UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA

OLAVO DE HOLANDA CAVALCANTI NETO

JOINT-DE: SISTEMA DE MAPEAMENTO OBJETO-ONTOLOGIA COM SUPORTE A OBJETOS DESCONECTADOS

Maceió
2014

Olavo de Holanda Cavalcanti Neto

JOINT-DE: SISTEMA DE MAPEAMENTO OBJETO-ONTOLOGIA COM SUPORTE A OBJETOS DESCONECTADOS

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.
Orientador: Prof. Dr. Ig Ibert Bittencourt Santana Pinto
Coorientador: Prof. Dr. Seiji Isotani

Maceió
2014

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Valter dos Santos Andrade

C376j Cavalcanti Neto, Olavo de Holanda.
Joint-de: sistema de mapeamento objeto-ontologia com suporte a objetos desconectados / Olavo de Holanda Cavalcanti Neto. – Maceió, 2014.
103 f. : il.

Orientador: Ig Ibert Bittencourt Santana Pinto.
Coorientador: Seiji Isotani.
Dissertação (Mestrado em Informática) – Universidade Federal de Alagoas. Instituto de Computação. Programa de Pós-Graduação em Informática. Maceió, 2014.

Bibliografia: f. 99-103.

1. Mapeamento - Objeto-ontologia. 2. Aplicações baseadas em ontologias. 3. Objetos desconectados. 4. Sistema de mapeamento. I. Título.

CDU:004.45

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PPGI
Instituto de Computação
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Membros da Comissão Julgadora da Dissertação de Mestrado de Olavo Holanda Cavalcanti Neto, intitulada: "JOINT-DE: Um Sistema de Mapeamento Objeto-Ontologia com Suporte a Objetos Desconectados", apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 22 de maio de 2014, às 14h00min, na Sala de Aula 04 do Instituto de Computação da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Ig Ibert Bittencourt Santana Pinto
UFAL – Instituto de Computação
Orientador

Prof. Dr. Seiji Isotani
ICMC - USP
Coorientador

Prof. Dr. Leandro Dias da Silva
Instituto de Computação – UFAL
Examinador

Prof. Dr. Evandro de Barros Costa
Instituto de Computação – UFAL
Examinador

Prof. Dr. Sean Wolfgand Matsui Siqueira
Instituto de Computação – UFAL
Examinador

"Se vi mais longe foi por estar de pé sobre ombros de gigantes."
Isaac Newton

AGRADECIMENTOS

Primeiramente a Deus que permitiu que eu trilhasse essa longa caminhada com sucesso, não somente nestes anos como mestrando, mas em todos os momentos da minha vida. Ele me deu forças para superar todas as dificuldades e obstáculos.

A Universidade Federal de Alagoas, principalmente ao Instituto de Computação, seu corpo docente, direção, coordenação e administração que oportunizaram as diversas escolhas que agora, como mestre, possuo. Agradeço pela compreensão pelos diversos erros por ser a primeira turma do mestrado, com a certeza de que aprendemos juntos e de que demos nosso melhor para tornar o Programa de Pós-Graduação em Informática uma excelência de mestrado.

Agradeço aos meus orientadores Ig Ibert Bittencourt e Seiji Isotani, por me proporcionar todo o conhecimento e orientação necessário para alcançar meus objetivos. Foram vocês que não só contribuíram para minha formação profissional, mas também me ensinaram lições que enriqueceram minha formação pessoal.

Agradeço aos meus colegas Endhe Elias, Thyago Tenório e Wilkson Eldon, que por muitas vezes me ajudaram com várias discussões e troca de opiniões acerca deste trabalho. Aos membros do projeto JOINT: Judson Bandeira, Williams Alcântara e Armando Barbosa que ao longo destes dois anos foram mais que parceiros de projeto.

Aos meus pais e irmãos, pelo amor, incentivo e apoio incondicional. A compreensão deles de que minha ausência em diversos momentos familiares era motivo de uma dedicação constante para um futuro de sucesso.

Finalmente, agradeço em especial a minha namorada Cindy Buarque, que por diversos momentos disponibilizou seu tempo e esforço para corrigir e revisar minha redação.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

RESUMO

Nos últimos anos, é crescente o desenvolvimento e o uso de ontologias na criação de aplicações mais inteligentes e eficazes que têm como objetivo solucionar problemas encontrados comumente na Web. Toda essa popularidade se deve ao fato de que ontologias tentam oferecer semântica aos dados consumidos pelas máquinas de forma que ela possa raciocinar sobre estes dados. Todavia, a larga adoção da Web Semântica pode ser ainda acelerada ao prover ferramentas sofisticadas que diminuam a barreira de desenvolvimento de aplicações baseadas em RDF e OWL. Desenvolvedores de aplicações com bancos de dados relacionais já estão acostumados com ferramentas como o Hibernate, que oferecem um mapeamento objeto-relacional e o gerenciamento de estados dos objetos. Na verdade, o principal estado de objeto que o Hibernate disponibiliza é o desconectado. Entretanto, a grande maioria dos sistemas de mapeamento objeto-ontologia (OOMS) apenas disponibiliza objetos persistentes. A grande diferença entre os dois tipos de objetos é que o primeiro tem seu ciclo de vida independente da conexão com o banco de dados RDF, já o último é limitado à conexão. Neste contexto, este trabalho propõe a criação de um sistema de mapeamento objeto-ontologia que suporta objetos desconectados, chamado JOINT-DE. Com este sistema, desenvolvedores de aplicações baseados em ontologias podem: i) utilizar os objetos oriundos do banco de dados RDF como objetos do modelo de negócio, transitando nas diversas camadas da aplicação; ii) utilizar esses objetos como objetos de transferência de dados (DTOs) entre subsistemas e iii) desenvolver pequenas transações com objetos desconectados que representam uma unidade longa de transação para o usuário da aplicação. Para exemplificar os benefícios do sistema proposto, um estudo de caso de uma aplicação real é apresentado, expondo as limitações arquiteturais dessa aplicação ao utilizar um OOMS existente na literatura, além de mostrar resultados favoráveis à implantação do JOINT-DE. Por fim, um experimento foi planejado e executado com o objetivo de comparar o JOINT-DE com outro OOMS bastante utilizado pela comunidade: Alibaba. As análises estatísticas realizadas nesse experimento apontaram resultados satisfatórios com relação ao JOINT-DE.

Palavras-chaves: Mapeamento Objeto-Ontologia. Aplicações Baseadas em Ontologias. Objetos Desconectados

ABSTRACT

In the last few years, it is increasing the development and use of ontologies in creating more intelligent and effective applications that aim to solve problems commonly found on the Web. This popularity is due to the fact that ontologies attempt to provide semantics to the data consumed by machines, so that they can reason about these data. However, the large adoption of the Semantic Web can be further accelerated by providing sophisticated tools that lower the barrier to the development of applications based on RDF and OWL. Developers of applications with relational databases are already familiar with tools like Hibernate, which provide an object-relational mapping and the management of the objects states. Actually, the main object state that Hibernate provides is the detached. Nevertheless, the great majority of the object-ontology mapping systems (OOMS) only provide persistent objects. The big difference between these two types of objects is that the former one has its life cycle independent of the underlying triple store connection, but the latter one is bounded to the connection. In this context, this paper proposes the creation of an object-ontology mapping systems that supports detached objects, called Joint-DE. With this system, developers of ontology-based applications can: i) use the objects coming from the triple store as objects of the business model; ii) use such objects as data transfer objects (DTOs) between subsystems and; iii) develop small transactions with detached objects that represent a long transaction unit for the application user. To illustrate the benefits of the proposed system, a case study of a real application is presented, outlining the architectural limitations of the application using an existing OOMS in the literature, as well as showing positive results to the use of JOINT-DE. Finally, an experiment was planned and executed aiming to compare the JOINT-DE with another OOMS widely used by the community: Alibaba. The statistical analyzes performed in this experiment showed satisfactory results with regard to JOINT-DE.

Keywords: Object-Ontology Mapping. Ontology-based Applications. Detached Objects.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, ontologias têm obtido bastante atenção na comunidade científica de computação. O termo, que teve origem na filosofia, se tornou uma palavra conveniente na área de computação para uma nova abordagem de representação de conhecimento sobre entidades do mundo real. Toda essa atenção se deve, também, ao fato de que ontologias tentam solucionar um dos maiores problemas no uso das máquinas para processar informação gerada por agentes humanos: alcançar a representação formal de um domínio real dentro de sistemas computacionais (HEPP et al., 2007).

Dessa forma, ontologias são consideradas como a maneira mais apropriada para facilitar a interoperabilidade entre sistemas heterogêneos, envolvidos em um domínio de interesse comum. Isso acontece porque as ontologias oferecem um entendimento compartilhado e uma formalização sobre um domínio que fazem com que os dados sejam mais facilmente interpretáveis por máquinas.

Como consequência, ontologias são aplicadas tanto como base para a Web semântica (BERNERS-LEE et al., 2001), quanto como ferramental para solucionar problemas em outras áreas da pesquisa e da indústria da computação. Por exemplo, aplicações de comércio eletrônico usam ontologias para fazer buscas parametrizadas, aperfeiçoar a navegação e integrar sistemas heterogêneos (DAS; WU; MCGUINNESS, 2001). A pesquisa na área de medicina tem utilizado ontologias com o intuito de relacionar e consultar bases de dados moleculares (BARD; RHEE, 2004). Outro segmento de indústria é o de sistemas de mídia, que têm usado este modelo para fazer inferência em dados em tempo real, entregando conteúdo atualizado para seus usuários (KIRYAKOV et al., 2010). Novas abordagens para a construção de sistemas educacionais têm aproveitado ontologias como representação para o modelo do estudante, o modelo pedagógico e o modelo de domínio (BITTENCOURT et al., 2009). Recentemente, a área de computação pervasiva lançou assistentes virtuais baseados em ontologias (CHEYER; GRUBER, 2010). Além de todas essas aplicações, ontologias estão sendo utilizadas em outros campos, tais como: sistemas de recomendação, recuperação de informação, engenharia de software e gerenciamento de conhecimento em organizações (BORGES et al., 2009) (POPOV et al., 2004) (SILVA et al., 2011) (HUANG; DIAO, 2008).

Toda essa difusão é consequência do crescente número de ferramentas e bibliotecas de software que permitem o desenvolvimento de aplicações semânticas. Atualmente, cerca de 215 ferramentas estão listadas no endereço *semanticweb.org*[1] , sendo que esse número não é exato e tende a crescer ainda mais com a popularização da Web Semântica. Dentre essas ferramentas, se destacam:

[1] Disponível em http://semanticweb.org/wiki/Tools

- **Editores de ontologias:** Muito úteis para modelar ontologias, auxiliando no processo de engenharia de ontologias. Os principais editores são Protégé (NOY; FERGERSON; MUSEN, 2000), TopBraidComposer[2] e NeOn Toolkit (HAASE et al., 2008);

- **Raciocinadores em ontologias:** Máquinas de inferências em ontologias, capazes de raciocinar sobre construtores da Linguagem de Ontologia para a Web (OWL) e do Arcabouço para Descrição de Recursos (RDF), deduzindo novas informações nas ontologias. Alguns exemplos de raciocinadores são FACT (TSARKOV; HORROCKS, 2006), Pellet (SIRIN et al., 2007), KAON2 (BOZSAK et al., 2002) e RacerPro (HAARSLEV et al., 2012);

- **Ferramentas de Armazenamento em Triplas:** Sistemas que funcionam como banco de dados RDF, com a diferença que seu armazenamento é feito baseado nas triplas do RDF. Dentre estas ferramentas, se destacam OWLIM (KIRYAKOV; OGNYANOV; MANOV, 2005), Virtuoso (ERLING; MIKHAILOV, 2009) e AllegroGraph (AASMAN, 2006);

- **Arcabouços RDF:** Arcabouços que permitem a manipulação e processamento de dados RDF, incluindo parsers, soluções de armazenamento, raciocínio e motor de consulta. As ferramentas mais usadas nesta categoria são Jena (MCBRIDE, 2002) e OpenRDF Sesame (BROEKSTRA; KAMPMAN; HARMELEN, 2002);

- **Sistemas de Mapeamento Objeto-Ontologia (OOMS):** Usados por desenvolvedores de aplicações baseadas em ontologias, estas ferramentas auxiliam no processo de desenvolvimento, mapeando a ontologia no código da aplicação através do paradigma de orientação a objetos. Dentre estas ferramentas, estão OpenRDF Alibaba[3], Jastor (SZEKELY; BETZ, 2006) e KOMMA (WENZEL, 2010).

Todavia, a larga adoção da Web Semântica pode ser ainda acelerada ao prover ferramentas sofisticadas que diminuam a barreira de desenvolvimento de aplicações baseadas em RDF e OWL. Barreira essa ainda muito alta devido ao simples fato de que os desenvolvedores de sistemas tradicionais com bancos de dados relacionais estão acostumados com ferramentas que facilitam o desenvolvimento. Tais soluções são chamadas de *sistema de mapeamento objeto-relacional* (ORMS).

Particularmente em Java, a ferramenta Hibernate é uma implementação de ORMS (BAUER; KING, 2006). Em aplicações orientadas a objetos que usam o padrão arquitetural de multicamadas, as três camadas seguintes são as mais comuns: i) Camada de Apresentação, provê a interface do sistema para o usuário final; ii) Camada de Negócio, implementa toda a lógica da aplicação com funcionalidades de negócio; iii) Camada

[2] TopBraid Composer: $http://www.topquadrant.com/products/TB_Composer.html$
[3] OpenRDF Alibaba: http://www.openrdf.org/alibaba.jsp

de Infraestrutura, conhecida também como camada de dados ou de persistência, responsável pelo acesso ao banco de dados e pela persistência[4] dos dados da aplicação nesse banco (BUSCHMANN; HENNEY; SCHIMDT, 2007). A ferramenta Hibernate, assim como qualquer outro ORMS, se encontra na última camada da aplicação: a camada de infraestrutura.

O Hibernate não apenas torna transparente para o desenvolvedor detalhes sobre o sistema de gerenciamento de banco de dados subjacente, como também oferece o gerenciamento dos estados dos objetos. Na verdade, um destes estados de objetos é o ponto fundamental do Hibernate, esse estado é chamado *desconectado* (do inglês detached) (BAUER; KING, 2005). Assim, uma *instância desconectada* é um objeto que já foi persistido no banco de dados, porém a conexão com o banco vinculada a ele já foi encerrada. Esse objeto pode ainda ter suas informações alteradas pelo desenvolvedor fora da camada de infraestrutura (e.g. na camada de negócios), sendo possível que ele seja posteriormente reconectado a uma nova conexão com o banco e tenha seus dados sincronizados e persistidos novamente. Essa funcionalidade permite, dentre outros pontos, que:

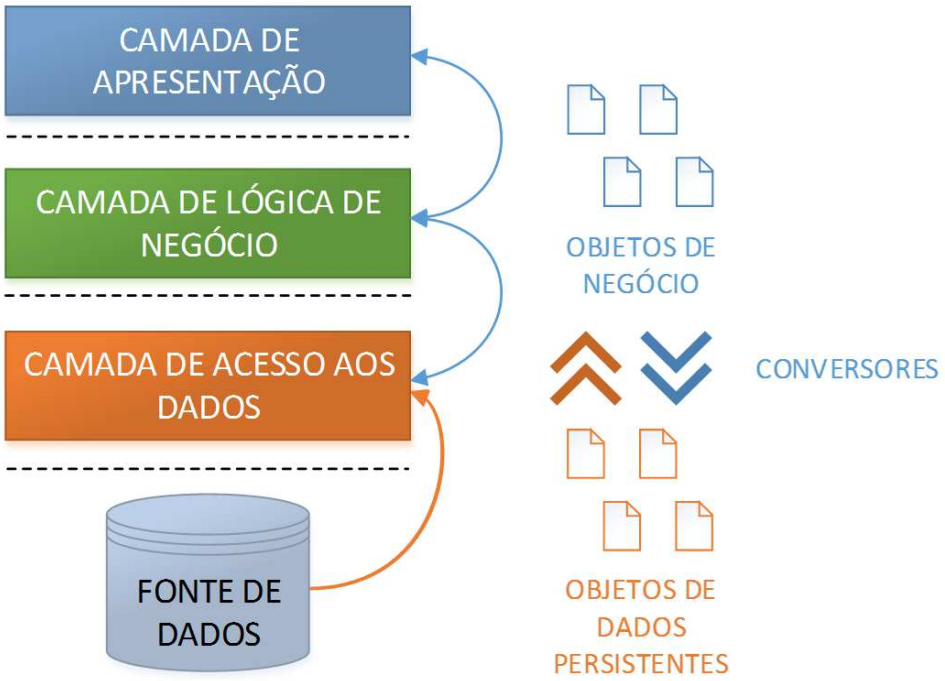
- Objetos *desconectados* sejam passados por diferentes camadas das aplicações, como se fossem objetos do modelo de negócio;
- Objetos *desconectados* também sejam utilizados como Objetos de Transferência de Dados (DTOs) entre subsistemas de uma aplicação, através de serialização[5];
- Quando transações longas são necessárias devido ao tempo de resposta do usuário, é recomendado quebrar essa transação em duas ou mais transações. Assim, objetos *desconectados* da primeira transação irão carregar os dados por toda a aplicação até a camada de apresentação. Depois, esses objetos serão modificados fora de uma transação e, posteriormente, reconectados a uma nova transação por meio de outra conexão.

Em contrapartida, os OOMS (sistemas de mapeamento objeto-ontologia mencionados anteriormente) que facilitam a construção de aplicações baseadas em ontologias apenas suportam objetos *persistentes*, ou seja, objetos cujos ciclos de vida estão limitados ao ciclo de vida da conexão com o banco. A partir do momento que essa conexão se encerra, esse objeto não pode mais ser manipulado. Dessa forma, estes objetos não podem ser usados como objetos do modelo de negócio da aplicação, necessitando de diversos conversores, como ilustra a Figura 1.

[4] A persistência de dados refere-se ao armazenamento não-volátil de dados, por exemplo, o armazenamento em um dispositivo físico como um disco rígido. O termo persistência é associado a uma ação que consiste em manter em meio físico recuperável.

[5] A serialização é o processo de salvar um objeto em um meio de armazenamento ou transmiti-lo por uma conexão de rede, em forma binária.

Figura 1 – Necessidade de conversores entre objetos persistentes e objetos de negócio.



Fonte: Elaborado pelo autor.

Alguns autores de OOMS concordam que é complexo dar suporte a objetos *desconectados* no contexto de ontologias. Isso ocorre pois a instância de uma ontologia possui um subgrafo de referência muito maior que uma instância convencional de banco de dados relacional, tornando custoso identificar e carregar de maneira automática esses subgrafos[6].

Neste contexto, este trabalho propõe uma evolução no código da ferramenta *Java Ontology Integrated Toolkit* (JOINT) (HOLANDA et al., 2013b) por meio de um sistema de mapeamento objeto-ontologia com suporte a objetos *desconectados* chamado JOINT-Detached (JOINT-DE). Assim, objetos recuperados pelo JOINT-DE, além de desconectados, serão serializados, o que permite a transição desses objetos entre diferentes camadas da aplicação até a camada de apresentação sem a necessidade do uso de DTOs. Além disso, o sistema proposto trata o problema de grandes subgrafos de uma instância por intermédio de um mecanismo de carregamento inicial das propriedades diretas da instância e o carregamento tardio, mas só quando necessário para o desenvolvedor, dos subgrafos que estas propriedades referenciam, viabilizando o sistema em termos de desempenho.

1.1 Objetivos

O principal objetivo deste trabalho é propor um sistema de mapeamento objeto-ontologia em linguagem de programação Java com suporte a objetos *desconectados*. Este

[6] Como exemplo, veja a resposta de James Leigh, criador do Alibaba, sobre a questão de instâncias desconectadas em: http://www.openrdf.org/forum/mvnforum/viewthread?thread=1819

sistema visa disponibilizar operações padrões de criação, recuperação, atualização e remoção de instâncias (CRUD, do inglês *create, retrieve, update and detele*); operações de consultas em SPARQL - *Simple Protocol and RDF Query Language*; além de um gerador de código Java que recebe ontologias e automaticamente cria código Java para a aplicação.

Desta forma, o trabalho lida tanto com aspectos mais gerais, como facilitar a criação de uma aplicação baseada em ontologias, quanto com problemas específicos, como carregamento tardio de instâncias no banco de dados. Apesar de ser um trabalho com enfoque em engenharia de software e banco de dados, as suas contribuições estão mais voltadas para a área de Ontologias. Seguem algumas dessas contribuições:

- Criação de operações CRUD para ontologias através do paradigma de orientação a objetos;
- Construção de um gerador automático de código Java a partir de arquivos OWL;
- Viabilização da execução de consultas em SPARQL com o retorno convertido em objetos Java correspondentes;
- Utilização da abordagem de objetos *desconectados*, presentes em ORMS tradicionais, em aplicações baseadas em ontologias;
- Criação de experimentos para avaliar as vantagens e desvantagens de um OOMS com objetos *desconectados* e de um OOMS com objetos *persistentes*.

1.2 Organização do Texto

Esta dissertação está dividida em sete capítulos. O Capítulo 1 introduz a problemática e os objetivos do trabalho proposto, enaltecendo a necessidade de criação de um OOMS com suporte a objetos *desconectados*, muito usados em aplicações com banco de dados relacional. No Capítulo 2, são apresentados os conceitos relacionados ao tema deste trabalho, como a teoria envolta da Web Semântica, Ontologias e algumas categorias de ferramentas que auxiliam o desenvolvimento de aplicações semânticas: Frameworks RDF, Ferramentas de Armazenamento de Triplas RDF e sistemas de mapeamento objeto-ontologia.

No Capítulo 3, são apresentados os trabalhos relacionados ao sistema proposto. Alguns sistemas de mapeamento objeto-ontologia são descritos bem como comparados com o JOINT-DE. Na conclusão do capítulo, uma tabela é apresentada mostrando as diferenças entre a presente proposta e os diversos sistemas detalhados.

O Capítulo 4 mostra como a ferramenta JOINT foi evoluída para suportar o JOINT-DE, tanto na arquitetura como em seu código, por intermédio de alguns diagramas de classes. Nesse capítulo, cada operação do JOINT-DE é descrita, tanto na perspectiva do desenvolvedor que está utilizando a ferramenta, quanto na sua operação interna, ilustrada por meio de frações de códigos e algoritmos.

No Capítulo 5, o estudo de caso é apresentado. Nele, é descrita a plataforma educacional online MeuTutor-ENEM, um sistema de acompanhamento individualizado baseado em ontologias para alunos que se preparam para o Exame Nacional do Ensino Médio (ENEM). Ademais, também é detalhado como a utilização do JOINT-DE na plataforma supriu algumas necessidades arquiteturais dela, além de apresentar alguns resultados quantitativos da implantação do JOINT-DE.

No Capítulo 6, um experimento foi projetado para avaliar, em termos de performance, a ferramenta proposta (JOINT-DE), em comparação a sua versão tradicional (JOINT), que utiliza a ferramenta OpenRDF Alibaba para o mapeamento. Cada operação de CRUD de cada ferramenta é avaliada e uma discussão geral é apresentada ao final do capítulo.

Por fim, no Capítulo 7, são apresentadas as considerações finais deste trabalho, bem como são definidos alguns trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo deste capítulo é apresentar a fundamentação teórica referente ao foco deste trabalho, fazendo uma pequena introdução sobre conceitos importantes da área que auxiliam na compreensão da pesquisa desenvolvida.

2.1 Web Semântica

A Web atual ainda é composta por uma grande quantidade de páginas estáticas ou dinamicamente geradas que se interligam. Tais páginas são feitas em uma linguagem chamada *Hyper Text Markup Language* (HTML), bastante utilizada para disponibilizar informações que são consumidas principalmente por humanos. As pessoas podem ler essas páginas e compreendê-las; todavia, os dados contidos nessas páginas não possuem um significado associado de modo a possibilitar sua interpretação semântica por parte dos computadores. Nesse cenário, surge a Web Semântica, que tem como ideia original a de estender a Web atual, descrevendo os dados e documentos atuais para que máquinas possam também entender e processar essa vasta coleção de informação (CARDOSO; HEPP; LYTRAS, 2007).

