribuição de processos entre diversas máquinas);

II) Utilização de esquema de armazenamento de dados flexível, o que permite que dados não estruturados sejam armazenados mais facilmente;

III) Menor tempo para recuperar informações em comparação com SGBDs baseados no modelo relacional;

IV) Utilização do princípio BASE ao invés do princípio ACID;

V) Consistência de dados nem sempre presente, seguindo o teorema CAP (Consistência, Disponibilidade e Tolerância a Partições).

Os princípios ACID, BASE e o teorema CAP serão abordados de maneira mais detalhada posteriormente na Subseção 3.1.1.

Não é necessária a predefinição de um esquema de dados durante a criação de um banco de dados NoSQL, contrastando com os bancos de dados relacionais onde é

necessário definir previamente como os dados serão armazenados (com a definição de tabelas, atributos, relacionamentos, restrições de chave, dentre outros). Cada item do banco de dados NoSQL que possui uma mesma classe ou um mesmo tipo podem ter esquemas diferentes, possibilitando que dados sejam armazenados de forma não estruturada, parcialmente estruturada ou totalmente estruturada. Esta característica garante às bases NoSQL uma das principais vantagens em comparação ao modelo relacional, dado que cerca de 80% de toda informação gerada não é estruturada (Brito, 2010).

Em bancos de dados que utilizam o modelo relacional, apesar da escalabilidade vertical (aumentar o carregamento em um servidor através da melhoria de itens de hardware) ser mais comumente utilizada, também é possível a implementação da escalabilidade horizontal (utilizar mais servidores de banco de dados e particionar a estrutura de dados de acordo com critérios estabelecidos). Porém, o uso elevado de threads ou diversos processos se conectando simultaneamente em uma mesma base de dados gera uma alta concorrência no modelo relacional, aumentando o tempo de acesso às tabelas de dados. Esta é uma grande vantagem dos bancos de dados NoSQL, dado que a ausência ou a flexibilidade dos esquemas permite a implementação da escalabilidade horizontal sem que o tempo de acesso à base de dados seja afetado pelo aumento da concorrência. Uma das técnicas utilizadas para implementação da escalabilidade horizontal é o Sharding. No Sharding, os dados são divididos em múltiplas tabelas armazenadas em diversos nós de uma rede, evitando a criação de uma cadeia de relacionamentos e consequentemente diminuindo o tempo de acesso e manipulação dos mesmos (Anderson, Lehnardt e Slater, 2009).

Há de se considerar ainda os bancos de dados relacionais que utilizam processamento paralelo como forma de melhorar seu desempenho. Através da utilização de vários processadores, os SGBDs paralelos são capazes de executar um grande número de transações simultâneas. Porém deve-se atentar a alguns detalhes para que os SGBDs relacionais paralelos alcancem seu máximo desempenho, o que aumenta sua dificuldade de implementação, como, por exemplo, o correto particionamento de dados dentro os diversos discos que compõem o sistema de armazenamento paralelo e um balanceamento de carga eficaz (Fernandes, 2015).

As características flexíveis dos modelos de dados NoSQL mencionadas anteriormente explicam também a possibilidade na redução de custos de infraestrutura quando comparadas com SGBDs relacionais. A alta escalabilidade horizontal dos bancos de dados NoSQL possibilita a utilização de diversos servidores com baixo custo, com menor capacidade de processamento (Brito, 2010).

5.4.1. Princípios ACID e BASE

Os sistemas gerenciadores de banco de dados NoSQL garantem em suas implementações o princípio BASE (Basically Available, Soft state, Eventual consistency – traduzido para Basicamente Disponível, Estado Leve, Eventualmente Consistente). Enquanto os sistemas gerenciadores de banco de dados que utilizam o modelo relacional garantem em suas implementações o princípio ACID (Atomicity, Consistency, Isolation, Durability – traduzido para: Atomicidade, Consistência, Isolamento e Durabilidade) (Nayak, Poriya e Poojary, 2013).

Os bancos de dados que utilizam o princípio BASE priorizam a disponibilidade e o desempenho, sendo responsabilidade do desenvolvedor antever possíveis problemas de consistência. Tal característica diferencia-se do princípio ACID, utilizado pelos bancos de dados relacionais, que priorizam consistência e isolamento, forçando ao final de cada transação a persistência dos dados para garantir a consistência. É importante ainda ressaltar que os bancos de dados que utilizam o princípio ACID são implementados seguindo o paradigma de que um único servidor concentra todos os dados da aplicação e gerencia o acesso simultâneo de vários usuários. No caso de SGBDs relacionais com suporte a cluster de servidores, há a necessidade de interação entre os servidores para garantir que os dados permaneçam consistentes quando acessados de forma simultânea por diversos usuários. De forma oposta, os bancos de dados que utilizam o princípio BASE trabalham com os dados de formas distribuída, não centralizando o acesso dos usuários (Pritchett, 2008). A Tabela 2 mostra as principais características dos princípios ACID e BASE.

Tabela 2. Características dos princípios ACID e BASE.

ACID	Atomicidade	Em uma transação envolvendo duas ou mais operações, ou a transação será executada totalmente ou não será executada. Isto é, após a finalização de uma transação, a base de dados não deve refletir resultados parciais da transação.
	Consistência	Uma transação nunca causa inconsistência nos dados. Em caso de falha, retorna todos os dados ao estado anterior ao início da transação.
	Isolamento	As transações não estão cientes das demais transações que ocorrem de forma simultânea.
	Durabilidade	Transações validadas são persistidas, de forma que em caso de falha ou reinício do sistema, os dados estarão disponíveis em seu estado correto.
BASE	Basicamente Disponível	Esta propriedade garante a disponibilidade dos dados no que diz respeito ao Teorema CAP. Sempre haverá resposta a uma solicitação. Porém essa

		resposta pode ser de falha na obtenção dos dados solicitados ou ainda os dados podem estar em um estado inconsistente ou em mudança.
	Estado Leve	O estado do sistema pode mudar com o tempo, mesmo durante os períodos sem entrada, podendo ocorrer mudanças devido à propriedade da "consistência eventual".
	Eventualmente Consistente	O sistema se tornará consistente quando deixar de receber entrada. Os dados serão replicados em algum momento, porém o sistema continuará recebendo dados e não verificará a consistência de todas as transações antes de passar para a próxima.

5.4.2. Teorema CAP

O teorema CAP (Consistency, Availability, Partition Tolerance), também denominado de conjectura de Brewer, foi desenvolvido por Eric Brewer e descreve o comportamento de uma coleção de nós interconectados que compartilham dados, sendo uma das principais motivações técnicas para o desenvolvimento de aplicações NoSQL (Fox e Brewer, 1999). De acordo com (Marungo, 2018), o Teorema CAP é baseado em três propriedades:

- Consistency (Consistência): significa que o sistema garante a leitura do dado mais atualizado possível. Leituras simultâneas de um dado no mesmo local (nó) em que este dado foi escrito ou de um local (nó) diferente retornam o mesmo dado. O comportamento de consistência é que garante que a aplicação não realize a leitura de dados obsoletos ou conflitantes, sendo que apenas dados atualizados serão lidos.

- Availability (Disponibilidade): O sistema é capaz de realizar a leitura/escrita de dados em um nó mesmo que haja outro nó do sistema com falha ou offline. Com isso, garante que nenhuma das requisições da aplicação cliente irá retornar um erro ou irá deixar o usuário aguardando por tempo indefinido, pois o sistema gerenciador de banco de dados sempre encontrará um nó disponível para realizar as transações solicitadas pela aplicação cliente.

- Partition Tolerance (Tolerância a Partições): A propriedade de Tolerância a Partições trata do número e do desempenho de componentes independentes do sistema interligados através da rede. Esta propriedade garante que o sistema continue operando mesmo no caso de uma falha na rede gerar uma partição, garantindo que ao menos a leitura dos dados continue ocorrendo normalmente.

O Teorema CAP estabelece ainda que, das três propriedades, somente duas podem coexistir simultaneamente. Desta maneira, há a possibilidade de três combinações distintas das propriedades, conforme a Figura 31 demonstra.

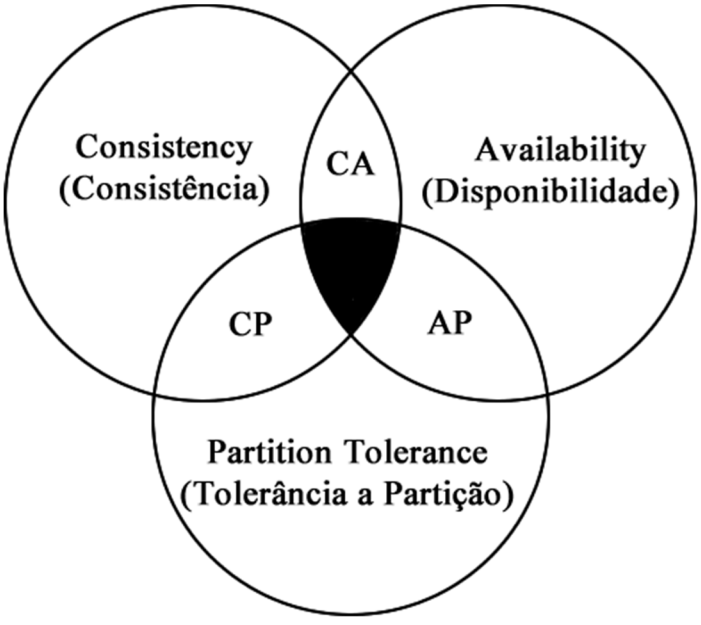

Figura 31. Propriedades do teorema CAP.

Realizando uma análise das combinações das propriedades do teorema CAP, tem-se:

- CA (Consistência e Disponibilidade): Nesta abordagem, mantêm-se a consistência e a disponibilidade do sistema, abrindo mão da tolerância a partições na rede. Tal implementação pode representar um risco para a aplicação cliente caso na mesma não esteja prevista a possibilidade de falha em algum dos servidores que integram um sistema de armazenamento distribuído. A falha que deve estar prevista é a perda de comunicação entre dois ou mais servidores, criando assim uma partição na rede. Caso essa previsão não exista na aplicação, a mesma fica indisponível até que a conectividade dos servidores que compõem o sistema de armazenamento distribuído se reestabeleça. Esta combinação também é conhecida como "abordagem otimista", pois as escritas nunca falham.

- AP (Disponibilidade e Tolerância a Partições): Escolher a propriedade da disponibilidade em detrimento à propriedade de consistência, permite que a aplicação realize uma leitura imediata dos dados, porém, os dados podem ser obsoletos ou imprecisos. Sistemas que abordam esta combinação requerem geralmente alta disponibilidade e escalabilidade, e para atenuar a falta da propriedade de consistência aplicam uma técnica denominada Eventual-Consistency. Sistemas com este tipo de abordagem permitem um número maior de escritas realizando sincronização posterior.

- CP (Consistência e Tolerância a Partições): A abordagem CP garante a consistência dos dados e permite que o banco de dados possua partições (nós) distribuídos pela rede, porém abre mão da garantia de disponibilidade do sistema. Tal combinação é conhecida popularmente como abordagem pessimista, pois escritas podem ser negadas. A combinação CP é comumente utilizada em sistemas que dependem de um alto nível de consistência, como, por exemplo, sistemas de aplicação bancária. As plataformas que utilizam a abordagem CP implementam um protocolo de consenso denominado Paxos, que possui a finalidade de aumentar a consistência do sistema, para evitar ao máximo a negação de escrita de dados. O protocolo Paxos implementa um algoritmo de consenso distribuído que opera sobre um conjunto de réplicas de dados com o objetivo de obter um consenso entre os servidores que gerenciam tais réplicas, para atualiza-los com um único valor em comum (Coulouris et al, 2013). A grande maioria dos sistemas NoSQL utilizam

a abordagem CP. Pode-se citar como exemplos os sistemas MongoDB (MongoDB, 2019) e Hadoop (Hadoop, 2019).

5.4.3. Tipos de sistemas NoSQL

Os sistemas NoSQL podem, inicialmente, ser divididos em dois grupos: orientados a agregados e não orientados a agregados. Sistemas orientados a agregados trabalham com conjuntos de objetos relacionados que se desejam tratar como unidade (Chandra, 2015). Pode-se classificar os sistemas NoSQL orientados a agregados em três diferentes categorias:

- Sistemas Chave/Valor: O modelo chave/valor permite a persistência de dados de forma schemaless (dados que não possuem um esquema de armazenamento predefinido). Os sistemas baseados em chave/valor realizam a persistência de dados em uma estrutura semelhante a uma tabela hash. O dado é armazenado e relacionado a um valor de índice em forma de um tipo primitivo. Devido ao fato de cada dado ser mapeado por um índice de tipo primitivo, a inserção e recuperação de dados tornam-se tarefas de baixo custo computacional. A maioria dos sistemas seguem dois principais métodos: put (chave, valor) – método para inserir um registro e get (chave) – método para recuperar um registro. Embora existam vantagens na velocidade e simplicidade no armazenamento e recuperação de dados, esse modelo de NoSQL gera uma grande quantidade de dados redundantes. Alguns sistemas que se utilizam do modelo chave/valor são: DynamoDb (Amazon, 2019), Couchbase (CouchDB, 2019), Riak (Riak, 2019), Azure Table Storage (Azure, 2019), Redis (Redis, 2019), Tokyo Cabinet (Fallabs, 2019), e Berkeley DB (Oracle, 2019).
- Baseado em Colunas: Os sistemas de banco de dados NoSQL baseado em colunas, faz uso de tabelas para representar entidades, agrupando os dados gravados na unidade de armazenamento através de colunas. Apesar de possuírem conceitos parecidos, os bancos de dados orientados a colunas atuam de maneira diferente dos bancos relacionais, onde o paradigma utilizado é de orientação a registros ou tuplas. No modelo orientado a colunas, os dados são indexados por uma tripla (linha, coluna e timestamp – número de segundos decorridos desde a meia noite de 01/01/1970, no fuso horário UTC sem considerar os segundos bissextos, até o

exato horário em que o dado foi gravado), onde linhas e colunas são identificadas por chaves e pelo timestamp, que permite diferenciar múltiplas versões de um mesmo dado. Este agrupamento lógico possibilita a redução do tempo de leitura e escrita em disco. Também facilita operações de busca e sumarização visto que o valor de cada coluna é armazenado de maneira sequencial. A capacidade de compactar informação em disco também é otimizada visto que os dados em uma mesma coluna são do mesmo tipo. Alguns exemplos de sistemas de banco de dados baseados em colunas são: Apache Cassandra (Cassandra, 2019) e o Apache HBase (HBase, 2019).

- Baseados em Documentos: os sistemas de banco de dados baseados em documentos são semelhantes aos sistemas de banco de dados baseados em chave/valor, porém, em vez de trabalhar somente com tipos primitivos, os sistemas baseados em documentos salvam qualquer formato descritivo de arquivos. Alguns exemplos são: arquivos de extensão xml, json, json-bloob, dentre outros. De maneira geral, os sistemas NoSQL orientados a documento não possuem esquema definido, ou seja, os documentos persistidos não precisam possuir estrutura em comum. Exemplos de sistemas NoSQL que trabalham no modelo baseados em documentos são o Apache CouchDB (CouchDB, 2019) e o MongoDB (MongoDB, 2019).
- Baseados em Grafos: o modelo orientado a grafos utiliza, para a persistência dos dados, conceitos provenientes da teoria dos grafos. Um grafo pode ser descrito como um conjunto de nós (que podem possuir seus próprios atributos) e arestas interligados entre si. Em um sistema de bancos de dados orientado a grafos, uma entidade é representada por um nó do grafo, que pode possuir também arestas, que representam relacionamentos, interligando os nós. Podem ser citados como exemplos de banco de dados orientado a grafos: Neo4J (Neo4J, 2019) e o GraphDB (GraphDB, 2019).

5.4 NEO4J

O Neo4J é o sistema de gerenciamento de banco de dados de grafos escolhido para o desenvolvimento do protótipo proposto neste estudo. Essa escolha baseia-se no desempenho para realização de consultas alcançado pelos bancos de dados orientados a

grafos em comparação com as demais arquiteturas de banco de dados existentes, sobretudo a arquitetura relacional.

Bancos de dados orientados a grafos são um tipo de bancos de dados NoSQL voltado para aplicações que gerenciam grandes quantidades de dados altamente conectados e com relacionamentos dinâmicos. Essas características são desejáveis no desenvolvimento de aplicações como redes sociais, sistemas de mineração de dados, dentro outros tipos de aplicações que exijam a manipulação de dados complexos (Sadalage e Fowler, 2012). A Figura 32 apresenta um exemplo de dados estruturados como grafo.

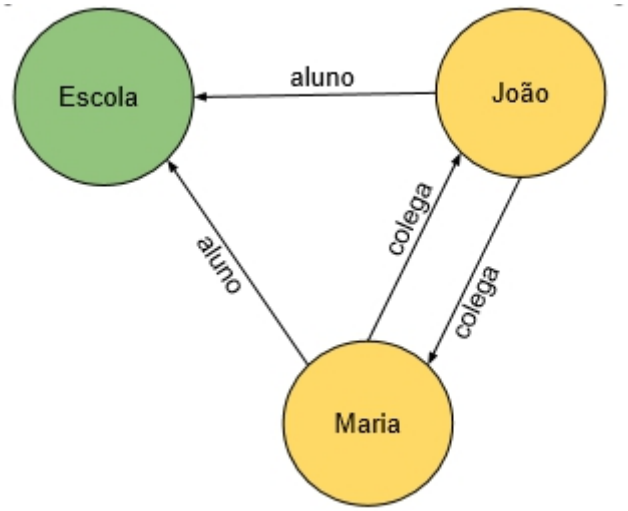

Figura 32. Dados estruturados como grafo.

A eficiência no desempenho para a realização de consultas complexas nos bancos de dados de grafos é fortemente influenciada pela maneira como os dados são modelados nesta arquitetura. Diferente do modelo relacional, o qual é baseado em um esquema definido, o modelo de banco de dados de grafos possui formato livre. Os nós que possuem um mesmo rótulo ou propriedades podem ter estruturas diferentes. No modelo relacional, como existe um esquema definido, os relacionamentos entre as tabelas de dados são realizados através de atributos de chave estrangeira, o que por vezes exige a criação de tabelas intermediárias. No modelo de grafos, os relacionamentos são representados por arestas, ocorrendo a interligação entre os nós (dados) na forma de relacionamentos (Sadalage e Fowler, 2012). Em um estudo comparativo de desempenho realizado por (Homrich e Mergen, 2018) entre o sistema gerenciador de banco de dados relacional MySQL e o sistema Neo4J, o tempo para realização de consultas com cruzamento de dados foi consideravelmente menor no banco de dados orientado a grafos. A Tabela 3

apresenta os resultados comparativos entre o MySQL e o Neo4J. Os testes de carga produzidos por Homrich e Mergen foram realizados através de um *framework* programado em linguagem Python, versão 3.6.4. Neste, os testes de carga de dados foram realizados apenas uma vez cada, com apenas um SGBD ativo por vez. Para a realização do estudo, os pesquisadores utilizaram uma máquina com processador Intel(R) Core(TM) i5-3230M, 6 GB de memória RAM DDR3 e sistema operacional Linux Mint 18.3 64 bits. Como dados para os SGBD's, foi utilizado o grafo da rede rodoviária do estado da Califórnia, obtidos do projeto *Stanford Network Analysis Project* (SNAP).

Tabela 3. Comparativo entre MySQL e Neo4j (Homrich e Mergen, 2018).

	Consulta de nós em até três níveis	Consulta de nós a três níveis
MySQL MyISAM	11,540125 s	11,522095 s
MySQL InnoDB	0,001099 s	0,000870 s
Neo4j	0,007750 s	0,006132 s

O Neo4J foi criado pela empresa Neo Technology e teve sua primeira versão (1.0) lançada no ano de 2010. Atualmente o sistema encontra-se na versão 3.5.1 lançado em dezembro de 2018. O Neo4J é distribuído em duas versões distintas: *Community* e *Enterprise.* A versão *Community* possui licença GPL (*General Puyblic License*), sendo esse o segundo fator determinante na escolha do Neo4J como sistema gerenciador de banco de dados do protótipo desenvolvido (Neo4J, 2019).

