criado não necessita de uma lógica que requer conhecimento específico como são os bancos de dados relacionais e sim da compreensão e representação gráfica (desenho) do problema (Robinson; *et al*; 2013).

Durante a implementação é necessário atribuir informações específicas aos vértices e arestas. Essas informações são conhecidas como atributos ou propriedades. Desta forma, para que uma pesquisa (*traversal*) possa obter uma maior precisão e desempenho é aconselhável nomear os relacionamentos favorecendo a elaboração de consultas complexas quando o volume de dados crescer. Como exemplo, um relacionamento entre dois amigos pode conter o nome (AMIGO_DE) facilitando a identificação do tipo de relação entre duas arestas (Robinson; *et al*; 2013).

Para Neotechnology (2013), no banco de dados, os vértices e arestas são chamados de nós (*nodes*) e relacionamentos (*relationships*) respectivamente. A ideia é armazenar dados que possuam características específicas e, para isso, tanto os vértices quanto as arestas podem ou não ter propriedades. Um vértice ou uma aresta podem começar com apenas uma propriedade e de acordo com que o volume de dados cresça, novas propriedades podem ser inseridas a fim de se obter uma maior precisão durante uma busca. A **Fig. 6** abaixo é uma representação gráfica desta estrutura:

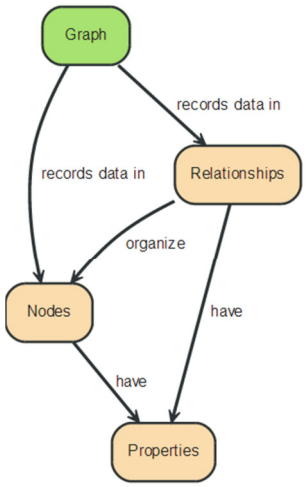

Fig. 6 - Estrutura básica de um GRAPHDB.
Fonte: Neotechnology (2013).

Com uma estrutura de grafos devidamente modelada, há a necessidade de implementar algoritmos de buscas para que o banco retorne os dados esperados pelo usuário. A pesquisa é denominada *Traversal,* que nada mais é do que a forma com que uma consulta irá navegar entre os nós para se chegar à informação desejada. A partir de um vértice inicial, é possível

é necessária a instalação, que é simples; basta apenas um duplo clique no arquivo baixado e seguir as instruções na tela.

3.4.4.2 Neo4J

Como descrito na seção 2.4, o Neo4J possui duas versões para manipulação de dados (*embedded* e *server).* Ambas versões necessitam do pacote completo do Neo4J que é disponibilizado para *download* no *site* oficial como mostra a **Fig. 16**. Esta pesquisa foi elaborada sobre a plataforma Windows, seguindo os procedimentos a seguir.