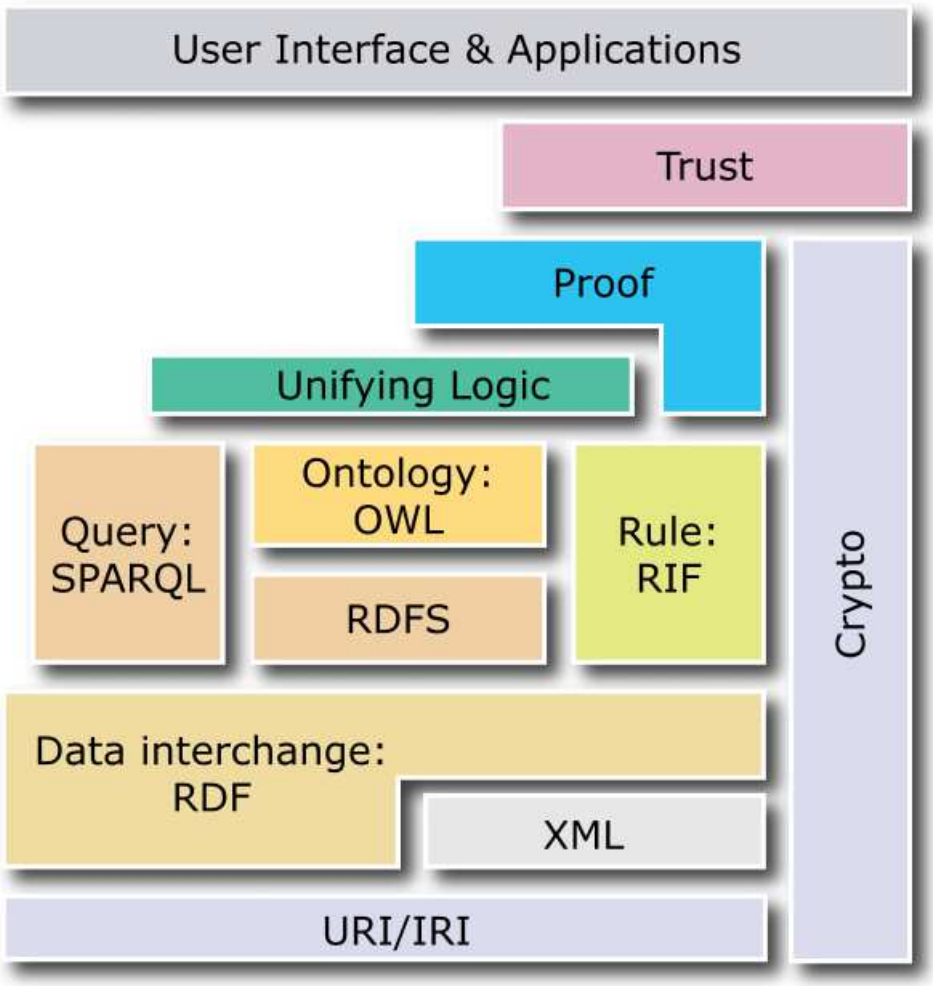
Surgindo como a nova geração da Web Atual, a Web Semântica tenta representar informação de tal forma que ela possa ser utilizada por máquinas não apenas com o intuito de exibir os dados, mas para automatização, integração e reuso através de diferentes aplicações (BOLEY; TABET; WAGNER, 2001). Vale ressaltar que o criador da *World Wide Web* (WWW), Tim Berners-Lee, foi quem iniciou a pesquisa relacionada a esta área (BERNERS-LEE et al., 2001). A Web Semântica vai trazer estrutura para o significado do conteúdo presente nas páginas Web, criando um ambiente em que agentes de software percorrem página por página, podendo facilmente realizar tarefas sofisticadas para os usuários finais, tais como agendamento de consultas médicas ou compras de pacotes de turismo. Tim Berners-Lee também propôs um modelo de camadas que foi e ainda é revisto pela *World Wide Web Consortium* (W3C). O modelo pode ser visto na Figura 2.

Os principais componentes, deste modelo, recomendados pela W3C, são descritos a seguir:

- RDF: O *Resource Description Framework* (RDF) é um framework para a criação de declarações em forma das conhecidas triplas RDF. Ele permite representar informações sobre os recursos na forma de grafos. A estrutura de qualquer elemento em RDF é vista como uma tripla da forma *<sujeito, predicado, objeto>*;

- RDFS: O RDF Esquema é uma extensão semântica do RDF e é responsável por prover um conjunto de recursos inter-relacionados. O objetivo do RDF-S é de pro-

Figura 2 – As camadas da Web Semântica.



Fonte: http://pt.slideshare.net/onlyjiny/linked-open-data-for-ecommerce

porcionar um conjunto básico de recursos que permita que documentos RDF sejam estruturados e validados. Utilizando RDFS é possível, por exemplo, criar hierarquias de classes e propriedades e descrever relacionamentos entre recursos;

- OWL: O *Web Ontology Language* (OWL) estende RDFS adicionando construções mais avançadas para descrever a semântica de declarações RDF. Ele permite expressividade de alto nível e inferência implícita, bem como possibilita ter restrições adicionais, como por exemplo, cardinalidade, as restrições de domínio e imagem e regras de união, disjunção, inversão e transitividade. Ele é baseado em lógica descritiva e, por isso, traz poder de raciocínio para a Web Semântica;

- SPARQL: O *Simple Protocol and RDF Query Language* (SPARQL) nada mais é do que uma linguagem de consulta para grafos baseado em RDF, ou seja, grafos na forma de triplas (Sujeito, Predicado, Objeto). Também é utilizado para extrair subgrafos de um RDF.

2.2 Ontologias

A Web Semântica se baseia em ontologias como suporte formal para representação de conhecimento e comunicação entre agentes de software. Desta forma, as ontologias têm um papel fundamental para que a Web Semântica possua um plano de conhecimento explicitamente declarado. A palavra ontologia vem do Grego ontos e logos, significando conhecimento do ser. Em filosofia, ontologia refere-se ao estudo do ser.

Em Computação, de maneira informal, uma ontologia define um conjunto de conceitos e suas relações, tais como a terminologia (vocabulário do domínio), definição explícita dos conceitos essenciais, suas classificações, taxonomias, relações e axiomas do domínio, incluindo hierarquias e restrições (DEVEDžIC, 2006). Formalmente, uma ontologia é uma especificação formal e explícita de uma conceitualização compartilhada (GRUBER, 1993). Para melhor entendimento, abaixo seguem detalhes sobre os termos citados nessa definição:

- Explícita: definições de conceitos, relações, restrições e axiomas;
- Formal: compreensível para agentes e sistemas;
- Conceitualização: modelo abstrato de uma área de conhecimento;
- Compartilhada: conhecimento consensual.

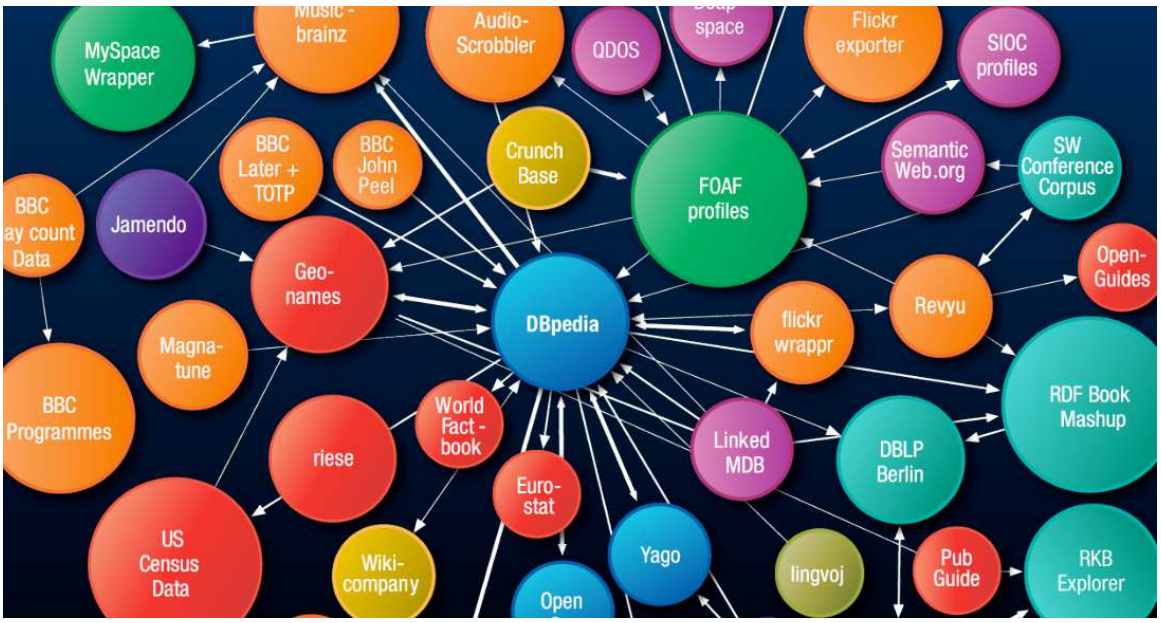
Pode-se concluir, a partir das características supracitadas sobre ontologias, que o conhecimento é explícito. Esse conhecimento equivale à descrição de determinada área do conhecimento, garantindo um conhecimento "consensual" sobre tal área. A partir do momento em que há um consenso, há a possibilidade e a viabilidade de compartilhar tais ontologias e integrá-las a outras áreas de conhecimento (através de outras ontologias), como mostrado na Figura 3.

Além disso, as ontologias têm o objetivo de serem expressas em uma linguagem formal que permita a compreensão por máquinas, provendo, assim, a automatização de atividades.

2.3 Desenvolvimento baseado em ontologias

A manipulação de instâncias é um importante passo no processo de desenvolvimento de aplicações baseadas em ontologias. Atualmente, existem duas abordagens principais utilizadas pelas ferramentas: desenvolvimento em triplas RDF e desenvolvimento orientado a objetos. Nas próximas seções, serão apresentadas as principais distinções e benefícios das abordagens mencionadas.

Figura 3 – Compartilhamento e integração de ontologias.



Fonte: http://www.w3.org/2010/Talks/1216-ghent-tbl/

2.3.1 Desenvolvimento orientado a triplas RDF

A maioria das APIs atuais ainda trabalham com o desenvolvimento baseado em triplas RDF. Dessa forma, os desenvolvedores da aplicação devem estar cientes de como funciona a ontologia em RDF, para poder manipular os dados através de cada tripla (sujeito, predicado e objeto) em código.

Para exemplificar, caso se deseje adicionar na ontologia um Recurso (sujeito) com várias propriedades inerentes a ele, são necessárias diversas linhas de código representando cada tripla do recurso, as quais capturam um único valor de cada propriedade. Similarmente, caso se deseje remover esse recurso, várias triplas devem ser removidas.

Por exemplo, várias linhas de código são necessárias para criar a instância *Alice*, da entidade *Person* com a propriedade de *name* tendo valor "Alice", usando a API do Sesame, que manipula instâncias através do desenvolvimento baseado em triplas RDF.

Figura 4 ilustra como adicionar este recurso em um repositório do Sesame.

2.3.2 Desenvolvimento orientado a objetos

Diferentemente de triplas RDF, aplicações orientadas a objetos manipulam dados através de objetos e seus atributos. Tais objetos são caracterizados por um conjunto de atributos e valores. Nesse sentido, faz-se necessária uma ferramenta que "traduza" as operações em objetos para a infraestrutura de triplas RDF da camada inferior. Algumas ferramentas foram criadas para prover esse paradigma para manipulação de instâncias em ontologias.

Com isso, os desenvolvedores não precisarão ter um conhecimento profundo da linguagem de representação da ontologia. Um objeto no código representa uma instância na

Figura 4 – Exemplo de código da ferramenta Sesame.

```
...
ValueFactory f = myRepository.getValueFactory();

// create some resources and literals to make statements out of
URI alice = f.createURI("http://example.org/people/alice");
URI name = f.createURI("http://example.org/ontology/name");
URI person = f.createURI("http://example.org/ontology/Person");
Literal alicesName = f.createLiteral("Alice");

RepositoryConnection con = myRepository.getConnection();
// alice is a person
con.add(alice, RDF.TYPE, person);
// alice's name is "Alice"
con.add(alice, name, alicesName);
...
```

Fonte: Elaborado pelo autor.

ontologia, seus atributos são mapeados com as propriedades das instâncias e as classes em RDF se tornam classes na linguagem de programação. Logo, para adicionar um recurso da ontologia, basta adicionar o objeto, facilitando, assim, o desenvolvimento dessas aplicações. Em comparação com o exemplo do Sesame (Figura 4), a Figura 5 mostra o mesmo recurso sendo adicionado ao repositório do Sesame usando a ferramenta Alibaba, que permite o desenvolvimento baseado no paradigma de orientação a objetos.

Figura 5 – Exemplo de código da ferramenta Alibaba.

```
...
ObjectConnection con = repository.getConnection();

// create a Person
Person alice = new Person();
alice.setName(“Alice");

// add a Person to the repository
con.addObject(alice);
...
```

Fonte: Elaborado pelo autor.

2.4 Ferramentas que auxiliam o desenvolvimento de sistemas semânticos

Nesta seção, serão detalhadas algumas categorias de ferramentas que auxiliam o desenvolvedor ao construir um sistema semântico, sendo responsáveis por determinada função na aplicação. Além disso, alguns exemplos de ferramentas em cada categoria serão apresentados e sempre que possível serão feitas analogias com sistemas de informação tradicionais que utilizam bancos de dados relacionais.

2.4.1 Frameworks para manipulação de RDF

A primeira ferramenta que se necessita ao construir um sistema baseado em ontologias são as APIs que oferecem a manipulação de triplas RDF e a conexão com o banco de dados RDF. Tais ferramentas funcionam como *middlewares* entre a aplicação e a ferramenta *triplestore*. Portanto, elas têm a mesma responsabilidade que um *Java Database Connectivity* (JDBC), com a diferença de que na área de ontologias essas ferramentas, geralmente, possuem um banco de dados RDF próprio. As APIs Sesame (BROEKSTRA; KAMPMAN; HARMELEN, 2002) e Jena (MCBRIDE, 2002) são as duas mais populares nesta categoria.

Sesame é um arcabouço Java para armazenamento e consulta em dados RDF. O Sesame é bem extensível e configurável no que diz respeito aos mecanismos de armazenamento (memória principal, arquivos binários ou banco de dados relacional), máquinas de inferência (RDFS), formatos de arquivo de ontologias (OWL, RDF ou N3) e linguagens de consulta (SPARQL e *Sesame RDF query language*). O Sesame oferece uma API para o usuário para que este possa ter acesso a seus repositórios, bem como uma interface HTTP que suporta o protocolo SPARQL. Várias extensões para o arcabouço foram desenvolvidas por terceiros.

Sesame é a principal ferramenta utilizada neste trabalho. Sua escolha deve-se a alta flexibilidade da API, que é uma das mais populares quando se trabalha com Web Semântica e ontologias. Além disso, a interface HTTP que o Sesame provê é muito útil para visualizar os dados que estão sendo manipulados durante o desenvolvimento.

Outra vantagem do Sesame é que, por ser extensível, o mecanismo de armazenamento pode ser alterado sem que haja qualquer impacto na aplicação construída com o Sesame. Dessa forma, pode-se desfrutar de toda a potencialidade da API e usar um repositório de alta performance como o OWLIM ou Virtuoso.

O Jena é um framework para construção de aplicações semânticas. Jena também é uma coleção de ferramentas e bibliotecas Java com o objetivo de suportar o desenvolvimento de sistemas baseados na web semântica.

A ferramenta inclui: uma API para leitura e escrita de dados RDF em arquivos; uma API para manipulação de ontologias em OWL e RDFS; um motor de inferência baseado em regras para raciocínio; mecanismos de armazenamento de grandes dados de triplas

RDF; um motor de consulta conforme a nova especificação do SPARQL.

2.4.2 Bancos de dados RDF

A segunda categoria a ser descrita possui as ferramentas que são responsáveis por armazenar RDF em algum tipo de banco de dados. Essas ferramentas são conhecidas por diversos nomes, como: banco de dados RDF, *triplestore*, repositórios de ontologias, entre outros. Em comparação com sistemas de informação tradicionais, essas ferramentas são similares aos Sistemas de Gerenciamento de Banco de Dados (SGBD). Contudo, apenas algumas ferramentas utilizam bancos de dados relacionais para armazenar as triplas em RDF (caso do Virtuoso), as outras utilizam um sistema de índice de arquivos (caso do OWLIM). A seguir, três bancos de dados RDF serão apresentados: OWLIM, Virtuoso e AllegroGraph.

OWLIM é uma extensão do Sesame que possui uma diversidade de repositórios semânticos (KIRYAKOV; OGNYANOV; MANOV, 2005). Esses repositórios possuem características como: armazenamento de RDF implementado em Java; alta performance; suporte à inferência das representações RDFS e OWL; escalabilidade e balanceamento de carga. OWLIM possui três versões de repositórios:

- OWLIM-Lite: é o repositório de maior performance e o único grátis. Apesar de ser em memória principal, ele possui um mecanismo de persistência em arquivos binários. Além disso, ele suporta inferência com dezenas de milhões de triplas mesmo em desktops atuais;

- OWLIM-SE: é o repositório de maior escalabilidade. Este repositório possui suporte à inferência e alta performance em consultas concorrentes. Ademais, ele consegue operar com o carregamento de dezenas de bilhões de triplas;

- OWLIM-Enterprise: é uma infraestrutura de *cluster* baseada nos repositórios do tipo OWLIM-SE. Esse repositório oferece infraestrutura escalável através do alto desempenho paralelizando consultas. Também oferece o balanceamento de carga e recuperação automática de falhas.

O Virtuoso é um servidor multiprotocolo que provê acesso ao banco de dados relacional interno através de ODBC/JDBC. Além de possuir um motor de busca SQL, o Virtuoso possui um servidor HTTP para usuários administradores, com terminais em diferentes protocolos (*e.g* Serviços Web) e linguagem de script interna (ERLING; MIKHAILOV, 2009).

Como o propósito do Virtuoso é ser um banco de dados universal, foi feita uma adaptação para suportar o armazenamento em triplas RDF. Nessa perspectiva, a ferramenta mapeia as triplas RDF em tabelas dentro do seu banco de dados relacional. O Virtuoso também oferece um motor de busca em SPARQL (com suporte a nova especificação 1.1),

que "traduz" as consultas em SPARQL feitas pelo desenvolvedor para a correspondente em SQL.

Assim como o OWLIM, o Virtuoso possui uma versão grátis; não bastasse isso, é código aberto. A grande diferença dessa versão para as versões pagas do Virtuoso é que as pagas possuem a opção de *cluster*. O virtuoso provê *drivers* de acesso tanto usando a API do Sesame como a do Jena.

AllegroGraph é um banco de dados RDF moderno e de alta performance (AASMAN, 2006). A ferramenta possui um mecanismo de persistência que faz uso, de maneira eficiente, da memória em combinação com o armazenamento interno baseado em disco. Dessa forma, AllegroGraph pode ser escalado para o armazenamento de bilhões de triplas RDF mantendo uma boa performance.

AllegroGraph também possui um motor de busca em SPARQL, inclusive suportando a nova especificação 1.1. Além disso, a ferramenta oferece suporte ao raciocínio em construtores RDFS e uma interface administrativa com diversos recursos. A versão grátis do AllegroGraph é limita ao armazenamento de 5 milhões de triplas; acima disso é necessário comprar a versão *Enterprise*.

2.4.3 Sistemas de mapeamento objeto-ontologia

Finalmente, a última categoria de ferramenta a ser detalhada neste trabalho engloba os sistemas de mapeamento objeto-ontologia (OOMS). Esse tipo de ferramenta tem como papel facilitar o desenvolvimento da aplicação semântica, ao permitir que o desenvolvedor foque na lógica da aplicação. Assim, não é necessário escrever códigos para conexão com o banco de dados RDF nem acessar os dados via triplas RDF, preservando as características de orientação a objetos da linguagem de programação.

Como o próprio nome diz, um sistema de mapeamento objeto-ontologia cria classes na linguagem de programação correspondentes às entidades em uma ontologia. Portanto, instâncias dessas entidades podem ser representadas na aplicação como objetos. Em sistemas de informação que usam bancos de dados relacionais, essas ferramentas são chamadas de sistemas de mapeamento objeto-relacional (ORMS), sendo alguma delas já bem consolidadas e amplamente utilizadas como a ADO.NET (ESPOSITO, 2002) e o Hibernate (BAUER; KING, 2006). Na área de ontologias, já existem algumas ferramentas, mas elas serão descritas no Capítulo 3 como trabalhos relacionados (já que a proposta deste trabalho também é uma solução OOMS).

Particularmente, em Java, a ferramenta Hibernate é uma implementação de ORMS. O Hibernate não apenas torna transparente para o desenvolvedor detalhes sobre o sistema de gerenciamento de banco de dados subjacente, mas essencialmente oferece o gerenciamento dos estados dos objetos. Esses estados estão descritos em sua documentação de referência online[1].

[1] Disponível em *http* : *//docs.jboss.org/hibernate/orm/*3.3*/reference/en/html/objectstate.html*.

Embora, o Hibernate defina esses estados de objeto, muitos dos ORMS e OOMS fornecem alguns ou todos esses estados de objetos. Nessa perspectiva, a descrição desses estados de objetos será abstraída quanto aos mecanismos do Hibernate e generalizada para facilitar a compreensão:

- Transientes: Um objeto é transiente se ele foi inicializado usando o operador new (em Java), ou seja, esse objeto é inicializado pela aplicação sem possuir qualquer vínculo ou correspondência com a conexão ao banco de dados (seja ele relacional ou em RDF). No caso do Java, se a aplicação remover a referência a esse objeto, o *garbage collector* irá destruir esse objeto. Se o desenvolvedor deseja que esse objeto transiente se torne um objeto persistente, então ele deve usar a conexão que o OOMS ou ORMS provê com o banco de dados para persistir os dados do objeto;

- Persistentes: Um objeto persistente tem uma representação correspondente ao banco de dados com um identificador. Ele pode ter sido previamente um objeto transiente e depois ter se tornado persistente, a partir do momento em que a ferramenta salvou seus dados, ou ele pode ter sido carregado direto pela ferramenta através da conexão com o banco de dados. Contudo, em ambos os casos, um objeto persistente é por definição limitado dentro do escopo da conexão. As mudanças feitas no objeto persistente, enquanto a conexão está aberta, são sincronizadas automaticamente com o banco de dados, sem a necessidade de um método *update* (se estiver numa transação as mudanças só serão salvas ao final dela);

- Desconectados: Um objeto desconectado é um objeto que já foi persistente, porém sua conexão com o banco de dados foi encerrada. Esse objeto ainda possui uma correspondência com o banco de dados, podendo ter suas informações alteradas pelo desenvolvedor. Porém, essas informações só serão salvas se o desenvolvedor utilizar a conexão da ferramenta para reconectar o objeto com o banco de dados, tornando-o persistente novamente (através do método *update*). Esse tipo de objeto pode ser usado como objeto de negócio das camadas superiores da aplicação ou até mesmo como objeto de transferência de dados.

3 TRABALHOS RELACIONADOS

Neste capítulo serão abordados alguns sistemas que suportam o desenvolvimento de aplicações baseadas em ontologias como uma solução de mapeamento objeto-ontologia. Cinco ferramentas serão apresentadas, com uma breve descrição sobre elas, relatando-se no que o sistema proposto difere de cada uma delas. Ao final do capítulo, uma sumarização das comparações será mostrada.

3.1 Jastor

Jastor é um gerador de código Java que cria *JavaBeans*[1] a partir de ontologias descritas em OWL (SZEKELY; BETZ, 2006).Com isso, os desenvolvedores podem convenientemente acessar uma ontologia armazenada em um modelo do Jena (vide Seção 2.4.1). O Jastor consegue gerar interfaces Java, suas implementações e fábricas, tudo baseado nas propriedades e hierarquia de classes descritas na ontologia.

Contudo, o Jastor não oferece ao desenvolvedor a possibilidade de trabalhar com objetos *desconectados*. Além disso, o projeto está descontinuado pois desde 2006 não tem uma atualização. Por causa disso, a ferramenta não é compatível com a mais nova versão do Jena (que suporta a nova especificação do SPARQL).

3.2 OWL2Java

Owl2Java é, também, um gerador de código, que cria uma simples API para trabalhar com dados em uma ontologia (ZIMMERMANN, 2010). A ferramenta permite uma fácil programação que opera com ontologias em OWL e RDF, abstraindo as peculiaridades dessas representações semânticas e da biblioteca que as acessa. O gerador de código cria classes Java para cada classe OWL presente na ontologia, com métodos estáticos para recuperar os indivíduos de cada classe e com *getters* e *setters* para todas as propriedades das classes.

Semelhante ao Jastor, OWL2Java tem como base a API do Jena e também não oferece o mapeamento objeto-ontologia através de objetos *desconectados*. A última atualização dessa ferramenta foi em 2010.

3.3 KOMMA

O *Knowledge Modeling and Management Architecture* (KOMMA) é um framework de aplicação para sistemas de software Java baseados em tecnologias da Web Semântica

[1] Segundo a especificação da Sun Microsystems, os JavaBeans são "componentes reutilizáveis de software que podem ser manipulados visualmente com a ajuda de uma ferramenta de desenvolvimento"

(WENZEL, 2010). KOMMA oferece funcionalidades para todas as camadas da aplicação semântica ao prover soluções para persistência com mapeamento de objeto-ontologia, gerenciamento de ontologias usando grafos nomeados e visualização específica de domínio. Além disso, a ferramenta disponibiliza edição direta de dados RDF.

A ferramenta é construída com a API do OpenRDF Sesame. Dessa forma, o Sesame faz o gerenciamento dos dados RDF no nível de triplas e o KOMMA faz o mapeamento de objetos para triplas RDF do Sesame.

Embora os objetos carregados pela ferramenta KOMMA tenham um sistema de cache de algumas propriedades, tais objetos ainda são vinculados à conexão com o repositório. Dessa forma, KOMMA provê apenas objetos *persistentes* para o desenvolvedor da aplicação.

3.4 Empire

Empire é uma implementação da Java Persistence API (JPA) para RDF, através do mapeamento de objeto-ontologia, permitindo consultas de bancos de dados RDF (armazenamento em triplas) com a linguagem SPARQL ou Sesame SeRQL (GROVE, 2010). JPA é uma especificação para gerenciamento de objetos Java, normalmente usada em conjunto com um RDBMS (padrão da indústria para ORMS em Java). Este mapeamento de classes Java para triplas RDF é obtido através do uso de anotações JPA padrões, as quais são estendidas com algumas especificações RDF, seja para *namespaces*, classes RDF ou propriedades RDF.

A ferramenta Empire provê um framework de persistência em Java para uso em projetos da Web Semântica, onde os dados são armazenados em RDF. Ao oferecer uma implementação do JPA, a ferramenta abstrai a manipulação de dados em RDF. Contudo, o objetivo maior da ferramenta é substituir implementações JPA existentes para bancos de dados relacionais, ao simplificar a mudança desses sistemas para sistemas baseados em modelos RDF. Assim como o KOMMA, o Empire faz uso da API do Sesame para manipulação em triplas RDF, mas o desenvolvedor pode configurar um *parser* para operar o Empire com a API do Jena.