O Neo4j é considerado robusto, escalável, de alto desempenho e apresenta transações que utilizam os princípios do paradigma ACID. Dentre outras vantagens que o Neo4J apresenta, podem-se citar: alta disponibilidade, alta velocidade de consulta aos grafos, linguagem de consulta aos grafos totalmente declarativas, inúmeras ferramentas de integração e grande capacidade de armazenamento de nós e arestas (Neo4J, 2019). A Tabela 4 apresenta o espaço destinado ao endereçamento dos nós, relacionamentos, propriedades e tipos de relacionamentos no sistema Neo4J. Destaca-se que em uma única instância de servidor é possível armazenar e manipular grafos contendo bilhões de nós e relações.

Tabela 4. Espaços para endereçamento no Neo4j (Homrich e Mergen, 2018).

Propriedades	Espaço para Endereçamento
Nós	2^{35}:aproximadamente 34 bilhões
Relacionamentos	2^{35}:aproximadamente 34 bilhões
Propriedades	2^{36} a 2^{38} dependendo do tipo da propriedade. Mínimo para 68 bilhões e máximo para 274 bilhões.
Tipos de Relacionamento *RelationshioTypes*	2^{15}:aproximadamente 32.000

O Neo4J utiliza a linguagem *Cypher*, desenvolvida exclusivamente para o uso com o mesmo, sendo esta considerada uma linguagem de consulta com sintaxe próxima da linguagem humana. O *Cypher* possibilita ao desenvolvedor manipular grafos de maneira bastante intuitiva, trabalhando de forma específica com nós e relacionamentos (Holzschuher e Peinl, 2013). Possui ainda uma sintaxe intuitiva para representar os padrões utilizados pelos comandos da linguagem, possibilitando a criação de uma série de consultas complexas, utilizando recursos tais como: operações, filtros, agregações, classificações e paginações, sendo este o principal motivo de adoção da linguagem *Cypher* no projeto. O Apêndice B desta dissertação apresenta de maneira detalhada as características da linguagem *Cypher,* bem como seus comandos e as vantagens encontradas em sua utilização que foram determinantes para a sua adoção nesta dissertação.

5.5 Síntese do Capítulo

Após a realização de pesquisas e levantamento de hipóteses, decidiu-se que a aplicação seria estruturada em uma arquitetura cliente-servidor, baseado em um navegador *web* para realizar a requisição e exibir a resposta do servidor da aplicação. Tal decisão foi tomada pensando no ponto de vista do usuário com pouco conhecimento específico da área de ciência da computação, para o qual a execução do navegador *web* é algo rotineiro, o que nem sempre acontece com a instalação e execução de um software, dado inclusive que cada sistema operacional possui suas particularidades para tal. Após tal decisão, dentre as tecnologias possíveis para o desenvolvimento optou-se pelo Node.JS devido a suas características arquiteturais que vão ao encontro ao estabelecido na arquitetura do

projeto e desempenho quando comparado as demais tecnologias de desenvolvimento para *web*. O sistema gerenciador de banco de dados Neo4J foi o escolhido para gerenciar o banco de dados da aplicação devido a suas características de armazenamento em modelo de grafos (estrutura similar a de armazenamento do sistemas de arquivos dos principais sistemas operacionais) e desempenho para armazenamento e consulta quando comparados com outros sistemas gerenciadores de banco de dados do tipo *NoSQL*.

6. Solução Proposta - visualizador hierárquico de metadados

O presente capítulo apresenta a aplicação, obtida como resultado desta dissertação, denominada “visualizador hierárquico de metadados”. Ao longo da seção será explorada sua interface, bem como seu funcionamento e utilização. O código fonte da solução proposta pode ser consultado no Apêndice C desta dissertação.

6.2 Apresentação detalhada da solução desenvolvida

Após a aplicação ser iniciada, a mesma é executada em um servidor local e pode ser visualizada por meio de qualquer navegador de internet através da URL http://localhost:3000. A página inicial apresenta um formulário onde é permitido ao usuário escolher em qual disco do sistema ele deseja realizar a busca e a profundidade (quantidade de níveis da árvore hierárquica do sistema de arquivos) que a aplicação deve percorrer. Também há a opção do usuário ir direto para a visualização dos diretórios e arquivos encontrados em uma busca anterior.

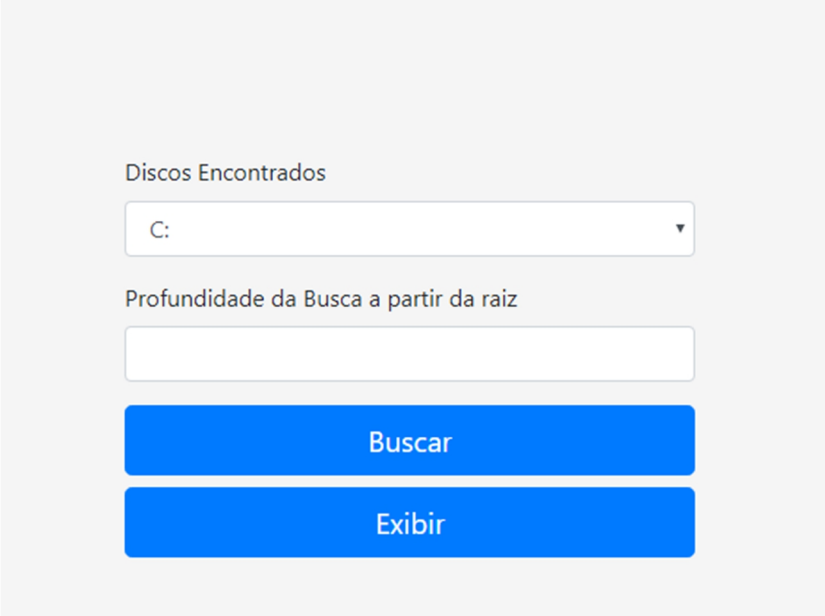

Figura 33. *Layout* da tela inicial do protótipo.

Caso o usuário da aplicação escolha a opção buscar, todos os diretórios e arquivos do disco selecionado serão percorridos até atingir a profundidade especificada. Caso o campo profundidade tenha ficado sem preenchimento, então todos os arquivos e diretórios do disco serão percorridos. Ao final da execução, o fluxo de dados redireciona o navegador

para a página de visualização dos dados. Caso o usuário escolha exibir, então um novo grupo de opções é apresentando, permitindo a escolha das cores dos nós para determinadas faixas de tamanho dos diretórios. Os tamanhos de diretórios disponíveis foram determinados apenas para teste da solução e podem ser facilmente personalizados via código fonte. A Figura 34 apresenta a tela de opções de visualização. Após a escolha das opções, ao clicar no botão exibir, será carregada a página de visualização dos dados. Caso não tenha sido feita nenhuma busca prévia, a tela de visualização se mostrará vazia.

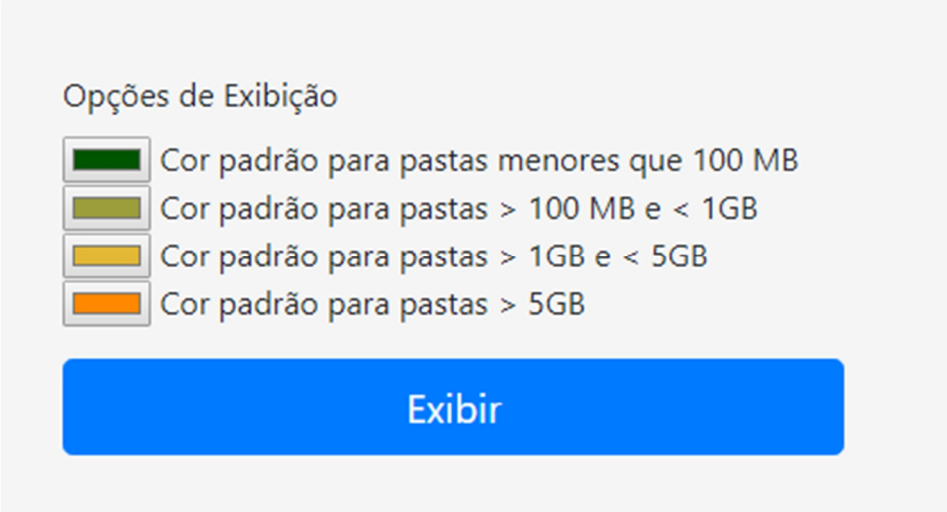

Figura 34. Opções de cores.

Ao ser carregada a página de visualização, a mesma exibe na forma de árvore hierárquica o disco onde foi realizada a busca. O disco será o nó raiz do grafo, bem como os diretórios e subdiretórios contidos no disco apresentados de maneira hierárquica como subníveis do grafo. A exibição é feita na forma de árvore hierárquica, porém a distribuição visual dos nós a partir da raiz é feita automaticamente de modo a melhor aproveitar o espaço do navegador, devido à grande quantidade de diretórios e arquivos. Da raiz partem as arestas que se ligam aos diretórios contidos na raiz. Destes, partem as arestas que se ligam aos subdiretórios e arquivos. A visualização da árvore hierárquica pode ser controlada com o *mouse*, sendo possível movimentar a árvore com o ponteiro do *mouse* e aproximar/afastar a visualização. A Figura 35 apresenta de maneira geral o *layout* da página de visualização.

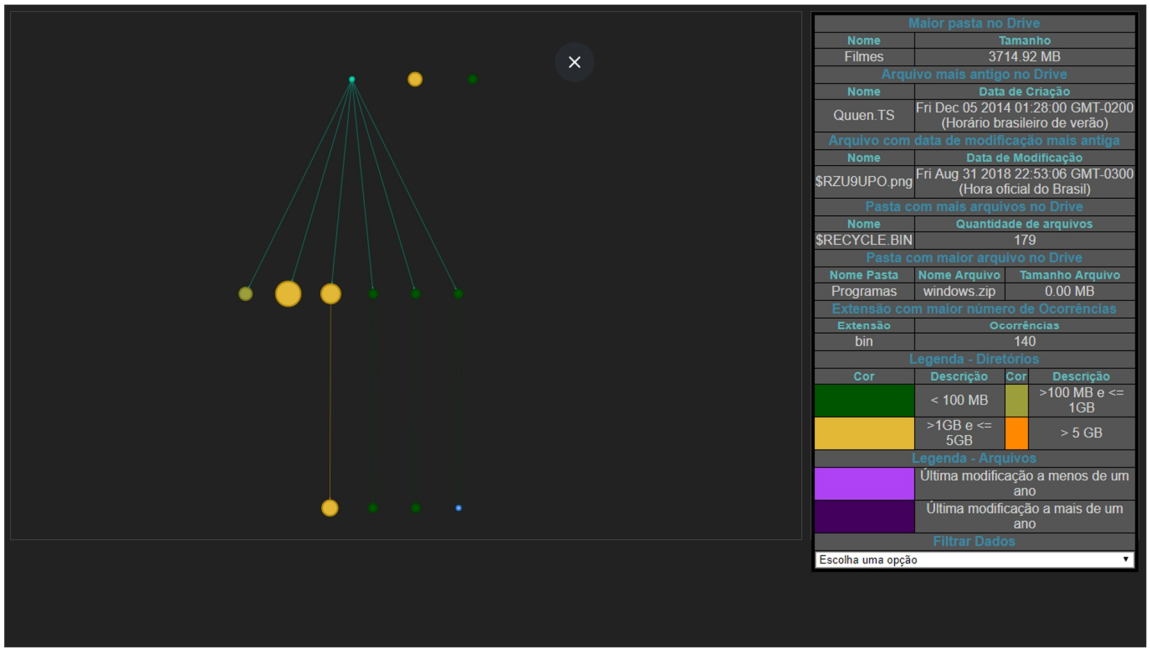

Figura 35. *Layout* da página de visualização.

A página de exibição dos arquivos e diretórios apresenta na cor azul claro o disco onde a busca foi realizada, sendo este o nó raiz do grafo. A Figura 36 apresenta o detalhe do nó raiz da árvore hierárquica, que representa o disco onde a busca foi realizada.

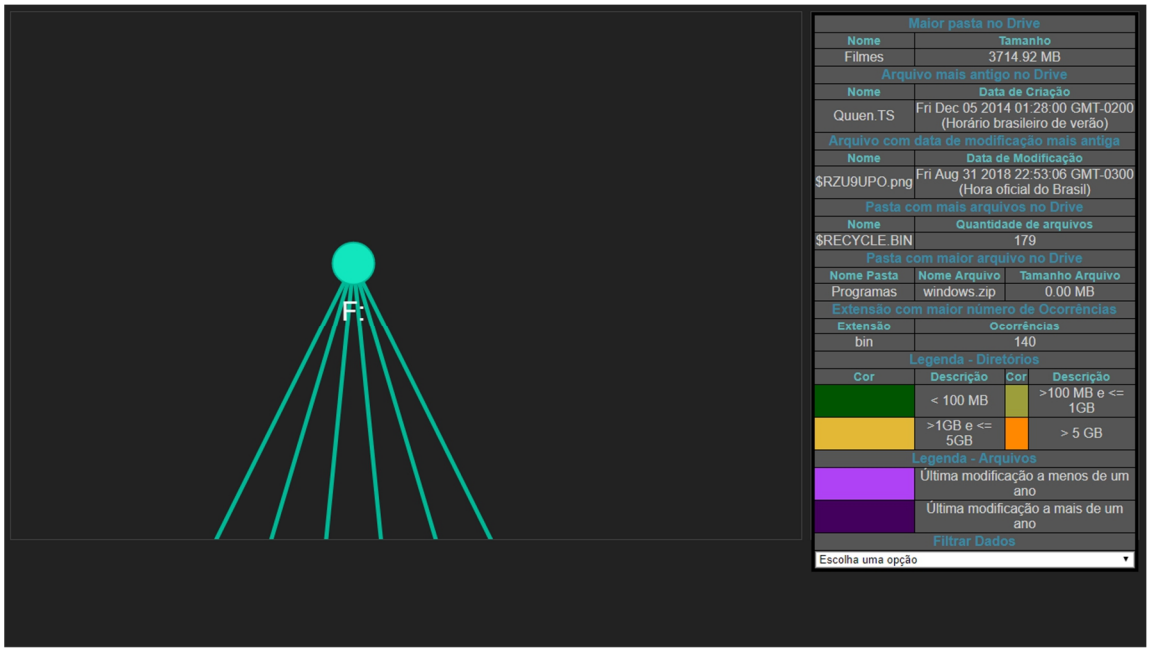

Figura 36. Nó raiz do grafo.

Diretórios e subdiretórios são apresentados como nós de diferentes tamanhos, sendo que esta diferença representa a diferença de tamanho em disco de cada diretório e

subdiretório, sendo a cor do nó também indicativo de seu tamanho. Diretórios exibidos na cor verde possuem tamanho em disco menor do que 100 MB. Diretórios apresentados na cor bege apresentam tamanho em disco entre 100 MB e 1 GB. Diretórios que são exibidos em cor amarelo claro apresentam tamanho em disco maior que 1 GB e menor ou igual a 5 GB. Por fim, os diretórios exibidos na cor laranja apresentam tamanho em disco maior do que 5 GB. A legenda apresentada no quadro exibido a direita da página de visualização detalha o esquema das cores utilizado para os diretórios e subdiretórios juntamente ao respectivo significado de cada uma. Logo abaixo de cada nó é apresentado o nome do diretório. A Figura 37 exibe em detalhes a diferença de tamanho e cores dos nós que representam diretórios e subdiretórios.

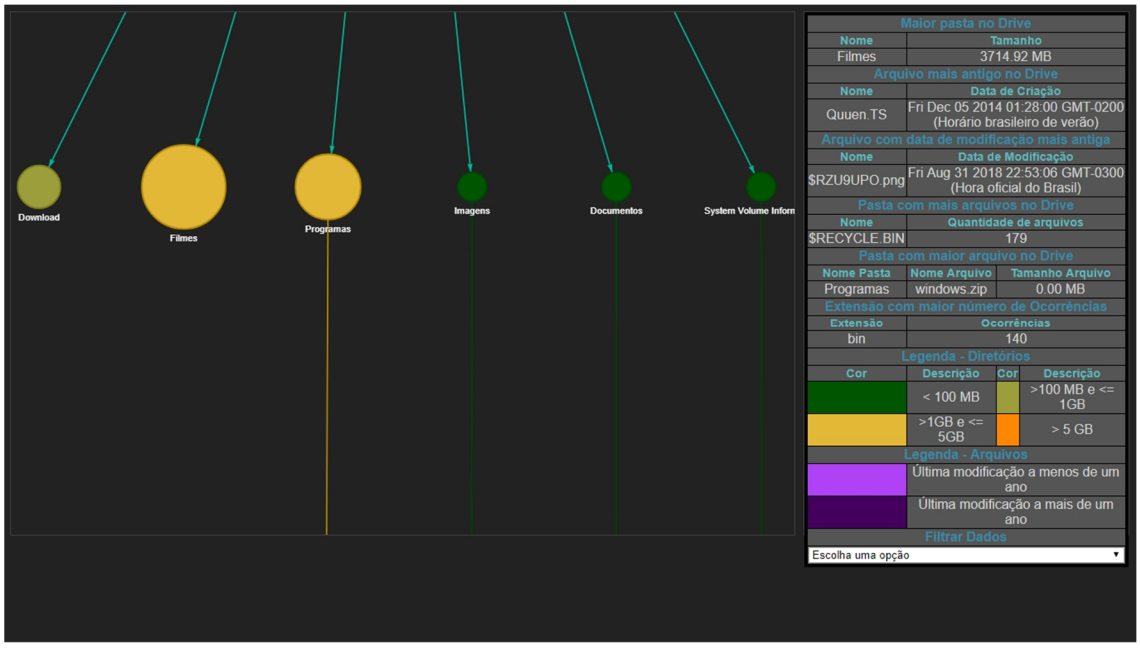

Figura 37. Representação de diretórios e subdiretórios.

Quando o ponteiro do *mouse* é posicionado sobre um dos nós que representa um subdiretório ou diretório é exibida uma janela *pop-up* contendo o nome do diretório e o exato tamanho que o mesmo ocupa em disco. A Figura 38 mostra tal recurso.

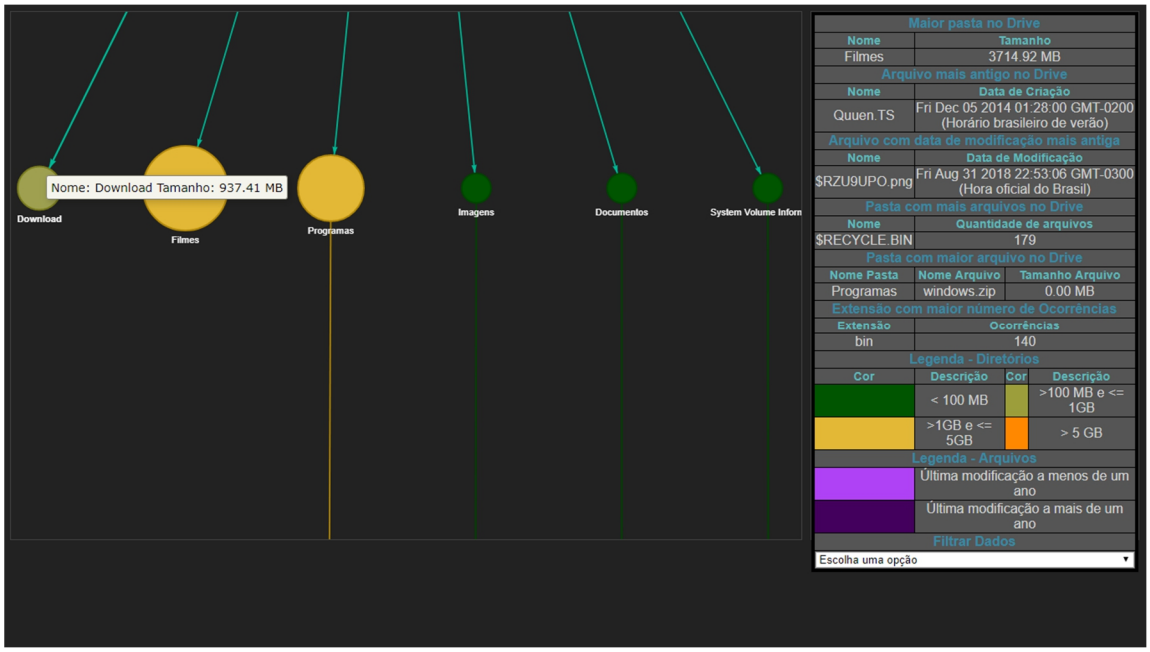

Figura 38. Janela com detalhes do nó.