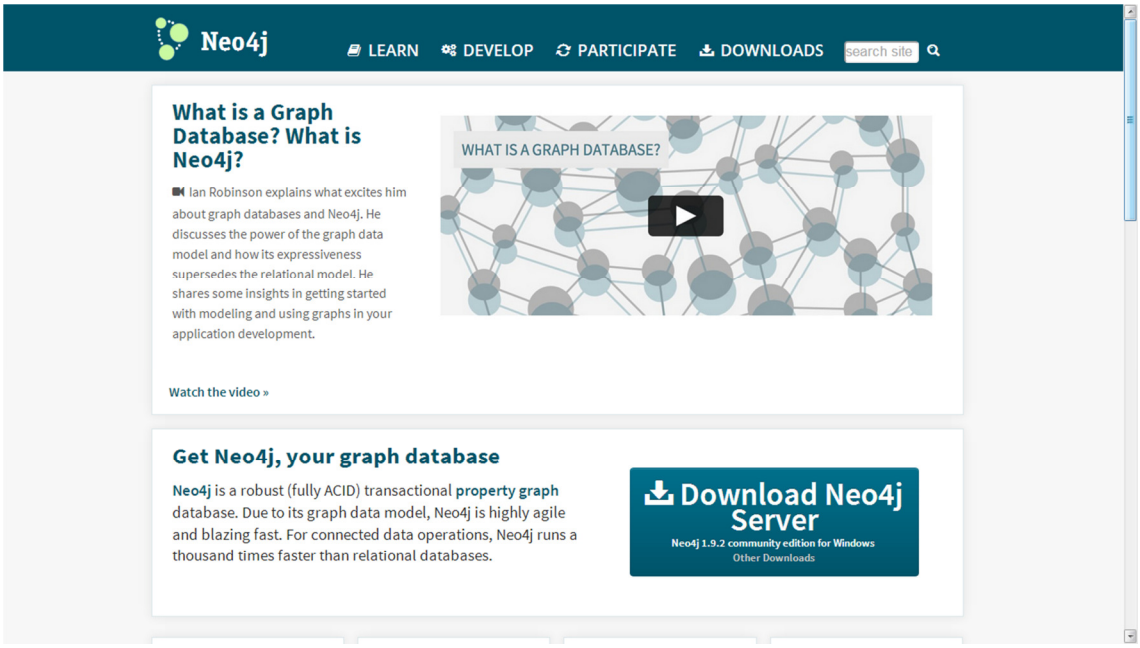

Fig. 16 - *Download* Neo4J. **Fonte:** Elaborado pelos autores (2013).

A pesquisa foi desenvolvida sobre a versão *Community* 1.9.2 do Neo4J. Esta versão não necessita de instalação, porém, após o *download*, o arquivo deve ser descompactado no disco local. O diretório gerado após a descompactação possui a seguinte estrutura:

- BIN: arquivos de *scripts* e demais executáveis;
- CONF: arquivos de configuração do servidor;
- DATA: arquivos de *logs* além da base de dados;
- DOC: documentação e demais informações da ferramenta;
- LIB: bibliotecas a serem utilizadas na versão *embedded;*
- PLUGINS: *scripts* para adição de funcionalidades;
- SYSTEM: arquivos de sistema necessários para a execução do servidor.

Segundo a Neotechnology (2013), os requisitos mínimos de *hardware* são: a) Processador Intel Core i3; b) Memória física de 2GB; c) Disco rígido de 10GB. Como o Neo4J é baseado na plataforma Java, também é necessário que o mesmo esteja instalado na versão sete. No entanto, esta pesquisa foi executada em dois diferentes computadores que possuem as seguintes configurações:

- *Software*: Windows 7 Ultimate Edition e Windows 7 Home Edition ambos 64bits; Navegador Google Chrome v28 e Java 7.25
- *Hardware*: processador Intel Core i3 2,13 GHz e Intel Core 2 Duo 2,2 GHz; memória física de 4GB e disco rígido de 320GB

O arquivo "Neo4J.bat" presente no diretório BIN foi utilizado para iniciar o servidor do Neo4J. Este arquivo contém comandos para a configuração do mesmo como serviço do sistema operacional. Para que seja possível o acesso à interface do servidor, este arquivo necessita ser executado. Com o serviço iniciado, o acesso aos recursos de manipulação do banco de dados foi realizado pelo navegador utilizando a porta 7474. A **Fig. 17** ilustra a tela de administração do servidor Neo4J, a tela principal (*Dashboard*) mostra a quantidade total de vértices e arestas presentes no banco de dados.