O Empire é, aparentemente, o único projeto que suporta o uso de objetos *desconectados*. Entretanto, o projeto ainda é inicial e está com muitas limitações, principalmente no que se refere a seu gerador de código Java a partir de ontologias. Cite-se, por exemplo, que o Empire não consegue gerar o código Java das ontologias usadas neste trabalho (algumas exceções internas da ferramenta foram lançadas) tanto no capítulo do cenário de implantação quanto no capítulo do experimento. O código do gerador dessa ferramenta ainda possui alguns TODO[2] (traduzindo "para fazer"), de modo que ainda não suporta

[2] O código da classe que gera interfaces Java a partir de arquivos de ontologias do Empire está disponível em: https://github.com/mhgrove/Empire/blob/master/core/main/src/com/clarkparsia/empire/codegen/BeanGe acessado em Janeiro de 2014.

que mais de uma ontologia seja gerada ao mesmo tempo, não havendo, até o momento, uma versão estável da ferramenta[3]. Algumas das principais diferenças entre o gerador de código do Empire e do JOINT-DE são apresentadas na Tabela 1.

Tabela 1 – Principais diferenças entre os geradores do Empire e do JOINT-DE.

Características	**Gerador do Empire**	**Gerador do JOINT-DE**
Quantidade de ontologias suportadas	1	1 ou mais
Tipo de Geração	1:1 – Cada classe na ontologia vira uma classe Java	1:N – Cada classe na ontologia vira uma classe e uma interface, podendo ainda ter interfaces que representem a união de classes
Blank Nodes (Nó de ligação em um grafo sem um tipo próprio)	Não trata	Trata como um RDF:List, buscando cada elemento que o blank node conecta
Construtores owl	Não considera	Gera anotações como owl:UnionOf, owl:deprecated, entre outras, mas não infere sobre estas anotações

Fonte: Elaborado pelo autor.

3.5 OpenRDF Alibaba/Elmo

Elmo é um gerenciador de entidades RDF que mapeia implementações *JavaBeans* em triplas RDF para repositórios Sesame (MIKA, 2005). Elmo provê interfaces Java estáticas para recursos RDF, ou seja, o desenvolvimento de aplicações com Elmo é feito através de orientação a objetos centrada no sujeito. Por ser um sistema orientado a objetos, ele disponibiliza um modo de agrupar comportamentos comuns e separar papéis dentro de interfaces e classes. Os modelos gerados pelo Elmo são simples em expressar os conceitos envolvidos.

O AliBaba foi desenvolvido como o sucessor do Elmo e, portanto, usa princípios similares ao este para o mapeamento de objeto-ontologia. Assim como as outras ferramentas, o Alibaba é uma implementação de sistema objeto-ontologia centrada no sujeito da tripla RDF. Além disso, a ferramenta oferece implementações *RESTful* de bibliotecas cliente e servidor para armazenamento distribuído de documentos e metadados RDF.

A ferramenta OpenRDF Alibaba/Elmo talvez seja a mais conhecida solução de OOMS. Contudo, seu sistema de mapeamento objeto-ontologia é feito usando classes proxies, ou seja, a cada chamada de *getter* e *setter* do desenvolvedor uma requisição individual é feita ao banco de dados. Dessa forma, OpenRDF Alibaba/Elmo não suporta também o uso de objetos *desconectados*.

[3] A última versão acessada do Empire foi a versão 0.8.

3.6 Comparação com o Sistema Proposto

Neste capítulo, algumas das principais ferramentas existentes atualmente na comunidade científica foram apresentadas. Estas ferramentas tem o objetivo de facilitar o desenvolvimento de aplicações baseadas em ontologias, através de um mapeamento objeto-ontologia.

Dentre os sistemas apresentados, quatro deles não oferecem ao desenvolvedor a possibilidade de programar com objetos *desconectados*, comportando apenas objetos *persistentes*. Apenas a ferramenta Empire suporta objetos *desconectados*, porém seu gerador de código Java a partir de ontologias ainda não está completamente estável, o que inviabiliza o uso do Empire para aplicações com ontologias um pouco mais complexas. Além disso, duas das ferramentas propostas estão descontinuadas: Jastor e OWL2Java. A Tabela 2 sumariza e compara os trabalhos relacionados com o JOINT-DE.

Tabela 2 – Comparativo entre os trabalhos relacionados e o JOINT-DE.

Ferramenta	**Objetos Desconectados**	**Objetos Persistentes**	**Gerador Estável**	**API Base**
Jastor		X	X	Jena
OWL2Java		X	X	Jena
Empire	X			Sesame
KOMMA		X	X	Sesame
Alibaba/Elmo		X	X	Sesame
JOINT-DE	X		X	Sesame

Fonte: Elaborado pelo autor.

4 JOINT-DETACHED

No Capítulo 3, foram discutidos alguns sistemas de mapeamento objeto-ontologia, tais ferramentas acessam ontologias armazenadas através de objetos correspondentes às suas instâncias. Foi observado também, que quase todas as ferramentas proveem ao desenvolvedor objetos vinculados à uma conexão com o repositório, chamados de objetos *persistentes*.

Todavia, muitas aplicações na linguagem Java, principalmente as de multicamadas orientadas a objetos, possuem uma necessidade de que seus objetos de negócios transitem entre as camadas, desde a persistência até a camada de interface do usuário. Dessa forma, objetos *persistentes* não podem ser utilizados como objetos de negócio da aplicação, pois tais objetos só vivem dentro do escopo da conexão em que ele foi recuperado ou criado.

A fim de suprir essa necessidade, o presente trabalho propõe uma evolução no código da ferramenta[1] JOINT(HOLANDA et al., 2013b), com o objetivo de oferecer aos desenvolvedores a manipulação de suas ontologias armazenadas através de objetos *desconectados*. Esse novo sistema de mapeamento objeto-ontologia é chamado de JOINT-DETACHED (JOINT-DE).

Este capítulo visa descrever o JOINT-DE. A Seção 4.1 aborda a mudança arquitetural na ferramenta JOINT para permitir o desenvolvimento por meio de objetos *desconectados*. Na Seção 4.2, o gerador de código, responsável pelo mapeamento automático entre as ontologias e o código Java, será detalhado. As outras Seções apresentam e exemplificam como funciona o JOINT-DE no que se refere os métodos de criar, recuperar, atualizar, remover e consultar objetos.

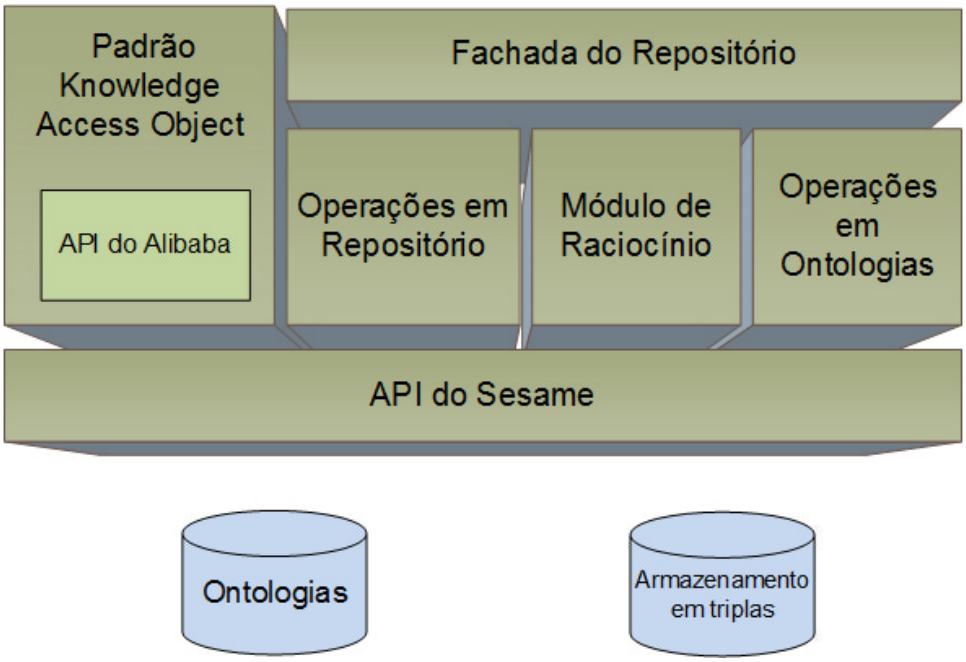
4.1 Reformulação Arquitetural do JOINT

JOINT-DE é uma evolução no código da ferramenta JOINT. Especificamente, essa evolução trata da mudança no código de mapeamento objeto-ontologia, onde foi retirado a ferramenta ALIBABA do JOINT e colocado um novo sistema visando oferecer aos desenvolvedores objetos *desconectados*. Para permitir essa evolução algumas mudanças foram realizadas na arquitetura original da ferramenta. Na Figura 6 a antiga arquitetura do JOINT é apresentada, onde pode-se observar que é uma arquitetura estritamente em camadas (BUSCHMANN; HENNEY; SCHIMDT, 2007), onde cada camada usa os serviços da camada inferior.

Segue uma descrição geral de cada camada (para maiores detalhes sobre esta arquitetura, ver (HOLANDA et al., 2013b)):

[1] O *Java Ontology Integrated Toolkit* - JOINT é uma ferramenta para auxiliar o desenvolvimento de aplicações semânticas. A ferramenta foi criada em 2012 por este autor, juntamente com outros integrantes do grupo de pesquisa NEES. O JOINT está disponível em: http://www.nees.com.br/pt/joint/.

Figura 6 – Arquitetura antiga do JOINT.



Fonte: Elaborado pelo autor.

- **API do Sesame:** Nessa camada se encontram os métodos necessários para conexão com as ferramentas de armazenamento de triplas RDF e manipulação dos dados em forma de triplas RDF (Sujeito, Predicado, Objeto);
- **Operações em Repositório:** Essa camada reúne serviços pertinentes à manipulação de repositórios do Sesame, possibilitando ao desenvolvedor operações de criação ou remoção de um repositório, limpar repositório (deletar todos os dados presentes) e criar cópias de backup de um repositório;
- **Módulo de Raciocínio:** Esse módulo tem como função inferir novos dados em um repositório através da execução de regras SWRL presentes no mesmo;
- **Operações em Ontologias:** Essa camada é responsável por agregar funcionalidades efetuadas em ontologia, como por exemplo: inserção e remoção de uma ontologia em um repositório e a geração de código Java a partir da API do Alibaba (na arquitetura antiga do JOINT, descrita aqui, além de estar presente no módulo KAO, o Alibaba também está presente no módulo de Operações em Ontologias);
- **Padrão KAO:** O Knowledge Object Access (KAO) é um padrão de persistência semelhante ao Data Access Object (DAO)[2], com a diferença de que o KAO não só trabalha com dados, mas trabalha com informações das ontologias. O padrão KAO tem como objetivo fornecer uma abstração do mecanismo de persistência utilizado,

[2] Detalhes sobre o padrão DAO em: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

proporcionando algumas operações específicas (como a criação, recuperação e remoção de instâncias, entre outros), sem expor detalhes sobre as conexões com o banco de dados RDF. A grande diferença entre o KAO e o DAO, é que ao usar o padrão KAO, o desenvolvedor irá criar classes concretas KAO para cada ontologia que ele deseja acessar e manipular. Essas classes herdam da classe abstrata predefinida *AbstractKAO*.

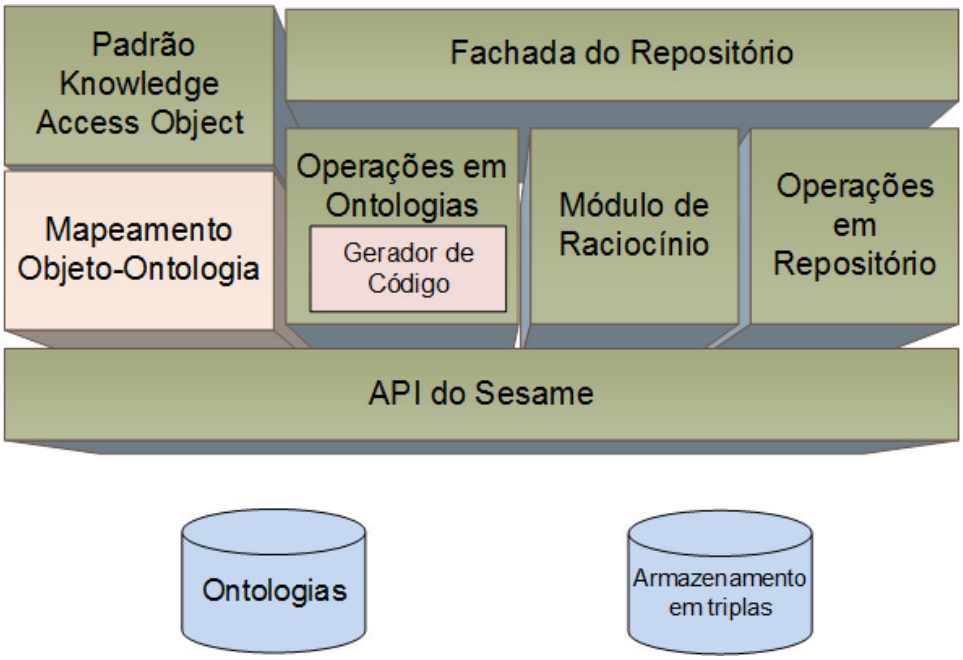
A ferramenta JOINT usa a API do Alibaba para fazer o mapeamento objeto-ontologia. Contudo, como foi visto na Seção 3.5, a ferramenta Alibaba não oferece a possibilidade de trabalhar com objetos *desconectados*. Portanto, para atingir os objetivos deste trabalho faz-se necessária a retirada dessa API da ferramenta JOINT.

Nesse sentido, um novo sistema de mapeamento objeto-ontologia foi desenvolvido para substituir a ferramenta Alibaba. O objetivo principal desse novo sistema é permitir que os objetos gerados automaticamente por ele possam ser objetos de negócio da aplicação, ou seja, tenham capacidade de transitar entre as diferentes camadas da aplicação sem a necessidade de manter uma conexão com o repositório ativa.

A Figura 7 ilustra como ficou a nova arquitetura do JOINT-DE. Uma nova camada foi adicionada representando o sistema implementado, chamada *Mapeamento Objeto-Ontologia*. Nessa camada, consiste os serviços de criar, recuperar, remover, atualizar e consultar instâncias em um repositório. O padrão KAO, que tem o papel de abstrair para o desenvolvedor qual mecanismo de persistência é utilizado, requer esta camada para acessar o repositório. Outra mudança na arquitetura é um novo gerador de código Java, presente na camada *Operações em Ontologias*.

Para ilustrar melhor o núcleo do sistema proposto, o diagrama de classes das camadas *Padrão KAO* e *Mapeamento Objeto-Ontologia* foi criado, veja a Figura 8. Assim como no JOINT antigo, a classe abstrata *AbstractKAO* é a principal classe deste pacote, nela contém as operações de criar, remover, atualizar (esta operação apenas no JOINT-DE), recuperar e consultar instâncias. Todavia, no JOINT antigo o *AbstractKAO* usava apenas a ferramenta Alibaba, que de fato era responsável por fazer o mapeamento de objeto-ontologia. Por conta da remoção da API do Alibaba, no JOINT-DE diversas outras classes tiveram que ser desenvolvidas ou modificadas para que a classe principal *AbstractKAO* pudesse manipular a ontologia por meio de objetos *desconectados*. A primeira classe desenvolvida é a classe estática *RepositoryFactory*, responsável pela criação do objeto *Repository*, usado para conexão com a ferramenta de armazenamento em triplas RDF. Esta classe usa o padrão de projeto *Singleton* (GAMMA et al., 1995), para que só haja uma instância de *Repository* na aplicação. Através do método *configureRepository*, o desenvolvedor pode configurar qualquer ferramenta de armazenamento em tripla para seu uso com o JOINT-DE, desde que essa ferramenta seja compatível com a interface *Repository* da API do Sesame.

Figura 7 – Arquitetura modificada para o JOINT-DE.



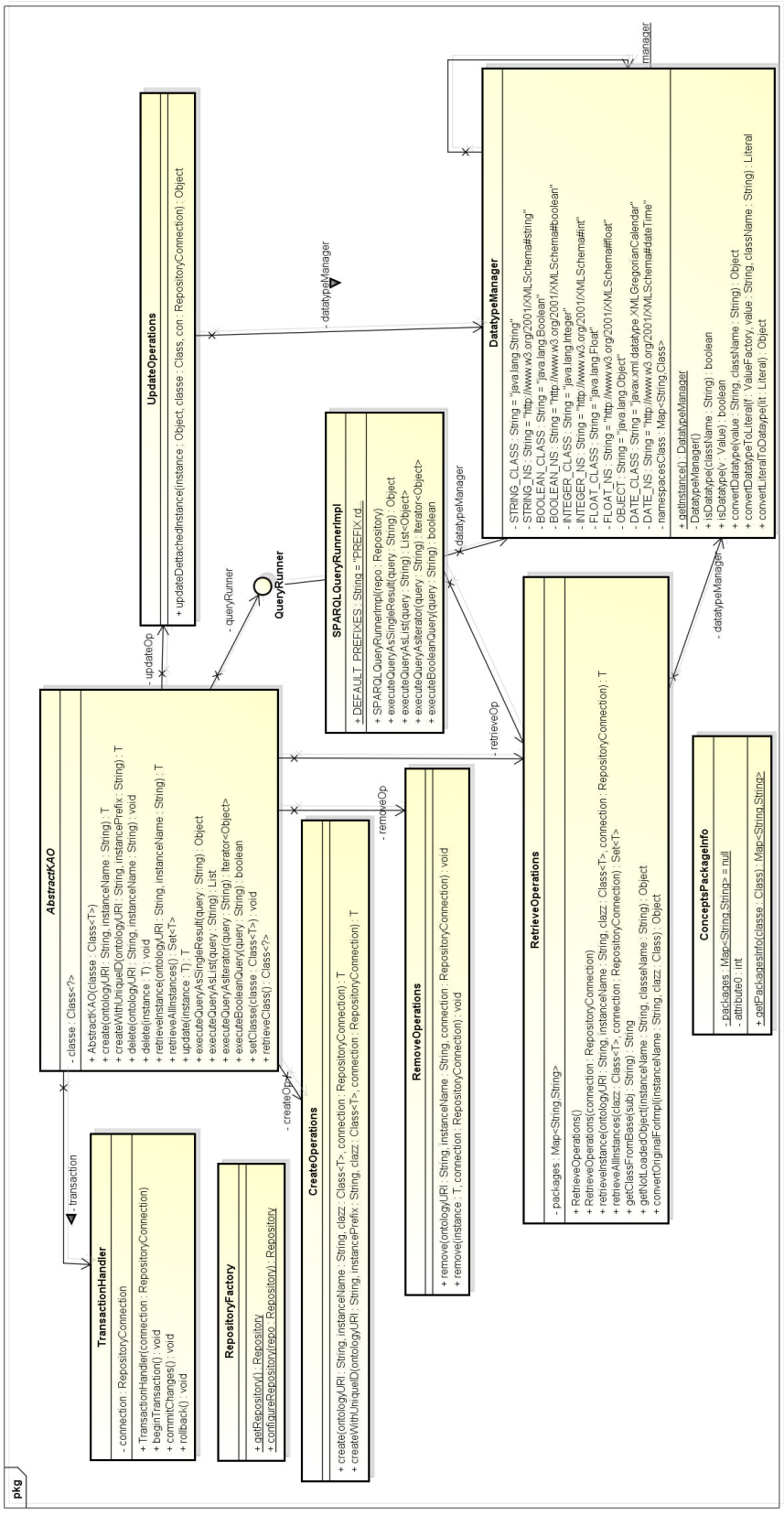
Fonte: Elaborado pelo autor.

Para cada operação CRUD e para operação de consulta desenvolveu-se uma classe responsável pelas suas implementações. Como pode ser visto na Figura 8, a classe *CreateOperations* contém a implementação dos dois métodos de criação de instância: *create*, cria-se uma instância no repositório com a URI da ontologia passada e o nome da instância; *createWithUniqueID*, cria-se uma instância no repositório com a URI da ontologia e um prefixo do nome da instância, o método então adiciona um identificador (ID) único para essa instância. Ambos os métodos retornam um objeto representando a instância criada.

Criou-se também, a classe *RemoveOperations*, responsável pela remoção de instâncias, que também possui dois métodos com distinção apenas nos parâmetros: *remove*, com a URI da ontologia e o nome da instância (que representa o ID da instância) o método remove ela do repositório; *remove*, neste método é passado o objeto que representa a instância e então ele é removido do repositório.

Em seguida, a classe *RetrieveOperations* foi criada com o objetivo de recuperar instâncias através de objetos *desconectados*. Esta classe tem dois métodos usados pela *AbstractKAO*: *retrieveInstance*, este método retorna um objeto que representa a instância dado a URI da ontologia e o nome da instância; *retrieveAllInstance*, já este método retorna um conjunto de objetos contendo todas as instâncias da classe desejada. Os outros métodos são de uso interno e serão detalhados na subseção 4.4. Além disso, a *RetrieveOperations* usa a classe estática *ConceptPackageInfo* para associar um determinado conceito da ontologia com sua respectiva classe no código Java.

Figura 8 – Diagrama de classes do sistema proposto.



Fonte: Elaborado pelo autor.

No que se refere as operações de CRUD, a última classe desenvolvida é a *UpdateOpera-*

tions. Ela só possui um método, o *updateDettachedInstance*, que basicamente consiste em atualizar um objeto *desconectado*, que foi modificado externamente, sincronizando suas informações atuais com o repositório.

Para executar as consultas, a classe *AbstractKAO* usa a interface *QueryRunner*, que é implementada pela classe *SPARQLQueryRunnerImpl*, fazendo consultas na linguagem SPARQL. Ao visualizar o diagrama de classes, nota-se que a *SPARQLQueryRunnerImpl* usa a *RetrieveOperations*, pois, uma vez executado determinada consulta, é necessário converter o resultado em objetos *desconectados*.

Entretanto, ao recuperar, atualizar ou consultar instâncias, é necessário saber em tempo de execução se uma determinada propriedade do objeto é outra instância ou um valor literal, ou seja, se é um tipo primitivo. Dessa forma, várias classes utilizam a *DatatypeManager*, responsável por serviços relacionados aos tipos de dados, como por exemplo: *isDatatype*, para verificar se determinado valor oriundo do repositório é um tipo de dados; *convertDatatype*, converter o valor em um objeto Java, seja *String*, *Integer*, *Boolean*, entre outros.

4.2 Gerador de Código

Nesta subseção o gerador de código Java do JOINT-DE será detalhado. O código deste gerador é uma extensão do código do gerador da ferramenta Alibaba.

A linguagem Java não suporta herança múltipla (GOSLING, 1995), ou seja, um objeto só pertence à uma determinada classe, não podendo ser uma instância de duas ou mais classes. Em contrapartida, indivíduos de uma ontologia podem ter múltiplos tipos associados à eles. Portanto, a solução encontrada pela ferramenta Alibaba foi gerar interfaces Java, uma vez que estas permitem a herança múltipla. Assim, uma interface Java pode herdar duas ou mais interfaces. Para implementar tais interfaces, a ferramenta Alibaba gera, em tempo de execução, proxies dinâmicos para que os desenvolvedores só possam acessar e manipular as interfaces geradas.

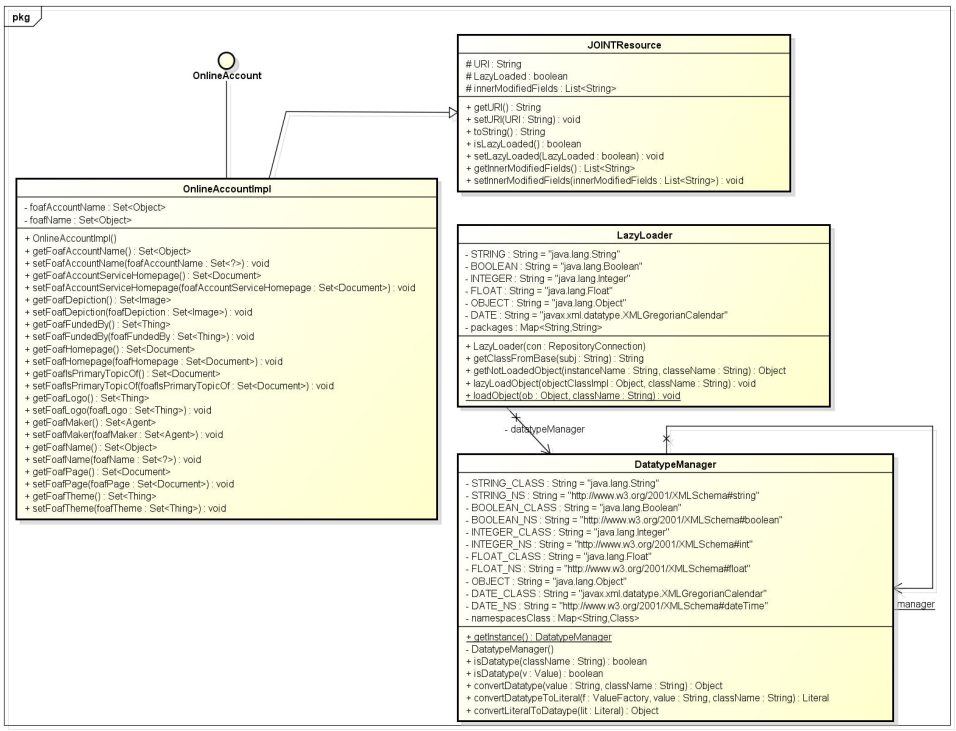
Para este trabalho, o gerador de código do Alibaba foi modificado e evoluído com o objetivo de não apenas gerar interfaces Java, porém classes concretas que implementam cada interface. Desenvolvedores ainda acessam e manipulam apenas as interfaces, porém ao invés de criar proxies dinâmicos em tempo de execução, JOINT-DE retorna um objeto da classe que implementa a determinada interface. Isso resulta, teoricamente, em uma menor sobrecarga da ferramenta em termos de desempenho, pois ao invés de criar implementações em tempo de execução com proxies a ferramenta usa códigos pré-compilados com as classes concretas geradas.