No lado direito da tela de visualização é exibido um quadro contendo informações importantes sobre o conteúdo do disco representado visualmente pela árvore hierárquica. São apresentadas as seguintes informações: nome e tamanho do maior diretório contido no disco; nome e data completa de criação do arquivo mais antigo presente em todo o disco; nome e data completa de modificação do arquivo que há mais tempo não é modificado no disco; nome e número de arquivos contidos no diretório com maior quantidade de arquivos; nome do diretório que contém o maior arquivo do disco, bem como o nome do arquivo e respectivo tamanho; extensão com maior número de ocorrências bem como o número de arquivos que possuem a extensão no disco; logo abaixo são apresentadas as cores utilizadas na representação de diretórios e subdiretórios junto de seus respectivos significados. Também são apresentadas as cores utilizadas na representação de nós de arquivos com seus respectivos significados. Por fim, apresenta-se uma caixa de opções onde a visualização pode ser alternada para exibir além de diretórios e subdiretórios os arquivos neles contidos. A Figura 39 demonstra o quadro de informações, legenda e opção de alternativa de visualização.

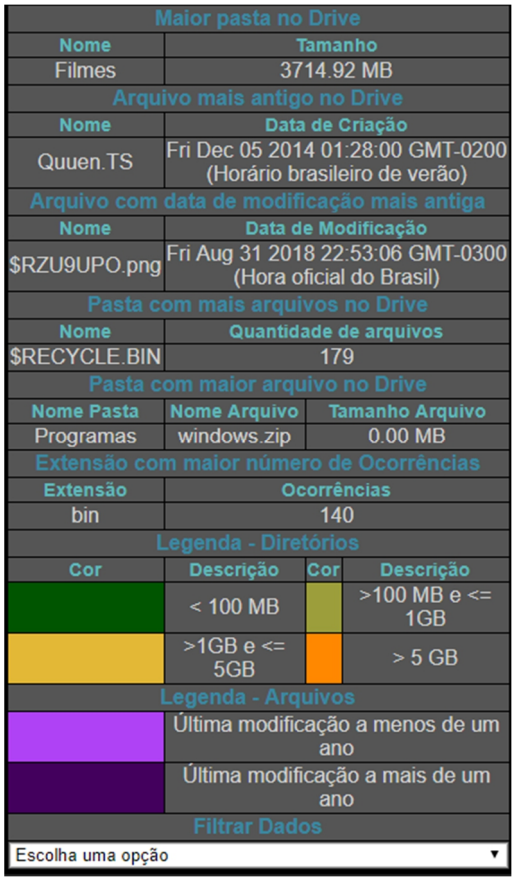

Figura 39. Quadro de informações e legendas.

Ao clicar na opção de filtrar visualização dos dados, o usuário pode escolher entre visualizar somente diretórios e subdiretórios ou visualizar os mesmos com os respectivos arquivos que estão neles contidos. A Figura 40 mostra as opções disponíveis na janela de filtro de visualização.

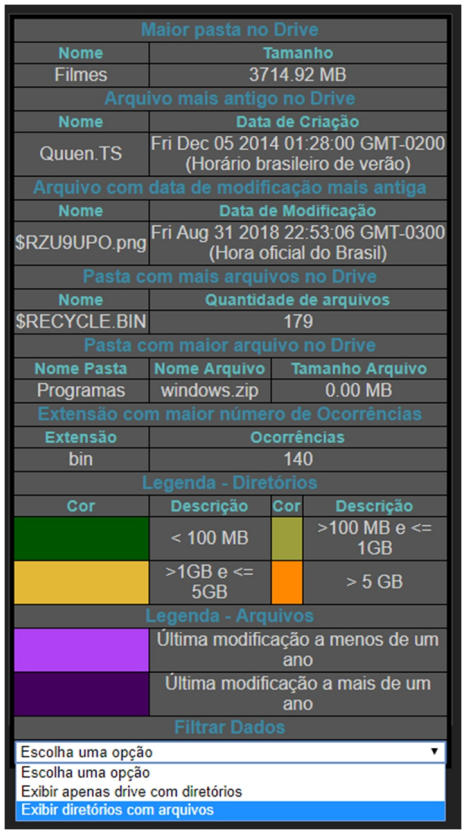

Figura 40. Opções do filtro de visualização.

Ao selecionar a visualização que exibe os arquivos, a tela é recarregada e os arquivos são exibidos na forma de nós, respeitando o *layout* de árvore hierárquica. A Figura 41 demonstra a tela de visualização exibindo os arquivos em seus respectivos diretórios e subdiretórios.

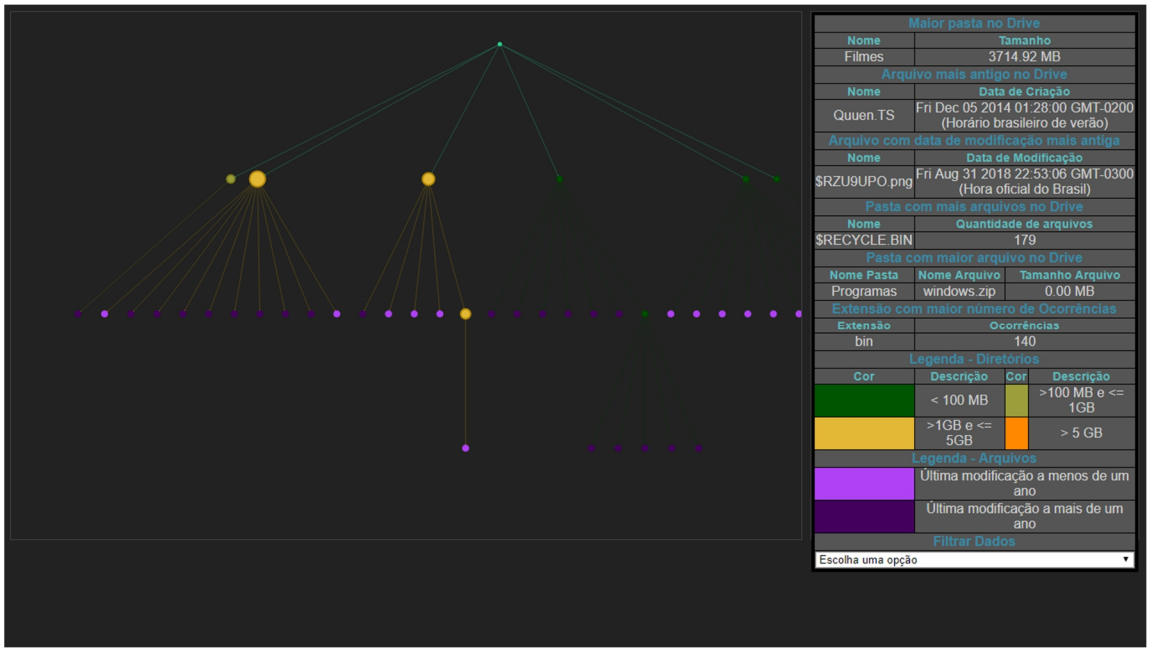

Figura 41. Visualização com arquivos.

Os nós que representam os arquivos também possuem tamanhos diferentes, representando a diferença de tamanho que os mesmos ocupam em disco. Os nós apresentam também cores diferentes em relação aos nós que representam diretórios. Os nós que representam arquivos podem possuir duas possíveis cores: roxo escuro caso o arquivo tenha sido modificado em período de tempo superior a um ano e roxo claro (lilás) caso sua última modificação tenha ocorrido a menos de um ano. Tal espaço de tempo foi estabelecido como padrão por ser o mesmo espaço de tempo utilizado pelos demais softwares correlatos, porém, cabe ressaltar que pode ser alterado facilmente, de acordo com a necessidade da organização. Pretende-se, futuramente, adicionar a funcionalidade deste padrão de tempo poder ser alterado visualmente pelo usuário. Com isto, é possível identificar arquivos que são frequentemente utilizados e arquivos que estão sem utilização. Logo abaixo de cada nó o nome do arquivo e sua extensão são exibidos. A Figura 42 apresenta como são mostrados os nós que representam arquivos.

Figura 42. Nós de arquivos.

Ao posicionar o ponteiro do *mouse* sobre um nó que representa um arquivo, uma janela pop-up é exibida. Nesta janela são apresentadas as informações de tamanho que o arquivo ocupa em disco e sua última data de modificação de maneira detalhada. A Figura 43 apresenta a janela pop-up exibida ao se posicionar o ponteiro do *mouse* sobre um nó de arquivo.

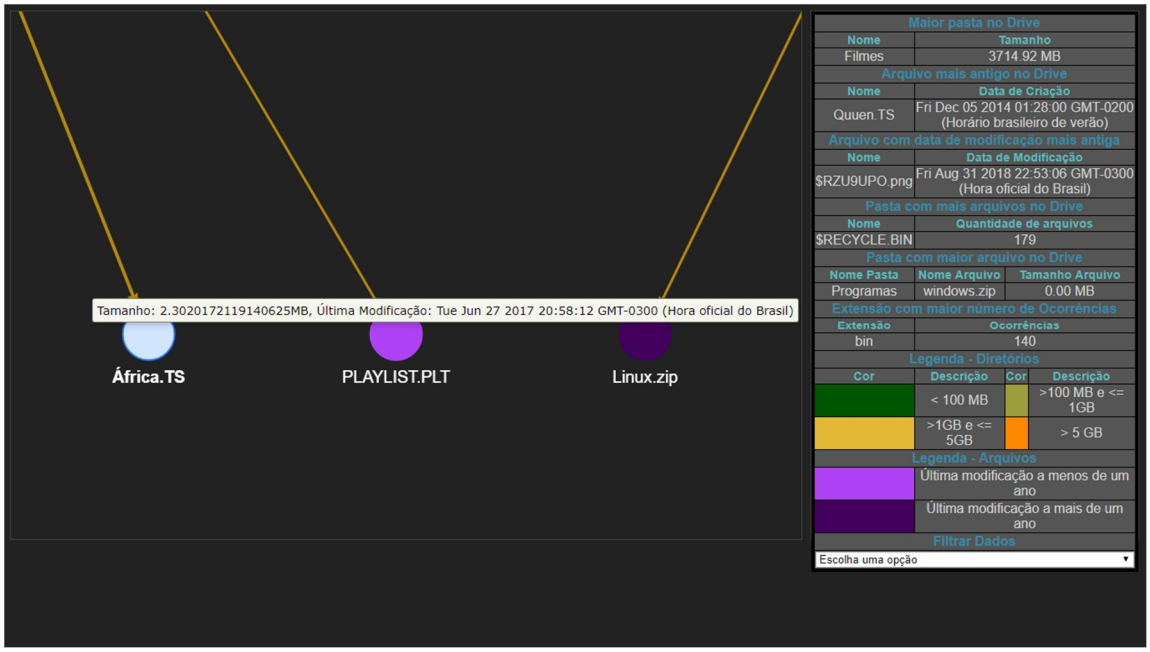

Figura 43. Janela pop-up com informações do arquivo.

Para a implementação de visualização foi utilizado o *framework* vis.js. O vis.js é uma biblioteca de visualização gráfica para representação de estruturas hierárquicas, porém não conta com integração ao Neo4J. O código fonte do *framework* possui licença *open source* e pode ser encontrado no endereço https://visjs.org/. Também foi utilizado o *framework* Bootstrap para a estilização do *layout* da página de visualização. O Bootstrap é um *framework* de código aberto para desenvolvimento de páginas *web* com HTML (Hyper Text Markup Language), CSS (Cascading Style Sheets) e JS (Javascript). O código fonte pode ser encontrado em https://getbootstrap.com/docs/4.3/getting-started/download/.

6.1 Metodologia e Justificativas

A solução proposta como resultado desta dissertação objetiva ser uma alternativa às demais soluções gratuitas existentes. Após o teste e análise das demais soluções, foram identificadas lacunas ou possíveis funcionalidades alternativas que não se fizeram presentes em nenhuma das soluções testadas, bem como foram identificadas características de usabilidade, como as apresentadas por Schneiderman (1996) que não estavam presentes em conjunto nas demais soluções. Com base nestas lacunas, desenvolveu-se o protótipo que será apresentado em detalhes na seção 6.2 desta dissertação.

Quando a solução apresentada nesta dissertação é comparada as demais soluções gratuitas para uso apresentadas anteriormente, pode se observar que é a única que apresenta as informações na tela em formato de árvore enraizada, deixando claro a hierarquia dos arquivos e diretórios contidos no espaço de armazenamento pesquisado, bem como trata-se da única que possuir o agrupamento das características de usabilidade: visão geral, zooming, filtragem, detalhes por demanda. A forma de visualização é considerada ponto chave da aplicação visto que a representação hierárquica através de uma árvore representando arquivos e diretórios é o tipo de visualização que mais se aproxima da maneira como se navega por arquivos e diretórios na maioria dos sistemas operacionais modernos e comumente utilizados pelo mercado, como por exemplo Microsoft Windows, Linux em suas diversas distribuições, Mac OSX, Android, dentre outros.

Observa-se também que a solução proposta nesta dissertação é a que visualmente apresenta o menor conjunto de informações em forma de texto, detalhando de forma textual apenas os itens sobre os quais o ponteiro do *mouse* for parado. A Figura 44 apresenta o comparativo entre o *software* Autopsy, que utiliza essencialmente visualização das informações na forma de texto e o protótipo resultado desta dissertação.

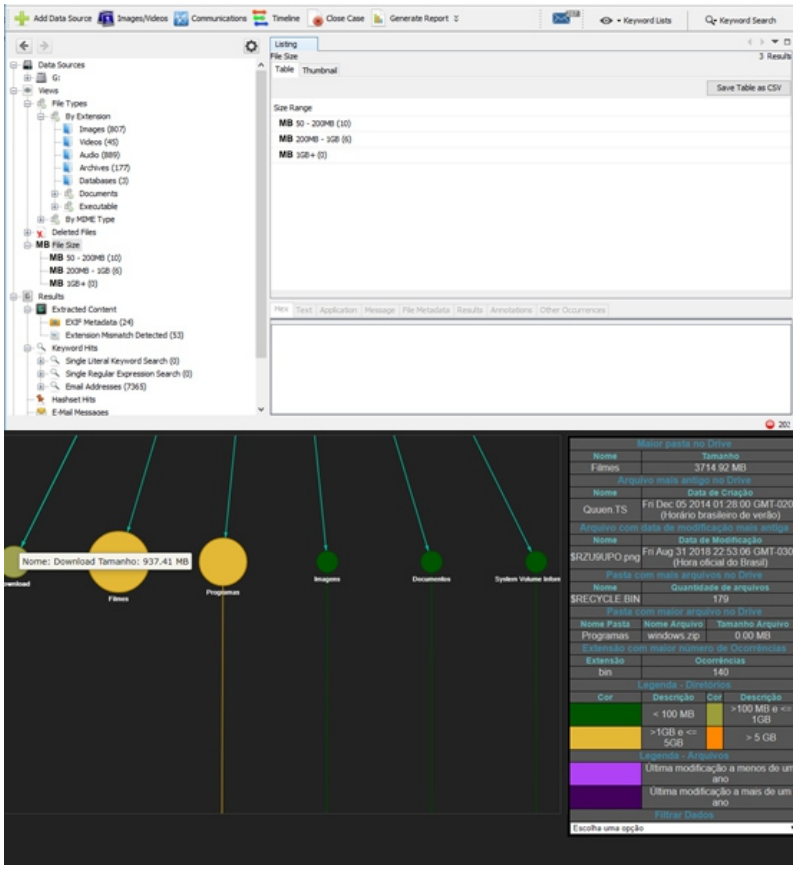

Figura 44. Comparativo com o *software* Autopsy.

Desta maneira, a visualização torna-se direta e objetiva quando comparada com a de outros *softwares* citados nesta dissertação.

Outra diferença é a forma escolhida para visualização. Enquanto os demais *softwares* existentes se fazem valer do mapa de árvore, o protótipo utiliza a visualização em forma de diagrama vértice-aresta, deixando clara ao usuário a hierarquia entre os diretórios e arquivos. A Figura 45 apresenta um comparativo da visualização gráfica - do mesmo conjunto de dados - entre a ferramenta WinDirStat e o protótipo desenvolvido.

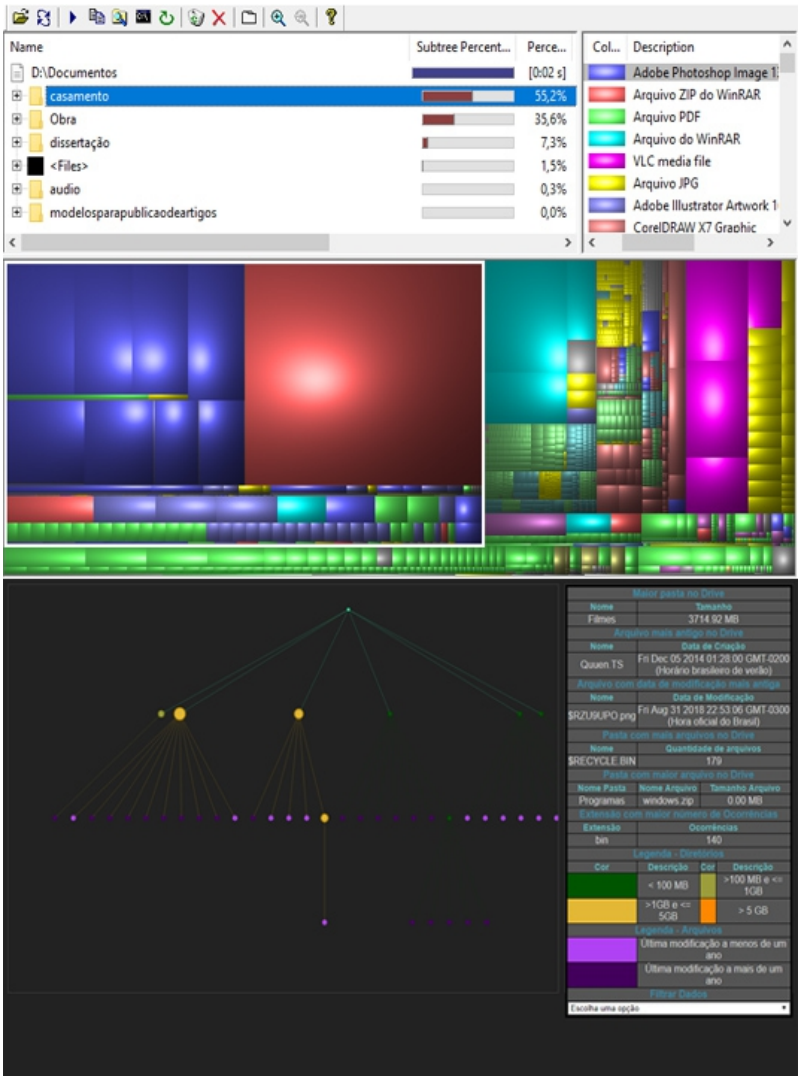

Figura 45. Comparativo de visualização com o software WinDirStat.

Pode se observar que a visualização gráfica do *software* WinDirStat, realizada por meio de um mapa de árvore, possui um número maior de elementos agrupados e representados por diversas cores. Porém, torna-se mais difícil identificar a hierarquia entre os elementos bem como suas informações complementares. No protótipo desenvolvido nesta dissertação, os elementos são mais espaçados e a hierarquia entre eles fica visualmente explícita ao usuário. O detalhamento de cada elemento (diretório ou arquivo) se faz presente somente quando o usuário passa o ponteiro do *mouse* sobre o mesmo.

As informações contidas na visualização no formato de árvore tendem a ser mais fáceis de assimilar pelo usuário quando comparadas com as informações contidas em uma visualização no formato de mapa de árvore - formato de visualização predominante nos demais *softwares* comparados nesta dissertação. Tal fato é corroborado pela pesquisa

realizada por Laubheimer (2019) que afirma que os mapas de árvore são um tipo de visualização de dados complexa, não sendo possível obter uma compreensão rápida do que é exibido. Assim, a falta de compreensão rápida passa a ser um problema visto que o principal requisito para qualquer tipo de informação apresentada em um painel é a sua rápida compreensão. Ainda de acordo com estudos conduzidos por Zhao, McGuffin e Chignell (2005) é possível identificar claramente uma topologia ou hierarquia em um diagrama de árvore, com a desvantagem de que os nós são distribuídos geralmente de maneira desigual. Já em um mapa de árvore, o espaço de visualização é usado com eficiência, porém a forma de exibição é menos familiar e mais difícil de interpretar quando comparado com uma visualização no formato de árvore.