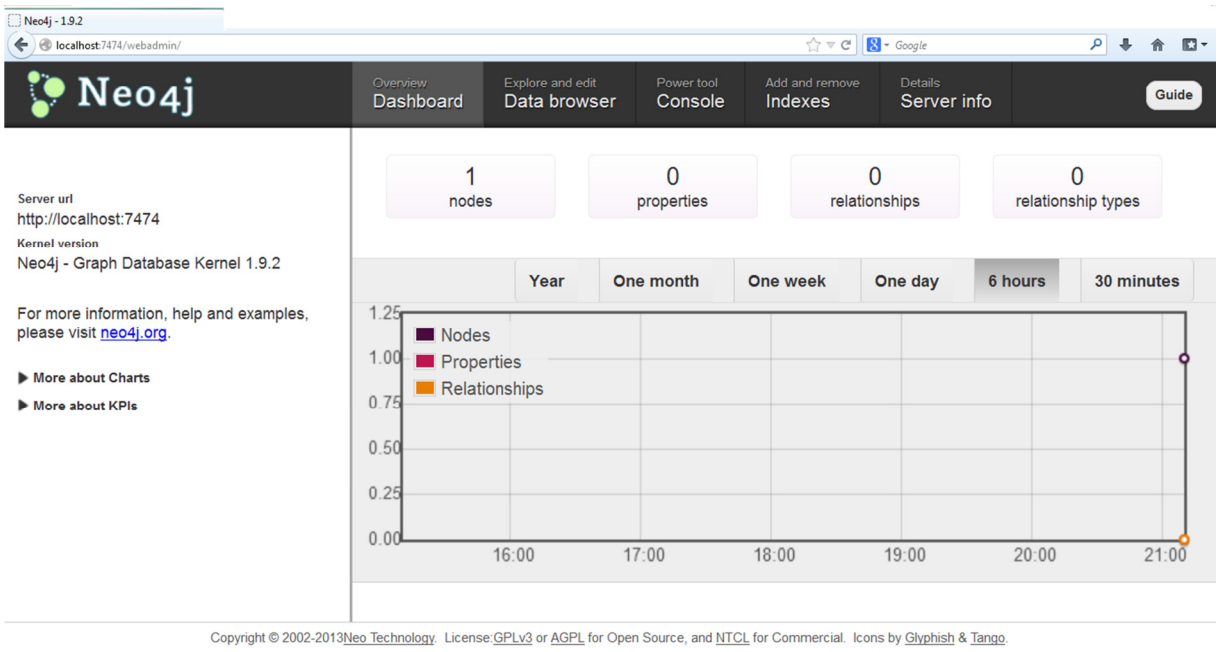

Fig. 17 - Tela de administração do Neo4J. **Fonte:** Elaborado pelos autores (2013).

Ainda na **Fig. 17**, a guia *Data Browser* pode ser utilizada para a criação dos elementos do banco de dados (vértices, arestas e propriedades) e também apresenta o grafo completo, que com efeitos visuais, auxiliam a análise do grafo como um todo. A guia *Console* possui o recurso de execuções dos comandos Cypher e Gremlin para a manipulação de dados no grafo.

Na guia *Indexes*, é possível criar novos índices bem como suas propriedades ou então, apagar os já existentes. Por fim, a guia *Server info* possui informações gerais do servidor, além dos recursos oferecidos pelo mesmo.

3.4.4.3 Eclipse

Nesta pesquisa, o ambiente para desenvolvimento de códigos Java foi a versão *Juno 4.2* do Eclipse. A versão em questão possui recursos para desenvolvimento de aplicações na plataforma *Java Enterprise Edition* (JEE), contudo, para esta pesquisa não houve a necessidade de utilização deste recurso e sim, o suporte a *plataforma Java Standard Edition* (JSE).

O *download* do Eclipse foi realizado pelo site oficial da ferramenta como mostra a **Fig. 18** abaixo:

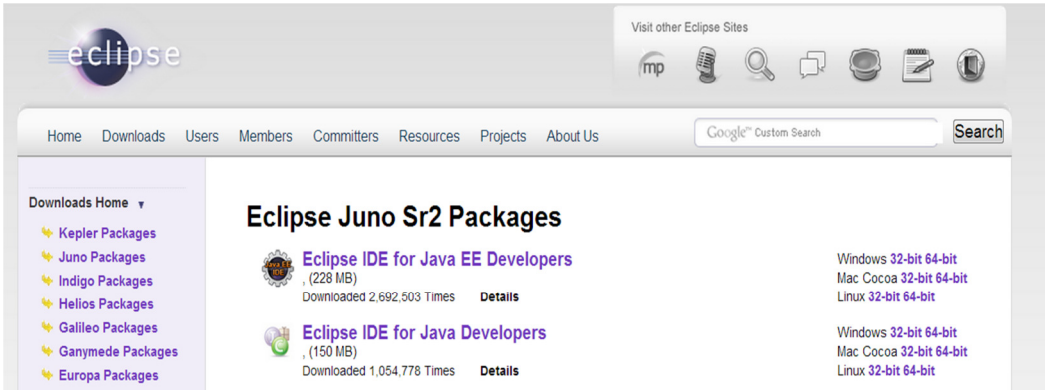

Fig. 18 - *Download* do Eclipse. **Fonte:** Elaborado pelos autores (2013).

Após o *download*, foi necessária a configuração do projeto para que a comunicação entre o código Java executado no Eclipse fosse realizada com os sistemas gerenciadores dos bancos de dados. Para esta configuração, foi necessário além dos arquivos com extensão (.jar) da biblioteca do Neo4J, o *download* do driver JDBC para o PostgreSQL versão 8.4 que é disponibilizado no *site* oficial do PostgreSQL.