Figura 9 ilustra uma parte do código gerado pelo JOINT-DE para a entidade *OnlineAccount* da ontologia *Friend of a Friend* (FOAF)[3]. Como pode ser visto nesse diagrama

[3] Disponível em: http://www.foaf-project.org/.

de classes, o sistema gera a interface *OnlineAccount* e a classe concreta *OnlineAccountImpl* que implementa a interface. Toda a classe concreta gerada pela ferramenta proposta herda a classe *JOINTResource*, a qual possui alguns métodos e atributos necessários para os serviços de recuperação, criação e remoção de instâncias, como por exemplo o atributo URI que guarda a URI da instância representada pelo objeto da classe concreta.

Figura 9 – Diagrama de classes do exemplo *OnlineAccount* da FOAF.



Fonte: Elaborado pelo autor.

Outro ponto que todas as classes geradas têm em comum é fato de chamarem internamente a classe estática *LazyLoader*. Esta classe possui o método estático *lazyLoadObject*, responsável por carregar as propriedades de uma instância de maneira tardia. Isso se deve ao fato de que o mecanismo de recuperação do JOINT-DE só carrega o primeiro nível de propriedades de uma instância, pois carregar todo o ciclo de referência de uma instância de uma ontologia é muito custoso. Este cache de primeiro nível ficará mais claro na subseção 4.4.

A Figura 10 mostra uma parte do código da interface *OnlineAccount*, onde cada propriedade na ontologia gera no código da interface os métodos *get* e *set* respectivos. Como mencionado anteriormente, para cada interface gerada, JOINT-DE gera uma classe concreta que a implementa. A classe *OnlineAccountImpl* pode ser vista na Figura 11.

A primeira observação que pode ser feita referente a Figura 11 é que para cada *getter* e *setter*[4] (e.g. *getFoafAccountName* e *setFoafAccountName*) da classe e para a própria

[4] *Getters* e *Setters* são expressões utilizadas para denominar métodos *get* e *set*, respectivamente. Um *getter* é um método que retorna o valor de uma propriedade específica do objeto. Já o *Setter* é um

Figura 10 – Parte da interface *OnlineAccount* gerada pelo JOINT-DE.

```
/** An online account. */
public interface OnlineAccount extends Thing {

    /** Indicates the name (identifier) associated with this online account. */
    Set<Object> getFoafAccountName();
    /** Indicates the name (identifier) associated with this online account. */
    void setFoafAccountName(Set<?> foafAccountName);

    /** Indicates a homepage of the service provide for this online account. */
    Set<Document> getFoafAccountServiceHomepage();
    /** Indicates a homepage of the service provide for this online account. */
    void setFoafAccountServiceHomepage(Set<? extends Document>
            foafAccountServiceHomepage);
}
```

Fonte: Elaborado pelo autor.

classe, existe uma anotação *Iri* associada à eles. Anotações *Iri* contém o valor da URI relativa à propriedade ou classe. Com essas anotações o sistema pode mapear os objetos para a ontologia armazenada em triplas RDF. Também pode-se observar que além de implementar a respectiva interface, a classe implementa a interface *Serializable*[5]. Portanto, objetos carregados com a ferramenta proposta podem ser serializados e enviados através de diferentes aplicações sem perder suas propriedades.

Cada propriedade da classe na ontologia é mapeada como atributo privado na classe Java gerada. Por um lado, cada método *setter* da classe quando chamado altera o valor do atributo *innerModifiedFields*. Este atributo contém a lista de campos que foram modificados externamente. Assim, quando o objeto é enviado para a atualização do sistema sabe-se o que vai ser atualizado e o que deve continuar inalterado no repositório. Por outro lado, todo o método *getter* checa se esse objeto em particular já foi carregado a partir do repositório ou se tem apenas o valor da propriedade URI, que representa a instância para que possa ser carregado.

4.3 Criando Objetos

Nesta seção será apresentado o funcionamento do serviço de criação de instâncias da ferramenta proposta, tanto na visão do desenvolvedor que usa a ferramenta como sua operação interna.

Para executar as operações de CRUD e fazer consultas em uma ontologia com o JOINT-DE é necessário a criação de uma classe KAO representando a ontologia que deseja ser manipulada. No caso do exemplo da FOAF, foi criada a classe *FoafKAO*, que herda a classe *AbstractKAO*.

método que insere um valor em uma propriedade específica.

[5] A serialização de uma classe em Java é permitida ao implementar a interface java.io.Serializable: http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html.

Figura 11 – Parte da classe concreta *OnlineAccountImpl* gerada pelo JOINT-DE.

```
/** An online account. */
@subClassOf({"http://www.w3.org/2002/07/owl#Thing"})
@Iri("http://xmlns.com/foaf/0.1/OnlineAccount")
public class OnlineAccountImpl extends JOINTResource implements OnlineAccount,
                                                                 Serializable {

    public OnlineAccountImpl(){
        this.innerModifiedFields = new ArrayList<>();
    }

    private Set<Object> foafAccountName;

    /** Indicates the name (identifier) associated with this online account. */
    @Iri("http://xmlns.com/foaf/0.1/accountName")
    public Set<Object> getFoafAccountName(){
        if(!this.isLazyLoaded())
            LazyLoader.loadObject(this, this.getClass().getName());
        return this.foafAccountName;
    }
    /** Indicates the name (identifier) associated with this online account. */
    @Iri("http://xmlns.com/foaf/0.1/accountName")
    public void setFoafAccountName(Set<?> foafAccountName){
         this.innerModifiedFields.add("FoafAccountName");
         this.foafAccountName = (Set<Object>) foafAccountName;
    }

    private Set<Document> foafAccountServiceHomepage;

    /** Indicates a homepage of the service provide for this online account. */
    @Iri("http://xmlns.com/foaf/0.1/accountServiceHomepage")
    public Set<Document> getFoafAccountServiceHomepage(){
        if(!this.isLazyLoaded())
            LazyLoader.loadObject(this, this.getClass().getName());
        return this.foafAccountServiceHomepage;
    }
    /** Indicates a homepage of the service provide for this online account. */
    @Iri("http://xmlns.com/foaf/0.1/accountServiceHomepage")
    public void setFoafAccountServiceHomepage(Set<? extends Document>
            foafAccountServiceHomepage){
         this.innerModifiedFields.add("FoafAccountServiceHomepage");
         this.foafAccountServiceHomepage = (Set<Document>)
                 foafAccountServiceHomepage;
    }
```

Fonte: Elaborado pelo autor.

A Figura 12 mostra o código de criação de uma instância do tipo *OnlineAccount* utilizando a *FoafKAO*. Primeiro cria-se um objeto da classe *FoafKAO*, passando por parâmetro a interface *OnlineAccount*. Em seguida, também cria-se um novo objeto com o método *create*, passando como parâmetro a URI da ontologia FOAF e o nome da instância. O sistema, então, gera a instância no repositório e retorna para o desenvolvedor um objeto *desconectado* com a implementação da interface passada (no caso do exemplo *OnlineAccount*). A partir deste ponto, o desenvolvedor pode utilizar o objeto como ele desejar, ressaltando que as alterações feitas no objeto não irão refletir no repositório até que o objeto seja atualizado.

É importante destacar que na criação de um aplicativo o desenvolvedor apenas utilizará a interface gerada. As classes concretas são usadas internamente pelo JOINT-DE, pois o

Figura 12 – Código de criação de uma instância com o JOINT-DE.

```
//...

String foaf = "http://xmlns.com/foaf/0.1/";
String nomeConta = "Alice_Conta";
//Cria o kao passando a interface que se deseja operar
FoafKAO kao = new FoafKAO(OnlineAccount.class);
//Cria o objeto passando a uri da ontologia e o nome da instância
OnlineAccount aliceConta = kao.create(foaf, nomeConta);

//...
//Usa o objeto da forma que desejar
//...
```

Fonte: Elaborado pelo autor.

sistema precisa retornar uma implementação da interface. No caso da operação *create*, o sistema primeiro inicia a sessão (conexão com o repositório) e a transação, pois assim, caso alguma exceção Java for lançada o sistema pode recuperar o estado anterior do repositório com um *rollback*[6]. Inclusive, todas as operações CRUD são feitas através de transações para garantir a integridade dos dados[7].

Depois de iniciada a transação, a ferramenta começa a usar Java *Reflection* (FORMAN; FORMAN, 2004) para, primeiro, recuperar a classe que implementa a interface *OnlineAccount*, que neste caso é a *OnlineAccountImpl*, criando uma nova instância dessa classe em tempo de execução. O sistema então adiciona uma tripla RDF no repositório representando a nova instância criada, onde o sujeito é a URI da instância, o predicado é o *rdf:type*[8] e o objeto da tripla é a URI da entidade que ela pertence. A URI da entidade é obtida através da anotação *Iri* presente na classe gerada. Logo após, JOINT-DE invoca dois métodos padrões de todas as classes geradas por ele: o primeiro é o *setURI*, para inserir a URI da instância no objeto que será retornado; e o segundo é o *setLazyLoaded*, para inserir o valor de *true* que informa que este objeto está com todas as suas propriedades recuperadas (no caso, como esta instância é nova, não tem propriedade para recuperar). Por fim, o sistema salva as mudanças feitas na transação, fecha a sessão e retorna o objeto criado. A Figura 13 apresenta o algoritmo da operação de criar instâncias.

[6] Em banco de dados, *rollbacks* são importantes para garantir a integridade dos dados, pois eles podem restaurar a base para uma cópia anterior mesmo depois de algum erro ter ocorrido durante uma operação.

[7] Diferente de banco de dados, as transações não garantem a consistência dos dados nas ontologias. A consistência de uma ontologia é verificada se seu modelo possui apenas uma interpretação.

[8] RDF:type significa "é um" na especificação do RDF, por exemplo: foaf:Alice é um (rdf:type) foaf:Person.

Figura 13 – Algoritmo que mostra de forma genérica a criação de instâncias.

Algoritmo - Criação de Instâncias	
Entrada: uriInstancia; nomeClasse.	
1	iniciaSessao();
2	iniciaTransacao();
3	Classe c = recuperarClassePeloNome(nomeClasse + 'Impl');
4	Objeto r = c.novaInstancia();
5	adicionarTriplaRDF(uriInstância, RDF.Type, c.recuperarAnotacao(Iri));
6	Metodo m1 = c.recuperarMetodo('setURI');
7	m1.invocar(r, uriInstancia)
8	Metodo m2 = c.recuperarMetodo('setLazyLoaded');
9	m2.invocar(r, true);
10	salvarMudancasTransacao();
11	fecharSessao();
12	**Retorne** r;

Fonte: Elaborado pelo autor.

4.4 Recuperando Objetos

Na perspectiva do desenvolvedor que usa o JOINT-DE, o código para recupera uma instância contida no repositório é bastante similar ao de código de criação, mudando apenas o método chamado no KAO, como mostra a Figura 14.

Figura 14 – Código de recuperação de uma instância com o JOINT-DE.

```
//...

String foaf = "http://xmlns.com/foaf/0.1/";
String nomeConta = "Alice_Conta";
//Cria o kao passando a interface que se deseja operar
FoafKAO kao = new FoafKAO(OnlineAccount.class);
//Recupera o objeto passando a uri da ontologia e o nome da instância
OnlineAccount aliceConta = kao.retrieveInstance(foaf, nomeConta);

//...
//Usa o objeto da forma que desejar
//...
```

Fonte: Elaborado pelo autor.

Todavia, internamente, a operação de recuperação de instância é mais complexa. Embora ela também deva retornar um objeto de uma classe concreta, esse objeto deve ter todos seus atributos carregados com os valores presentes no repositório. Dessa forma, assim como na operação de criação, o sistema inicia recuperando a classe concreta e criando um novo objeto da classe. A fim de inserir valores em todos os atributos do objeto antes de retorná-lo, todos os métodos *setter* são recuperados. Depois, para cada método recuperado, o JOINT-DE recupera qual a propriedade na ontologia que aquele método está mapeado usando a anotação *Iri*. A URI da propriedade é usada em conjunto com a URI

da instância para verificar os dados presentes no repositório, ou seja, o sistema recupera todas as triplas RDF que tenham como sujeito a URI da instância e como predicado a URI da propriedade (os objetos destas triplas são os dados da propriedade).

A partir deste momento, a ferramenta começa a invocar cada *setter* com os dados obtidos no repositório. Se o dado for nulo, ele simplesmente invoca o método passando nulo como parâmetro. Senão, é preciso verificar se esse método representa uma propriedade funcional ou multivalorada[9]. Em ambos os casos, o sistema também checa se os dados recuperados são literais (i.e. tipos primitivos) ou são outras instâncias da ontologia. Gerando uma combinação dessas condições, quatro casos se formam. No primeiro caso, propriedade é funcional e o dado é literal, o método *setter* é invocado passando o dado convertido em um objeto de uma classe nativa do Java. No segundo caso, a propriedade é funcional e o dado é uma instância, o método *setter* é invocado passando o dado convertido como um objeto da classe concreta que ele pertence. Esse objeto possui apenas carregado a propriedade URI e a propriedade *lazyLoaded* como *false*, indicando que o objeto não foi carregado completamente. Nos outros dois casos restantes o processo ocorre de maneira similar, com a diferença de que ao invés de converter um dado, converte-se um conjunto de dados. Por fim, é invocado para o objeto principal, os métodos *setURI* com a URI da instância e *setLazyLoaded* com *true* e, então, retorna-se o objeto carregado. A Figura 15 mostra o algoritmo com a operação interna da recuperação de instância.

Vale ressaltar, que o algoritmo de recuperação de instância não é recursivo, o que implica que o objeto recuperado não carrega todo o seu ciclo de referência. O sistema apenas carrega as propriedades no primeiro nível, pois se tratando de ontologia, carregar todo o ciclo (grafo RDF) de uma instância é muito custoso.

Entretanto, para o desenvolvedor isso é transparente, pois se ele chamar um método *getter* que possui uma instância que não foi carregada o sistema abre uma conexão em tempo de execução e recupera essa instância em particular. Isso ocorre por que todo o método *getter* verifica se o objeto que está manipulando já foi carregado, caso não tenha sido chama-se a classe *LazyLoader* com o método estático *loadObject*.

Para exemplificar, veja o cenário em que um desenvolvedor recupera uma instância da classe *OnlineAccount*, ele recebe um objeto *desconectado* com todas as suas propriedades carregadas. Então, caso ele chame o método desse objeto *getFOAFAccountUserName*, que possui como retorno uma String, ele vai receber de retorno o valor normalmente. Porém, se ele chamar o método *getFOAFAccountOf*, que possui como retorno um *Person*, a instância de *Person* retornada não vai estar completamente carregada, ela possuirá apenas os valores dos atributos URI e *lazyLoaded*. A partir deste ponto, se ele chamar qualquer método *getter* com esse objeto não carregado, o JOINT-DE perceberá que suas propriedades não foram recuperadas e irá carregá-las em tempo de execução.

[9] Uma propriedade funcional em ontologias é uma propriedade que só possui um único valor. Por outro lado, uma propriedade multivalorada possui um conjunto de valores associado a ela.

Figura 15 – Algoritmo que mostra de forma genérica a recuperação de instâncias.

```
Algoritmo – Recuperação de Instâncias
Entrada: uriInstancia; nomeClasse.
1   iniciaSessao();
2   iniciaTransacao();
3   Classe c = recuperarClassePeloNome(nomeClasse + 'Impl');
4   Objeto r = c.novaInstancia();
5   Lista metodos = c.recuperarTodosMetodosSetter();
6   Para cada m em metodos
7           predicado = m.recuperarAnotacao(Iri);
8           Lista objetos = recuperarTriplas(uriInstancia, predicado, null);
9           Se triplas == null
10                  m.invocar(r, null);
11          Senão
12                  Se propriedadeFuncional(m)
13                          Se tipoDeDados(objetos)
14                                  m.invocar(r, converterTipoDeDados(objetos[0]));
15                          Senão
16                                  m.invocar(r, recuperarObjetoVazio(objetos[0]));
17                  Senão
18                          Se tipoDeDados(objetos)
19                                  m.invocar(r, converterTipoDeDadosEmLista(objetos));
20                          Senão
21                                  m.invocar(r, recuperarObjetoVazioEmLista(objetos));
22  Metodo m1 = c.recuperarMetodo('setURI');
23  m1.invocar(r, uriInstancia)
24  Metodo m2 = c.recuperarMetodo('setLazyLoaded');
25  m2.invocar(r, true);
26  salvarMudancasTransacao();
27  fecharSessao();
28  Retorne r;
```

Fonte: Elaborado pelo autor.

4.5 Atualizando Objetos

A próxima operação a ser detalhada é a de atualização de instâncias. A Figura 16 mostra na perspectiva do desenvolvedor como atualizar uma instância. Embora, nesta figura o objeto esteja sendo recuperado e atualizado pelo mesmo KAO, isso não é necessário. Na verdade, a instância pode ter sido recuperada em outra parte da aplicação do desenvolvedor, depois enviada por diversas camadas da arquitetura e alterada em alguma delas. No momento que a aplicação devolve a instância para o KAO atualizar, ele recupera as mudanças feitas no objeto e sincroniza com os dados no repositório.

O sistema proposto consegue descobrir quais propriedades daquele objeto foram alterados através do atributo *innerModifiedFields*, presente em todas as classes geradas pelo JOINT-DE. Toda vez que o desenvolvedor chama um método *setter* de qualquer objeto *desconectado*, esse método, internamente, adiciona seu nome no atributo *innerModified-*

Figura 16 – Código de atualização de uma instância com o JOINT-DE.

```
//...

String foaf = "http://xmlns.com/foaf/0.1/";
String nomeConta = "Alice_Conta";
//Cria o kao passando a interface que se deseja operar
FoafKAO kao = new FoafKAO(OnlineAccount.class);
//Recupera o objeto passando a uri da ontologia e o nome da instância
OnlineAccount aliceConta = kao.retrieveInstance(foaf, nomeConta);

//...
//Usa o objeto da forma que desejar
//...
Set<String> nomesConta = new HashSet<>();
aliceConta.setFoafAccountName(nomesConta);

// Atualiza para salvar as mudanças
kao.update(aliceConta);

//...
```

Fonte: Elaborado pelo autor.

Fields. Como consequência, no momento que o JOINT-DE for atualizar a instância, o atributo *innerModifiedFields* vai conter a lista de todos os métodos que foram alterados pelo desenvolvedor, impedindo a atualização desnecessária de propriedades que não foram alteradas.

Levando isto em consideração, fica mais fácil explicar como funciona a operação de atualização de uma instância. Em primeiro lugar, o sistema cria uma lista de triplas RDF, essa lista guardará todas as triplas que serão adicionadas ao repositório no fim da operação. Após isso, a classe concreta e a URI da instância que está sendo atualizada são recuperados. Ademais, a ferramenta recupera todos os métodos *getter*, cuja as respectivas propriedades foram modificadas pelo desenvolvedor (baseado no atributo *innerModifiedFields*).

Para cada método recuperado, o algoritmo faz uma série de operações. A primeira é verificar qual a URI da propriedade associada a este método. Depois, ele remove todas as triplas RDF no repositório que tenham como sujeito a URI da instância e predicado a URI da propriedade. Dessa forma, os dados antigos desta propriedade presentes no repositório são removidos. JOINT-DE, então, invoca o método *getter* da instância a fim de recuperar o valor atual da propriedade no objeto, tratando esse retorno como um objeto qualquer.

Em seguida, o sistema começa a verificar de que tipo esse retorno pertence e se o método representa uma propriedade funcional ou não. Se o retorno for um objeto nulo, nada se tem a fazer, pois as triplas RDF já foram removidas do repositório. Apenas se o retorno for diferente de nulo é que ele representa novos dados que devem ser adicionados ao repositório. Neste caso, é verificado se a propriedade é funcional ou multivalorada. Para o caso de propriedade funcional, a ferramenta checa se o retorno é um tipo de dado e, assim,

converte o objeto que pertence à uma classe Java para um literal RDF e adiciona na lista de novas triplas a tripla RDF com a URI da instância como sujeito, a URI da propriedade como predicado e o literal RDF como o objeto. Se o retorno não for um tipo de dado e sim uma outra instância, então, o algoritmo recupera a URI dessa instância e adiciona a tripla RDF mudando só objeto que será essa URI (note que neste caso a propriedade representa uma relação entre duas instâncias). Para o caso de propriedade multivalorada, o algoritmo opera de maneira similar. A diferença é que o retorno representa uma lista de valores para serem atualizados e, para cada valor uma tripla RDF deve ser adicionada à lista de novas triplas.

Figura 17 – Algoritmo que mostra de forma genérica a atualização de instâncias.

Algoritmo – Atualização de Instâncias

Entrada: instancia; nomeClasse.

```
1  iniciaSessao();
2  iniciaTransacao();
   Lista novasTriplas = novaLista();
3  Classe c = recuperarClassePeloNome(nomeClasse + 'Impl');
4  uriInstancia = instancia.recuperarURI();
5  Lista metodos = recuperarTodosMetodosGetterModificados(instancia);
6  Para cada m em metodos
7      predicado = m.recuperarAnotacao(Iri);
8      removerTriplas(uriInstancia, predicado, null);
9      Objeto retornoM = m.invocar(instancia);
10     Se retornoM != nulo
11         Se propriedadeFuncional(m)
12             Se tipoDeDados(retornoM)
13                 Literal obj = converterLiteral(retornoM);
14                 novasTriplas.add(uriInstancia, predicado, obj);
15             Senão
16                 uriObjeto = retornoM.recuperarURI();
17                 novasTriplas.add(uriInstancia, predicado, uriObjeto);
18         Senão
19             Lista retornosM = converterRetornoEmLista(retornoM);
20             Se tipoDeDados(retornoM)
21                 Para cada o em retornosM
22                     Literal obj = converterLiteral(o);
23                     novasTriplas.add(uriInstancia, predicado, obj);
24             Senão
25                 Para cada o em retornosM
26                     uriObjeto = o.recuperarURI();
27                     novasTriplas.add(uriInstancia, predicado, uriObjeto);
28 adicionarColecaoTriplas(novasTriplas);
29 instancia = limparTodosMetodosGetterModificados(instancia);
30 salvarMudancasTransacao();
31 fecharSessao();
32 Retorne instancia;
```

Fonte: Elaborado pelo autor.

Depois de percorrer todos os métodos *getter* modificados, a lista de novas triplas conterá todas as triplas que devem ser adicionadas no repositório para que o objeto passado como parâmetro esteja sincronizado com o repositório. O sistema, então, adiciona essa lista de triplas no repositório em uma única operação (diminuindo o custo de adição de triplas individuais). Por fim, o atributo *innerModifiedFields* é modificado para que seu estado inicial, inferindo que a instância neste momento está atualizada. A Figura 17 apresenta o algoritmo da operação de atualização.

4.6 Removendo Objetos

A última operação de CRUD é a de remoção de instâncias. Assim como nos métodos de criar e recuperar instância, o desenvolvedor cria um KAO passando a interface que ele deseja operar e, ao chamar o método *delete*, passa como parâmetros a URI da ontologia e o nome da instância. A Figura 18 mostra o código de remoção de instâncias na perspectiva do desenvolvedor.

Figura 18 – Código de remoção de uma instância com o JOINT-DE.

```
//...

String foaf = "http://xmlns.com/foaf/0.1/";
String nomeConta = "Alice_Conta";
//Cria o kao passando a interface que se deseja operar
FoafKAO kao = new FoafKAO(OnlineAccount.class);
//Remove o objeto passando a uri da ontologia e o nome da instância
kao.delete(foaf, nomeConta);

//...
```

Fonte: Elaborado pelo autor.

Internamente, a operação de remoção de instâncias é a mais simples de todas as operações CRUD. Isso se deve ao fato de que ela é a única que não usa a API de *Reflection* do Java. Embora, como todas as outras operações, seu algoritmo esteja dentro de uma unidade de transação, ele só precisa fazer duas operações: i) remover todas as triplas RDF que tem como sujeito a URI da instância que será apagada; ii) remover todas as triplas RDF que tem como objeto essa mesma URI. Dessa forma, o sistema remove todas as triplas da instância, inclusive, aquelas em que outras instâncias a referenciam como objeto. A Figura 19 mostra o algoritmo da operação de remoção de uma instância.

4.7 Consultando Objetos

Enfim, o último serviço a ser detalhado é o de consultar objetos. Ao usar consultas SPARQL em um repositório, o desenvolvedor na maioria das vezes consulta determinadas instâncias, ou seja, a consulta em SPARQL retorna uma ou um conjunto de instâncias.

Figura 19 – Algoritmo que mostra de forma genérica a remoção de instâncias.

Algoritmo - Remoção de Instâncias	
Entrada: uriInstancia; nomeClasse.	
1	iniciaSessao();
2	iniciaTransacao();
3	removerTriplas(uriInstancia, null, null);
4	removerTriplas(null, null, uriInstancia);
5	salvarMudancasTransacao();
6	fecharSessao();

Fonte: Elaborado pelo autor.

O JOINT-DE provê que os retornos dessas consultas sejam também na forma de objetos *desconectados*.

Para fazer uma consulta SPARQL com o JOINT-DE, o desenvolvedor deve colocar a consulta dentro da classe KAO que ele está operando, pois os métodos de consultas da classe *AbstractKAO* são do tipo *protected* (só classes que a herdam podem chamar esses métodos). Na Figura 20 pode-se ver um exemplo de consulta na *FoafKAO*, onde a consulta tem como objetivo recuperar todas as instâncias de *Person*.

Figura 20 – Consultas dentro da classe *FoafKAO*.

```
public class FoafKAO extends AbstractKAO {

    public <T extends Object> FoafKAO(Class<T> classe) {
        super(classe);
    }

    public List<Person> recuperarTodasPessoas(){
        String query = "PREFIX rdf:<" + RDFUris.RDF + "> "
                + "PREFIX foaf:<http://xmlns.com/foaf/0.1/> "
                + "SELECT ?p "
                + "WHERE {?p rdf:type foaf:Person}";

        return this.executeQueryAsList(query);
    }
}
```

Fonte: Elaborado pelo autor.