Pode-se determinar, portanto, que os mapas de árvore são frequentemente usados para visualizar conjuntos grandes de dados, otimizando o espaço de visualização quando comparado com a visualização em árvore, porém a grande quantidade de informações distribuídas uniformemente e sem espaços entre si pode sobrecarregar visualmente os usuários. Conforme afirma Laubheimer (2019), um mapa de árvore pode tornar-se apenas um grande conjunto de retângulos minúsculos, pequenos demais até mesmo para possuir rótulos textuais. Ainda, de acordo com a complexidade e quantidade de retângulos em um mapa de árvore, a hierarquia entre diretórios e arquivos pode tornar-se indiscernível.

Outro ponto a se destacar entre os *softwares* existentes e o protótipo desenvolvido nesta dissertação é o que diz respeito às cores. Em nenhum dos demais *softwares* pesquisados o usuário possui a opção de definir quais cores ele deseja na exibição de diretórios e arquivos. Tal fato pode ser considerado uma desvantagem, dado que o uso das cores diversificadas é fundamental para uma ferramenta de visualização gráfica. Estudos comprovam que o uso das cores possui efeito psicológico no ser humano, conforme afirma Farina, Peres e Bastos (2011) a visualização das cores faz penetrar através de nossos olhos uma variedade de ondas com diferentes potencias, as quais atuam sobre os centros nervosos do cérebro e modificam nossas atividades sensoriais, emocionais e afetivas. Desta forma, possibilitar ao usuário escolher de maneira personalizada as cores para a visualização dos diretórios e arquivos permite que o mesmo configure cores mais chamativas para destacar itens que mais lhe interessam e cores menos chamativas para itens que sejam de menor interesse no momento da visualização de acordo com sua subjetividade. Alguns dos softwares gratuitos relacionados sequer possuem a preocupação

no uso de cores na representação dos diretórios e arquivos, conforme se pode observar por comparativo na Figura 46 entre o *software* File System Visualizer e o protótipo obtido como resultado desta dissertação.

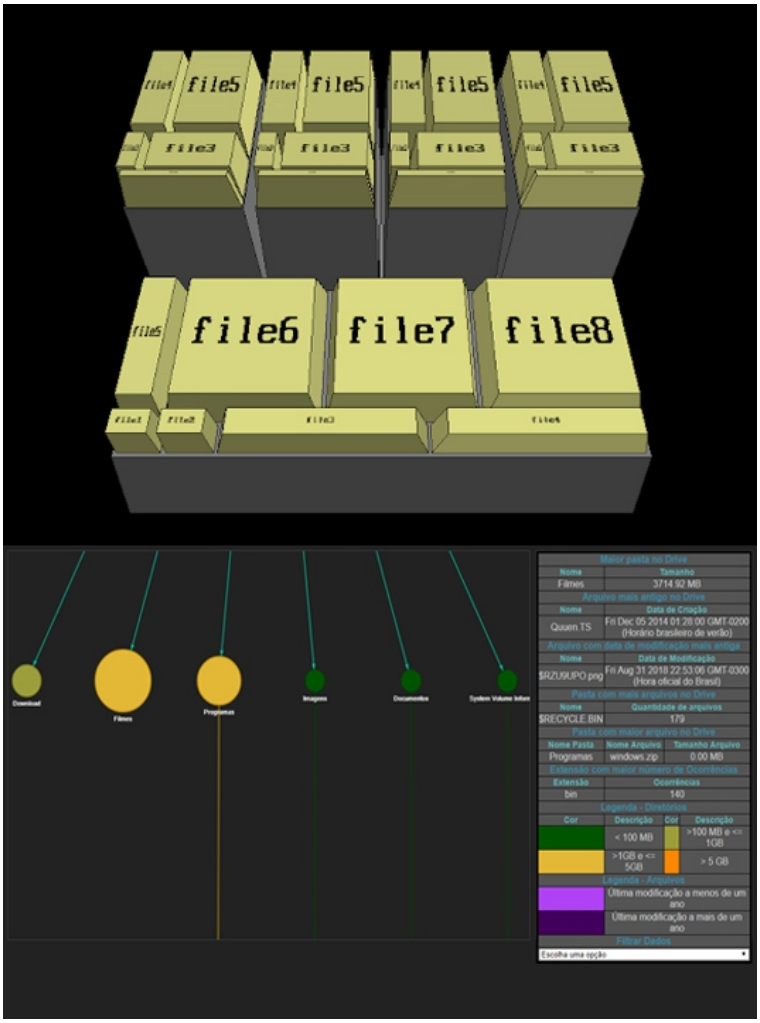

Figura 46. Comparativo de cores com o software File System Visualizer.

Sendo assim, o protótipo de *software* obtido como resultado desta dissertação diferencia-se dos demais *softwares* para visualização e avaliação de diretórios e arquivos através de metadados por entregar ao usuário final uma alternativa no modo de visualização, utilizando árvores em contraponto aos demais que utilizam mapa de árvore e permitindo também a personalização das cores dos nós que representam os arquivos e diretórios, tornando-o mais simplificado e didático.

6.3 Síntese do Capítulo

A implementação da solução proposta nesta dissertação buscou manter uma interface simples e fácil de ser manipulada por qualquer tipo de usuário, sobretudo dos usuários que não possuem elevados conhecimentos nas áreas relacionadas à ciência da computação. A partir da seleção do disco a ser analisado, a tela de visualização gerada apresenta toda a estrutura de diretórios e arquivos do disco de forma clara, diferenciando diretórios e arquivos através de cores e tamanhos. A tabela de legendas e resumo do disco exibida na interface auxilia no entendimento das informações exibidas no grafo. Além disso, usuários com maiores conhecimentos podem através de códigos escritos na linguagem CSS alterar detalhes do esquema de cores da visualização facilmente.

7. Conclusão

Este capítulo apresenta a conclusão desta dissertação, abordando uma visão sobre as contribuições da pesquisa, possíveis trabalhos futuros e considerações finais. A seção 7.1 destaca as principais contribuições que a pesquisa proporciona em relação aos pontos estabelecidos como objetivos. A seção 7.2 apresenta uma discussão sobre possíveis trabalhos futuros, apontando inclusive as limitações encontradas no desenvolvimento dessa dissertação. Na seção 7.3 são apresentadas as considerações finais com caráter crítico-analítico.

7.1 Contribuições

Esta pesquisa tem como objetivo pesquisar e analisar as diversas técnicas e ferramentas voltadas a visualização de dados, bem como analisar as características desejadas de usabilidade no desenvolvimento de uma ferramenta ou técnica de visualização. Também foi realizada uma pesquisa buscando entender o estado da arte das ferramentas de visualização de dados com licença não proprietárias disponíveis. Prover para o usuário final uma capacidade na tomada de decisões e a avaliação de dados em grandes quantidades de arquivos armazenados, através da visualização de maneira simples e didática do que se tem de fato armazenado é uma preocupabação e busca inerente de todas as técnicas de visualização. Não saber exatamente o que se tem de fato armazenado torna-se um problema, gerando na maioria das vezes gastos desnecessários com armazenamento visto que torna-se difícil otimizar o uso do espaço disponível. Com base nos estudos e pesquisas realizadas dispostos nos capítulos desta dissertação, foi proposto como resultado, visando a aplicação prática das técnicas de visualização e conceitos de usabilidade análisadas, o desenvolvimento modular de um protótipo de sistema voltado à realização de busca, mapeamento e armazenamento em forma de árvore hierárquica de um conjunto de metadados de arquivos armazenados, exibindo de forma didática ao usuário o resultado deste mapeamento. Tal aplicação busca ser uma alternativa mais acessível ao entendimento e manipulação por parte dos usuários em geral quando comparada com outras aplicações do genêro. Para tal, levou se em consideração as características desejáveis de usabilidade apresentadas por Schneiderman (1996) e apresentadas em capítulo anterior, sendo possível implementar a maior parte delas, sendo:

visão geral, zooming, filtragem, detalhes por demanda. Com a implementação deste conjunto de características, associadas a uma visualização clara e precisa da hierárquia dos dados e o uso de cores personalizadas pelo usuário na representação de diretórios e arquivos bem como a utilização do recurso de tamanhos variáveis nos elementos gráficos, pretende-se facilitar a tomada de decisões e a otimização dos arquivos armazenados por parte dos usuários. Assim, a visualização é capaz de proporcionar a análise de diretórios e arquivos de maneira simples e acessível a qualquer tipo de usuário propiciando rapidamente a identificação, por exemplo, de arquivos e diretórios raramente utilizados ou que consomem muito espaço em disco, permitindo uma melhor tomada de decisão sobre os mesmos.

O protótipo foi testado em um sistema operacional desktop (Microsoft Windows 10) e que já conta com ferramentas para a realização de buscas e filtros para encontrar arquivos e otimizar espaços de armazenamento. Porém, ainda com o teste limitado a apenas alguns milhares de arquivos a visualização hierárquica, seleção personalizada de cores, a combinacação das características de usabildiade e os filtros desenvolvidos na aplicação, mostraram-se promissores na tarefa de visualizar e encontrar arquivos não acessados ou criados há muito tempo e que estavam apenas ocupando espaço de forma desnecessária. Através dos metadados é possível saber o que se tem armazenado em cada diretório e entender o que é cada arquivo. Como a premissa dessa dissertação é que a aplicação obtida como resultado dos estudos seja uma alternativa de ferramenta de visualização agregando características e técnicas não presentes de forma conjunta em nenhuma das demais aplicações comparadas e que seja viável para o uso, a mesma mostra-se uma alternativa a ser considerada, ainda que careça de aprimoramentos.

7.2 Trabalhos Futuros

Entre as limitações encontradas no desenvolvimento desta dissertação, destaca-se a dificuldade em encontrar estudos relevantes a respeito do *data assessment* através de técnicas de visualização hierárquica de dados, sendo este um tema não muito explorado pela comunidade acadêmica. Outra dificuldade relevante encontrada ao longo do projeto foi referente ao desenvolvimento de formas de visualização dos dados contidos na base de dados do Neo4J. Por tratar-se de um modelo de dados que ainda não é comercialmente popular, os *frameworks open source* que provêm visualização sobre os dados armazenados

na base são raros e com recursos limitados. Por fim, outra dificuldade encontrada no desenvolvimento do protótipo foi relativa ao teste do mesmo. Os testes do protótipo foram realizados no sistema operacional Windows em sua versão 10, que possui algumas limitações no retorno de metadados, ocultando informações que seriam de interesse relevante como, por exemplo, o tamanho total que um diretório ocupa no dispositivo de armazenamento.

Dadas as limitações encontradas no desenvolvimento do protótipo e as limitações da pesquisa realizada, é possível citar os seguintes trabalhos futuros principais:

- Aprimoramento do módulo de visualização com o acréscimo de filtros para os dados e arquivos baseados em testes de usabilidade com possíveis grupos de usuários.
- Desenvolver um método para o cálculo do tamanho total que um diretório ocupa no sistema de armazenamento, de modo a não depender dos metadados de diretório.

7.3 Considerações finais

Uma ferramenta de visualização de metadados de arquivos é um recurso de muito potencial, tanto em questões ligadas a auxílio na tomada de decisões sobre as informações quanto em operações de data assessment. Adicionalmente, é extremamente útil para companhias e usuários que possuem grandes quantidades de arquivos armazenados em sistemas. Considerando que a utilização de espaço de armazenamento cresce diariamente, conseguir otimizar tal uso pode contribuir para a economia de recursos financeiros e computacionais. Cabe ressaltar que a aplicação comercial da ferramenta obtida como resultado dessa dissertação ainda não é algo trivial, visto que a mesma ainda carece de aprimoramentos atualmente indisponíveis nas tecnologias utilizadas.

Tecnologias como o Node.JS e a base de dados de grafos Neo4J tem sido atualizadas de forma constante com novos recursos e funcionalidades. Assim, podem-se vislumbrar, em um futuro próximo, novos trabalhos para o aprimoramento da ferramenta e do tema, sendo este de grande relevância para o mercado da tecnologia da informação.

Referências

3D File System Visualizer. 2019. http://fsv.sourceforge.net/. (acesso em 26 de Abril de 2019).

Abowd, Gregory D, Joëlle Coutaz, e Laurence. Nigay. "Structuring the Space of Interactive System Properties." *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for HumanComputer Interaction*, 1992.

Amazon. *Amazon DynamoDB.* 2019. https://aws.amazon.com/pt/dynamodb/.

Anderson, J. Chris, Jan Lehnardt, e Noah Slater. *CouchDB: The Definitive Guide.* O'Reilly Media, 2009.

Autopsy. 2019. https://www.sleuthkit.org/autopsy/ (acesso em 26 de Abril de 2019).

Azure, Microsoft. *Armazenamento de Tabelas.* 2019. https://azure.microsoft.com/pt-br/services/storage/tables/ (acesso em 15 de Agosto de 2019).

Barr, Jeff. *Amazon S3 – The First Trillion Objects.* 2013. https://aws.amazon.com/pt/blogs/aws/amazon-s3-the-first-trillion-objects/ (acesso em 23 de Janeiro de 2019).

Bertin, J. *Graphics and Graphic Information Processing.* New York: Walter de Gruyte, 1981.

Brito, R. W. "Banco de Dados NoSQL x SGBDs Relacionais." *InfoBrasil.* 2010. http://www.infobrasil.inf.br/userfiles/27-05-S4-1-68840Bancos%20de%20Dados%20NoSQL.pdf (acesso em 29 de Setembro de 2018).

Bundt, Jacob. "A graphical file system visualization tool for operating systems." *University of Northern Iowa UNI ScholarWorks.* 2018. https://scholarworks.uni.edu/hpt/348/ (acesso em 05 de Fevereiro de 2019).

Calder, Brad. *Windows Azure Storage – 4 Trillion Objects and Counting.* Julho de 2012. https://azure.microsoft.com/pt-br/blog/windows-azure-storage-4-trillion-objects-and-counting/ (acesso em 23 de Janeiro de 2019).

Cassandra. *Apache Cassandra.* 2019. http://cassandra.apache.org/ (acesso em 02 de 11 de 2019).

Cava, Ricardo, Paulo Roberto Gomes Luzzardi, e Carla Dal Sasso Freitas. *The Bifocal Tree: a Technique for the Visualization of Hierarchical Information Structures.* Pelotas, 2003.

Chandra, Deka Ganesh. "BASE analysis of NoSQL database." *Future Generation Computer Systems*, 2015.

Chaniotis, Ioannis K., Kyriakos-Ioannis D. Kyriakou, e Nikolaos D. Tselikas. "Is Node.js a viable option for building modern web applications? A performance evaluation study." *Springer-Verlag Wien 2014.* 2014. 1023-1044.

Chi, E. H. et al. "Visualizing the evolution of web ecologies. ." *Conference on human factors in computer systems*, 1998.

Conrod, Jay. *A tour of V8: full compiler.* 2012. http://jayconrod.com/posts/51/a-tour-of-v8-full-compiler (acesso em 13 de Março de 2019).

—. *A tour of V8: garbage collection.* 2013. http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection (acesso em 14 de Março de 2019).

CouchDB. *Couch Database.* 2019. https://www.couchbase.com/.

Coulouris, G, J Dollimore, T Kindberg, e G Blair. *Sistemas Distribuídos-: Conceitos e Projeto.* Bookman Editora, 2013.

Dahl, Ryan. *Node.js.* 2009. http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html (acesso em 13 de Março de 2019).

Dastani, M. " The role of visual perception in data visualization." *Journal of Visual Languages and Computing.* , 2002: pp. 601-622.

Dulclerci, Sternadt Alexandre, e João Manuel R. S. Tavares. " Factores da Ppercepção Visual Humana na Visualização de Dados ." *Congresso Ibero Latino-Americano sobre Métodos Computacionais em Engenharias*, 2007.

Express.JS. *Express.JS.* 2019. http://expressjs.com/ (acesso em 15 de Março de 2019).

Fallabs. *Tokyo Cabinet: a modern implementation of DBM.* 2019. https://fallabs.com/tokyocabinet/ (acesso em 02 de 11 de 2019).

Farina, Modesto, Clotilde Perez, e Dorinho Bastos. *Psicodinâmica das cores em Comunicação.* São Paulo: Blucher, 2011.

Fernandes, Fabiano da S. *Banco de Dados vs Mapreduce em Algoritmos para Grafos.* Março de 2015.

Few, Stephen. *Now you see it: simple visualization techniques for quantitative analysis.* Analytics Press, 2009.

Fox, A, e E. A Brewer. "Harvest, Yield, and Scalable." *Proceedings of the Seventh Workshop on Hot Topics in*, 1999.

Fraga, Marcelo Caramuru Pimentel, e William Geraldo Sallum. *Sistemas Operacionais II.* 2016.

Gershon, N., e S. Eick. "Information Visualization." *IEEE Computer Graphics and Applications*, 1997: p. 29-31.

GraphDB. *GraphDB.* 2019. http://graphdb.ontotext.com/ (acesso em 02 de 11 de 2019).

Hadoop. *Apache Hadoop.* 2019. https://hadoop.apache.org/.

HBase. *Welcome to Apache HBase.* 2019. https://hbase.apache.org/ (acesso em 02 de 11 de 2019).

Holmquist, L.E. "Focus+Context Visualization with Flip Zooming and the Zoom Browser." E*xtended Abstract of Conference on Human Factors in Computer Systems*, 1997: p.263-264.

Holten, Danny. "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data." *IEEE transactions on visualization and computer graphics*, 2006: 741-748.

Holzschuher, F., e R. Peinl. *Performance of Graph Query Languages.* 2013. http://www.edbt.org/Proceedings/2013-Genova/papers/workshops/a29holzschuher.pdf (acesso em 15 de Março de 2019).

Homrich, Émerson P., e Sergio L. S. Mergen. "Comparação entre MySQL e Neo4J para o Acesso a Dados Complexos Usando Linguagens Declarativas." *XIV Escola Regional de Banco de Dados* . Rio Grande, 2018. 32-41.

Hunger, M, e P. Neubauer. *Neo4j: Java-based NoSQL Graph Database.* 2010. http://www.infoq.com/news/2010/02/neo4j-10 (acesso em 15 de Março de 2019).

Jerding, D.F., e J.D. Stasko. " The Information Mural: A technique for displaying and navigating large information spaces." *IEEE Transactions on Visualization and Computer Graphics*, 1998: p. 257-271.

Johnson, B, e B. Shneiderman. "Tree-maps: A space-filling approach to the visualization of hierarchical information structures." *IEEE Visualization'91.* San Diego: IEEE, 1991. 284-291.

Kandel, Sean, Ravi Parikh, Andreas Paepcke, e Joseph Hellerst. "Profiler: Integrated Statistical Analysis and Visualization for Data Quality Assessment." *International Working Conference on Advanced Visual Interfaces.*, 2012: 547-554.

Keim, Daniel A. "Visual Exploration of Large Data Sets." *Communications of ACM 44.* 2001. 38-44.

Lamping, J., R. Rao, e P. Pirolli. ""A Focus+Context Technique Based in Hyperbolic Geometry for Visualizing Large Hierarchies." *CHI'95 ACM Conference on Human Factors in Computing Systems.* 1995. 401-408.

Laubheimer, Page. *Nielsen Norman Group.* 29 de Setembro de 2019. https://www.nngroup.com/articles/treemaps/ (acesso em 15 de Junho de 2020).

Leung, Andrew W., Minglong Shao, Timothy Bisson, Shankar Pasupathy, e Ethan L. Mille. "Spyglass: Fast,Scalable Metadata Search for Large-Scale StorageSystems." *7th USENIX Conference on File and Storage Technologies.* San Francisco, 2009.

Lóscio, B. F, H. R. Oliveira, e J. C. S. Pontes. "NoSQL no desenvolvimento de Aplicações Web Colaborativas." *VIII Simpósio Brasileiro de Sistemas Colaborativos.* 2011.

Luzzardi, Paulo Roberto Gomes. "Critérios de Avaliação de Técnicas de Visualização de Informações Hierárquicas." *Tese de Doutorado em Ciência da Computação, Universidade Federal do Rio Grande do Sul*, Março de 2003.

Mackinlay, Jock D., George G. Robertson, e Stuart K. Card. “The Perspective Wall: Detail and context smoothly integrated.” *CONFERENCE ON HUMAN FACTORS IN COMPUTER SYSTEMS*, 1991: p.173-176.