A **Fig. 19** apresenta o conteúdo do projeto Java elaborado no Eclipse onde foram criadas as classes *GraphDB* e *RelationalDB* contendo os códigos de inserção para os dados fictícios do banco de dados orientado a grafos e relacional respectivamente[5].

[5] Os códigos para popular os bancos de dados estão descritos na seção 3.4.6

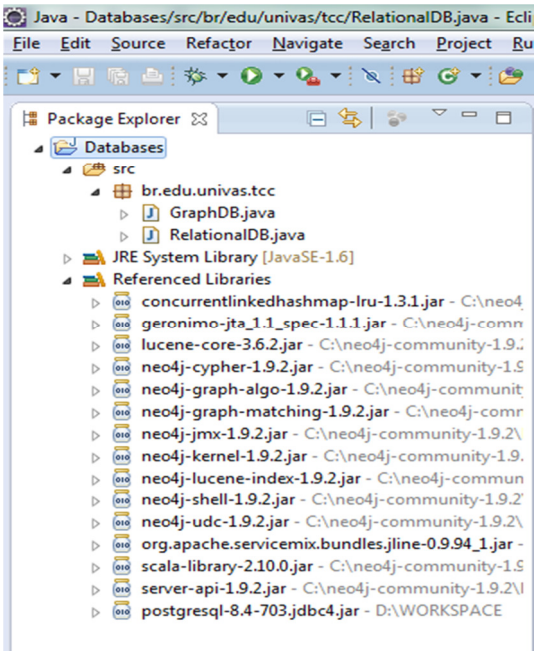

Fig. 19 - Estrutura do projeto no Eclipse.
Fonte: Elaborado pelos autores (2013).

No projeto criado foram adicionadas as bibliotecas do Neo4J, localizadas no diretório "LIB" na estrutura raiz do mesmo e o driver JDBC obtido após o *download*.

3.4.5 Consultas

Com as modelagens projetadas, foi necessária a elaboração das consultas para serem executadas em ambos bancos de dados. Assim como os tipos de dados a serem trabalhados, as consultas foram definidas por meio de análise das redes sociais e reuniões entre os participantes desta pesquisa.

Foram definidas as seguintes perguntas para a elaboração dos comandos de consulta a partir de um determinado usuário:

1. Quais são seus contatos?
2. Quem é da família?
3. Quais os contatos que moram em determinada cidade?
4. Quais os contatos que moram em determinada cidade, que também trabalham em determinada empresa?
5. Do que os amigos gostam?
6. Quais os grupos que os amigos participam?

As perguntas foram executadas nos bancos de dados PostgreSQL e Neo4J utilizando as linguagens SQL e Cypher respectivamente, cujos códigos são discutidos abaixo.

vértices em memória, utilizaram-se listas, separadas pelo tipo de dado tratado e com a constante que possui a quantidade total de vértices a serem criados.

```
GraphDatabaseService graphDb;
Node listUsuarios[] = new Node[usuarios];
Node listCidades[] = new Node[cidades];
Node listInteresses[] = new Node[interesses];
Node listGrupos[] = new Node[grupos];
Node listEscolas[] = new Node[escolas];
Node listEmpresas[] = new Node[empresas];
```

Fig. 20 – Listas de objetos *Node*. **Fonte:** Elaborado pelos autores (2013).

A **Fig. 21** apresenta a forma como o objeto graphDb foi instanciado. Para tal procedimento, foi necessário informar por meio do método EmbeddedDatabaseBuilder, o caminho completo em que a base de dados se encontra. O método setConfig() presente nas linhas 3 e 4 foi utilizado para ativar a indexação automática do Neo4J, esta configuração é necessária para facilitar a localização de vértices que possuem determinado valor em sua propriedade. Nesta pesquisa, o índice foi configurado com as propriedades "*name*" e "tipo". Por fim, o método newGraphDatabase() cria uma instância para o objeto graphDb, que foi utilizado para execuções dos métodos necessários ao se trabalhar com o banco de dados Neo4J.

```
1. graphDb = new GraphDatabaseFactory().
2.   newEmbeddedDatabaseBuilder( "C:/neo4j-community-1.9.2/data/graph.db").
3.   setConfig( GraphDatabaseSettings.node_keys_indexable, "name,tipo" ).
4.   setConfig( GraphDatabaseSettings.node_auto_indexing, "true" ).
5.   newGraphDatabase();
```

Fig. 21 - Instância do objeto *GraphDatabaseFactory*. **Fonte:** Elaborado pelos autores (2013).