Vale observar que o desenvolvedor não precisa converter o retorno do método de consulta para a instância que ele deseja operar. Internamente, o sistema executa a consulta através da API do Sesame e então, verifica se o retorno é um literal (convertendo para a classe nativa Java respectiva) ou uma instância. Se for uma instância ele usa o método de recuperar instância, já detalhado anteriormente. É bom frisar, que no caso das consultas, o sistema descobre a que classe a instância pertence usando mecanismos internos da API do Sesame. Dessa forma, não é necessário passar a interface *Person*, por exemplo, como

parâmetro no método de consulta. Também em razão disto, se o desenvolvedor utilizar o KAO apenas para acessar os métodos de consulta dentro dele, ele poderá passar como parâmetro em seu construtor qualquer interface sem gerar nenhum impacto no algoritmo de consultas. A Figura 21 mostra um exemplo de consulta usando o JOINT-DE.

Figura 21 – Código de consulta em ontologias usando o JOINT-DE.

```
//...

//Cria o kao passando a interface que se deseja operar
FoafKAO kao = new FoafKAO(OnlineAccount.class);
//Consulta todos os objetos do tipo Person
List<Person> pessoas = kao.recuperarTodasPessoas();

//...
//Usa os objetos da forma que desejar
//...
```

Fonte: Elaborado pelo autor.

5 ESTUDO DE CASO

Neste capítulo, será descrita a implantação do JOINT-DE em uma aplicação real chamada MeuTutor-ENEM. O MeuTutor-ENEM foi escolhido como um bom cenário, pois ele foi projetado com uma arquitetura em multicamadas, além de necessitar que seus objetos do modelo de negócio sejam provenientes das suas ontologias. Este capítulo mostra as limitações da aplicação semântica MeuTutor-ENEM ao utilizar OOMS que apenas provê objetos *persistentes*, e, em seguida, enfatiza os benefícios obtidos com a mudança para o JOINT-DE.

5.1 MeuTutor-ENEM

O MeuTutor-ENEM[1] é um ambiente educacional web inteligente que busca realizar o acompanhamento da aprendizagem dos alunos de forma personalizada, focando na qualidade do ensino e no desempenho dos estudantes. O ambiente auxilia alunos do ensino médio a se prepararem para o Exame Nacional do Ensino Médio (ENEM), disponibilizando as seguintes disciplinas para estudo: Matemática, Português, Literatura, Biologia, Química, Física, História, Geografia, Inglês e Espanhol.

O MeuTutor-ENEM foi projetado com base em três aspectos: ambiente gamificado (KAPP, 2012), aprendizagem personalizada e experiência social. Para motivar o aluno a continuar aprendendo, o sistema faz uso de elementos de jogos (ambiente gamificado), tais como troféus, missões, pontos e níveis. O objetivo principal da Gamificação é aumentar o engajamento dos usuários por meio do uso de técnicas semelhantes àquelas presentes em jogos (FLATLA et al., 2011), fazendo com que os usuários se sintam no controle de suas ações e se motivem com as tarefas (PAVLUS, 2010).

Uma definição de Gamificação segundo Kapp (KAPP, 2012) é: "*Gamificação é o uso de mecânica, ideias e estética de jogos (e.g. contexto, feedback rápido, competição, fases, conquistas, pontos e etc.), para engajar pessoas, motivar ações, promover aprendizado e solucionar problemas*".

Nesse contexto, é possível observar na Figura 22 alguns elementos de jogos presentes na plataforma. Os troféus recompensam os estudantes por determinadas ações no sistema, como por exemplo acertar 10 questões seguidas. Os níveis representam a evolução do conhecimento dos estudantes em cada disciplina. Já os pontos consistem no elemento de jogo mais básico na plataforma; a maioria das interações do estudante com o sistema é recompensada com pontos. Por fim, os rankings e missões possibilitam, respectivamente, ao estudante competir com os amigos e maximizar sua evolução no sistema.

[1] A plataforma está disponível no link: www.enem.meututor.com.br.

Figura 22 – Elementos de jogos presentes no MeuTutor-ENEM.

Fonte: http://enem.meututor.com.br

No MeuTutor-ENEM, as questões e vídeo-aulas se adequam aos alunos (aprendizagem personalizada). A aprendizagem personalizada tem por objetivo aplicar métodos, técnicas e tecnologias com vistas a identificar limitações e necessidades específicas de um estudante ou de um grupo de estudantes de modo a personalizar a aprendizagem. Dentre as abordagens conhecidas para a personalização da aprendizagem, o MeuTutor-ENEM faz uso de Sistemas Tutores Inteligentes (BITTENCOURT et al., 2009)(DEVEDŽIC, 2006) e Mineração de Dados Educacionais (WITTEN; FRANK, 2005).

Com o desenvolvimento de técnicas de mineração de dados educacionais é possível identificar e sanar (ou pelo menos minimizar) alguns problemas educacionais, principalmente em ambientes virtuais de aprendizagem. Por exemplo, é possível verificar quais ações pedagógicas são ineficazes e identificar métodos que proporcionem melhores benefícios educacionais ao aluno (PAIVA et al., 2012). No MeuTutor-ENEM, o aluno ao estudar na plataforma, tem seu perfil de interações no sistema minerado com o objetivo de verificar quais os seus pontos fracos (e fortes), identificando, assim, qual o próximo recurso educacional que deverá ser oferecido a ele: seja outro problema para desafiá-lo, seja um vídeo para suprir sua deficiência sobre determinado conceito. A Figura 23 ilustra o momento em que vídeos são recomendados para o aluno quando ele erra uma questão.

Na verdade, todo esse conceito de personalização automática da aprendizagem no MeuTutor-ENEM provém do uso de técnicas da área de Sistemas Tutores Inteligentes (STIs). STIs são sistemas educacionais que fazem uso de recursos da inteligência artificial para personalizar a aprendizagem de acordo com a necessidade de cada estudante.

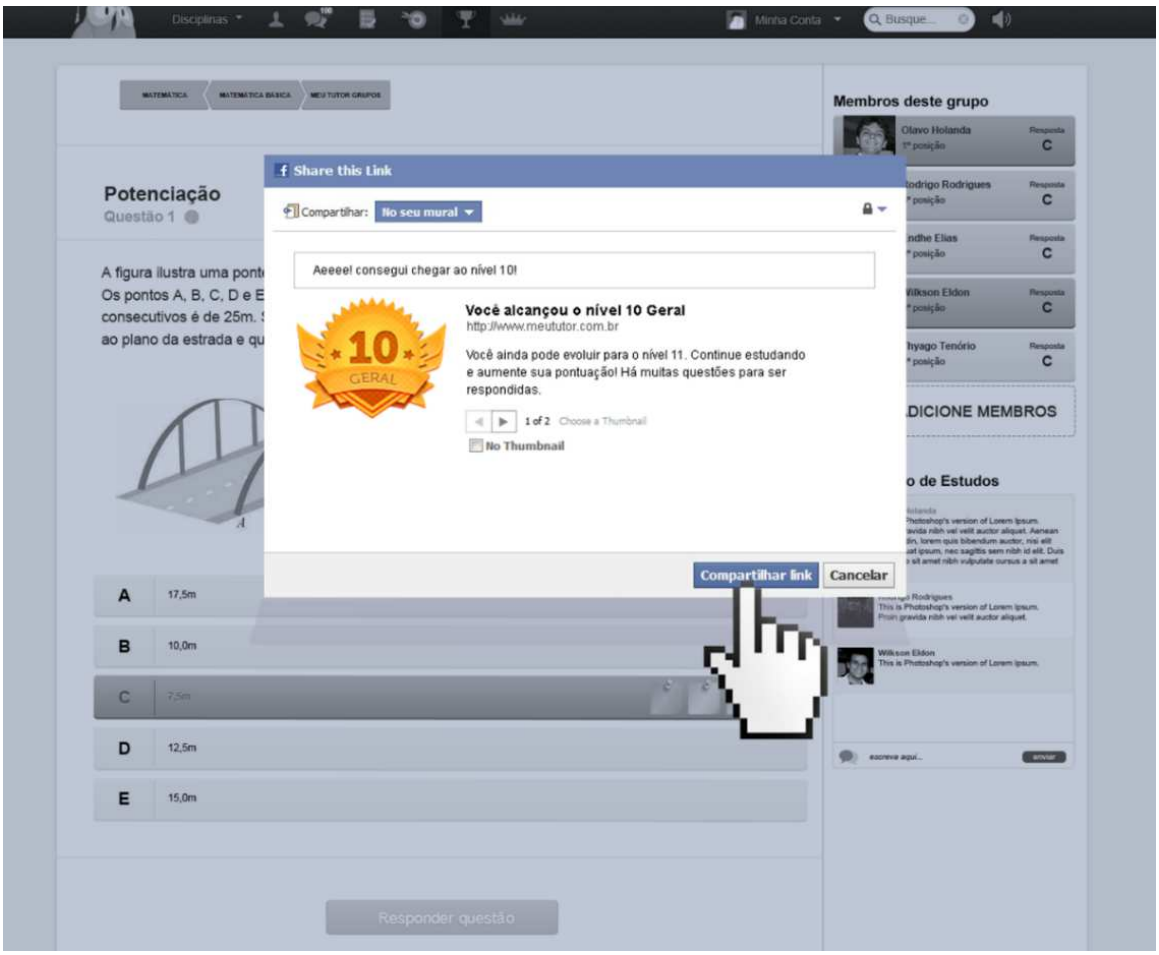
O MeuTutor-ENEM é um STI que usa diferentes técnicas para tutorar o aluno dentro da plataforma. Dentre essas técnicas, destacam-se a estratégia pedagógica baseada em algoritmos genéticos (MITCHELL, 1998) e a técnica *knowledge tracing* (CORBETT; ANDERSON, 1994) para verificação da aquisição de conhecimento do aluno.

Figura 23 – Tela de recomendação de vídeo para o aluno.

Fonte: http://enem.meututor.com.br

Embora, o MeuTutor-ENEM tenha sido projetado com o intuito de proporcionar autonomia no aprendizado do aluno, outros estudantes podem ajudá-lo na construção do conhecimento por meio de técnicas da aprendizagem colaborativa (ISOTANI et al., 2009). Para tanto, o aluno pode criar grupos de estudo e convidar outros usuários para estudar um determinado assunto, debatendo as questões e argumentando sobre cada opinião. Além disso, como se observa na Figura 24, o aluno pode compartilhar suas realizações no Facebook, potencializando sua experiência social. Todos esses aspectos foram embutidos na plataforma de uma maneira intuitiva para que o estudante possa aproveitar ao máximo todos os recursos disponíveis.

Figura 24 – Compartilhamento no Facebook.



Fonte: http://enem.meututor.com.br

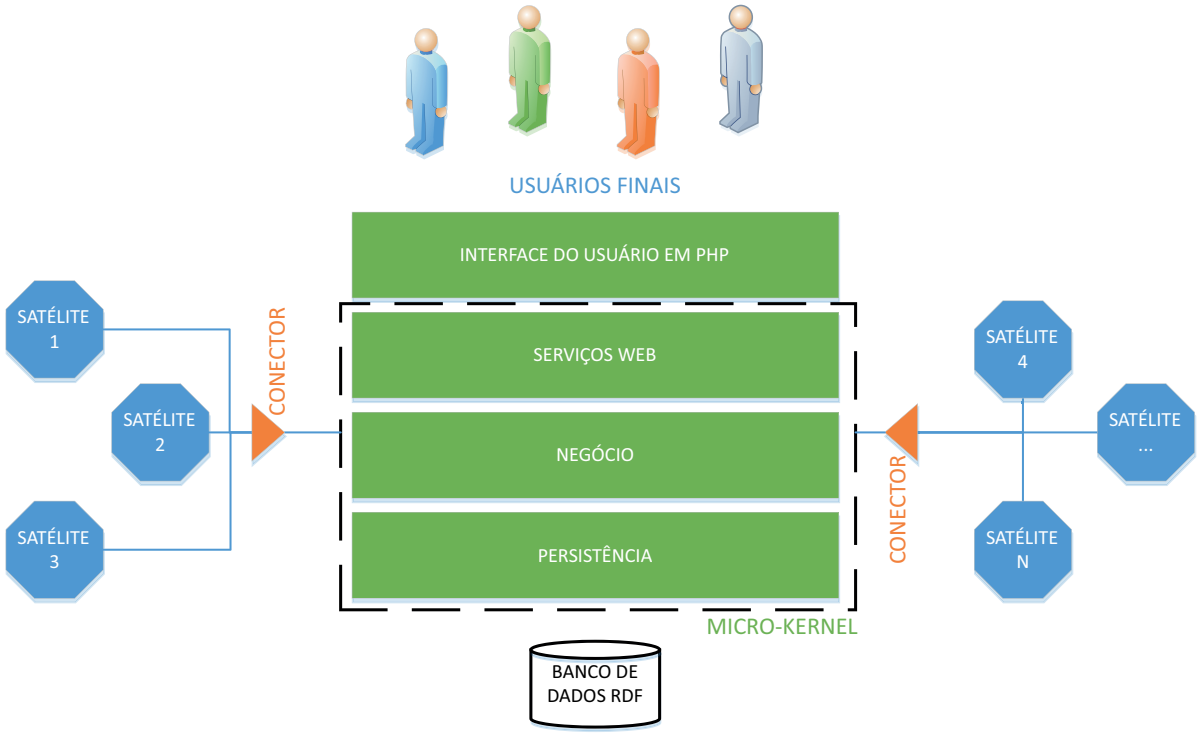
5.1.1 Visão geral da arquitetura

O MeuTutor-ENEM foi desenvolvido como um produto de uma linha de produto de software[2] semântica (SILVA et al., 2010). Nessa perspectiva, ele possui uma certa variabilidade no que se refere à implementação de algumas de suas funcionalidades internas. Por exemplo, a plataforma pode detectar que um determinado estudante não está aprendendo ao usar uma estratégia pedagógica baseada em problemas e, então, ela modifica, em tempo de execução, o componente responsável pela estratégia pedagógica para uma abordagem baseada em problemas e vídeo-aulas.

A arquitetura do MeuTutor-ENEM foi projetada analisando essa necessidade de variabilidade de algumas de suas funcionalidades. Todo o sistema é baseado em componentes que são unidades de softwares que encapsulam uma série de funcionalidades. Por questões de espaço, o diagrama de componentes da plataforma foi omitido, mas a visão geral da arquitetura pode ser vista na Figura 25.

[2] Uma linha de produtos de software é um conjunto de sistemas de software que têm determinadas funcionalidades em comum, sendo desenvolvidos tendo esta mesma base e variando apenas alguns

Figura 25 – Visão geral da arquitetura do MeuTutor-ENEM.



Fonte: Elaborado pelo autor.

Para tratar a variabilidade da plataforma, utilizou-se o padrão arquitetural microkernel (BUSCHMANN; HENNEY; SCHIMDT, 2007). Todos os componentes que não possuem variabilidade são colocados dentro do microkernel da arquitetura. Esses componentes são chamados de obrigatórios, pois eles estão presentes em qualquer cenário de configuração do MeuTutor-ENEM (e.g. componentes *Login* e *Cadastro*).

A variabilidade da arquitetura é feita com o uso dos conectores. Os conectores são responsáveis por chavear qual componente satélite será utilizado pelo microkernel. Por exemplo, um estudante visualiza uma questão de múltipla escolha que o sistema recomendou, mas não se adapta a este tipo de problema; neste caso, o conector responsável pelo componente de problemas pode mudar o satélite, que antes era múltipla escolha, para problemas de verdadeiro ou falso.

Como também pode ser observado na arquitetura, além de usar o padrão microkernel, outro padrão arquitetural também é utilizado: o de multicamadas. O próprio microkernel é dividido em três camadas: i) a camada de Serviços Web, responsável por agregar todos os serviços web que a plataforma, construída na linguagem Java, disponibiliza; ii) camada de Negócio, na qual se encontram todas as funcionalidades que representam a lógica do sistema; iii) camada de Persistência, responsável pelo acesso e comunicação com o banco de dados RDF. Note também que a camada de serviços web é utilizada pela sua camada superior, sendo esta responsável pela interface do usuário (UI). Vale frisar que a camada

aspectos.

de UI foi construída usando a linguagem PHP.

Contudo, o foco do JOINT-DE é na camada de persistência. Sendo assim, os detalhes da camada de persistência atual do MeuTutor-ENEM serão apresentados, demonstrando-se, ainda, como esta camada e o OOMS utilizado dentro dela limitam em alguns aspectos a plataforma MeuTutor-ENEM.

5.2 Camada de Persistência

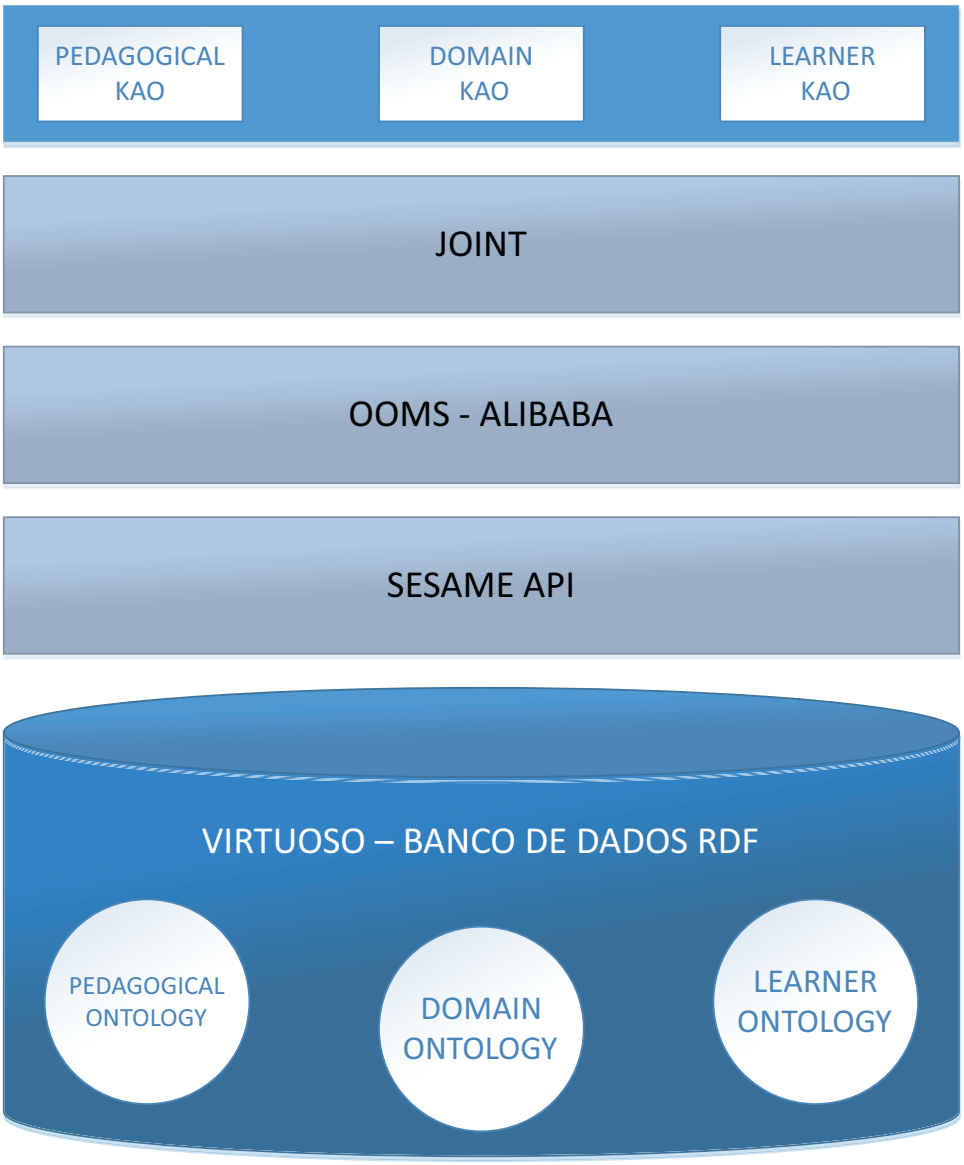
A camada de persistência do MeuTutor-ENEM é responsável pelo acesso e conexão com o banco de dados RDF. Para isso, é necessário o uso de algumas ferramentas, como pode ser visto na Figura 26. A primeira ferramenta que a plataforma utiliza é o JOINT. Dessa forma, seguindo o padrão KAO recomendado pelo JOINT, para cada ontologia presente no banco de dados RDF é necessário criar uma classe concreta KAO referente à ontologia. No cenário da aplicação MeuTutor-ENEM, as principais ontologias utilizadas são: *Pedagogical*, responsável por conter as estratégias pedagógicas; *Domain*, descreve as características do domínio que será aprendido; e *Learner*, modela toda a informação do estudante e suas interações com o sistema.

O JOINT, por sua vez, utiliza duas ferramentas: o Alibaba e o Sesame. A primeira possui o papel de mapear os objetos Java com as entidades presentes nas ontologias, permitindo a manipulação dessas ontologias por meio do paradigma de orientação a objetos. Já a segunda ferramenta, é responsável por acessar e prover uma conexão com o banco de dados RDF subjacente, sendo também responsável por outros mecanismos, como o controle de transações, por exemplo. A última ferramenta utilizada na camada de persistência do MeuTutor-ENEM é o Virtuoso (vide Seção 2.4.2), onde são armazenadas as ontologias mencionadas anteriormente.

O principal problema do uso dessas ferramentas no MeuTutor-ENEM é que a plataforma foi projetada em multicamadas e orientada a objetos. Portanto, a aplicação necessita que os objetos, oriundos da camada de persistência, transitem pelas outras camadas. Contudo, para atender tal requisito, usando um OOMS como o Alibaba que apenas provê objetos *persistentes*, seria necessário que as outras camadas da aplicação (e.g. Serviços Web) acessassem a camada de persistência com o intuito de controlar a conexão com o banco de dados RDF e manter os objetos *persistentes* "acessíveis". Todavia, isso contraria o padrão arquitetural de camadas, o qual afirma que uma camada só pode acessar a camada imediatamente inferior a ela.

Para não violar a arquitetura da aplicação, os desenvolvedores do MeuTutor-ENEM implementaram uma variação do padrão Open Session in View (WADIA et al., 2008). Em resumo, a camada de persistência era responsável por abrir uma conexão e retornar para a camada de negócio o objeto vinculado a essa conexão. O sistema, então, marca a conexão com um limite de tempo de vida, ou seja, o objeto deve ser usado antes que o tempo de vida da conexão chegue ao fim. Dessa forma, quando esse limite de tempo é

Figura 26 – Visão interna da camada de persistência do MeuTutor-ENEM.



Fonte: Elaborado pelo autor.

atingido, o sistema encerra a conexão e todos os objetos vinculados a ela não podem mais ser manipulados.

Embora esse mecanismo desenvolvido permita a transparência com relação às conexões entre as camadas da aplicação, ele implica em outro sério problema. Se o limite de tempo de vida de uma conexão for curto, as funcionalidades que podem demorar mais que esse limite, ao necessitar dos objetos serão prejudicadas, vez que a conexão já estará encerrada. Por outro lado, se o limite for longo, muitas conexões ficarão abertas mesmo sem serem utilizadas, acarretando um desnecessário uso de recursos.

Além desses problemas, a aplicação MeuTutor-ENEM faz uso de vários conversores de objetos *persistentes* em objetos de transferência de dados (DTOs), a fim de que estes possam ser enviados, por meio de serviços web, para a camada de interface do usuário.

5.3 Aplicação do JOINT-DE

Tendo em vista os problemas encontrados, não somente no desenvolvimento do MeuTutor-ENEM mas em outras aplicações que possuem requisitos arquiteturais semelhantes, surgiu a necessidade de evoluir o JOINT para suportar objetos *desconectados*.

O processo de substituição do JOINT tradicional para o JOINT-DE na plataforma MeuTutor-ENEM foi bem simples. Como o gerador de código Java do JOINT-DE é uma extensão modificada do gerador da ferramenta Alibaba (preservando as interfaces Java), o impacto da mudança para o JOINT-DE como novo OOMS se restringiu à camada de persistência.

Com o JOINT-DE, as conexões ficam abertas apenas o tempo necessário para se carregar o objeto na memória, de modo que, posteriormente, ele pode ser usado pelas camadas superiores a qualquer momento. Com isto, o problema de manter várias conexões abertas sem utilização é resolvido. Vale ressaltar que, com objetos *desconectados*, as camadas superiores não precisam acessar nem mesmo conhecer o mecanismo de persistência que está sendo utilizado pela aplicação, mantendo-se a transparência entre os vários níveis da arquitetura.

Ademais, os objetos retornados pelo JOINT-DE podem ser serializados, ou seja, podem atuar como objetos de transferência de dados. Dessa forma, os vários conversores de DTOs presentes no código do MeuTutor-ENEM podem ser excluídos[3], removendo essa etapa de conversão em quase todos os serviços web do sistema. Sumarizando, a implantação do JOINT-DE no projeto do MeuTutor-ENEM implicou nos seguintes benefícios:

- Permitir que os objetos *desconectados* transitassem entre as camadas da aplicação;
- Manter a transparência entre o mecanismo de persistência e as camadas da aplicação;
- Reduzir o número de conexões abertas com o banco de dados RDF;
- Permitir que os objetos oriundos do banco de dados RDF pudessem ser usados como DTOs, removendo a necessidade de conversores.