Maidin, Cian. *Why Node.JS is becoming the go-to technology in the enterprise.* 2014. http://www.nearform.com/nodecrunch/node-jsbecoming-go-technology-enterprise/ (acesso em 13 de Março de 2019).

Markets, Markets and. *Cloud Storage Market by Type.* Fevereiro de 2018. https://www.marketsandmarkets.com/Market-Reports/cloud-storage-market-902.html (acesso em 21 de Janeiro de 2019).

Marungo, Fumbeya. “A Primer on NoSQL Databases for Enterprise Architects: The CAP Theorem and Transparent Data Access with MongoDB and Cassandra .” *Proceedings of the 51st Hawaii International Conference on System Sciences* , 2018: 4621-4630.

MongoDB. *MongoDB - The database for modern applications.* 2019. https://www.mongodb.com/.

Muniz, Matheus Henrique da Silva, e Ricardo Costa P Santos. “Comparação de Performance de Processamento entre Bases de Dados Relacionais e Bases de Dados NoSql.” 2018.

Munzner, T. “H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space.” *IEEE Information Visualization’97.* Phoenix: IEEE Computer Society, 1997. 2-10.

Nayak, Ameya, Anil Poriya, e Dikshay Poojary. “Type of NOSQL Databases and its Comparison with Relational Databases.” *International Journal of Applied Information Systems*, 2013: 16-19.

Neo4J. *Neo4J.* 2019. http://neo4j.org (acesso em 15 de Março de 2019).

Nielsen, Jakob. *Usability Engineering.* São Francisco: Morgan Kaufmann, 1993.

NPM. *NPM.* 2019. https://docs.npmjs.com/getting-started/what-isnpm (acesso em 15 de Março de 2019).

Ogden, Max. *Callback Hell.* 2012. http://callbackhell.com/ (acesso em 14 de Março de 2019).

Oliveira, Samuel Silva de. “BANCOS DE DADOS NÃO-RELACIONAIS: UM NOVO PARADIGMA PARA ARMAZENAMENTO DE DADOS EM SISTEMAS DE ENSINO COLABORATIVO .” *Revista da escola de administração pública do amapá*, 2014: 184-194.

Oracle. *Oracle Berkeley DB.* 2019. https://www.oracle.com/database/technologies/related/berkeleydb.html (acesso em 02 de 11 de 2019).

Panzarino, Onofrio. *Learning Cypher.* Packt Publishing Ltd, 2014.

Pritchett, Dan. “BASE: An Acid Alternative.” *Queue - Object-Relational Mapping*, 2008: 48-55.

Redis. *Redis.* 2019. https://redis.io/ (acesso em 15 de Agosto de 2019).

Rekimoto, Jun, e Mark Green. “The Information Cube: Using Transparency in 3D Information Visualization.” *ANNUAL WORKSHOP ON INFORMATION TECHNOLOGIES AND SYSTEMS,*, 1993: p.125-132.

Reuters. *Global Cloud Services Market and Cloud Storage Industry Size, Share, Growth Statistics, Industry Analysis 2018 and Forecast to 2025.* 18 de Outubro de 2018. https://www.reuters.com/brandfeatures/venture-capital/article?id=59412 (acesso em 21 de Janeiro de 2019).

Riak. *Riak: Enterprise NoSQL Database.* 2019. https://riak.com/ (acesso em 15 de Agosto de 2019).

Robertson, G., J. Mackinlay, e S. Card. “Cone Trees: Animated 3D Visualizations of Hierarchical Information.” *CHI’91 ACM Conference on Human Factors in Computing Systems.* 1991. 189-194.

Robertson, George G., Stuart K. Card, e Jock D. Mackinlay. “Cone Trees: Animated 3D Visualizations of Hierarchical Information.” *CONFERENCE ON HUMAN FACTORS IN COMPUTER SYSTEMS*, 1991: p.189-194.

Rockenbach, Dinei A., Nadine Anderle, Dalvan Grieble, e Samuel Souza. “Estudo Comparativo de Bancos de Dados NoSQL.” 2014.

Rydning, John. *Worldwide Hard Disk Drive Forecast, 2018–2022.* Maio de 2018. https://www.idc.com/getdoc.jsp?containerId=US43766218 (acesso em Janeiro de 2019).

Sadalage, P. J, e M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 2012.

Saliba Júnior, Edwar. *Introdução a Arquitetura de Sistemas Operacionais.* 2017.

Schonhage, S.P.C, A van Ballegooij, e A.P.W. Eliëns. "3D Gadgets for Business Process Visualization: a case study." *VRML/Web 3D - 2000 Conference*, 2000.

Seagate. *A demanda por dispositivos de armazenamento em um mundo de dados conectado.* 2019. https://www.seagate.com/br/pt/tech-insights/demand-for-storage-devices-in-connected-world-master-ti/ (acesso em 21 de Janeiro de 2019).

Shneiderman, B. "Tree visualization with Treemaps: a 2d space filling approach." *ACM Transactions on Graphics*, 1992: 92-99.

SILVA, M. da. "A polissemia do termo" arquivo"." *Encontros Nacionais de Pesquisa e Pós-Graduação em Ciência da Informação*, 2017.

Silva, Victória de Abreu. *Preservação digital: um estudo sobre padrões de metadados.* 2018.

Solo, A.M.G., e M.M. Gupta. "Perspectives on computational perception and cognition under uncertainty." *Proceedings of IEEE International Conference on Industrial Technology 200*, 2000: pp. 221-224.

Spence, R., e M. Apperley. "Data base navigation: an office environment for the professional." *Behavior and Information Technology*, 1982: p.43-54.

Stonebraker, Michael. *SQL Databases v. noSQL Databases.* Abril de 2010. http://cacm.acm.org/blogs/ blog-cacm/50678 (acesso em 15 de Setembro de 2018).

Tanenbaum, Andrew S, e Albert S. Woodhul. *Sistemas Operacionais: Projetjos e Implementação.* Bookman Editora, 2009.

Technavio. *Global Hard Disk Drive Market 2011-2015.* 2015. https://www.technavio.com/content/global-hard-disk-drive-market-2011-2015 (acesso em 21 de Janeiro de 2019).

The Sleuth Kit. 2019. https://www.sleuthkit.org/sleuthkit/ (acesso em 26 de Abril de 2019).

Tilkov, Stefan, e Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Program." *The Functional Web*, 2010: 80-83.

Tukey, J.W. *Exploratory data analysis.* Addison Wesley, 1977.

Veritas. *Veritas Data Insight.* 29 de Dezembro de 2019. https://www.veritas.com/pt/br/insights/data-insight (acesso em 29 de Dezembro de 2019).

Wheeler, Ric. *One Billion Files: Scalability Limits in Linux File Systems.* Agosto de 2010.

WinDirStat. 2019. https://windirstat.net/.

Zhao, Shengdong, Michael J. Mcguffin, e Mark H Chignell. "Elastic hierarchies: Combining treemaps and node-link diagrams." *EEE Symposium on Information Visualization*, 2005: 57-64.

Apêndice A – Detalhes do framework Node.js

O Node.js foi criado por um desenvolvedor de nome Ryan Dahl, quando este viu-se diante de um problema aparentemente sem solução: informar em tempo real e corretamente a um usuário que realiza a transferência de um arquivo entre sua máquina e um servidor o progresso de *upload* (Dahl 2009). As funções e métodos existentes para retornar o progresso do *upload* ao usuário exigiam consultas frequentes ao servidor, retornando valores percentuais. Tais valores então eram inseridos em um objeto disponível no DOM (*Documento Object Model*) da aplicação *web*. Para Dahl tal método era um problema, dado ineficiência para lidar concorrentemente com múltiplos uploads, visto que cada *upload* geraria uma série de respostas para informar o percentual completado. Para Ryan, a solução para tais inconvenientes era o envio de maneira automática por parte do servidor de atualizações de progresso do *upload* ao cliente, com base em mudanças e sem o envio de requisições frequentes ao servidor. Paralelamente ao desenvolvimento do projeto de Dahl, as grandes empresas desenvolvedoras de navegadores *web* investiam quantidades significativas de recursos em uma acirrada competição pelo desenvolvimento de interpretadores Javascript com desempenho superior aos já existentes. Isso gerou um aumento acentuado na utilização e popularidade do Javascript, aumentando também a quantidade de projetos *open source* voltados para a melhora de performance da linguagem, um gargalo histórico da mesma e que durante muito tempo inviabilizou sua utilização em grandes projetos. Neste contexto histórico, aspectos da linguagem como a sintaxe simples, próxima da sintaxe da linguagem C, orientação a eventos nativa e a abertura do código fonte do compilador Javascript V8 *Engine* por parte da equipe do Google, fizeram do Javascript uma escolha óbvia para o desenvolvimento do projeto de Dahl.

É importante ressaltar que o Node.js não foi o primeiro projeto de implementação da linguagem JavaScript no *back-end* de aplicações. Diversos outros projetos com diferentes características, (pode-se citar: Helma4, AppEngine JS SDK5 e RingoJS6) surgiram anos antes do desenvolvimento do Node.js, porém todos os projetos adotavam a Java Virtual Machine (JVM) como interpretador base no servidor, de modo que tais projetos apenas portavam a linguagem JavaScript para o servidor, não possuindo características de operações de entrada e saída não bloqueantes, execução assíncrona, *single-threading* ou programação orientada a eventos, características essenciais para o

sucesso do Node.js. Portanto, qualquer comparação entre o Node.js e as implementações de Javascript no servidor anteriores a ele deve levar tais fatos em consideração, sendo mais correto comparar o Node.js com as plataformas e *frameworks* que possuem características semelhantes (Tilkov e Vinoski 2010). Nas subseções que se seguem, são apresentados o funcionamento do compilador Chrome V8 *Engine* bem como características da linguagem Javascript que foram fundamentais para o sucesso do projeto Node.js e adoção do mesmo neste projeto de dissertação.

Compilador Chrome V8 Engine

Segundo (Conrod, 2012) o compilador Javascript de alto desempenho Chrome V8 *Engine* foi desenvolvido pelo Google e é utilizado nativamente pelo navegador Google Chrome. Pode ainda ser utilizado de forma totalmente independente ou embarcada em *softwares* desenvolvidos na linguagem C++, visto que possui código *open source* e foi escrito na linguagem C++. Desde o princípio do projeto, o V8 *Engine* possui como foco principal a alta performance, buscando solucionar problemas apresentados comumente pelo Javascript, como por exemplo a limitação de aplicabilidade e os códigos complexos utilizados no desenvolvimento de aplicações devido à característica de *single threading*. Três pontos fundamentais foram definidos pela equipe de desenvolvimento do projeto V8 quanto ao seu funcionamento:

- Compilação do código Javascript de maneira direta para linguagem de máquina nativa;
- Melhoria na gestão de memória, com a realização de interrupções para realização de *garbage collection* e atribuição mais rápida de objetos na memória;
- Aumento na eficiência de acesso a chamadas de funções e propriedades através da adoção do conceito de classes ocultas.

Uma das principais características que permitiram ao V8 *Engine* obter um melhor desempenho na execução de Javascript dentre todos os interpretadores foi a substituição da interpretação dos códigos para a compilação JIT (Conrod, 2012). A ideia de compilação *just-in-time* começou a surgir após pesquisas sobre a linguagem LISP, realizadas por McCarthy ainda na década de 1960. Nas pesquisas, McCarthy menciona à compilação de funções de maneira suficientemente rápida para que não seja necessário o salvamento da saída do compilador (Maidin, 2014). Na compilação JIT, durante a execução de um *script*,

é gerado um código de máquina a cada função em linguagem Javascript, à medida em que são executadas, evitando a geração de códigos intermediários, como na linguagem Java. Tal característica permite que durante execução de grandes bibliotecas de *scripts*, o compilador foque na execução das funções somente no momento em que são chamadas, não se mantendo ocupado compilando todas as funções presentes no *script*. No caso do Chrome V8 *Engine*, há uma complexa infraestrutura de compilação, da qual fazem parte um compilador base (denominado de Full Codegen) cuja tarefa é a geração rápida porém não otimizada de código nativo de máquina e um compilador otimizador (chamado de Crankshaft) responsável por compilar o código de maneira não tão veloz porém otimizada (Conrod, 2012). O *script* é compilado inicialmente utilizando o compilador Full Codegen, enquanto paralelamente uma *thread* é aberta para que o analisador de programas (denominado *profiler*) do V8 *Engine* faça a avaliação das funções que são críticas e das funções que são utilizadas com maior frequência durante a execução do *script*. Quando uma função crítica é identificada pelo analisador, o compilador Crankshaft fica encarregado de realizar a compilação da mesma no próximo ciclo de execução, de maneira concorrente ao compilador Codegen (Conrod, 2012).

De acordo com Conrod (2012) a compilação realizada pelo Chrome V8 *Engine* é dividida em fases distintas, sendo elas:

- Parsing: Nessa etapa, o *script* é analisado traduzido em uma AST (árvore sintática abstrata), que será utilizada pelos compiladores Codegen e Crankshaft, sendo desalocada da memória após o uso, dado que sua frequência de utilização é baixa e seu processo de reconstrução é simples.
- Análise de escopo: nesta fase o V8 estabelece o uso das variáveis segundo o escopo do *script*. De maneira similar a etapa anterior, tanto o compilador Codegen quanto o compilador Crackshaft utilizam a análise de escopo. Usando a AST gerada na etapa anterior em conjunto com informações de escopos e tipos, o compilador inicia a construção de um grafo de controle de fluxo.
- Otimizações: nessa etapa, o grafo de controle de fluxo criado na fase anterior da compilação passa por diversos processos de otimização. Ao término da etapa de otimização, o grafo de controle de fluxo (já otimizado) é utilizado como base para a geração de um segundo grafo: o grafo de representação de código de baixo-nível usado no V8. Por fim, uma série de instruções é gerada pelo compilador Crankshaft

para cada instrução presente no grafo de representação de código de baixo nível. Ao final dessa fase, o código é empacotado em um objeto do tipo Code e a execução pode ser realizada pelo V8.

Hidden Class

É possível adicionar propriedades em tempo de execução a objetos na linguagem Javascript, visto que a mesma é uma linguagem dinâmica. Porém tal característica, apesar de oferecer flexibilidade ao desenvolvedor, necessita que pesquisas dinâmicas (*dynamic lookups*) sejam executadas sempre que uma nova propriedade é chamada, para encontrar sua posição correta na memória (Maidin, 2014). Em scripts grandes o acesso a memória e as propriedades, portanto, tendem a ser muito mais demorados quando comparados ao acesso a variáveis de instâncias em outras linguagens como Java ou C++.

De modo a reduzir o tempo de busca por propriedades, o V8 *Engine* não utiliza pesquisas dinâmicas (Conrod, 2012). No lugar dessa característica foi implementado o conceito de classes ocultas (*hidden class*). Tais classes representam objetos armazenados na memória, assim, o acesso às propriedades de um objeto é direto, de modo similar ao que é feito em linguagens compiladas. Conforme novas propriedades são adicionadas ao objeto, o mesmo muda de classe oculta (Maidin, 2014). A Figura 47 abaixo exemplifica o processo realizado pelas classes ocultas.

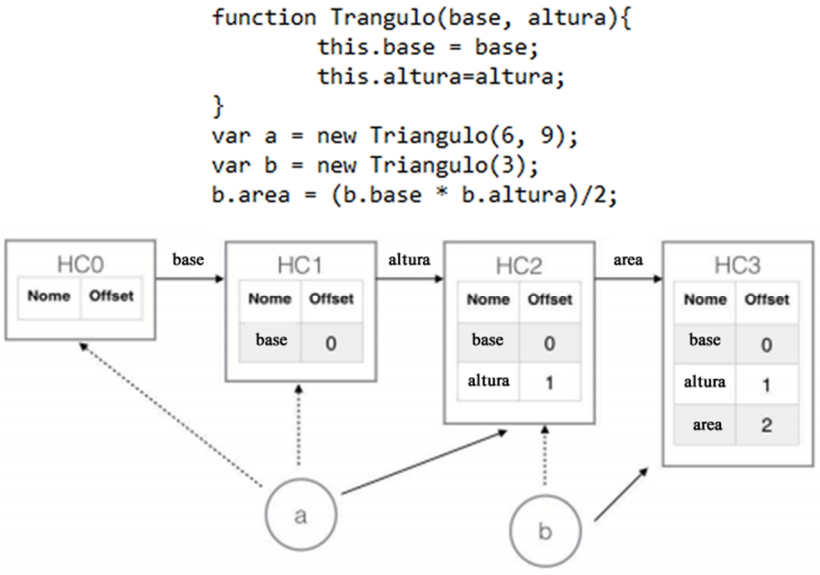

Figura 47. Exemplo de *hidden class*.

Na Figura 47 é possível entender como se dá a geração das *hidden classes* do objeto Triangulo. Durante a execução dos comandos presentes na linha 5, o comando *new* gera uma nova instância do objeto. Como não há representação da classe na memória, é criada a *Hidden Class* 0, sem qualquer propriedade definida. Ao instanciar o Triangulo, é executada a linha 2, onde é definida a propriedade base. Como trata-se de uma propriedade não existente na *Hidden Class* 0, ela é estendida gerando a *Hidden Class* 1, que recebe o valor de *offset* 0. O valor de *offset* é utilizado como índice de acesso direto da propriedade na memória. É acrescentado a *Hidden Class* 0 um ponteiro denominado base, indicando que caso algum objeto aponte para *Hidden Class* 0 receba o campo base como nova propriedade, este objeto deverá utilizar a *Hidden Class* 1. Quando a linha 3 é executada, é gerada a *Hidden Class* 2, onde é acrescentada a propriedade altura com *offset* 1. Em seguida é adicionado um ponteiro denominado altura na *Hidden Class* 1, que faz referência a *Hidden Class* 2. Retornando a execução do código, quando a variável b é instanciada como já existem classes que atendem às propriedades necessárias de b, nenhuma nova *hidden class* é criada. Assim, a variável b passa a apontar para a *Hidden Class* 2. Quando uma nova propriedade é inserida no objeto da variável b (área), a *Hidden Class* 2 deixa de atender a necessidade do código e então uma nova *hidden class* é criada, a *Hidden Class* 3. A variável b passa a apontar para ela e *Hidden Class* 2 ganha um novo ponteiro indicando para a *Hidden Class* 3.

Portanto, fica evidente o grau de reuso de código do Node.JS. Se uma nova instância for criada utilizando a classe Triangulo, as *hidden classes* criadas anteriormente serão compartilhadas. É importante ressaltar que para o conceito de *hidden class* ser de fato uma vantagem para a aplicação, deve-se declarar as propriedades dos objetos.

NodeJS *Garbage Collector*

É imprescindível o gerenciamento da memória de forma cuidadosa durante a execução de um programa, de modo a prevenir a ocorrência de inúmeros problemas, os quais incluem vazamentos de memória. O coletor de lixo do Node.js possui como principal função a identificação de áreas da memória que podem ser liberadas para uso, permitindo assim libera-las para outros processos ou reaproveita-las para alocação de outras funções do próprio programa. O *garbage collector* do Node.js é bloqueante, próprio e preciso, ou seja, ele interrompe o processamento do programa para realizar a execução de um ciclo

de coleta e processa, na maioria dos ciclos, apenas uma pequena parte da pilha de objetos (Conrod, 2013).

Valendo-se do fato de que os objetos possuem tendência a um clico de vida curta, a pilha de objetos é dividida em duas partes principais: novo espaço na memória, sendo este o local onde novas instâncias de objetos são criadas e o antigo espaço da memória, no qual objetos que já passaram anteriormente por ciclos de limpeza são armazenados (Conrod, 2013). A cada mudança de espaço realizadas nos objetos do programa, todos os ponteiros referentes aos objetos alterados são atualizados. Todo espaço de memória é constituído de páginas de alocação contíguas com espaços que podem variar entre 1 *Mega Byte* a 8 *Mega Bytes*. Tal espaço varia conforme a heurística utilizada no projeto (Maidin 2014). A designação de objetos no espaço novo de memória é realizada por um processo menos complexo, utilizando apenas um ponteiro, o qual é incrementado quando um novo objeto solicita um espaço (Conrod, 2013).