Com a instância do objeto GraphDatabase e com as listas organizadas pelo tipo de vértices, o código prossegue com as rotinas que criam os vértices no banco de dados e armazena-os às listas. A **Fig. 22** ilustra a rotina de inserção de vértices do tipo "grupos", o método createNode() presente na linha 2 é utilizado para criar o vértice e nas linhas 3 e 4 as propriedades, que, como parâmetro, recebem o nome e o valor correspondente. Por se tratar de dados fictícios, a propriedade "*name*" recebeu valores aleatórios como: "grupo1", "grupo2", "grupo3", etc...

```
1. for(int i=0;i<grupos;i++){
2.    listGrupos[i] = graphDb.createNode();
3.    listGrupos[i].setProperty( "name", "grupo"+i);
4.    listGrupos[i].setProperty( "tipo", "grupo");
5.  }
```

Fig. 22 - Criando vértices. **Fonte**: Elaborado pelos autores (2013).

Esta rotina foi repetida para o preenchimento das demais listas de vértices e suas respectivas propriedades. Após este processo, foi necessária a ligação destes por meio de arestas. A **Fig. 23** apresenta a rotina para tal ligação entre vértices do tipo "usuário" e "grupos". O método presente na linha 2 armazena um vetor de números inteiros gerados aleatoriamente e que representam os índices de determinada lista de vértices. Tal método será discutido na **Fig. 24**. Com o vetor de números preenchido, os comandos das linhas 4 e 5 criam uma relação (aresta) do tipo PARTICIPA entre cada usuário com grupos aleatórios da lista de vértices "listGrupos[]". A quantidade de grupos de que cada usuário participa foi definida previamente por meio da constante `limiteGrupos`. Assim como os vértices, a rotina de inserção de arestas também se repetiu para os diferentes tipos, respeitando a lógica dos tipos de dados no grafo.

```
1. for(int i=0;i<usuarios;i++){
2.   Integer list[] = getNodes(grupos,limiteGrupos);
3.     for(int k=0;k<limiteGrupos;k++){
4.       listUsuarios[i].createRelationshipTo
5.       (listGrupos[list[k]],RelTypes.PARTICIPA);
6.      }
7. }
```

Fig. 23 - Criando arestas**. Fonte:** Elaborado pelos autores (2013).

A lógica de geração de números aleatórios está ilustrada na **Fig. 24**. O método recebe como parâmetro, o total de vértices e a quantidade de arestas do tipo em questão. Os números gerados através da instância da classe `Random()` representam os índices das listas de vértices criadas anteriormente. A variável criada na linha 8, juntamente com os códigos das linhas 10 à 15 são verificações feitas para descartar números já existentes no vetor. Nos casos em que os números forem repetidos, a função `nextInt()` presente na linha 9 gera outro valor, em caso contrário, o código da linha 17 insere o número gerado ao vetor que é posteriormente retornada na linha 22.

```
1. public static Integer[] getNodes(int populacao, int limite){
2.   Integer[] nodes = new Integer[limite];
3.   Random generator = new Random();
4.      for(int i=0;i<limite;i++){
5.         nodes[i] = 0;
6.       }
7.      for(int i=0; i<limite; i++){
8.        int existNode = 0;
9.        int selectedNode = generator.nextInt(populacao);
10.          for(int k=0;k<nodes.length;k++){
11.                  if(nodes[k] == selectedNode){
12.                        existNode = 1;
13.                        break;
14.                  }
15.          }
16.          if(existNode == 0){
17.              nodes[i] = selectedNode;
18.          }else{
19.              i--;
20.            }
21.         }
22.      return nodes;
23. }
```

Fig. 24 - Método para valores aleatórios. **Fonte:** Elaborado pelos autores (2013).

Para o modelo de banco de dados relacional, a classe *RelationalDB* foi criada, mantendo a mesma lógica de inserção de dados do modelo orientado a grafos, no entanto, diferentemente das listas de vértices, os valores das listas foram de números inteiros que representam o valor do campo ID de cada registro nas tabelas. Desta forma, o método "getNodes()" da classe *GraphDB* foi reutilizado, alterando apenas o nome para "getIndexes()", porém, a lógica de geração de números aleatórios foi a mesma.