5.4 Resultados Comparativos

Após a implantação do JOINT-DE no MeuTutor-ENEM, alguns testes foram realizados a fim de comparar quantitativamente os resultados de sua implantação com a infraestrutura antiga do MeuTutor (entenda infraestrutura a camada de persistência da aplicação, contudo em ambos os casos o Virtuoso é utilizado como ferramenta de armazenamento de triplas). Três métodos Java do MeuTutor-ENEM foram analisados (cada

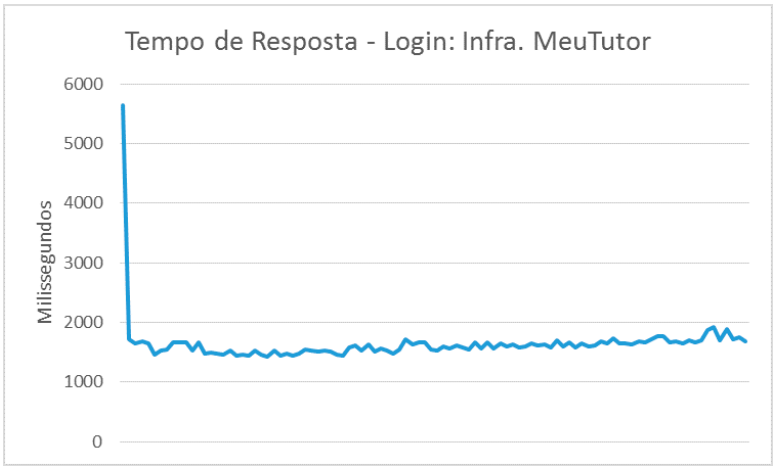
[3] Os DTOs do MeuTutor-ENEM não foram excluídos, pois eles não são representações fieis dos objetos oriundos do banco de dados RDF. Dessa forma, se estes objetos se tornassem DTOs seria necessária uma mudança no código PHP da camada de interface do usuário que requisita os DTOs.

um deles representa um serviço Web): i) Validar login, método que envolve várias atualizações de objetos acerca da estatística de acesso do usuário; ii) Cadastrar usuário, este método cria todas as informações necessárias para que o usuário comece a usar o sistema, portanto, várias instâncias são criadas neste método e; iii) Recuperar troféus do usuário, método que, basicamente, recupera instâncias de troféus que o usuário obteve no sistema.

Em cada teste executado, 100 requisições foram realizadas para cada método com o JOINT-DE e 100 com a infraestrutura antiga do MeuTutor. Todas as execuções foram feitas usando uma máquina com a seguinte configuração: processador Intel(R) Core(TM) i5-2450M 2,50 GHz; Memória RAM de 6,00 GB; Disco Rígido de 640 GB e 5400 RPM e Sistema Operacional Windows 7 Home Premium 64 bits. Os dados foram coletados usando o Netbeans Profiler embutido na IDE versão 7.3.1.

O primeiro teste executado envolveu o método de validar login do usuário usando a infraestrutura do MeuTutor. O gráfico de tempo de resposta durante a execução deste teste é apresentado na Figura 27. Pode-se observar dois pontos neste gráfico, o primeiro é com relação ao início do gráfico, onde as primeiras chamadas dos métodos do MeuTutor-ENEM geralmente possuem um tempo de resposta maior. Isso se deve ao fato, de que no início, o código do MeuTutor-ENEM realiza configurações de componentes para poder funcionar adequadamente, por esta razão que o tempo de resposta é maior no início do gráfico. Contudo, vale ressaltar, que essas configurações necessitam de dados das ontologias, portanto, dependem do mecanismo de persistência interno da aplicação. O segundo ponto que pode ser observado neste gráfico é que após este início, o tempo de resposta estabiliza de uma forma que permanece no intervalo de 1500 a 2000 milissegundos (ms).

Figura 27 – Tempo de resposta no método de validar login com a infra. do MeuTutor.

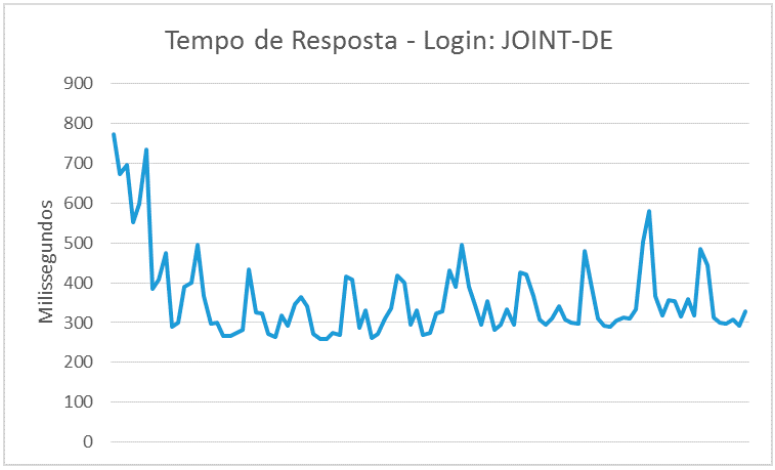


Fonte: Elaborado pelo autor.

Comparando este gráfico com o gráfico de tempo de resposta do validar login usando o JOINT-DE, apresentado na Figura 28, pode-se observar que o tempo de resposta reduziu significativamente. Apesar de também possuir um início que requer mais tempo (embora

ainda assim seja menor), o tempo de resposta permaneceu em um intervalo de 250 a 600 milissegundos.

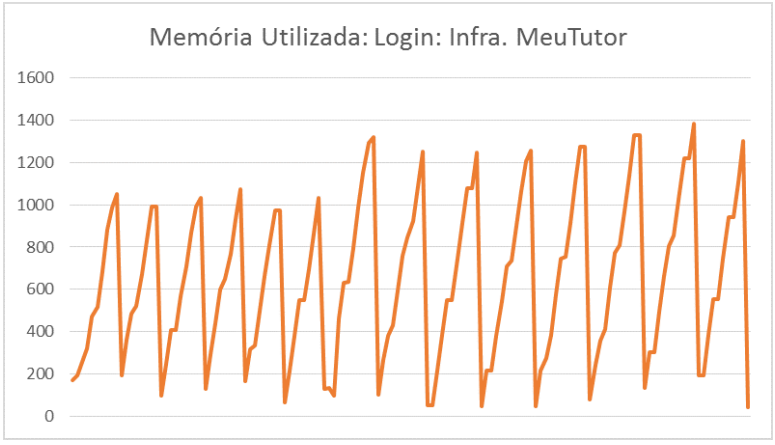
Figura 28 – Tempo de resposta no método de validar login com o JOINT-DE.



Fonte: Elaborado pelo autor.

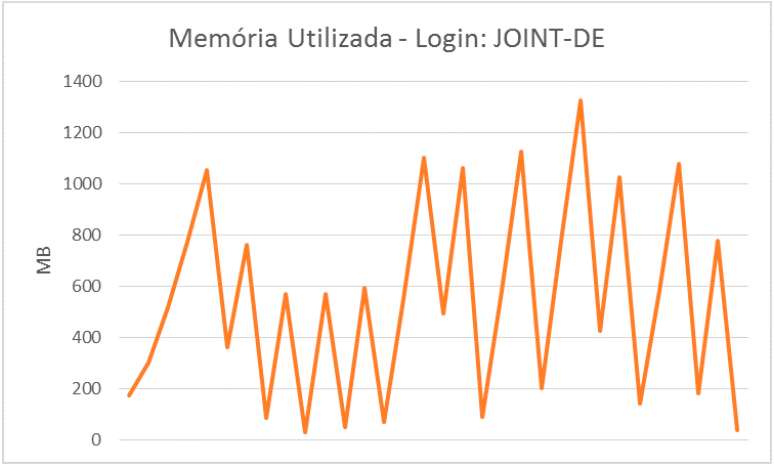
Além do tempo de resposta de cada teste, outro dado coletado foi a memória utilizada. O gráfico de memória utilizada durante a execução do teste com o método de validar login com a infraestrutura do MeuTutor é apresentado na Figura 29. É evidente a oscilação de memória que existe durante a execução deste teste, mas pode ser observado neste gráfico que o teste chegou perto de usar 1400 MegaBytes (MB) de memória. Estes dados não são muito diferentes dos dados de memória obtidos no teste do validar login usando o JOINT-DE, apresentado na Figura 30. Embora o gráfico do JOINT-DE aponte, em parte da execução, um consumo menor de memória, ele também chega perto dos mesmos 1400 MegaBytes.

Figura 29 – Memória utilizada no método de validar login com a infra. do MeuTutor.



Fonte: Elaborado pelo autor.

Figura 30 – Memória utilizada no método de validar login com o JOINT-DE.



Fonte: Elaborado pelo autor.

Os dados mais precisos com relação aos dois testes executados usando o método de validar login são apresentados na Tabela 3. Levando estes dados em consideração, pode-se concluir que neste método, houve uma melhora substancial no desempenho ao utilizar o JOINT-DE em comparação com a infraestrutura antiga do MeuTutor. Analisando as médias de tempo de resposta com cada persistência (368 milissegundos com o JOINT-DE e 1642 milissegundos com a infraestrutura do MeuTutor), o uso do JOINT-DE acarretou em uma melhora de aproximadamente 75% no desempenho do validar login. Por outro lado, a utilização de memória não obteve mudanças significativas ao mudar o mecanismo de persistência para o JOINT-DE. Isto mostra, que mesmo o JOINT-DE possuindo um consumo maior de memória ao carregar as propriedades dos objetos em memória para torna-los objetos *desconectados*, este consumo foi compensado através da redução de conexões simultâneas com o banco de dados RDF. O pico de conexões simultâneas foi de apenas 1 para o JOINT-DE, pois não houve requisições paralelas que necessitassem de mais de uma conexão simultânea. Em compensação, a infraestrutura do MeuTutor, ao utilizar um mecanismo de tempo de vida útil para conexões abertas, obteve um pico de 148 conexões simultâneas.

Tabela 3 – Dados comparativos dos testes executados no método de validar login.

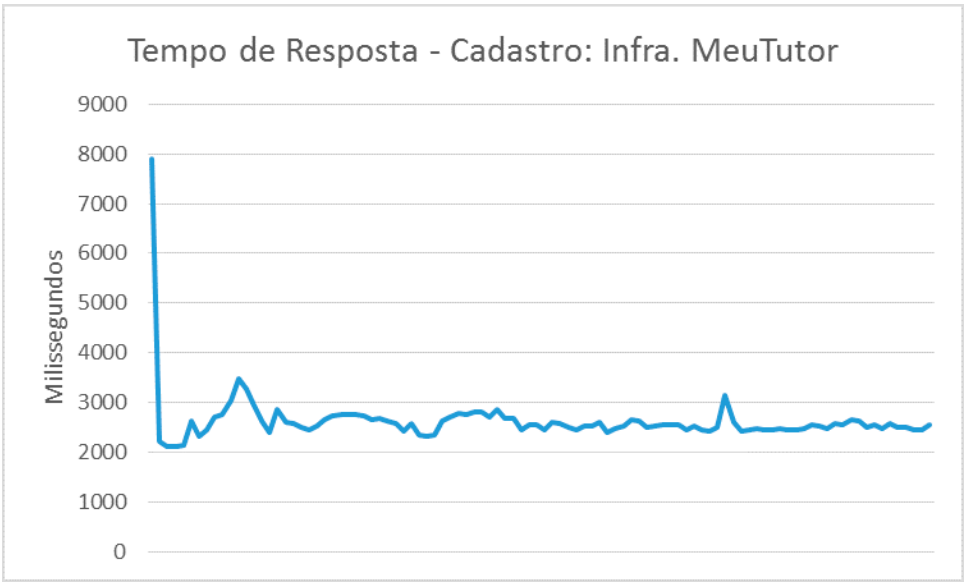
Ferramenta	**Máx. de Tempo de Resposta**	**Média de Tempo de Resposta**	**Máx. de Memória**	**Média de Memória**	**Máx. de Conexões Simultâneas**
JOINT-DE	773 ms	368 ms	1324 MB	544 MB	1
Infra. MeuTutor	5645 ms	1642 ms	1383 MB	647 MB	148

Fonte: Elaborado pelo autor.

O próximo teste executado está relacionado ao método de cadastrar usuário na pla-

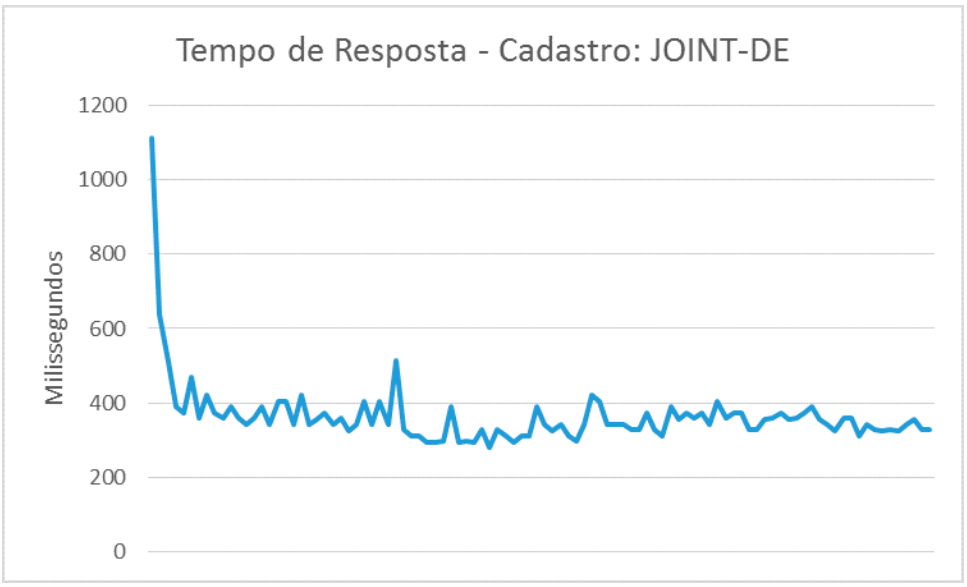
taforma. Os gráficos com os tempos de resposta usando a infraestrutura do MeuTutor e usando o JOINT-DE são apresentados, respectivamente, nas Figuras 31 e 32. Tal como ocorreu no método de validar login, o uso do JOINT-DE reduziu o tempo de resposta significativamente no método de cadastrar usuário. Como pode ser analisado nos gráficos, o tempo de resposta usando a infraestrutura do MeuTutor variou principalmente no intervalo de 2000 a 4000 milissegundos, enquanto que o gráfico de tempo de resposta usando o JOINT-DE permaneceu na sua maior parte em um intervalo de 200 a 600 milissegundos.

Figura 31 – Tempo de resposta no método de cadastrar usuário com a infra. do MeuTutor.



Fonte: Elaborado pelo autor.

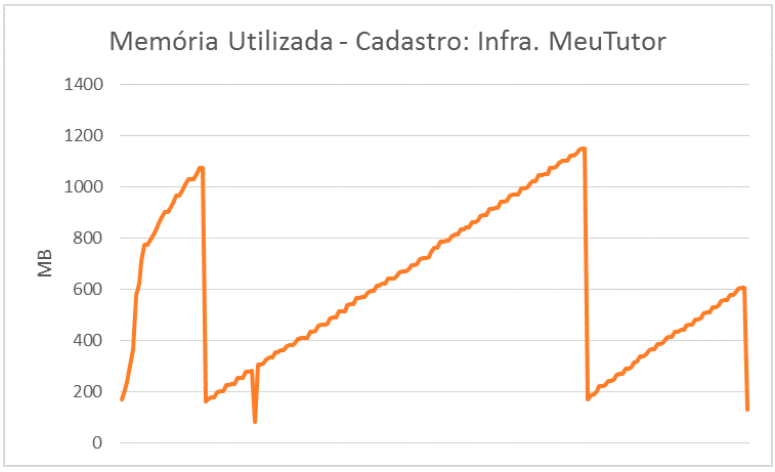
Figura 32 – Tempo de resposta no método de cadastrar usuário com o JOINT-DE.



Fonte: Elaborado pelo autor.

No que diz respeito a utilização de memória, o gráfico do teste executado com o método de cadastrar usuário usando a infraestrutura do MeuTutor é apresentado na Figura 33. Assim como no teste de validar login, existe uma oscilação de memória utilizada pelo teste de cadastrar usuário, todavia, o pico de memória atingido pelo teste não passou os 1200 MegaBytes.

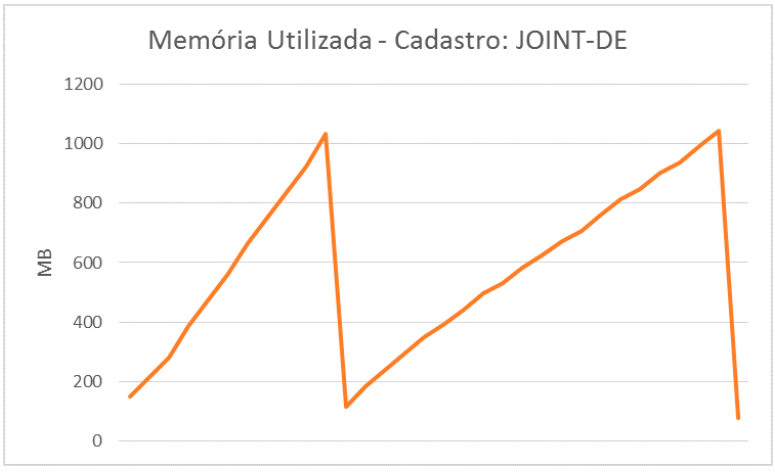
Figura 33 – Memória utilizada no método de cadastrar usuário com a infra. do MeuTutor.



Fonte: Elaborado pelo autor.

Embora o gráfico de memória utilizada envolvendo o método de cadastrar usuário com o JOINT-DE, apresentado na Figura 34, tenha uma semelhança com o gráfico relativo a infraestrutura do MeuTutor, ele possui uma oscilação de memória a menos. Isso está diretamente relacionado ao seu respectivo gráfico de tempo de resposta, pois como a execução do teste foi mais rápida, não foi necessário um terceiro pico de memória. Além disso, o pico de memória atingido por este teste foi um pouco menor que o anterior, não passando os 1100 MegaBytes.

Figura 34 – Memória utilizada no método de cadastrar usuário com o JOINT-DE.



Fonte: Elaborado pelo autor.

Os dados mais precisos com relação aos dois testes executados usando o método de cadastrar usuário são apresentados na Tabela 4. Tal como no método de validar login, houve uma melhora substancial no desempenho ao utilizar o JOINT-DE em comparação com a infraestrutura antiga do MeuTutor no método de cadastrar usuário. Analisando as médias de tempo de resposta (364 milissegundos com o JOINT-DE e 2627 milissegundos com a infraestrutura do MeuTutor), o uso do JOINT-DE acarretou em uma melhora de aproximadamente 86% no desempenho ao cadastrar usuários.

A utilização de memória também não apresentou resultados diferentes com relação ao método de cadastrar usuário, pelo mesmo motivo relacionado ao número de conexões simultâneas. Enquanto o JOINT-DE só necessitou de no máximo 2 conexões simultâneas, a infraestrutura do MeuTutor necessitou de 823, o que afetou o seu consumo de memória. É bom frisar que, apesar das 100 requisições não serem em paralelas, o método de cadastrar usuário possui alguns processos concorrentes internos (Java *Threads*) necessitando dessas 2 conexões simultâneas por parte do JOINT-DE.

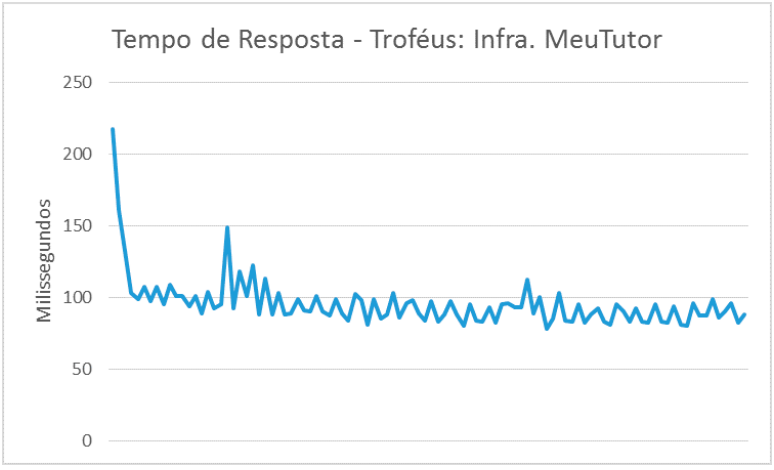
Tabela 4 – Dados comparativos dos testes executados no método de cadastrar usuário.

Ferramenta	**Máx. de Tempo de Resposta**	**Média de Tempo de Resposta**	**Máx. de Memória**	**Média de Memória**	**Máx. de Conexões Simultâneas**
JOINT-DE	1112 ms	364 ms	1040 MB	571 MB	2
Infra. MeuTutor	7910 ms	2627 ms	1151 MB	605 MB	823

Fonte: Elaborado pelo autor.

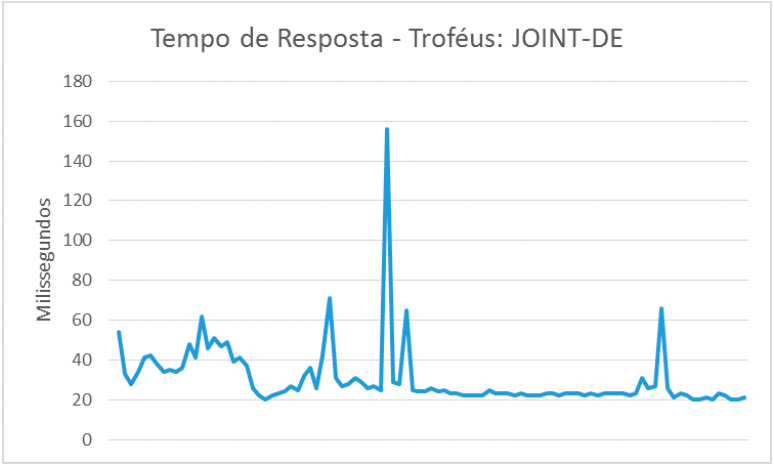
Por fim, o último teste executado envolveu o método de recuperar troféus do usuário. Os gráficos com os tempos de resposta deste método usando a infraestrutura do MeuTutor e usando o JOINT-DE são apresentados, respectivamente, nas Figuras 35 e 36.

Figura 35 – Tempo de resposta no método de recuperar troféus com a infra. do MeuTutor.



Fonte: Elaborado pelo autor.

Figura 36 – Tempo de resposta no método de recuperar troféus com o JOINT-DE.



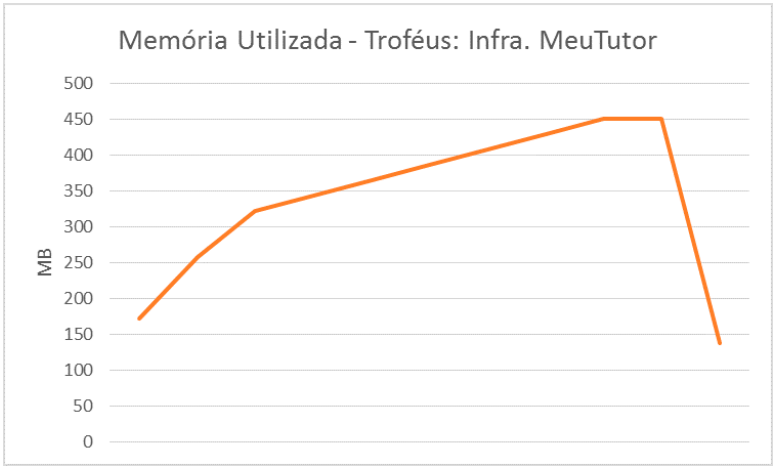
Fonte: Elaborado pelo autor.

Não diferente dos outros dois métodos, o uso do JOINT-DE também reduziu o tempo de resposta no método de recuperar troféus. Pode-se observar nos gráficos que o tempo de resposta usando a infraestrutura do MeuTutor variou principalmente no intervalo de 75 a 150 milissegundos, enquanto que o gráfico de tempo de resposta usando o JOINT-DE permaneceu na sua maior parte em um intervalo de 20 a 160 milissegundos (apenas um pico que ultrapassou 80 milissegundos).

Tendo em vista que o método de recuperar troféus é bem mais rápido independente de qual mecanismo de persistência ele use, os gráficos de memória são bem mais simples se comparados aos outros dois métodos. A Figura 37 mostra o gráfico de utilização de memória no método de recuperar troféus com a infraestrutura do MeuTutor, o pico de consumo de memória foi perto dos 450 MegaBytes. Já o gráfico de memória do método de recuperar troféus usando o JOINT-DE, apresentado na Figura 38, aponta um pico menor perto dos 300 MegaBytes.

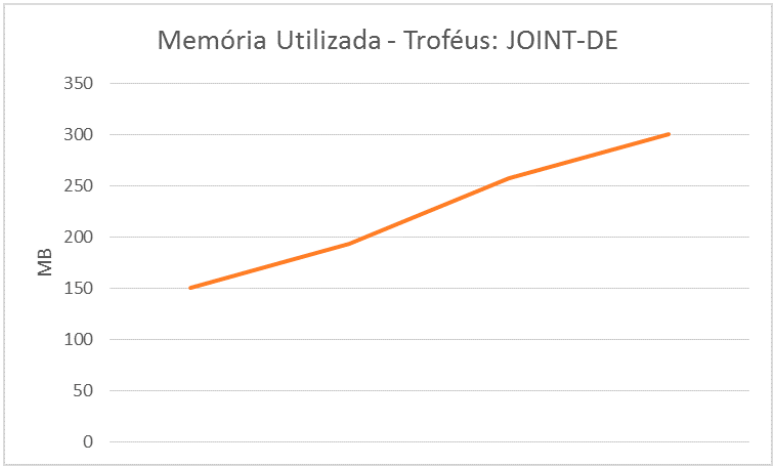
A Tabela 5 apresenta os dados obtidos envolvendo os dois testes com o método de recuperar troféus do usuário. Apesar do método ser mais rápido que os outros e, por isso, aparentar que não houve diferença significativa entre os dois testes com relação ao tempo de resposta, comparando as médias de tempo de resposta (30 milissegundos com o JOINT-DE e 115 milissegundos com a infraestrutura do MeuTutor), o método obteve uma melhora de tempo em aproximadamente 74%. No que diz respeito a utilização de memória, usando o JOINT-DE o sistema consumiu bem menos memória. Analisando as médias de memória (193 MB com o JOINT-DE e 338 com a infraestrutura do MeuTutor), conclui-se que o uso do JOINT-DE, pela razão de abrir menos conexões que a infraestrutura do MeuTutor, reduz em geral o consumo de memória utilizada pelo sistema.