Arquitetura do NodeJS

Apesar do V8 *Engine* ser eficaz, não seria possível somente com seus recursos implementar de maneira completa um servidor, pois há a necessidade de uma interface entre suas funcionalidades e o sistema operacional. Para suprir tal lacuna, uma série de funcionalidades extras escritas em C e C++ foram acrescentadas por Dahl, em forma de camadas, complementando ou estendendo as bibliotecas presentes na V8 *Engine* (Dahl, 2009). A Figura 48 apresenta um diagrama esquemático das camadas que compõem a arquitetura do Node.js

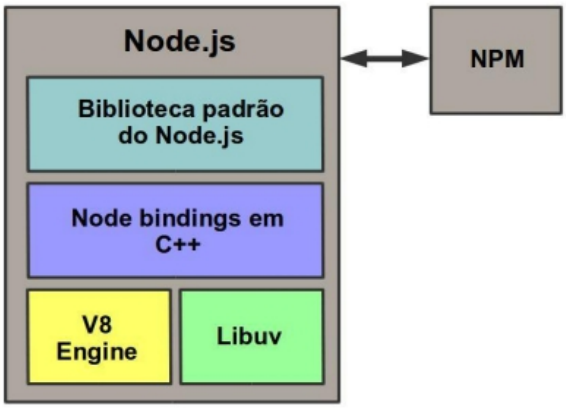

Figura 48. Arquitetura do Node.JS.

O V8 *Engine* é responsável por realizar a compilação da linguagem Javascript enquanto paralelamente uma biblioteca, denominada Libuv, gerencia o *loop* de eventos e as operações de entrada e saída assíncronas não bloqueantes, pilares fundamentais do Node.js (Conrod, 2012).

A biblioteca Libluv, escrita na linguagem C, garante a interface entre as funcionalidades do Node.js e as operações de baixo nível, como leitura e escrita, *loop* de eventos, acesso à rede, *sockets*, sistema de arquivos, dentre outras operações executadas junto ao sistema operacional (Conrod 2012). A gestão das funcionalidades da Libuv é feita internamente, através de um *pool* de *threads*, que organiza e gerencia a execução das funções em *threads*. A biblioteca Libuv juntamente com o *loop* de eventos do Node.js fornece os conjuntos de funções necessárias para a manipulação e uso de *handles* e *requests*. Objetos que possuem a função de executar operações complexas e de longa duração, como chamadas ao *callback* a cada iteração do *loop* de eventos são chamados de *handles*. Em contrapartida, objetos que possuem ciclos de vida curtos durante a execução, são denominados *requests*. Os *requests* são executados sobre um *handle* ou diretamente sobre o *loop* de eventos (Conrod, 2012).

Na camada acima da V8 *Engine* e da Libluv, encontra-se a camada denominada Node *Bindings*. Nessa camada, são disponibilizadas as Interfaces de Programação de Aplicações (API) necessárias para o desenvolvimento de aplicações com o Node.js. Um exemplo de interface disponível na camada Bindings do Node.js é a API de conexão a banco de dados. Nativamente a V8 *Engine* não fornece uma API para conexão e manipulação de banco de dados, neste caso a *binding* fornece uma interface entre a V8 e a biblioteca de banco de dados desejada. Pode-se dizer, portanto, que a camada Node *Bindings* atua entre os módulos fornecidos pela biblioteca padrão do Node.js e a interface nativa do V8 *Engine* (Conrod, 2012).

A camada situada acima da Node Bindings, denominada Biblioteca Padrão do Node.Js, armazena os módulos próprios do *framework*, todos desenvolvidos utilizando a linguagem Javascript. No momento do desenvolvimento de uma aplicação, cada módulo presente na camada de bibliotecas do Node.js necessita ser importado através do comando *require*(‘nome do modulo’). Tal comando disponibiliza para a aplicação todas as funções

do módulo importado. A aplicação é executada, então, sobre toda essas camadas, fazendo uso de todos os módulos disponibilizados pelo Node.js

Loop de Eventos do NodeJS

Dentre as funcionalidades do Node.js, o loop de eventos é a mais relevante para o correto funcionamento da plataforma. Desenvolvido com base em projetos como o Ruby Event Machine, o *loop* de eventos do Node.js é implementado pela biblioteca Libluv, que é executada em uma *thread* única e com uma estrutura do tipo FIFO (*First In, First Out*), sendo responsável por interceptar todos os eventos gerados pela aplicação e executar as funções relativas a cada evento (Tilkov e Vinoski, 2010). Quando um evento é disparado pela aplicação, o mesmo é alocado na fila de processamento. A cada iteração do *loop* de eventos, o primeiro evento da fila é executado. Durante a iteração, qualquer outro evento gerado é alocado no final da fila (Dahl, 2009). Quando a execução do evento termina, o *loop* de eventos verifica novamente a fila e inicia o processo de execução de um próximo evento. Apesar do loop de eventos ser executado em single *thread*, a execução dos eventos não necessariamente é realizada na mesma *thread*. Quando uma requisição que faz uso de algum recurso bloqueante, como uma chamada ao sistema de arquivos ou uma operação em um banco de dados, o servidor do Node.js anexa a requisição uma função chamada *callback*. A função *callback* é executada quando a requisição é finalizada. O *loop* de eventos dispara a função *callback*, executando o processamento com a resposta gerada pela requisição (Tilkov e Vinoski, 2010). A Figura 49 abaixo demonstra o esquema de funcionamento do *loop* de eventos do Node.js.

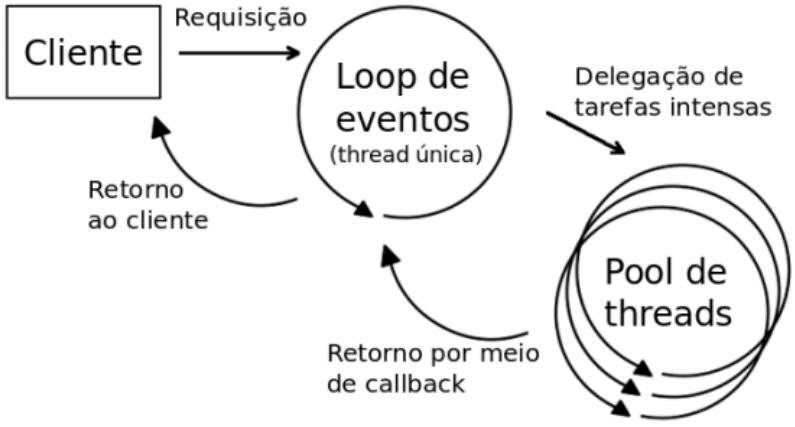

Figura 49. *Loop* de eventos do Node.JS.

Conforme observa-se na Figura 49, o *loop* de eventos é executado em uma única *thread* e fica responsável por gerenciar toda fila de eventos com as requisições geradas pela aplicação. Todas as requisições que possuem alta complexidade de processamento são delegadas para alguma outra *thread* auxiliar presente em um pool de *threads*. Quando a execução é finalizada, a *thread* realiza o retorno do processamento ao loop de eventos através da função de *callback* e fica disponível para alocação novamente no pool de *threads*. Por fim, o loop de eventos retorna as informações solicitadas a aplicação. Caso algum outro evento seja gerado durante a execução de um evento anterior, o mesmo é alocado no final da fila de eventos (Tilkov e Vinoski, 2010). O funcionamento do loop de eventos evidencia as características definitivas para a compreensão das aplicabilidades do Node.js.

Operações de entrada e saída não bloqueantes

Comumente, as operações de entrada ou saída realizadas por aplicações ou pelo sistema operacional são nativamente operações bloqueantes, isto é, quando ocorre uma solicitação para escrita ou leitura, a aplicação em execução é bloqueada e fica aguardando a resposta da operação. Em aplicações que tratam grandes quantidades de requisições de forma simultânea, o bloqueio é um problema grave, visto que até mesmo um curto período de tempo sem execução da aplicação pode gerar travamentos devido ao grande número de requisições que se acumulam aguardando tratamento.

No modelo de arquitetura no qual o Node.js foi estruturado, requisições com processos de alta complexidade, que normalmente gerariam bloqueios prolongados, são alocadas nas *threads* auxiliares disponíveis no pool de *threads* do loop de eventos, operando de maneira assíncrona. Assim, o *loop* de eventos permanece disponível para atender as demais requisições feitas pela aplicação e alocadas na fila de eventos, até que a operação de alto custo computacional seja processada completamente, enviando seu resultado para aplicação através da função de *callback* (Tilkov e Vinoski, 2010).

O trecho de código exposto na Figura 50 abaixo contém uma função para consulta em um banco de dados. Em sua execução, ao invés do Node.js bloquear a execução da aplicação enquanto aguarda a resposta do banco de dados, a operação de busca no banco é repassada ao *loop* de eventos e somente após o retorno da consulta estar disponível a função de *callback* é disparada para prosseguir com o processamento dos resultados

obtidos, atuando portanto de maneira assíncrona e não bloqueante (Tilkov e Vinoski, 2010).

```
consulta_finalizada = function (resultado) {
                    executar_algo_usando(resultado);
}
query('SELECT * FROM postagens WHERE id = 1', consulta_finalizada);
```

Figura 50. Exemplo de *callback*.

Dentre as vantagens desse modelo de execução, pode-se citar a redução considerável do uso de memória e processamento por parte do servidor. Enquanto no modelo de execução mais comumente utilizado, baseado em *threads*, existe a exigência de alocação de uma nova *thread* para cada evento iniciado, no modelo de execução do Node.js apenas a *thread* do *loop* de eventos se mantém em execução de maneira constante e novas *threads* somente são iniciadas de acordo com a necessidade. Com a menor intensidade de uso da memória e do processador do servidor, a capacidade de lidar com requisições simultâneas sofre um aumento significativo. Contudo, as vantagens e a maneira flexível de execução de um código assíncrono podem tornar-se um problema ao longo do processo de desenvolvimento de uma aplicação complexa.

Dada a maneira com que a arquitetura do Node.js foi concebida, um único evento pode disparar não somente um, mas vários *callbacks*, e cada *callback* pode disparar como resultado outras funções de retorno. Tal recorrência de funções é popularmente conhecido como *callback hell*. Para evita-lo, o desenvolvedor deve tomar o máximo de cuidado com o fluxo de execução previsto na escrita de código da aplicação (Ogden, 2012).

Biblioteca ASYNC

Uma das maneiras de prevenir a ocorrência de *callback hell* é utilizar durante a escrita de código a biblioteca denominada Async. A biblioteca Async fornece um conjunto de aplicações de implementação simples, desenvolvidas de maneira específica para lidar com a assincronicidade e controle de fluxo de execução no código da aplicação. São disponibilizadas pela biblioteca em torno de 70 funções que estabelecem padrões de fluxo de controle de execução assíncrona, tais como paralelização e serialização do código. Assim, a biblioteca facilita a programação da aplicação ao oferecer funções que

identificam o fluxo de execução, auxiliando o desenvolvedor a não cometer erros típicos gerados pela falta de domínio na programação orientada a eventos presente no Node.js, como por exemplo a utilização de variáveis sem a prévia atribuição de valores (Ogden, 2012).

NPM

O *Node Package Manager* (NPM) surgiu em 2009 como ferramenta de código aberto para realizar o gerenciamento e disponibilização de bibliotecas Javascript, visando o compartilhamento e reuso de código por parte de programadores do mundo todo. Não obstante seu desenvolvimento tenha sido pensado para instalação e uso junto ao Node.js, o NPM é um projeto independente, e na atualidade e utilizado em conjunto com outros projetos Javascript. O funcionamento da ferramenta baseia-se no encapsulamento de um módulo de código em um pacote e o envio do módulo encapsulado para registro no repositório NPM. Dessa maneira, tal módulo fica disponível para utilização por parte de outros programadores. Os pacotes registrados no repositório contém um arquivo denominado package.json, no qual são listadas todas as características e dependências do pacote, como por exemplo nome, descrição, versão, dentre outras. Através das informações a respeito dos pacotes cadastrados no repositório, é possível que um desenvolvedor faça uma busca por pacotes que atuem e solucionem problemas específicos encontrados no desenvolvimento de uma aplicação. Através da listagem de dependências encontrada no arquivo package.json, também é possível compartilhar as dependências internas do pacote, contribuindo no desenvolvimento de outros pacotes. O comando npm install, que deve ser executado na pasta da aplicação onde se encontra o arquivo package.json através do prompt de comando, permite que o Node.js faça uma varredura do projeto A partir do comando npm install, que deve ser realizado na linha de comando a partir da pasta do projeto em que se encontra o arquivo package.json, o Node.js varre o arquivo, lendo e baixando qualquer dependência nele listada. Os módulos baixados são salvos em uma pasta específica, nomeada node_modules, em que estarão disponíveis para importação no código da aplicação (NPM, 2019).

Express.js

O pacote Express.js adiciona uma camada de abstração sobre o módulo HTTP presente nativamente no Node.js, oferecendo uma sintaxe mais flexível para a construção de aplicações. O objetivo do módulo Express.js é facilitar o processo de desenvolvimento de APIs, através de sintaxes mais amigáveis para o desenvolvimento de rotas, validação de URLs e suporte a adoção de pacotes *middleware* desenvolvidos por terceiros. Assim, o desenvolvedor inicia um projeto com uma estrutura inicial sobre a qual pode desenvolver a aplicação mantendo a uniformidade do código e organização das rotas (Express.JS, 2019).

Apêndice B – Linguagem Cypher

Na sintaxe utilizada na linguagem *Cypher* para a manipulação de grafos, os nós do grafo são representados por parênteses, e.g. (n). Cada nó do grafo pode possuir um conjunto de propriedades. As arestas do grafo, que representam as relações entre os nós, são constituídas pelo conjunto de símbolos traço (-) e maior (>) formando uma "seta", e.g. (m)->(n). As arestas também podem possuir identificadores (*labels*) associadas a elas. Neste caso a identificação da aresta é representada pelo uso dos símbolos de colchetes [ ], e.g. (m)-[*label*]->(n) (Hunger e Neubauer, 2010)

Alguns dos principais comandos e recursos da linguagem Cypher, de acordo com (Panzarino, 2014) são:

- MATCH: o comando match da linguagem Cypher possui funcionalidade semelhante a do comando *select* da linguagem SQL. O comando recebe um padrão como parâmetro e busca por dados que atendam o padrão informado na consulta. É possível utilizar cláusulas para filtrar o resultado da consulta. A Figura 51 apresenta um exemplo de utilização do comando *match*.

```
MATCH (p:Pessoa)-[:GOSTA]->(c:Comida)
WHERE p.nome <> 'Maria' AND c.nome = 'Pudim'
RETURN p
```

Figura 51. Exemplo do comando Match.

- *CREATE*: o comando *create* pode ser considerado equivalente ao comando *insert* presente na linguagem SQL. Através do comando *create* que novos registros são inseridos na base de dados do Neo4J. O comando recebe atributos com dados que estarão contidos nos nós ou arestas a serem criadas. Múltiplos nós e arestas podem ser criadas em um único comando *create*. A Figura 52 demonstra o uso do comando *create*:

```
MATCH (p:Pessoa) WHERE p.nome = 'Maria'
CREATE (p)-[:GOSTA]->(:Comida { nome: 'Arroz' })
```

Figura 52. Exemplo do comando Create.

- *REMOVE*: O comando remove é similar ao comando delete da linguagem SQL. Sua função é remover nós e arestas (incluindo suas *labels* e propriedades). Pode

ser utilizado em conjunto com os comandos *match* e *return*. A Figura 53 demonstra o uso do comando remove associado aos comandos *match* e *return*:

```
MATCH (a { name: 'Andy' })
REMOVE a.age
RETURN a.name, a.age
```

Figura 53. Exemplo do comando Remove.

- *RETURN*: O comando *return* é utilizado para retornar apenas parte de uma consulta realizada com o comando *match*, sendo útil para refinar os dados a serem exibidos em tela. A Figura 54 demonstra o uso do comando *return*.

```
MATCH (n { name: 'B' })
RETURN n
```

Figura 54. Exemplo do comando Return.

- *MERGE*: A cláusula *merge* verifica se um padrão solicitado (nós) existe no grafo ou se necessita ser criado. Caso o padrão não exista, o comando *merge* cria o mesmo. É importante ressaltar que o comando não buscará por padrões parcialmente existentes, mas tentará encontrar uma correspondência exata a todo o padrão. Caso não encontre, o padrão inteiro é criado. Pode-se entender o merge como uma combinação dos comandos *match* e *create*. A Figura 55 abaixo demonstra o uso do comando *merge*.

```
MERGE (robert:Critic)
RETURN robert, labels(robert)
```

Figura 55. Exemplo do comando Merge.

- *WHERE*: Não é um comando da linguagem Cypher, mas um operador utilizado para a realização de filtros e cláusulas de critério quando combinado com o comando *match*.

```
MATCH (n)
WHERE n.age < 30
RETURN n.name, n.age
```

Figura 56. Exemplo do comando Where.

- *SET*: A cláusula *set* é usada para atualizar ou definir valores de propriedades em nós e identificadores em nós (relacionamentos).

```
MATCH (n { name: 'Andy' })
SET n.surname = 'Taylor'
RETURN n.name, n.surname
```

Figura 57. Exemplo do comando Set.

- *FOREACH*: A cláusula *foreach* é utilizada para atualizar dados em uma lista, elemento por elemento, sejam eles nós de um caminho ou resultado de agregação.

```
MATCH p =(begin)-[*]->(END )
WHERE begin.name = 'A' AND END .name = 'D'
FOREACH (n IN nodes(p)| SET n.marked = TRUE )
```

Figura 58. Exemplo do comando Foreach.

- *WITH*: divide uma consulta em múltiplas partes distintas, permitindo que as partes da consulta sejam encadeadas, utilizando os resultados como ponto de início ou critérios na consulta seguinte.

A linguagem Cypher conta ainda com um conjunto de operadores que podem ser utilizados com os comandos acima listados, possibilitando a construção de *querys* complexas de forma intuitiva, sendo esse outro motivo para a adoção da linguagem Cypher no projeto (Panzarino, 2014). A Tabela 5 apresenta os operadores disponíveis para utilização na linguagem Cypher.

Tabela 5. Operadores disponíveis na linguagem Cypher.

Tipo de Operador	Operador(es)
Operador de Agregação	DISTINCT
Operadores de Propriedades	. para acessar propriedades estáticas, [ ] para acessar propriedades dinâmicas, = para substituir todas as propriedades, += para substituir propriedades específicas.
Operadores Matemáticos	+, -, *, /, %, ^
Operadores de Comparação	=, <>, <, >, <=, >=, IS NULL, IS NOT NULL
Operadores de Comparação específicos para *Strings*	STARTS WITH, ENDS WITH, CONTAINS
Operadores Booleanos	AND, OR, XOR, NOT

Operadores de *String*	+ para concatenação, =~ para correspondência de expressões regulares
Operadores Temporais	+ e - para realizar operações exclusivamente entre durações (instantes temporais), * para operações entre durações temporais e números
Operadores de Listas	+ para concatenação, IN para verificar a existência de um elemento em uma lista, [ ] para acessar elementos dinamicamente.

Apêndice C – Código da Solução Proposta

O protótipo está estruturado em uma árvore de arquivos no padrão MVC (*Model, View, Controler*) sendo o arquivo index.js, presente na raiz do projeto, responsável por iniciar o fluxo da aplicação. Todos os módulos importados pelo servidor do Node.js, como por exemplo os *drivers* para conexão ao Neo4J, arquivos do Express.js e Walkdir se encontram no diretório node_modules. O diretório *config* contém o arquivo de configuração do módulo Express.js. O diretório *routes* contém os arquivos responsáveis por receber as requisições interceptadas pelo servidor e iniciar a busca por arquivos e seus metadados, bem como também são responsáveis por inicializar e manipular a base de dados do Neo4J. No diretório *views* encontram-se os arquivos responsáveis pela camada de visualização do protótipo. A Figura 59 apresenta a estrutura de arquivos do protótipo.

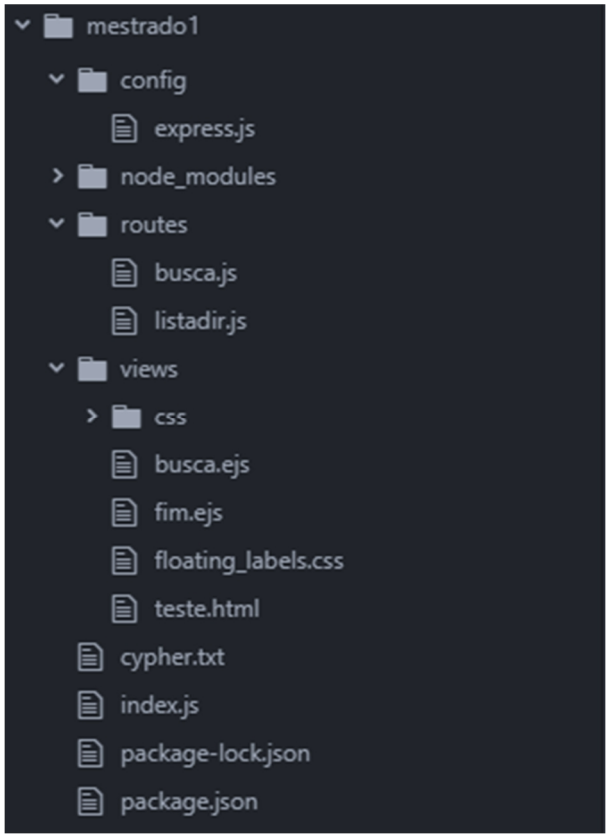

Figura 59. Estrutura de arquivos do protótipo.