A **Fig. 25** ilustra o processo de inserção de grupos à entidade "grupo" no banco de dados relacional, utilizando objetos do tipo `PreparedStatement` para executar os comandos SQL. Assim como no modelo orientado a grafos, o processo repetiu-se para os demais tipos de dados, respeitando os tipos de relacionamento entre eles.

```
1. PreparedStatement state, psSeq;
2. state = conn.prepareStatement("insert into grupo values(?,?)");
3. psSeq = conn.prepareStatement("select nextval('grupo_seq') ");
4.     for(int i=0;i<grupos;i++){
5.         ResultSet rsSeq = psSeq.executeQuery();
6.         rsSeq.next();
7.         id = rsSeq.getInt(1);
8.         listGrupos[i] = id;
9.         state.setInt(1, id);
10.        state.setString(2, "grupo"+id);
11.        state.execute();
12.    }
```

Fig. 25 - Inserindo dados em tabela com *PreparedStatement*. **Fonte:** Elaborado pelos autores (2013).

A **Fig. 26** contêm os códigos de relacionamento entre usuários e grupos utilizando a tabela "usuario_grupo". O código insere grupos aleatórios a partir do vetor "list[]" que possui valores que representam os índices da lista "listGrupos[]", a quantidade de grupos de que cada usuário participa foi definida previamente por meio da constante *limiteGrupo*.

```
1. state = conn.prepareStatement("insert into usuario_grupo
values(?,?)");
2. for(int i=0;i<usuarios;i++){
3.    Integer list[] = getIndexes(grupos,limiteGrupos);
4.    for(int k=0;k<limiteGrupos;k++){
5.       state.setInt(1,listUsuarios[i]);
6.       state.setInt(2,listGrupos[list[k]]);
7.       state.execute();
8.       }
```

Fig. 26 - Criando relacionemos no modelo relacional. **Fonte**: Elaborado pelos autores (2013).

O código descrito nesta sessão foi executado utilizando bases de dados vazias afim de evitar a duplicidade de dados.

3.5 Resultados

Os resultados desta pesquisa foram obtidos após as etapas de definição dos tipos de dados, modelagem dos bancos de dados, elaboração de consultas e execução das mesmas nos bancos PostgreSQL e Neo4J para o modelo relacional e orientado a grafos respectivamente.

A fim de se obter resultados que comprovem os estudos teóricos presentes nos materiais de referência desta pesquisa, as consultas foram executadas com diferentes quantidades dados inseridos aos bancos, ou seja, para a primeira análise de desempenho de

4 DISCUSSÃO DE RESULTADOS

Neste capítulo serão discutidos os resultados obtidos após a implementação das duas tecnologias de banco de dados estudadas nesta pesquisa. Os resultados foram gerados com base nas estruturas definidas durante a fase de desenvolvimento.

4.1 Tratando o limite de memória virtual

Durante a elaboração do algoritmo de inserção, que teve como finalidade criar uma rotina para o preenchimento de dados aleatórios às estruturas, ocorreu um erro durante a execução no modelo baseado em grafos. Segundo Alves e Carvalho (2012), este erro foi ocasionado devido à limitação de memória virtual utilizada pela *Java Virtual machine* (JVM), conhecida como *Java Heap*, gerando uma exceção como mostrado na **Fig. 27**.

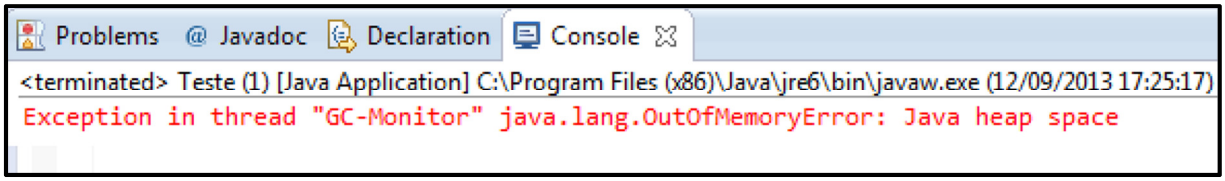

Fig. 27 - Erro Java Heap. **Fonte:** Elaborado pelos autores (2013).

Como descrito pela Neotechnology(2013), o espaço na memória reservado para a execução do processo em Java foi alterado por meio do parâmetro –Xmx1024m nos argumentos de execução do Eclipse. Para garantir a execução do código e respeitando o limite de memória física das estações, os algoritmos foram configurados para serem executados com 1024MB.