Figura 37 – Memória utilizada no método de recuperar troféus com a infra. do MeuTutor.



Fonte: Elaborado pelo autor.

Figura 38 – Memória utilizada no método de recuperar troféus com o JOINT-DE.



Fonte: Elaborado pelo autor.

Tabela 5 – Dados comparativos dos testes executados no método de recuperar troféus.

Ferramenta	Máx. de Tempo de Resposta	Média de Tempo de Resposta	Máx. de Memória	Média de Memória	Máx. de Conexões Simultâneas
JOINT-DE	156 ms	30 ms	300 MB	193 MB	1
Infra. MeuTutor	217 ms	115 ms	451 MB	338 MB	200

Fonte: Elaborado pelo autor.

6 EXPERIMENTO

Este capítulo descreve o experimento realizado visando avaliar, em termos de desempenho[1] e de utilização de memória, o JOINT-DE. Dessa forma, o experimento compara o OOMS JOINT-DE com a versão tradicional do JOINT que utiliza o OOMS Alibaba (durante este capítulo, ao ler Alibaba, leia-se JOINT utilizando o OOMS Alibaba). O capítulo está dividido em três seções: planejamento, execução e análise dos dados. O experimento foi conduzido seguindo alguns princípios da engenharia de software experimental (WOHLIN et al., 2012).

6.1 Planejamento do Experimento

Nesta seção, será detalhado o planejamento do experimento que foi projetado para este trabalho. Dentro do planejamento, encontra-se a definição da questão de pesquisa e derivação de hipóteses, a seleção das variáveis dependentes e independentes, a identificação da unidade experimental e a seleção do modelo experimental que será utilizado.

O experimento adotado é do tipo comparativo, no qual será comparada a ferramenta proposta JOINT-DE com a ferramenta Alibaba. O contexto do experimento é acadêmico, posto que executado em um laboratório, executando algoritmos com um conjunto de dados fictícios. Nas próximas subseções, o planejamento do experimento será descrito.

6.1.1 Hipóteses de Pesquisa

Como mencionado no início deste capítulo, este experimento visa avaliar o JOINT-DE com relação ao desempenho e à utilização de memória. Levando isto em consideração, duas questões de pesquisa são apresentadas:

Q1 - Existem diferenças de desempenho entre os OOMS JOINT-DE e Alibaba com relação às operações de *create*, *retrieve* e *update*[2]? Se sim, qual ferramenta é a melhor?

A questão de pesquisa acima implica nas seguintes hipóteses de pesquisa:

- **H1-0:** O desempenho para as ferramentas no método de criação de objetos é igual.
- **H1-1:** O desempenho para as ferramentas no método de criação de objetos é diferente.
- **H2-0:** O desempenho para as ferramentas no método de recuperação de objetos é igual.

[1] No contexto deste capítulo, desempenho se refere unicamente ao tempo gasto pelas ferramentas em suas operações.

[2] A operação delete não foi considerada, pois o Alibaba não dispõe desta funcionalidade.

- **H2-1:** O desempenho para as ferramentas no método de recuperação de objetos é diferente.
- **H3-0:** O desempenho para as ferramentas no método de atualização de objetos é igual.
- **H3-1:** O desempenho para as ferramentas no método de atualização de objetos é diferente.

Caso alguma das hipóteses nulas seja refutada (indicando que há diferença), testes estatísticos serão executados para que se possa concluir que ferramenta tem melhor desempenho e em que método.

Todavia, é necessário avaliar, além do desempenho da ferramenta, a utilização de memória que ela demanda. Isto porque não adianta uma ferramenta ter um bom desempenho mas consumir muito da máquina, tornando sua utilização inviável.

Q2 Existem diferenças de utilização de memória entre os OOMS JOINT-DE e Alibaba com relação às operações de *create*, *retrieve* e *update*? Se sim, qual ferramenta utiliza menos memória?

- **H4-0:** A utilização de memória para as ferramentas no método de criação de objetos é igual.
- **H4-1:** A utilização de memória para as ferramentas no método de criação de objetos é diferente.
- **H5-0:** A utilização de memória para as ferramentas no método de recuperação de objetos é igual.
- **H5-1:** A utilização de memória para as ferramentas no método de recuperação de objetos é diferente.
- **H6-0:** A utilização de memória para as ferramentas no método de atualização de objetos é igual.
- **H6-1:** A utilização de memória para as ferramentas no método de atualização de objetos é diferente.

Da mesma forma, caso alguma das hipóteses nulas seja refutada (indicando que há diferença), testes estatísticos serão executados para que se possa concluir que ferramenta utiliza menos memória em um determinado método.

A Tabela 6 define formalmente as hipóteses de pesquisa supracitadas. T e M são as funções que retornam, respectivamente, o valor do tempo de execução e a média de memória utilizada, com relação tanto aos sistemas de mapeamento objeto-ontologia F1 (JOINT-DE) e F2 (Alibaba) quanto às operações de criação (O1), recuperação (O2) e atualização (O3) de objetos.

Tabela 6 – Definição formal das hipóteses de pesquisa.

Hipótese	Hipótese Nula	Hipótese Alternativa
H1	$H1-0: T(F1,O1) = T(F2,O1)$	$H1-1: T(F1,O1) \neq T(F2,O1)$
H2	$H2-0: T(F1,O2) = T(F2,O2)$	$H2-1: T(F1,O2) \neq T(F2,O2)$
H3	$H3-0: T(F1,O3) = T(F2,O3)$	$H3-1: T(F1,O3) \neq T(F2,O3)$
H4	$H4-0: M(F1,O1) = M(F2,O1)$	$H4-1: M(F1,O1) \neq M(F2,O1)$
H5	$H5-0: M(F1,O2) = M(F2,O2)$	$H5-1: M(F1,O2) \neq M(F2,O2)$
H6	$H6-0: M(F1,O3) = M(F2,O3)$	$H6-1: M(F1,O3) \neq M(F2,O3)$

Fonte: Elaborado pelo autor.

6.1.2 Seleção de Variáveis

Seguindo com o planejamento, é necessário definir as variáveis contidas no experimento. Primeiro, as variáveis independentes, também chamadas de fatores, serão apresentadas. Logo em seguida, as variáveis dependentes, no caso as métricas, serão detalhadas.

As variáveis independentes são as seguintes:

- OOMS: Esta variável especifica qual o sistema de mapeamento objeto-ontologia que será avaliado;
- Operações em Objetos: Esta variável contém as operações que serão executadas por cada OOMS.

Os níveis dos fatores apresentados acima são definidos na Tabela 7. As variáveis dependentes (métricas) são definidas abaixo:

- Tempo de execução por iteração (T): Esta variável representa o tempo de execução de uma unidade de iteração de uma operação específica do OOMS, seja create, retrieve ou update;
- Média da Memória Utilizada (M): Esta variável captura a média de memória utilizada pela execução do experimento para cada operação em cada OOMS no intervalo de medição.

Tabela 7 – Níveis dos fatores.

Fator	Nível	Descrição
Operação	O1	Método de criação de objetos
	O2	Método de recuperação de objetos
	O3	Método de atualização de objetos
OOMS	F1	Ferramenta JOINT-DE
	F2	Ferramenta Alibaba

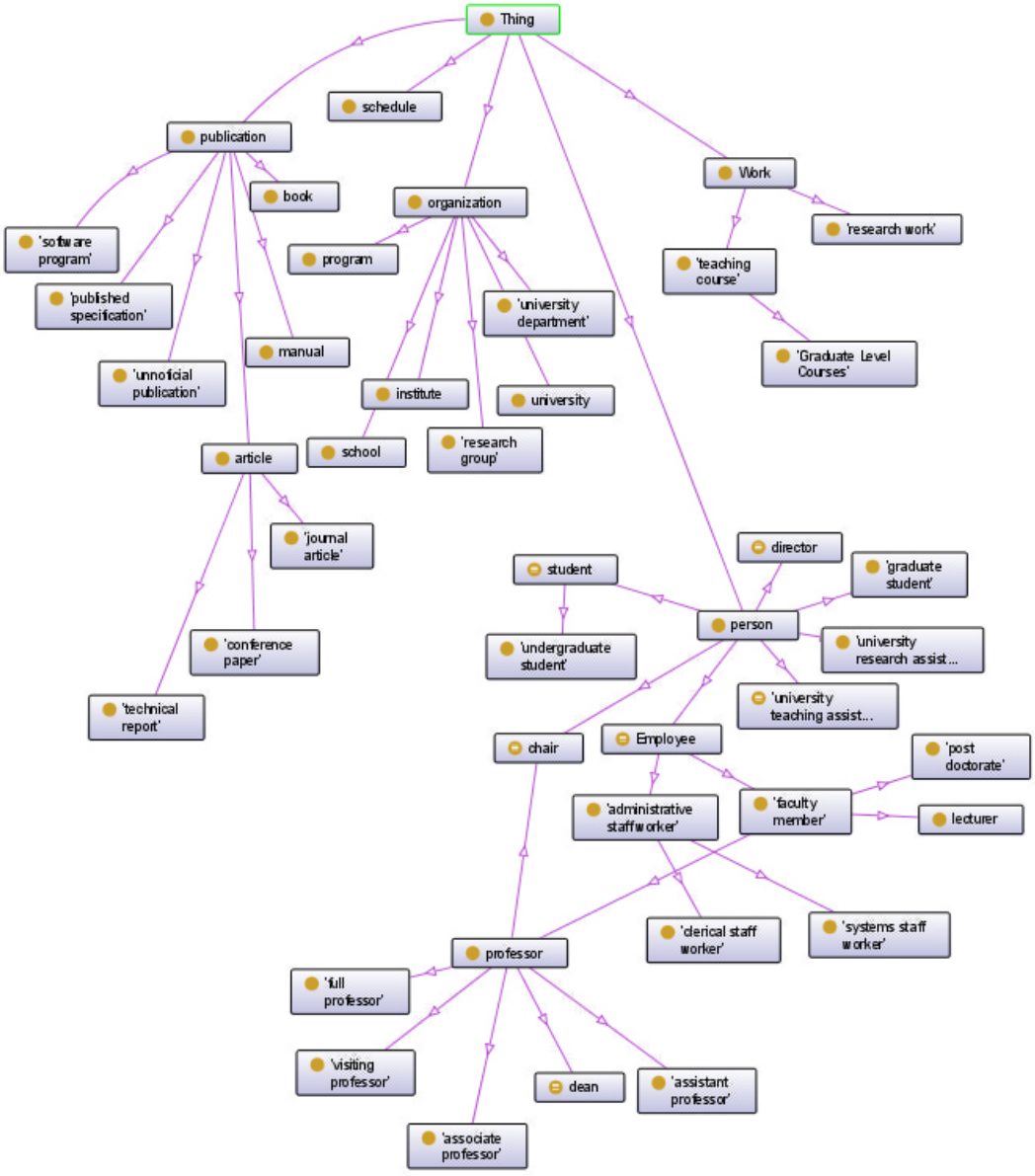
Fonte: Elaborado pelo autor.

6.1.3 Unidades de Experimento

A unidade experimental deste estudo empírico são instâncias da entidade *Department* do conjunto de dados *Lehigh University Benchmark* (LUBM). O LUBM é amplamente utilizado pela comunidade da área de ontologias para avaliações de tecnologias semânticas. Com o LUBM, é possível criar um grande conjunto de dados que representa uma ontologia real. Além de outros componentes, o LUBM possuí:

- Ontologia: A ontologia do LUBM é chamada de Univ-Bench. O domínio desta ontologia é de universidades acadêmicas. A Figura 39 mostra a taxonomia desta ontologia;

Figura 39 – Taxonomia da ontologia Univ-Bench.



Fonte: Elaborado pelo autor.

- Gerador de dados: Esta ferramenta gera instâncias em OWL para a ontologia Univ-Bench. Estes dados podem ser reproduzíveis e customizáveis, permitindo que o gerador seja configurado em três aspectos: a semente para geração de dados aleatórios; o número de universidades que serão geradas (não somente as instâncias, mas todo o subgrafo da universidade) e o índice inicial das universidades.

Para este experimento, o gerador do LUBM foi configurado com a semente aleatória padrão 1, com 1000 universidades e índice 0. Com esta configuração, a ferramenta gerou aproximadamente 20000 arquivos, num tamanho total de 19 GB de disco e mais de 137 milhões de triplas RDF. Por fim, esses arquivos foram armazenados no Virtuoso para que fossem acessados na execução do experimento pelos OOMS.

Todo o experimento foi executado considerando apenas instâncias da entidade *Department*. A escolha do modelo *Department* se deve ao fato de que suas instâncias possuem dados suficientes para este experimento. Ao gerar 1000 universidades, é gerado em torno de 20000 (vinte mil) instâncias de *Department*.

6.1.4 Seleção do Projeto Experimental

Levando em consideração as diversas classificações de experimento (JURISTO; MORENO, 2001), o presente experimento é classificado como fatorial completo com repetições. O experimento é repetido 10 vezes (este número foi escolhido por conta de restrições de tempo para execução e análise dos dados). Dessa forma, como o experimento possui dois fatores: OOMS, com 2 níveis (JOINT-DE e Alibaba) e; Operação, com 3 níveis (*create*, *retrieve* e *update*), multiplicando esses níveis resulta em um total de 6 tratamentos, sendo executado 10 vezes, totalizando 60 execuções. A Tabela 8 descreve cada um dos tratamentos.

Tabela 8 – Definição dos tratamentos.

Número do Tratamento	**Operação**	**OOMS**
1	O1	F1
2	O2	F1
3	O3	F1
4	O1	F2
5	O2	F2
6	O3	F2

Fonte: Elaborado pelo autor.

6.2 Execução do Experimento

Esta seção descreve a execução do experimento que foi planejado na seção anterior, ou seja, descreve a execução do micro *benchmark* projetado para os dois OOMS. *Benchmarks*

são programas desenvolvidos especificamente para avaliar um aplicativo computacional. No contexto do software Java, *benchmarks* são programas Java com o objetivo de medir o desempenho ou outra característica de um ou mais elementos de um sistema que são executados em Java. Esses elementos podem incluir todo o conjunto hardware e software ou ser limitado a determinadas funcionalidades do programa Java. Este último é chamado de micro *benchmark*, pois ele tem um foco mais especifico (HUNT; JOHN, 2011).

Nas próximas subseções, serão apresentados o micro *benchmark* realizado no contexto deste experimento, quais foram os instrumentos utilizados, bem como a narrativa de execução do experimento.

6.2.1 Java Micro *Benchmark*

Os micro *benchmarks* são criados usualmente por desenvolvedores para analisar uma determinada funcionalidade do sistema. Contudo, um micro *benchmark* em Java pode gerar conclusões erradas devido às várias peculiaridades da Máquina Virtual Java (JVM), pois ela pode fazer otimizações em alguns trechos do código e em outros não. Nesta perspectiva, alguns pontos devem ser considerados ao desenvolver esses micro *benchmarks*.

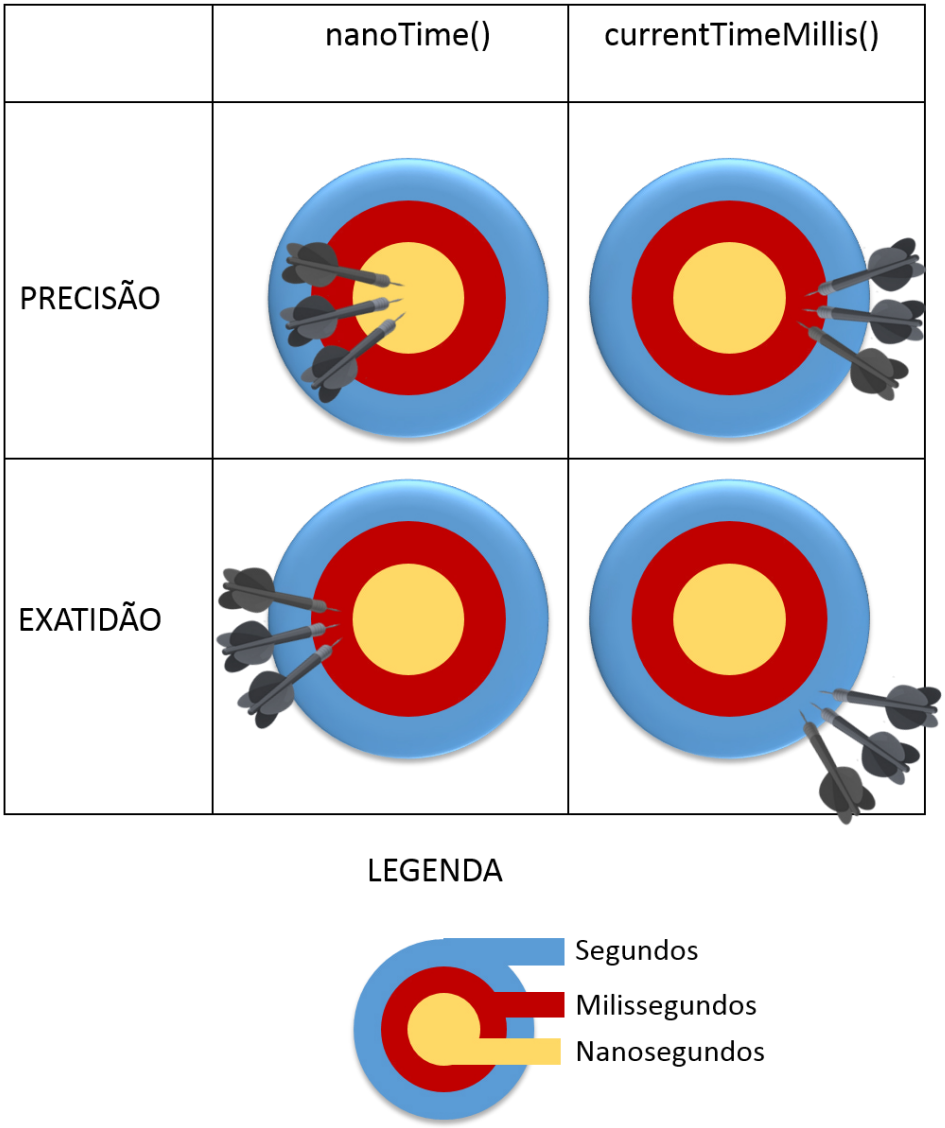
O primeiro ponto a ser considerado quando se desenvolve um micro *benchmark* Java é o período de aquecimento da JVM. Um período de aquecimento fornece à JVM a oportunidade de coletar informação sobre o programa em execução e, assim, produzir otimizações dinâmicas sobre o programa. Por padrão, se o programa está sendo executado em uma JVM edição servidor, é recomendado rodar o mesmo bloco de código 10.000 vezes para que a JVM transforme esse código em nativo de máquina. Se for JVM edição cliente, basta 1.500 iterações para que ele comece a produzir código nativo de máquina (HUNT; JOHN, 2011). Para o caso deste experimento, o micro *benchmark* foi executado em uma JVM edição servidor e, por essa razão, todas as execuções possuem um intervalo de aquecimento de 10.000 iterações e um intervalo de medida de 2.000 iterações contendo o mesmo código.

Outro ponto que deve ser considerado ao desenvolver micro *benchmarks* Java é o Java *garbage collector*. O Java *garbage collector* é um mecanismo da JVM que libera objetos que estão alocados na memória pelo fato de não estarem mais sendo utilizados pelo programa. Os efeitos que o *garbage collector* pode produzir em um código são difíceis de determinar (SHIRAZI, 2003). Contudo, uma prática comum ao fazer *benchmarks* é chamar o *garbage collector* algumas vezes antes do código de execução do experimento, para que seja minimizado o efeito dele durante o restante do código. Ele é chamado mais de uma vez, pois alguns objetos podem necessitar de múltiplos *garbage collectors* para serem liberados da memória.

O último ponto considerado nesse experimento foi a API de Tempo Java. O comum em vários micro *benchmarks* Java é o uso do **System.currentTimeMillis()** para saber o tempo atual em milissegundos. Dessa forma, o valor do tempo é medido no início da exe-

cução e no final, sendo, posteriormente, subtraídos os valores a fim de saber o tempo gasto durante a execução. Contudo, em versões mais recentes do Java, foi introduzido o método **System.nanoTime()**, que retorna o tempo com precisão de nanosegundos. Apesar deste último método retornar em nanosegundos, ele não possui uma exatidão em nanosegundos. O mesmo ocorre com o **System.currentTimeMillis()**; neste, o valor retornado é em milissegundos, mas a exatidão não. Portanto, se o tempo gasto na execução do experimento está na casa dos milissegundos é recomendado que se utilize o **System.nanoTime()**, já que este possui uma exatidão maior (KUPERBERG; KROGMANN; REUSSNER, 2009). Essas diferenças de precisão e exatidão entre as APIs do Java são ilustradas na Figura 40.

Figura 40 – Diferenças entre as APIs de Tempo em Java.



Fonte: Elaborado pelo autor.

Considerando os pontos supracitados, foi desenvolvido um micro *benchmark* para cada tratamento descrito na Tabela 8. A Figura 41 apresenta o código do micro *benchmark* para o tratamento 1, no qual se está avaliando o JOINT-DE (F1) na operação *create* (O1).

O tratamento 4, envolvendo o Alibaba (F2) na operação *create* (O1), foi desenvolvido de maneira bastante similar.

Figura 41 – Micro *benchmark* da operação *create* com o JOINT-DE.

```
public static void main(String[] args) throws IOException {

    //chama o garbage collector para liberar possiveis objetos na memoria
    System.gc();
    System.gc();
    System.gc();

    int warmUpCycles = 10000;
    int testCycles = 2000;
    CRUDBenchmarkDetached bck = new CRUDBenchmarkDetached();
    System.err.println("Aquecendo a JVM ...");
    bck.runCreateBenchMarkJOINTDE(warmUpCycles, 0);
    System.err.println("Finalizado o aquecimento da JVM.");
    System.err.println("Entrando no intervalo de medição ... " +
            System.currentTimeMillis());
    long nanosIteracao = bck.runCreateBenchMarkJOINTDE(testCycles, 10000);
    System.err.println("Intervalo de medição concluído.");
    System.err.println("Nano segundos por iteração :"
            + nanosIteracao);
    System.err.println("Finalizado benchmark!");
}

 private long runCreateBenchMarkJOINTDE(int iterations, int skip) {

    //cria o kao passando a classe de departamento
    UniversityKAO kao = new UniversityKAO(Department.class);

    //pega o tempo em nano segundos do inicio do teste
    long startTime = System.nanoTime();

    //executa o create um número de iterações passado como parâmetro desde
    // o skip até iterações mais skip
    for (int i = skip; i < iterations + skip; i++) {
        //recebe o nome da instância
        StringBuilder instanceName = new StringBuilder();
        instanceName.append("http://www.Department");
        instanceName.append(i);
        instanceName.append(".UniversityX.edu");

        //cria a instância
        kao.create(ontologyURI, instanceName.toString());
    }
    //pega o tempo em nano segundos do final do teste
    long elapsedTime = System.nanoTime();
    //retorna o tempo gasto pelo número de iterações
    return (elapsedTime - startTime) / iterations;
}
```

Fonte: Elaborado pelo autor.

No caso da operação *retrieve*, embora o método principal (*main*) permaneça inalterado, a execução do benchmark é diferente. Para os tratamentos 2 e 5, que envolvem a recuperação de instâncias da entidade *Department* com ambas as ferramentas, o micro

benchmark desenvolvido apenas recupera o objeto sem recuperar quaisquer propriedades dele. A Figura 42 apresenta o código do micro *benchmark* para a operação *retrieve* com a ferramenta Alibaba.

Figura 42 – Micro *benchmark* da operação *retrieve* com o Alibaba.

```
private long runRetrieveBenchMarkAlibaba(int iterations, int skip)
        throws IOException {

    //carrega o arquivo com o nome de todas as instâncias
    File f = new File("C:\\DepartmentInstances.txt");
    BufferedReader reader = new BufferedReader(new FileReader(f));

    //pula o número de instâncias passado como parâmetro
    for (int i = 0; i < skip; i++) {
        reader.readLine();
    }

    //cria o kao passando a classe de departamento
    UniversityKAO kao = new UniversityKAO(Department.class);

    //pega o tempo em nano segundos do inicio do teste
    long startTime = System.nanoTime();

    //executa o retrieve um número de iterações passado como parâmetro
    for (int i = 0; i < iterations; i++) {

        //recebe o nome da instância
        String instanceName = reader.readLine();

        //recupera a instância
        Department d = kao.retrieveInstance(ontologyURI, instanceName);

        //finaliza a transação e começa outra.
        kao.save();
    }

    //pega o tempo em nano segundos do final do teste
    long elapsedTime = System.nanoTime();
    //retorna o tempo gasto pelo número de iterações
    return (elapsedTime - startTime) / iterations;
}
```

Fonte: Elaborado pelo autor.

Por fim, foram criados os dois micro *benchmarks* para a operação de atualização de objetos. Para os tratamentos 3 e 6 os micro *benchmarks*, em cada iteração, recuperavam uma instância de *Department*, recuperavam uma determinada instância de *University* e adicionavam essa instância em duas propriedades do objeto da classe *Department* (*affiliatedOrganizationOf* e *subOrganizationOf*). Depois o objeto era atualizado chamando o método *update* (para o caso do JOINT-DE, tratamento 3) ou chamando o método *save*, que salva as mudanças feitas nos proxies dinâmicos (para o caso do Alibaba, tratamento 6). O micro *benchmark* do tratamento 3 pode ser visto na Figura 43.