A aplicação desenvolvida como protótipo está estruturada através de um servidor HTTP, que inicializa o módulo Express.js para a criação de rotas, instanciando um servidor do Node.js na porta 3000 da máquina. A partir de então, todas as requisições que forem enviadas através da porta 3000 serão interceptadas pelo servidor do Node.js,

atribuindo o destino previsto para cada uma. Uma mensagem é deixada no console da máquina onde o servidor é iniciado, indicando que o mesmo está em funcionamento corretamente. A Figura 60 apresenta o código do arquivo index.js.

```
var app = require('./config/express')()
var http = require('http');
var server = http.createServer(app);
var io = require('socket.io').listen(server);
server.listen(3000,function(){
  console.log("servidor rodando");
});
```

Figura 60. Código do arquivo index.js.

A Figura 61 abaixo apresenta o conteúdo do arquivo *package*.json, responsável por controlar todas as dependências da aplicação e mantê-las atualizadas junto ao repositório NPM. É possível observar ainda que o arquivo mantém as informações a respeito do desenvolvedor e da aplicação, sendo estes dados necessários para correta identificação caso seja feito o compartilhamento da aplicação no NPM.

```
{
  "name": "mestrado1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ricardo",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.18.3",
    "ejs": "^2.6.1",
    "express": "^4.16.3",
    "express-load": "^1.1.16",
    "neo4j-driver": "^1.6.3",
    "socket.io": "^2.1.1",
    "walkdir": "0.0.12"
  }
}
```

Figura 61. Código do arquivo package.json.

O arquivo express.js, presente no diretório *config* da aplicação, é responsável por importar e instanciar os módulos Express.js, Body-parser, Express-load e a *view engine* EJS.

O módulo *middleware* Body-parser é responsável por analisar todos os objetos de requisição de entrada, através do comando req.body, antes da execução dos *handlers* http e seus manipuladores. No protótipo desenvolvido, o Body-parser é utilizado para interceptar as requisições da visualização quanto a qual drive do sistema de arquivos deverá ser scaneado pela aplicação.

O *middleware* Express-load fornece a capacidade de carregar automaticamente modelos, rotas, esquemas, configurações, controladores, mapas de objetos e módulos do Node.js em uma variável de instância do Express.js a partir de diretórios ou arquivos especificados, facilitando no desenvolvimento de aplicações no padrão MVC e possibilitando uma separação lógica de arquivos. No projeto desenvolvido, o Express-load é utilizado para carregar na variável global *app* todo o conteúdo do diretório *routes*.

Ainda no arquivo express.js, configura-se o motor de visualização. No protótipo foi utilizada o EJS. O Embedded JavaScript (EJS) permite de maneira fácil e simples transportar dados do *back-end* da aplicação para o *front-end*, utilizando códigos Javascript no HTML das páginas de visualização, evitando o processamento de itens de visualização no *back-end*.

Todos esses módulos em conjunto com os arquivos do diretório *routes* são carregados na variável *app*, sendo esta disponibilizada ao servidor do Node.js através do comando *return*. A Figura 62 apresenta o código do arquivo express.js.

```
1  var express = require('express');
2  var bodyParser=require('body-parser');
3  var load = require('express-load');
4  module.exports = function(){
5    var app = express();
6    app.set('view engine','ejs');
7    app.use(bodyParser.urlencoded({extended: true}));
8    load('routes').into(app);
9    return app;
10   }
```

Figura 62. Código do arquivo express.js.

O arquivo busca.js faz a listagem dos *drives* de disco presentes na máquina onde o protótipo será executado, para que seja escolhido pelo usuário o *drive* de disco onde a busca de arquivos e metadados será processada. O arquivo recebe como parâmetros de uma função todo o conteúdo da variável *app* definida no arquivo express.js. Com os *middlewares* definidos pela variável *app*, o arquivo busca.js implementa uma escuta para a rota padrão do servidor ('/'), que ao ser chamada na visualização dispara a execução de uma função de *callback*. A função de *callback* inicialmente importa a biblioteca *child_process* e a armazena na variável denominada exec. A biblioteca *child_process* é utilizada quando há a necessidade de utilizar programas ou comandos nativos do sistema operacional, criando para isso um processo paralelo a aplicação no processador, porém com uso de memória compartilhada com o processo da aplicação. Através da variável exec, o comando de sistema wmic get logical disk getcaption é executado, obtendo como resposta uma função *callback*, que pode retornar ou erros de aplicação ou a listagem dos *drives* registrados no sistema operacional. Ao retornar os *drives* de disco presentes no sistema operacional, são executadas funções de *String* da linguagem Javascript, para eliminar caracteres indesejáveis e manter apenas o rótulo lógico de cada um dos discos encontrados. Ao final, a variável contendo os rótulos de discos é enviada para renderização na página busca.ejs.

```
module.exports = function(app){
app.get('/',function(req,res){
  var exec = require('child_process').exec;
       exec('wmic logicaldisk get caption', function(err, stdout, stderr) {
           if(err || stderr) {
               console.log("erro ao abrir o diretório raiz" + err + stderr);
               return;
           }else{
             var discos = stdout.slice(7).trim();
             discos = discos.split('\r\r\n');
             discos=JSON.stringify(discos).replace( /\s/g, '' );
             discos=JSON.parse(discos);
             console.log(discos);
             res.render('busca.ejs',{'lista':discos});
           }

       });

   });
```

Figura 63. Código do arquivo busca.js.

Após o usuário escolher em uma lista presente na página busca.ejs em qual disco de armazenamento a busca por arquivos será executada, o código do arquivo listar.js é executado. A variável opções recebe o rótulo do *drive* de disco e as opções de busca selecionadas na visualização pelo usuário, como por exemplo a profundidade de níveis hierárquicos que a busca de arquivos deve percorrer na árvore de diretórios. A variável buscar é responsável por carregar o código do arquivo listadir.js.

Em seguida, o método buscar presente na variável busca é executado, recebendo como parâmetros as opções de busca, o disco onde será realizada a busca e uma *callback* que será executada como resposta após a finalização do processo.

```
21  app.post('/listar',function(req,res){
22        var opcoes =req.body;
23        var buscar=require('./listadir');
24        buscar.busca(opcoes.profundidade,opcoes.discos,res);
25        console.log(opcoes.discos);
26    });
27  app.get('/exibir',function(req,res){
28      res.render('fim.ejs');
29
30    });
31  }
```

Figura 64. Rotas listas e exibir.

O arquivo listadir.js contém todo o código responsável pela busca de metadados e gravação dos mesmos em uma base de dados de grafos do Neo4J. Inicialmente na codificação são carregadas as bibliotecas necessárias para o correto funcionamento da busca, o *driver* de conexão entre o Node.JS e a base de dados e é feita a conexão com o servidor do Neo4J, como pode ser observado na Figura 64. A biblioteca Walkdir possui código *open-source* sob a licença MIT, podendo ser instalado através do NPM. A biblioteca foi desenvolvida para percorrer recursivamente uma árvore de diretórios, emitindo eventos com base no que encontrar. Como realiza a caminhada pelo sistema de arquivos de forma recursiva, possui um alto desempenho.

```
var walk = require('walkdir');
var neo4j = require('neo4j-driver').v1;
var driver = neo4j.driver("bolt://localhost", neo4j.auth.basic('neo4j', '1234'));
var session= driver.session();
const pasta= require('path');
```

Figura 65. Bibliotecas e conexão do arquivo listadir.js.

Ainda no arquivo listadir.js, a função busca recebe como parâmetros a profundidade de busca definida pelo usuário na interface de visualização, o disco escolhido para busca e uma *callback* que é disparada após a execução completa da função. Em seguida são instanciadas as variáveis utilizadas durante a execução da função. A variável *emitter* inicia a execução da biblioteca Walkdir através da função walk.sync, que estabelece a caminhada através da árvore de diretórios será sequencial e síncrona, garantindo assim que todos os subdiretórios e arquivos sejam lidos pela função. A função walk.sync retorna, a cada diretório ou arquivo verificado duas respostas: o caminho do diretório ou arquivo através da variável path e no caso de arquivos, retorna seus metadados através da variável *stat*. É através da variável *path* que o aplicativo obtém a extensão do arquivo lido, ficando esta armazenada na variável tipo, e obtém também o *array* com o caminho completo a partir do disco selecionado até o arquivo lido, ficando este armazenado na variável arquivo_array. A variável *param* armazena todos os metadados de arquivos, deixando-os preparados para posterior gravação na base de dados do Neo4J.

```
exports.busca = function(profundidade,discos,res){
  var raiz=discos+"\\";
  var i =0;
  var j=0;
  var tipo, datac, datam, arquivo, arquivo_array, tamanho;
  var depth = profundidade;
  var emitter = walk.sync(raiz,{follow_symlinks: false, no_recurse: false,max_depth: profundidade, track_inodes: true },
  function(path, stat) {
  tipo = (path.split(".").pop());
  if(tipo!=path){
    i++;
    datac=stat.birthtime;
    datam=stat.mtime;
    tamanho=stat.size;
    arquivo_array = path.split('\\');
    arquivo=arquivo_array[(arquivo_array.length-1)];
    var param={
      'tipo':String(tipo),
      'arquivo':String(arquivo),
      'datac':String(datac),
      'datam':String(datam),
      'tamanho':String(tamanho),
      'unidade': String(discos)
    }
```

Figura 66. Função de busca presente no arquivo listadir.js.

A função executa uma *query* Cypher para cada situação encontrada durante a busca de arquivos, sendo as mesmas: Arquivos encontrado na raiz do disco, diretórios encontrados na raiz do disco, arquivos encontrados em diretórios, subdiretórios. Após a execução de cada *query*, é feita a verificação de um contador (j). Se o contador possuir a mesma quantidade de diretórios e arquivos verificados então redireciona-se o navegador para a página de visualização da estrutura de diretórios e arquivos encontrados no disco.

```
if((arquivo_array.length)==3){
  let query = "MERGE (d:Drive { unidade: {unidade}}) MERGE (a:Arquivo { nome:{arquivo}, criação:{datac}, modificação : {datam},
  session
  .run(query,param)
  .then(function(result){
          j++;
          console.log('gravou no banco',path);
          if(j==i){
              res.render('fim.ejs');
          }
    })
  .catch(function(error){
    console.log(error);
  });
}else{
  diretorio=arquivo_array[(arquivo_array.length-2)];
    var param={
    'tipo':String(tipo),
    'arquivo':String(arquivo),
    'datac':String(datac),
    'datam':String(datam),
    'tamanho':String(tamanho),
    'unidade': String(discos),
    'diretorio':String(diretorio),
  }
  let query = "MERGE (p:Pasta { nome: {diretorio}}) MERGE (a:Arquivo { nome:{arquivo}, criação:{datac}, modificação : {datam},
  session
  .run(query,param)
  .then(function(result){
```

Figura 67. Função de gravação de metadados na base do Neo4J.

A visualização dos arquivos e diretórios armazenados na base de dados do Neo4J, bem como a visualização das opções de busca são feitas através de uma página *web*, facilitando a compatibilidade da aplicação entre os diversos sistemas operacionais existentes, visto que é necessário apenas um navegador para a exibição dos resultados.

Após a execução da aplicação, o navegador exibe uma página *web* onde são listados os discos existentes no computador onde a aplicação está sendo executada. A Figura 68 abaixo apresenta a codificação da página inicial da aplicação.

```
<body>
  <script src="//code.jquery.com/jquery-3.3.1.min.js"> </script>
  <form class="form-signin" action="/listar" method="post">
    <div class="form-group">
        <label for="exampleFormControlSelect1">Discos Encontrados</label>
        <select class="form-control" id="exampleFormControlSelect1" name="discos">
          <% for(var i=0; i<lista.length; i++){ %>
          <option value="<%=lista[i]%>"><%=lista[i]%></option>
        <%}%>
        </select>
      </div>
      <div class="form-group">
        <label for="pb">Profundidade da Busca a partir da raiz</label>
        <input type="text" name="profundidade" id="pb" required maxlength="2" class="form-control" />
      </div>
    <button class="btn btn-lg btn-primary btn-block" type="submit">Buscar</button>
    <a href="/exibir" class="btn btn-lg btn-primary btn-block">Exibir    </a>
    <p class="mt-5 mb-3 text-muted text-center">&copy; 2017-2018</p>
  </form>
```

Figura 68. Codificação da página index do protótipo.

Anexo I – Comprovante de Submissão de Artigo

O artigo derivado deste trabalho de dissertação de mestrado foi submetido inicialmente ao congresso The International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises - WETICE 2020, a ser realizado em Bayonne, França. Após dupla revisão as cegas o artigo não foi aceito para publicação no congresso por um dos avaliadores. O mesmo foi então revisado de acordo com as sugestões dos avaliadores e submetido novamente para o congresso 12th The International Conference on Information Visualization Theory and Applications – IVAPP 2021, a ser realizado online, entre os dias 8 a 10 de fevereiro de 2021, congresso este cuja temática de publicação é mais adequada para o tema da presente dissertação.

Segue abaixo o comprovante de submissão do artigo (Figura 69). O resultado da dupla avaliação às cegas será publicado em novembro de 2020.

Gmail **Ricardo Leardini Lobo <ricardoleardinilobo@gmail.com>**

IVAPP 2021 - Paper Submission

1 mensagem

PRIMORIS Information System <system@insticc.org> 15 de setembro de 2020 13:19
Para: ricardoleardinilobo@gmail.com

Dear Prof. Ricardo Lobo,

Thank you for your submission. We would like to inform you that your paper has been successfully submitted to IVAPP:

Title: Hierarchical file viewer through metadata
Paper number: 9
Paper Area: General Data Visualization
Uploaded on Date/Time: 2020-09-15 5:19:00 PM
Submission type: Regular Paper
Co-Authors: Eduardo Javier Huerta Yero and Marta Ines Velazco Fontova
---

To access the IVAPP conference management system, please go to the following address: https://www.insticc.org/Primoris/ and enter your username and password.
We recommend that you use this possibility now, to check that all submitted information is correct. By using your author's area in the conference management system you have access to several options including re-submitting the paper or updating the list of co-authors. All co-authors will have access to all relevant information concerning the paper.

Please note that all papers submitted to IVAPP will be peer-reviewed by at least two reviewers from the international program committee, who are independent of the conference editorial board.

Best regards,
Ana Rita Paciência
IVAPP Secretariat

This message has been sent automatically, so DO NOT reply to this email. If you need help please contact: ivapp.secretariat@insticc.org

Figura 69 – Comprovante de Submissão de Artigo

Anexo II – Artigo Submetido

Hierarchical file viewer through metadata

Ricardo Leardini Lobo[1], Eduardo Javier Huerta Yero[1] and Marta Inez Velasco Fontova[1]
[1]*UNIFACCAMP, Campo Limpo Paulista, Brasil*
ricardoleardinilobo@gmail.com, huerta@cc.faccamp.br, marta.velazco@gmail.com

Keywords: Data Evaluation, Hierarchical Data Visualization, File Storage, Graphics Trees

Abstract: The amount of electronic data available for analysis grows every day. It is a fact that this data is scattered across various platforms and formats, such as the Internet, where millions of documents can be accessed. Common desktop users can have thousands of files and documents stored on their systems, and one must also consider the large data storage servers used by large corporations where hundreds of millions of files are stored. In an effort to deal with the difficulty of understanding what is actually stored in a hierarchical file system and making decisions, more and more users and corporations are turning to building tools for didactic analysis and visualization of information through metadata. Determining which files have been stored and how the storage system is structured hierarchically is of vital importance to business or academic institutions. Understanding the nature of the stored data and how it is used allows you to strategize to acquire new storage systems when they are needed. It is possible to evaluate and discard or store in less expensive storage media files that are not frequently used, detect data duplication, meet compliance requirements, among others. This article analyses some of the data visualization tools currently available and proposes as a result the construction of a file visualization tool through metadata organized in a hierarchical structure in the form of a tree, in order to allow in a simplified way decision making about files allocated in large-scale storage systems.

1 Introduction

Storage solutions over the past few years have naturally migrated from on-premise and off-premise storage to online storage, given the emergence and growth of cloud computing. New paths have been opened in the area of information technology, such as: Big Data, Corporate Mobility and solutions for storing large amounts of files and information data on remote servers. This has helped and is helping data ecosystems to become increasingly interconnected, generating increasing demand for digital storage capacity (Seagate,2019). In 2017 the overall size of the cloud services market was $291 million and it is expected to reach $983 million by the end of 2025, with an annual growth rate of 16.4% over the period 2018-2025 (Reuters, 2018). The Cloud Storage Market report, published by (Markets, 2018) estimates that the cloud storage market is expected to grow 23.7% per year, reaching an added value of $88.91 billion by 2022. Demand for cloud storage is driven by many factors, such as the growing adoption of hybrid cloud

storage and the growing need for enterprise mobility. As technology and demand for storage capacity increase and consumers increasingly value a combination of options such as high performance, economy and high availability as the ideal storage solution.

The use of computing technologies for mass storage not only offers cost and availability benefits, but also presents unsolved problems that can directly interfere with cost, security and efficiency in the search for information and files. Information can be considered the life force not only of modern business, but also of modern life. Large corporations and organizations currently deal with petabytes of information, which grows year by year (Markets, 2018). Computer systems currently store large amounts of data at all times. Second (Markets, 2018) over the next three years, more data and files will be generated than during the entire previous human history. Much information is captured and stored automatically by sensors and monitoring systems. Many of the simple transactions that are now part of our daily lives, such as paying for food and clothes with a credit card or using the telephone, are usually recorded for future reference by computers. The exacerbated generation of data and archives today means that mass storage systems are hired only to take care of the storage and preservation of such archives and data. However, after some time, with a large mass of files and data stored, finding valuable information is difficult. File discovery is an intense work process, involving the collection and analysis of hundreds or thousands of data from stored information. More than that, knowing exactly what has been stored and optimizing decision making processes and storage expenditures becomes almost impossible without a graphical data visualization tool, since most current file and data storage systems do not allow you to visualize files and metadata in a simplified and effective way.

Issues such as analysis of stored data quality, duplicity and file integrity are just some of the problems that contribute to the storage of large amounts of files and data on external servers becoming a problem for organizations. The use of automated methods can help identify some anomalies, but the determination of what constitutes an error depends on the context of each file, thus requiring a human evaluation of the stored content (Kandel, Parikh,Paepcke and Hellerst, 2018). For this reason, analyzing and understanding the files and information that has been stored has become an increasingly difficult task, and although there are some visualization tools to facilitate the analysis of files and data by implementing data assessment policies, analysts often need to manually build the

necessary views, which require time and a significant level of knowledge in the technologies employed. Finding and correcting data quality problems can also be a task with a high financial cost: it is estimated that the evaluation and cleaning of duplicate or inconsistent data is responsible for up to 80% of the cost of data hosting projects (Kandel, Parikh,Paepcke and Hellerst, 2018).

2 RESEARCH METODOLOGY

In order to obtain materials aiming to obtain consistent content for the theoretical basis of this article, searches were made on the following academic paper bases: Google Academic, IEEE Xplore, ACM DL and Springer Link.

The base searches were performed using the following keywords in English: data assessment, metadata search, large-scale storage systems, data quality assessment, hierarchical view, data visualization and hierarchical data view. In the Portuguese language the keywords used were: data evaluation, hierarchical data visualization, metadata search. There were also searches for articles in Google search engine, focusing on finding commercial solutions related to the prototype proposed in this article. The following keywords were used to find commercial solutions: data assessment and data visualization.

The article selection process was carried out following these steps:

1) Pre-selection of articles based on the analysis of titles and reading of abstracts.

2) Full reading of pre-selected articles. At this point, articles whose content was not relevant to the main focus of the search (keyword used to search and obtain the article) were discarded.