Mesmo com a alteração do limite de memória, a execução do código não obteve sucesso. Este erro é descrito por Alves e Carvalho (2012) como *stop-the-world,* ocasionado pela pouca memória, impossibilitando a atuação do *Garbage Collector.*

Para resolver este problema, a **Fig. 28** apresenta o método que também foi utilizado ao fim de cada rotina de inserção para realizar o envio de informações às bases de dados (*Commit*), sem que haja o acúmulo de objetos na memória da JVM.

```
static int count = 0;
  private static void intermediateCommit(GraphDatabaseService graph){
    if(count % 1000 == 0) {
       tx.success();
       tx.finish();
       tx = graph.beginTx(); }
  count++;     }
```

Fig. 28 - Método de *Commit*. **Fonte**: Elaborado pelos autores (2013).

Com o propósito de se obter resultados precisos sem que haja diferenças de desempenho de nenhuma das arquiteturas de dados por conta do *cache*, a média do tempo de resposta das consultas em cada banco de dados foi calculada a partir da segunda execução[12].

A figura **Fig. 29** apresenta o gráfico[13] com o tempo de resposta obtido pelas estruturas de dados a partir da segunda até a décima execução da **consulta 1.** Percebeu-se que o tempo de resposta da consulta realizada no banco de dados relacional, foi em média 307,75 ms, já o banco de dados orientado a grafos obteve média de 15,5 ms, uma diferença de aproximadamente 94%.

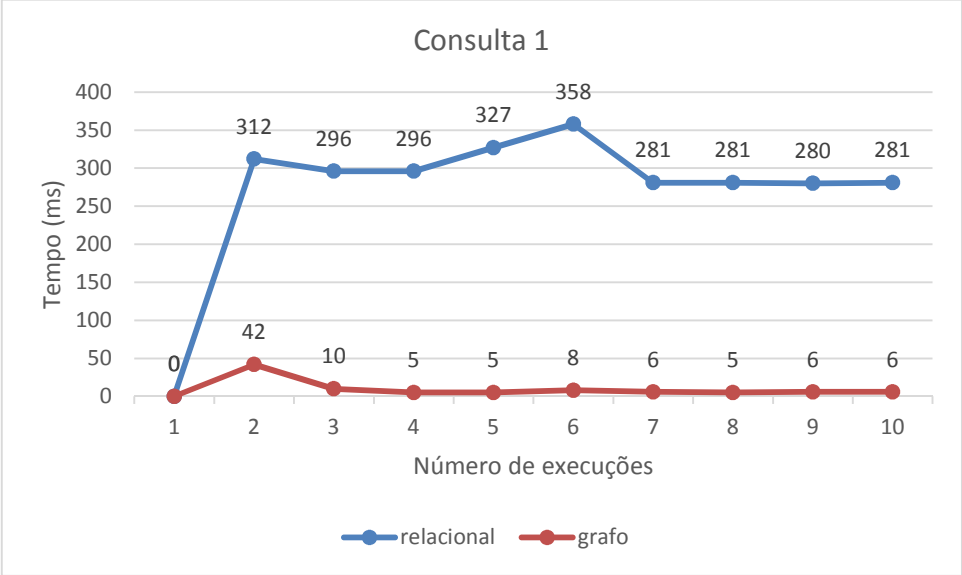

Fig. 29 - Gráfico da consulta 1. **Fonte**: Elaborado pelos autores (2013).

A **Fig. 30** apresenta o gráfico com os tempos de resposta de ambas as bases de dados para a **consulta 3**. Essa consulta contém informações de diferentes tipos de dados. Para os comandos de busca, foi necessário realizar *JOIN* de várias tabelas no modelo relacional e a navegação por diferentes tipos de arestas no modelo orientado a grafos. Novamente, o banco de dados orientado a grafos se mostrou mais ágil com média de 20,5 ms contra 327,25 ms do modelo relacional, uma diferença de aproximadamente 93,7 %.

[12] Os tempos obtidos com a primeira execução estão presentes nos apêndices desta pesquisa.
[13] Os gráficos das consultas 2,5 e 6 estão presentes nos apêndices desta pesquisa.

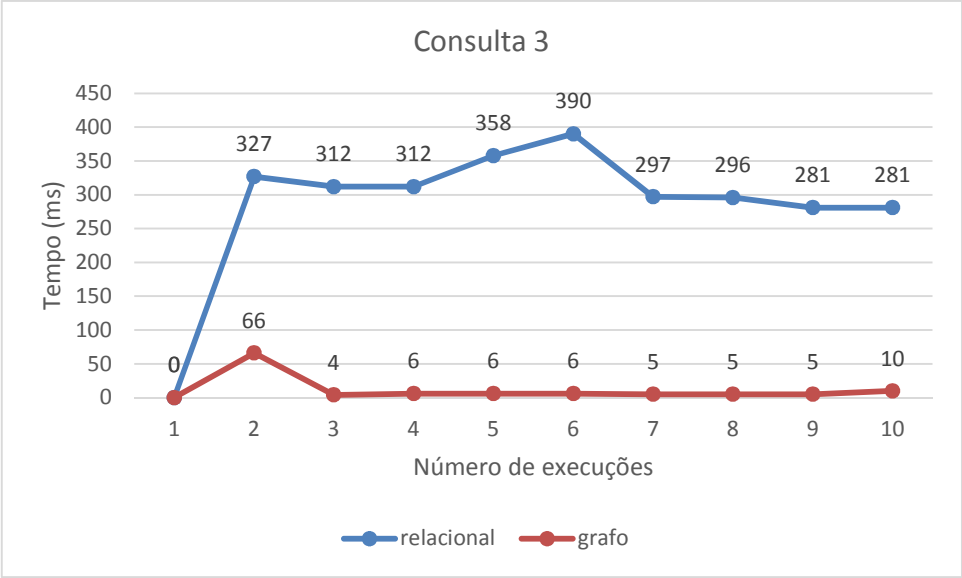

Fig. 30 - Gráfico da consulta 3. **Fonte**: Elaborado pelos autores (2013).

A execução da **consulta 4** nas bases de dados está representada na **Fig. 31**. Esta consulta possui uma complexidade maior dentre as já apresentadas, pois além de necessitar de dados com diferentes tipos, também foi necessário realizar consultas distintas dentro da principal. Desta forma, o uso de JOIN no modelo relacional e tipos de arestas no banco de dados orientado a grafos foi utilizado com maior frequência, devido à sua complexidade.

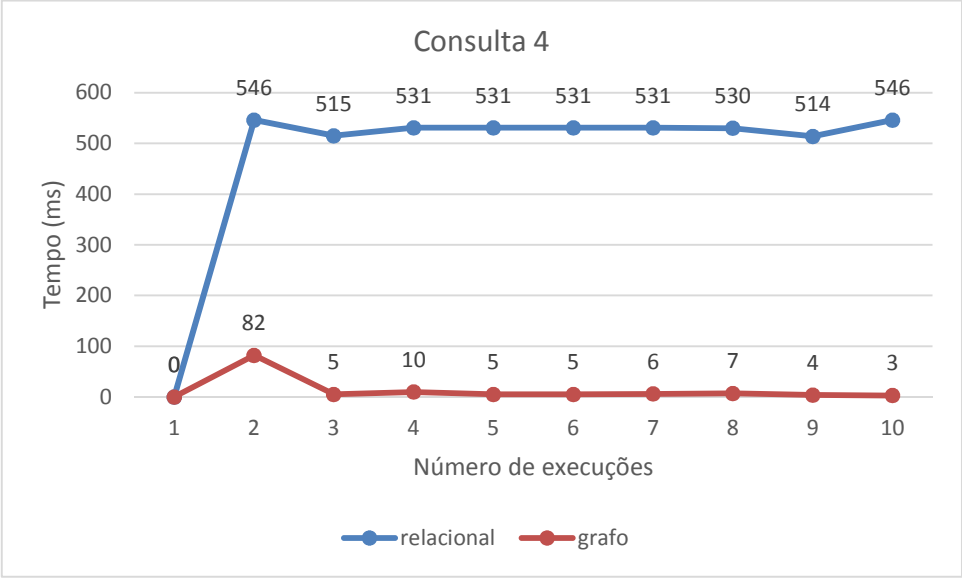

Fig. 31 - Gráfico da consulta 4. **Fonte**: Elaborado pelos autores (2013).

A **Tabela 10** apresenta as médias das duas arquiteturas de banco de dados nas demais consultas. Percebeu-se que a média se manteve, com números maiores para o banco de dados relacional e sem grandes variações de tempo entre uma execução e outra. No geral, o banco de dados orientado a grafos apresentou-se entre 85% e 96% mais ágil em comparação com o modelo relacional.