3) Synthesis and evaluation of the results obtained by reading the articles with contributions considered relevant to the theoretical basis of this article.

3 DATA VISUALIZATION

A variety of new paradigms and visualization frameworks have been developed in recent years. However, achieving flexible views of information contained in large storage spaces, i.e. pre-processing large amounts of information, displaying information context, and supporting a variety of exploration filters are still considered recurrent problems in the area of computer research (Keim, 2001).

A visual representation provides a degree of ease in the analysis of the information it contains much greater than in numerical or textual type representations. This leads to a strong demand for visual exploration techniques and tools and makes them indispensable in conjunction with automatic exploration techniques (Keim, 2001). The term visualization can be conceptualized in a general way, according to (Few, 2009) as the visual representation of information, and this term can be directly associated to three other terms with slightly different meanings, being them: Data Visualization, Information Visualization and Scientific Visualization. According to (Few, 2009), the term Data Visualization can be used to cover the various types of visual representations that support data exploration and analysis. The terms Information Visualization and Scientific Visualization can be considered as sub terms of Data Visualization. Both refer to specific types of representations (Few, 2009).

The visualization of hierarchical information structures is an important topic in the archive and metadata visualization research (Keim, 2001). Most research in this area focuses on the challenge of displaying large hierarchies in an understandable way. A file hierarchy can be represented as a tree. It is often necessary to navigate through the file hierarchy to find a specific file. Anyone who has done this has probably experienced some of the problems involved in viewing graphics: "Where am I?" Where's the file I'm looking for? (Bundt, 2018).

The display of information in hierarchically organized structures such as family trees, organizational charts, file systems, and class diagrams can usually be represented by structures consisting of edges and nodes. However, in addition to the display of data, users and analysts need to explore the data visualization to extract relationships between the information. Second (Tukey, 1977) and (Bertin, 1981), exploratory data analysis and the need for graphic applications to support this process led to the development of many visualization techniques, where visualization design plays as relevant a role in data analysis as the interaction mechanisms provided to analysts. In other words, a didactic

visualization of the data is just as important for an analyst as the analysis techniques employed in the information gathering (Cava, Luzzardi, Freitas, 2003).

Over the last decades, several visualization techniques have been developed to allow navigation and analysis in sets of information displayed in a hierarchical manner. Among the forms of visualization we can highlight: Tree Map (Johnson, Shneiderman, 1991) (Shneiderman, 1992), Cone Trees (Robertson, Mackinlay, Card, 1991) and the Hyperbolic Navigator (Lamping, Rao, Pirolli, 1995). One can also cite the use of graphs, represented by edges and nodes to represent a file system or the page structure of a site (Lamping, Rao, Pirolli, 1995), (Munzner, 1997). These visualization techniques convey a better perception of some attributes or provide additional interaction capabilities in data evaluation, as they provide a visual representation of the entire information space, as well as a detailed view of some selected item or region of interest. Each of the various visualization techniques aimed at exploiting existing hierarchically arranged information tries to incorporate extra resources to improve the user's perception, in order to achieve the objective of supporting the process of data assessment or decision making (Cava, Luzzardi, Freitas, 2003).

One of the best known tree views is the rooted tree (Holten, 2006). The root tree, as can be seen in Figure 1, is an example of a tree visualization based on the intuitive representation of nodes: the relationship between parent and child nodes is represented by lines that interconnect them. The layout, viewed from top to bottom, positions child nodes below their respective parent node, this being the most common root tree layout.

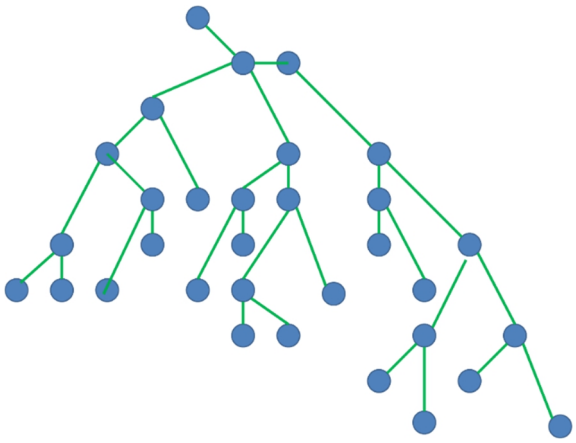

Figure 1: The root tree.

4 ANALYSIS OF EXISTING SOLUTIONS

There are a number of similar graphical visualization tools that share the common file system information visualization focus. Below is a list of some of the similar software, along with brief descriptions of each product.

4.1 Freeware or Open Source Solutions

- The Sleuth Kit software is a collection of command line operated tools aimed at examining disk images. The software functions are expandable via modules. Sleuth Kit scans the operating system file system to find hidden or deleted files, allowing you to examine the disk layout and extract partitions. Can display all metadata and file attributes, hashes and details of the metadata structure (The Sleuth Kit, 2019).
- The Autopsy application is a free file and metadata analysis tool used for forensic analysis. Its main purpose is to fight cyber crime, but it can be used to recover personal data. Allows you to compile usage reports about when certain events occurred on your computer, locate corrupted, hidden or dangerous files, recover deleted files from unallocated space, and extract metadata from images and search for threats. Autopsy is The Sleuth Kit's graphical interface and displays the data collected from TSK commands (Autopsy, 2019).
- The File System Visualizer software is aimed at viewing the files contained in directories on a 3D map. It is visualized where the files are stored in memory, displaying the directories as rectangles, and the files are visualized as blocks inside the directories. The blocks are proportional in size to the storage space used by the file. There is also an interface in the form of a 2D hierarchical tree, displaying the directories and their subdirectories (3D File System Visualizer, 2019).
- Windows Directory Statistics (WDS), also known as Win DirStat, is a program that scans the Windows file system creating a list of directories and subdirectories in tree map format and a list of extensions. It displays a visual representation of memory usage. The directory list and tree map visually display the files and directories, while the extension list acts as a descriptor to show details about the filesystem and its components. In tree map, each rectangle represents a different file and is proportional to the storage space used by the file. Likewise, directories are formed from these rectangles and include subdirectories and other files. The colors represent the type of file (WinDirStat, 2019).

4.2 Commercial Solutions

- The Data Insight software developed and licensed by Veritas has as main objective to increase the governance over stored data, thus reducing storage costs. The tool indexes the data stored on a disk through its metadata, allowing the user to view information from the stored files through filters such as file creation date, access permissions, date of last modification, among others. A demo version of the software can be accessed at https://riskanalyzer.apps.veritas.com/#!/admin/analyze. Veritas Data Insight software has a license cost of approximately US$ 30.00 per machine.

5 PROPOSED ALTERNATIVE

The solution proposed as a result of this article aims to be a viable alternative to the other existing free solutions. When comparing the other free solutions for use presented previously, it can be observed that it is the only one that presents the information on screen in a rooted tree format, making clear the hierarchy of the files and directories contained in the researched storage space. This is a characteristic considered a key point of the application, because it reduces the user's learning curve, since the hierarchical

representation through a tree representing files and directories is the type of visualization that is closest to the way files and directories are navigated in most modern operating systems commonly used by the market, such as Microsoft Windows, Linux in its various distributions, Mac OSX, Android, among others.

The information contained in a tree view tends to be easier to assimilate by the user when compared to the information contained in a tree view – the predominant view format in the other softwares compared in this article. This fact is corroborated by (Laubheimer, 2019) research which states that tree maps are a complex type of data visualization and that when observing it, it is not possible to obtain a quick understanding of what is being displayed, this being a problem since the main requirement for any type of information presented in a panel is its quick understanding. Also according to studies conducted by (Zhao, Mcguffin, Chignell, 2005) it is possible to clearly identify a topology or hierarchy in a tree diagram, with the disadvantage that the nodes are usually unevenly distributed. In a tree map, however, the visualization space is used efficiently, but the form of display is less familiar and more difficult to interpret when compared with a tree map.

5.1 Proposed Solution Architecture and Technological Choices

The application is structured in a client-server architecture, based on a web browser to perform the request and display the response of the application server, which is responsible for manipulating the file system and the database where the information regarding the metadata of the files found will be stored. The implemented client-server architecture can be visualized with three distinct levels. In the three-level architecture, there is an intermediate level shared between a client, that is, the computer requesting resources, equipped with a user interface (browser) in charge of the presentation and an application server (also called middleware). The task of the application server is to provide the requested resources by interacting with the file system, providing the necessary data to the database server. Figure 2 presents a three-level model of the client server architecture used by the application.

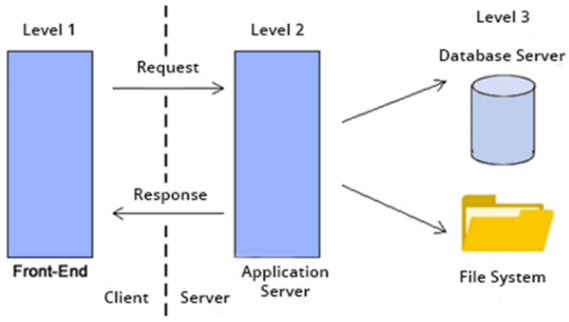

Figure 2: Proposed Solution Architecture.

The data model adopted in the project corresponds to a graph in the form of a rooted tree, which is a related graph (there is a path between any two of its nodes) and acyclic (it has no cycles). The root node of the generated graph tree is the logical unit of the file system where the search was performed. Each subsequent node represents a subdirectory. The depth of a node is considered to be the distance from the node to the root node. A set of nodes with the same depth is called a tree level. The greatest depth of a knot is the height of the tree. Each level of the tree has as nodes the subdirectories found internally in the directory represented by the node that is one level up.

Three technologies were initially considered for the development of the application as a result of this article, being them: PHP language with Apache server support, PHP language with Nginx server support and Node.js. Although the PHP language has more simplified syntax when compared to the syntax used in Node.js, it has had considerable drawbacks, such as performing blocking input and output operations and a disadvantage in computational performance. For these conclusions, the study “Is Node.js a viable option for building modern web applications? A performance evaluation study.” by Chaniotis, Kyriakou e Tselikas was considered (Chaniotis, Kyriakos, Tselikas, 2014).

The Neo4J database management system was chosen as the application database, due to its storage architecture and performance. Neo4j does not implement a scheme in the same sense that relational databases do. The definição of entities (nodes), relationships and properties is not definida in way fixa in the management system of the database such as tables, foreign keys, etc. in relational banks. Thus, the definitions of the nodes, relationships and their properties are definidas and maintained - optionally - by the application using the database. For this choice, the study “Comparison between MySQL and Neo4J for Accessing Complex Data Using Declarative Languages.” by Homrich e Mergen was considered (Homrich, Mergen, 2018).

6 RESULTS

The result application runs on a local server. The home page (Figure 3) presents a form where the user is allowed to choose on which disk of the system he wants to perform the search and the depth (number of levels of the hierarchical tree of the file system) that the application must go through. There is also the option for the user to go straight to the view of the directories and files found in a previous search, choosing the colors of nodes that represent directories and files.

If the user of the application chooses the search option, all the directories and files on the selected disk will be scrolled to the specified depth. If the depth field is left unfilled, then all files and directories on the disk will be scrolled. At the end of the execution, the data flow redirects the browser to the display page, even redirection that is performed if the user chooses the display option.

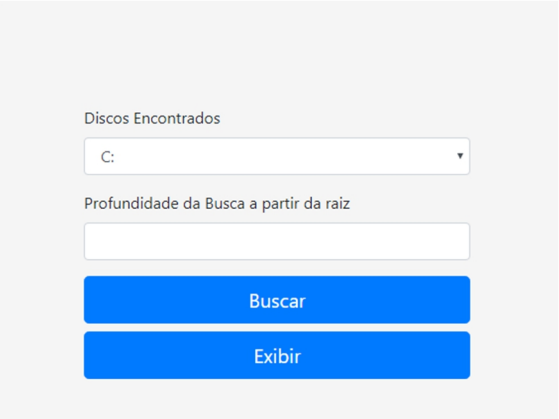

Figure 3: Application Index

When loaded, the page displays in the form of a hierarchical tree the disk on which the search was performed, this being the root node of the graph, as well as the directories and subdirectories contained in the disk, which are presented in a hierarchical manner as sublevels of the graph. The display is in the form of a hierarchical tree, but the visual distribution of the nodes from the root is done automatically in order to make the best use of the browser space, due to the large amount of directories and files. From the root are broken the edges that connect to the directories contained in the root. From these, the edges that connect to the subdirectories and files break. The hierarchical tree view can be controlled with the mouse, and you can move the tree with the mouse pointer and zoom in/out of the view. Figure 4 shows the layout of the view page in general.

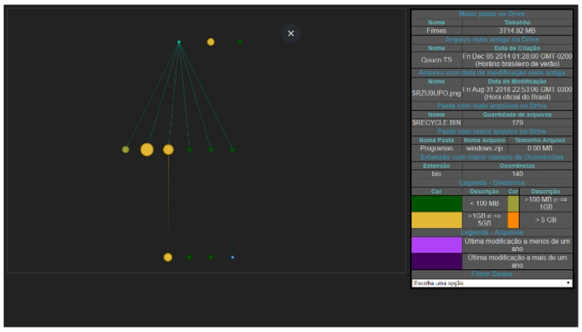

Figure 4: Layout of the view page.

Directories and subdirectories are presented as nodes of different sizes, and this difference represents the difference in disk size of each directory and subdirectory, and the color of the node also indicates its size. The legend displayed in the frame to the right of the preview page details the color scheme used for the directories and subdirectories together with their respective meaning. Just below each node is the name of the directory. Figure 5 shows in detail the difference in size and colors of the nodes representing directories and subdirectories.

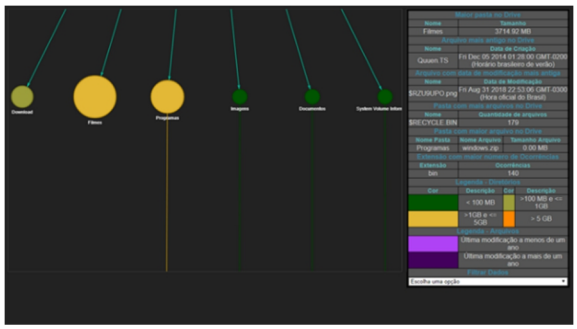

Figure 5: Node Size and Colors.

When the mouse pointer is positioned over one of the nodes representing a subdirectory or directory, a pop-up window containing the directory name and the exact size it occupies on disk is displayed.

The nodes representing the files also have different sizes, representing the difference in size that they occupy on disk. The knots also have different colors compared to the knots that represent directories. With this, it is possible to identify files that are frequently used and files that are unused. Just below each node the file name and its extension are displayed. When you position the mouse pointer over a node that represents a file, a pop-up window appears. This window displays the size information that the file occupies on

disk and its last modification date in detail. Figure 6 shows the pop-up window that appears when you position the mouse pointer over a file node.

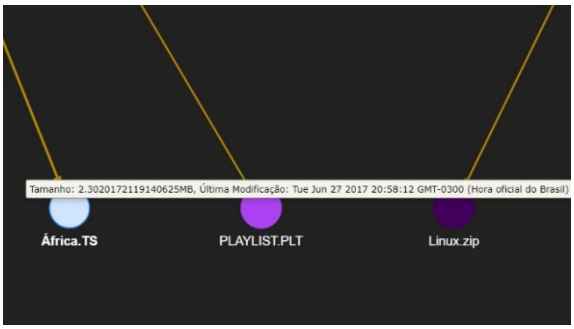

Figure 6: Information Pop-Up.

7 FINAL CONSIDERATIONS

The use of a file metadata visualization tool is a very potential resource, both in issues related to assisting in making decisions about information and in data assessment operations, being extremely useful for companies and users who have large amounts of files stored in systems that do not have a visual tool for analysis of what is actually stored. Considering that the use of storage space grows daily, being able to optimize such use can contribute to the economy of financial and computational resources.

In the prototype developed in this article, the elements are more spaced out and the hierarchy between them becomes visually explicit to the user. The detailing of each element (directory or file) is only present when the user passes the mouse pointer over it.

Another point to highlight between the existing software and the prototype developed in this article is the one concerning colors. In none of the other researched softwares the user has the option to define which colors he wants in the display of directories and files. This fact can be considered a disadvantage, given that the use of diversified colors is fundamental for a graphic visualization tool. In this way, allowing the user to choose in a personalized way the colors for the visualization of directories and files allows the user to configure more attractive colors to highlight items that interest him the most and less attractive colors for items that are of less interest at the time of visualization according to their subjectivity. It should be noted, however, that the practical application of the tool obtained as a result of this article in large storage systems is not yet somewhat trivial, but

needs improvement, currently unavailable due to the need for further improvement and development of the technologies used.

REFERENCES

Seagate. 2019. *A demanda por dispositivos de armazenamento em um mundo de dados conectado.* https://www.seagate.com/br/pt/tech-insights/demand-for-storage-devices-in-connected-world-master-ti/.

Reuters. 2018. *Global Cloud Services Market and Cloud Storage Industry Size, Share, Growth Statistics, Industry Analysis 2018 and Forecast to 2025.* https://www.reuters.com/brandfeatures/venture-capital/article?id=59412.

Markets, Markets and. 2018. "*Cloud Storage Market by Type*." https://www.marketsandmarkets.com/Market-Reports/cloud-storage-market-902.html

Kandel, Sean, Ravi Parikh, Andreas Paepcke, e Joseph Hellerst. 2012. *"Profiler: Integrated Statistical Analysis and Visualization for Data Quality Assessment."* International Working Conference on Advanced Visual Interfaces., 547-554.

Keim, Daniel A. 2011. *"Visual Exploration of Large Data Sets."* Communications of ACM 44. 38-44.

Few, Stephen. 2009. "*Now you see it: simple visualization techniques for quantitative analysis."* Analytics Press.

Bundt, Jacob. 2018. *"A graphical file system visualization tool for operating systems."* University of Northern Iowa UNI ScholarWorks. https://scholarworks.uni.edu/hpt/348/.

Tukey, J.W. 1977. "*Exploratory data analysis".* Addison Wesley.

Bertin, J. 1981. *Graphics and Graphic Information Processing.* New York: Walter de Gruyte.

Cava, Ricardo, Paulo Roberto Gomes Luzzardi, e Carla Dal Sasso Freitas. 2003. "*The Bifocal Tree: a Technique for the Visualization of Hierarchical Information Structures".* Pelotas.

Johnson, B, e B. Shneiderman. 1991. *"Tree-maps: A space-filling approach to the visualization of hierarchical information structures."* IEEE Visualization'91. San Diego: IEEE, 284-291.

Shneiderman, B. 1992. *"Tree visualization with Treemaps: a 2d space filling approach."* ACM Transactions on Graphics, 92-99.

Robertson, G., J. Mackinlay, e S. Card. 1991. *"Cone Trees: Animated 3D Visualizations of Hierarchical Information."* CHI'91 ACM Conference on Human Factors in Computing Systems. 189-194.

Lamping, J., R. Rao, e P. Pirolli. 1995. *"A Focus+Context Technique Based in Hyperbolic Geometry for Visualizing Large Hierarchies."* CHI'95 ACM Conference on Human Factors in Computing Systems. 401-408.

Munzner, T. 1997. *"H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space."* IEEE Information Visualization'97. Phoenix: IEEE Computer Society, 2-10.

Holten, Danny. 2006. *"Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data."* IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, 741-748.

The Sleuth Kit. 2019. https://www.sleuthkit.org/sleuthkit.

Autopsy. 2019. https://www.sleuthkit.org/autopsy/.

3D File System Visualizer. 2019. http://fsv.sourceforge.net/.

WinDirStat. 2019. https://windirstat.net/.

Laubheimer, Page. Nielsen Norman Group. 2019. https://www.nngroup.com/articles/treemaps/

Zhao, Shengdong, Michael J. Mcguffin, e Mark H Chignell. *"Elastic hierarchies: Combining treemaps and node-link diagrams."* EEE Symposium on Information Visualization, 57-64.

Chaniotis, Ioannis K., Kyriakos-Ioannis D. Kyriakou, e Nikolaos D. Tselikas. 2014. *"Is Node.js a viable option for building modern web applications? A performance evaluation study."* Springer-Verlag Wien 2014. 1023-1044.

Homrich, Émerson P., Sergio L. S. Mergen. 2018. *"Comparação entre MySQL e Neo4J para o Acesso a Dados Complexos Usando Linguagens Declarativas."* XIV Escola Regional de Banco de Dados . Rio Grande, 32-41.