Figura 43 – Micro *benchmark* da operação *update* com o JOINT-DE.

```
private long runUpdateBenchMarkJOINTDE(int iterations, int skip)
        throws IOException {

     //carrega o arquivo com o nome de todas as instâncias
    File f = new File("C:\\DepartmentInstances.txt");
    BufferedReader reader = new BufferedReader(new FileReader(f));

    //pula o número de instâncias passado como parâmetro
    for (int i = 0; i < skip; i++) {
        reader.readLine();
    }

    long totalTime = 0;

    //cria o kao passando a classe de departamento
    UniversityKAO kao = new UniversityKAO(Department.class);
    //executa o retrieve um número de iterações passado como parâmetro
    for (int i = 0; i < iterations; i++) {

        //recebe o nome da instância
        String instanceName = reader.readLine();

        //recupera a instância e a instância de universidade
        Department d = kao.retrieveInstance(ontologyURI, instanceName);
        Department o = kao.retrieveInstance(ontologyURI,
                "http://www.Department0.UniversityFicticia.edu");
        //pega o tempo em nano segundos do inicio do teste

        long startTime = System.nanoTime();
        Set<Organization> affiliatedOrganizationOf =
                d.getAffiliatedOrganizationOf();
        affiliatedOrganizationOf.add(o);
        d.setAffiliatedOrganizationOf(affiliatedOrganizationOf);
        Set<Organization> subOrganizationOf = d.getSubOrganizationOf();
        subOrganizationOf.add(o);
        d.setSubOrganizationOf(subOrganizationOf);

        kao.update(d);

        //pega o tempo em nano segundos do final do teste
        long elapsedTime = System.nanoTime();
        totalTime += (elapsedTime - startTime);
    }
    //retorna o tempo gasto pelo número de iterações
    return totalTime / iterations;
}
```

Fonte: Elaborado pelo autor.

6.2.2 Preparação e Instrumentação

Inicialmente, é necessário preparar o ambiente em que os micro *benchmarks* serão executados. Para isso, duas etapas são importantes: i) remover todos os programas que estão executando em segundo plano na máquina, para não ocorrer nenhuma influência externa;

ii) configurar todo o ferramental que será utilizado pelo experimento. As ferramentas que são utilizadas neste experimento, bem como suas versões são descritas a seguir:

- NetBeans IDE – 7.3.1;
- NetBeans Profiler – Embutido na IDE;
- Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode);
- Projeto R – 3.0.2;
- Virtuoso RDF Store – 6.1.6;
- OpenRDF Sesame API – 2.6.10;
- OpenRDF Alibaba – 2.0-rc7.

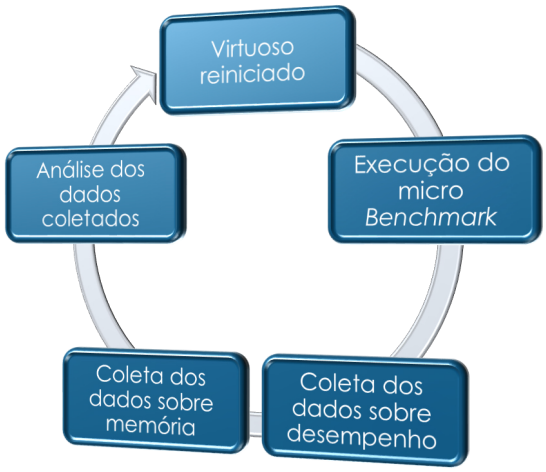
Todo o experimento foi feito em uma máquina com a seguinte configuração: processador Intel(R) Core(TM) i5-2450M 2,50 GHz; Memória RAM de 6,00 GB; Disco Rígido de 640 GB e 5400 RPM e Sistema Operacional Windows 7 Home Premium 64 bits.

6.2.3 Narrativa de Execução

A Figura 44 apresenta os passos de execução de cada micro *benchmark*, que são descritos abaixo:

1. Antes de executar o micro *benchmark*, o banco de dados RDF Virtuoso é reiniciado para que ele volte ao estado inicial de uso de memória;
2. Execução do micro *benchmark* através do Java Netbeans Profiler;
3. Depois da execução, os dados relativos à métrica "tempo gasto por iteração" (T) são coletados e separados em um arquivo texto;
4. Depois, são coletados os dados relativos à memória durante toda a execução do micro *benchmark* e, de posse de todos os dados, a média de memória utilizada é calculada e separada também em um arquivo texto;
5. Os dados, então, são analisados através de métodos estatísticos para responder as hipóteses de pesquisa do experimento.

Figura 44 – Passos de execução de cada micro *benchmark*.



Fonte: Elaborado pelo autor.

6.2.4 Análise de Ameaças à Validade

Embora todo o experimento tenha sido planejado para minimizar possíveis ameaças que comprometam suas conclusões, existem algumas que devem ser mencionadas. A primeira ameaça é com relação ao tamanho da amostra do experimento, o número escolhido de 10 repetições pode influenciar em um baixo poder estatístico para a análise dos dados. Devido à restrição de tempo para finalizar este experimento, não foi possível a execução de mais repetições dos micro *benchmarks* e consequentemente a análise desses dados.

A outra possível ameaça é a aparição de eventos randômicos durante a execução dos micro *benchmarks*, como por exemplo a inicialização de um programa em segundo plano não esperado. Por fim, a terceira ameaça envolve a seleção da unidade experimental, onde é considerado apenas um modelo que possuí características próprias, o que dificulta a generalização dos resultados além do escopo estudado.

Levando isto em consideração, na próxima subseção, os dados obtidos a partir da execução do experimento serão analisados.

6.3 Análise dos Dados

O experimento foi executado seguindo todo o planejamento descrito anteriormente neste capítulo, depois da execução de cada micro *benchmark* os dados eram coletados com relação ao tempo e memória utilizada. Após todas as repetições serem executadas para cada tratamento o arquivo texto com todos os dados resultantes foi analisado por meio da ferramenta R. De fato, dois *scripts* em R foram construídos para análise desses dados.

Ao longo desta seção, uma análise descritiva de todos os dados obtidos no experimento será conduzida, esta análise envolve a apresentação gráfica dos histogramas para cada mé-

trica, operação e ferramenta, diagramas de caixa comparativos (*boxplots*) e a sumarização das estatísticas. Logo após, a verificação das hipóteses de pesquisa será apresentada através da análise dos intervalos de confiança e execução de testes de hipóteses. Por fim algumas conclusões serão delineadas.

Realizando uma breve análise dos dados é possível ter noção das respostas para as questões de pesquisa levantadas no experimento, além de permitir um melhor entendimento do comportamento das variáveis e detectar possíveis padrões e/ou inconsistências nos dados. Como mencionado anteriormente, as variáveis e níveis definidos no planejamento do experimento são:

Variáveis:

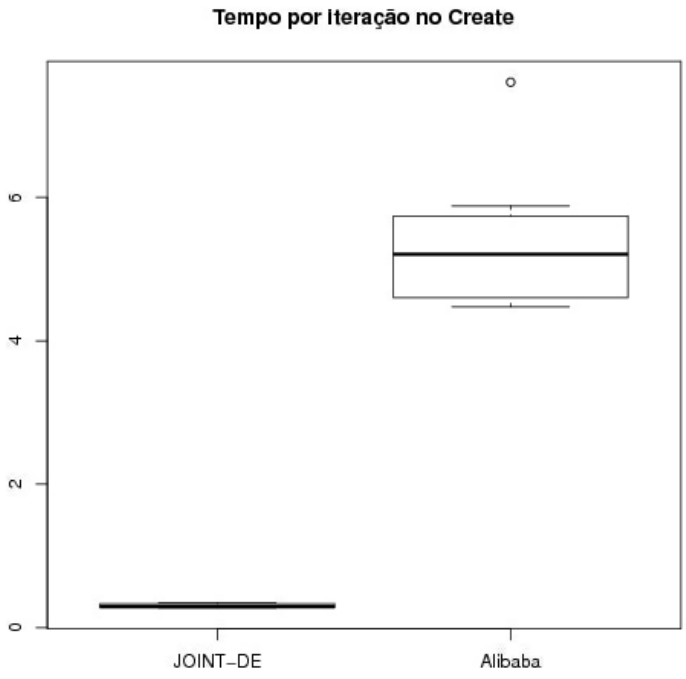
- Tempo por Iteração(T), em milissegundos;
- Média de Memória Utilizada(M), em megabytes.

A seguir, serão analisados os dados de cada uma das variáveis estudadas.

6.3.1 Análise Descritiva do Tempo por Iteração

A Figura 45 apresenta os diagramas de caixa do tempo para operação de criação de instâncias para ambas as ferramentas. Esta figura sugere que o desempenho do JOINT-DE para criação de instâncias é melhor que o desempenho do OOMS Alibaba, no entanto,

Figura 45 – Diagramas de caixa com comparativo de Tempo no *Create*.

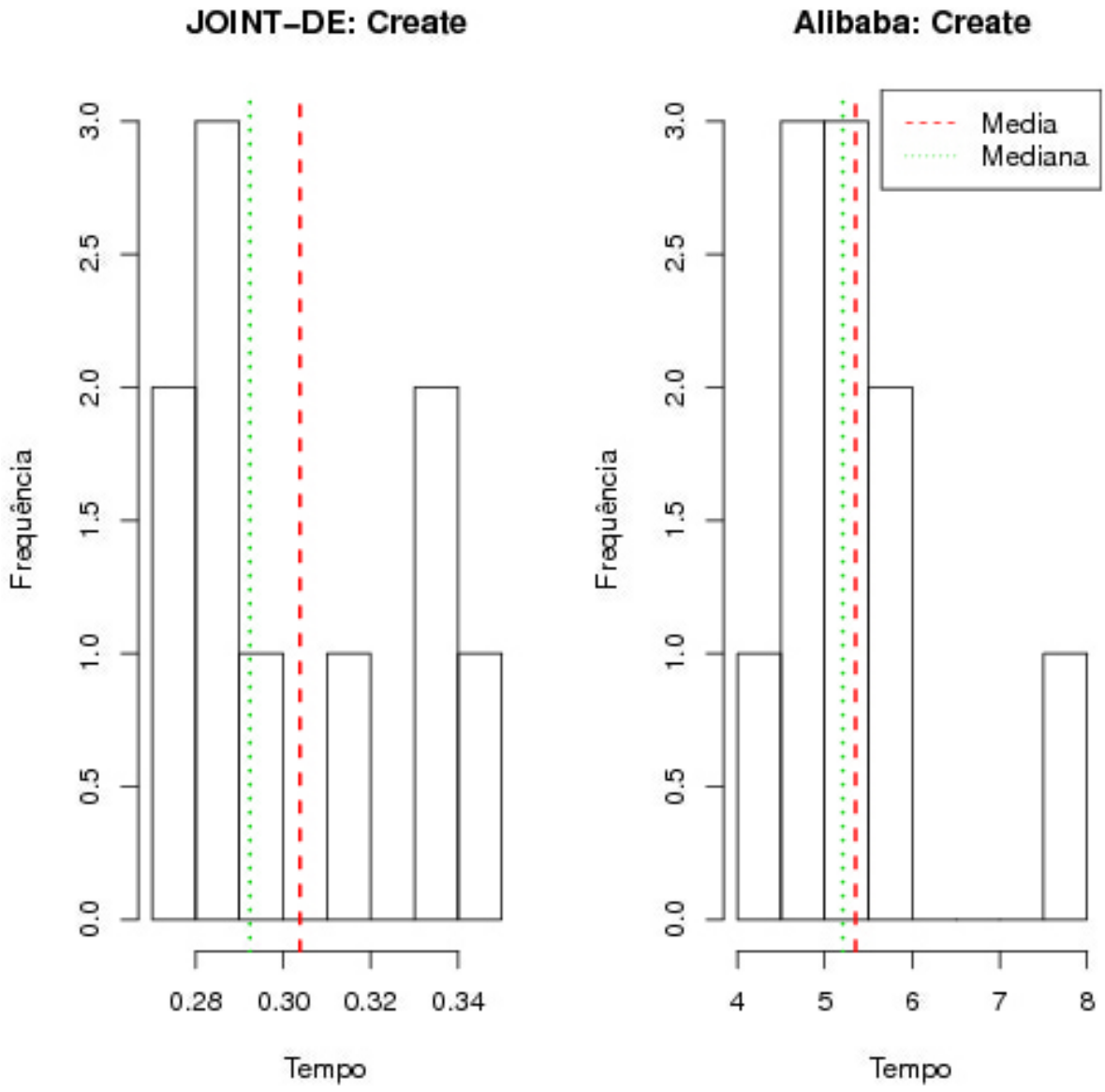


Fonte: Elaborado pelo autor.

ainda não foram geradas evidências estatísticas para afirmar isso, tais afirmações só poderão ser feitas quando os testes estatísticos forem realizados (especificamente na subseção de Verificação das Hipóteses 6.3.3).

A Figura 46 apresenta os histogramas de cada ferramenta também para a operação *create*, estes histogramas mostram que as distribuições dos dados do tempo por iteração em ambas as ferramentas não aparentam ser normais. Contudo, só com métodos estatísticos será possível afirmar se a distribuição desses dados é normal ou não. Vale ressaltar que a verificação de normalidade dos dados é pré-requisito para escolha de qual teste de hipótese estatístico será utilizado para analisar as hipóteses.

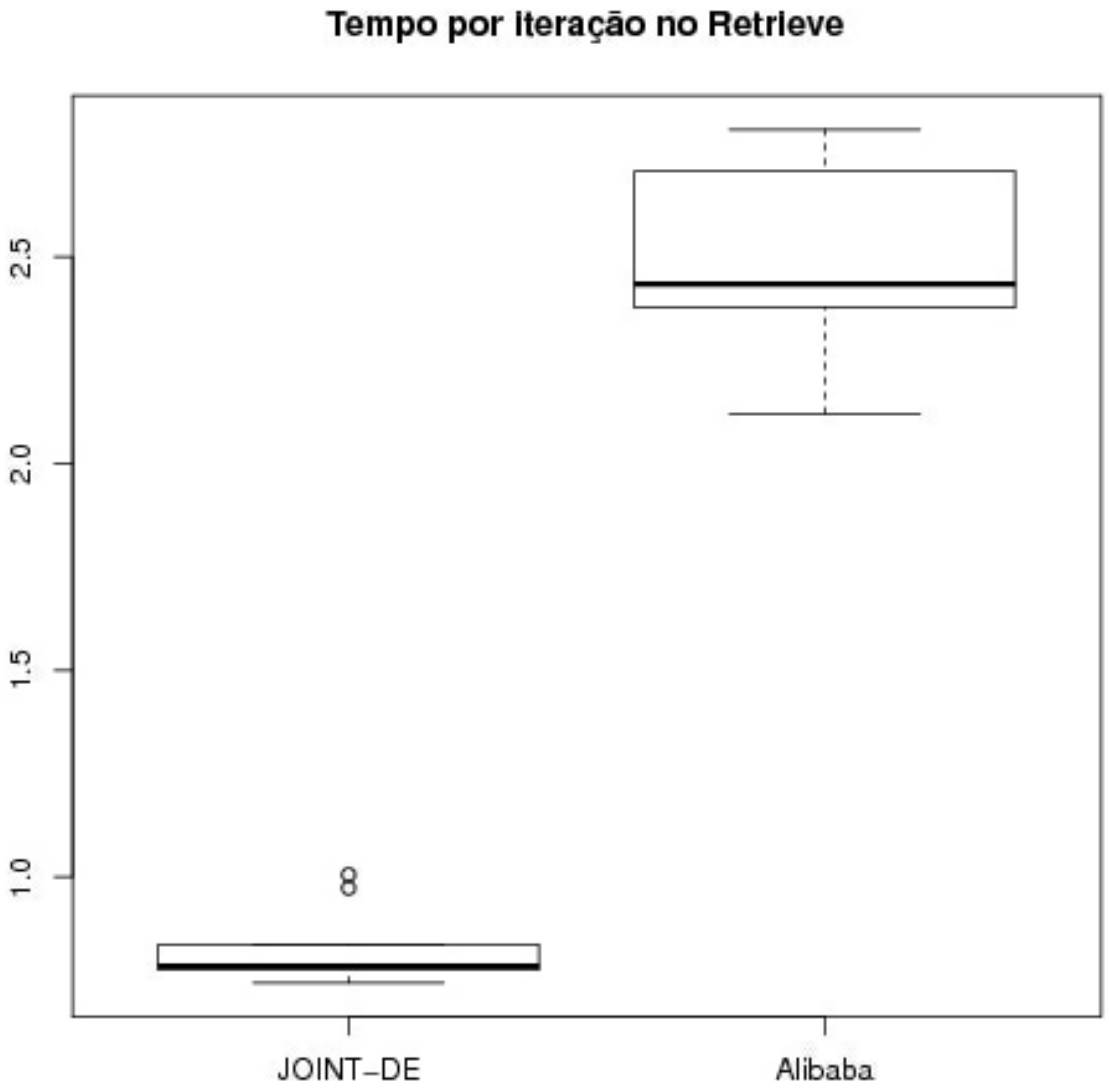
Figura 46 – Histogramas da variável Tempo na operação *Create*.



Fonte: Elaborado pelo autor.

A próxima operação a ser analisada é operação de recuperação de instâncias (*retrieve*). A Figura 47 apresenta os diagramas de caixa do tempo para operação de recuperação de instâncias para ambas as ferramentas. Assim como no *create*, os diagramas de caixa apontam que o desempenho do JOINT-DE para recuperação de objetos é melhor que o desempenho do OOMS Alibaba. Entretanto, também faz-se necessário evidências estatísticas para corroborar tal afirmação.

Figura 47 – Diagramas de caixa com comparativo de Tempo no *Retrieve*.

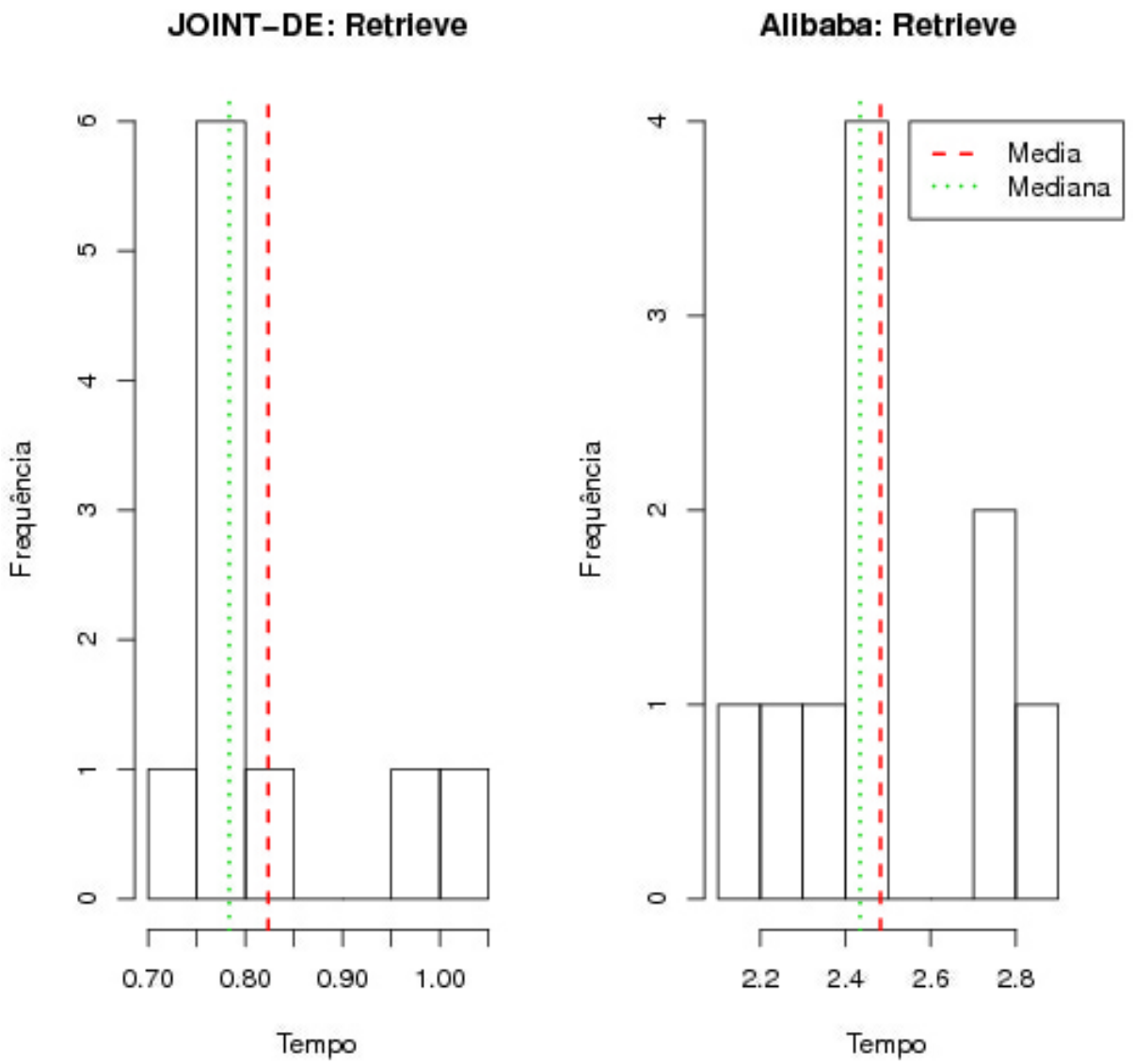


Fonte: Elaborado pelo autor.

Os histogramas de cada ferramenta com relação ao tempo na operação *retrieve* são apresentados na Figura 48, estes histogramas não sugerem explicitamente nenhum tipo

de distribuição dos dados do tempo por iteração em ambas as ferramentas. Para verificar a normalidade desses dados é necessário executar algum teste de normalidade.

Figura 48 – Histogramas da variável Tempo na operação *Retrieve*.

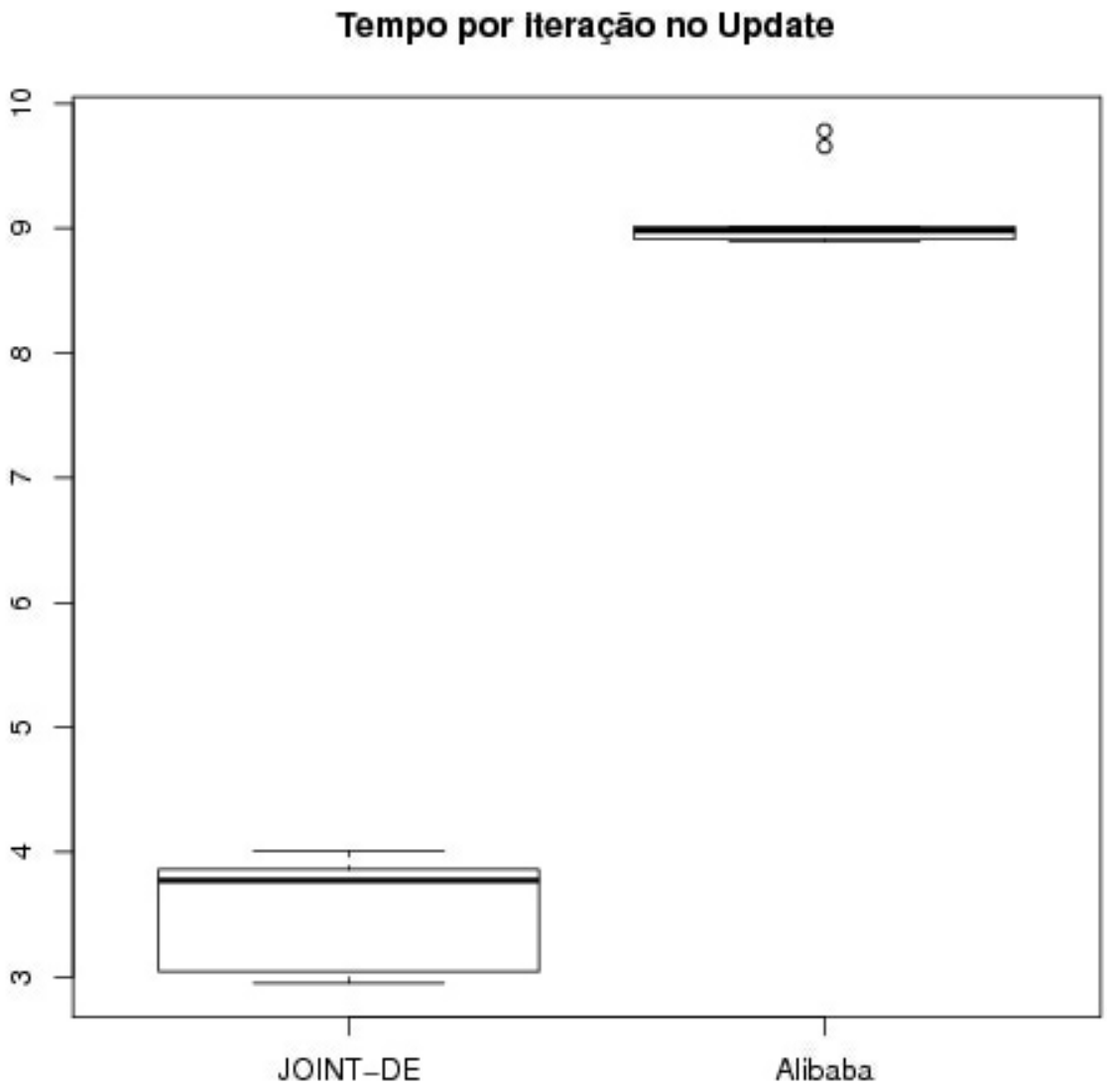


Fonte: Elaborado pelo autor.

A última operação a ser analisada graficamente com relação a variável Tempo por Iteração (T) é a operação de atualização de instâncias (*update*). A Figura 49 apresenta os diagramas de caixa do tempo para operação de atualização de instâncias para ambas as ferramentas. Não diferente dos demais, os diagramas de caixa da operação *update* também sugerem que o desempenho do JOINT-DE para atualização de objetos é melhor que o desempenho do OOMS Alibaba.

Os histogramas de cada ferramenta com relação ao tempo na operação *update* são

Figura 49 – Diagramas de caixa com comparativo de Tempo no *Update*.

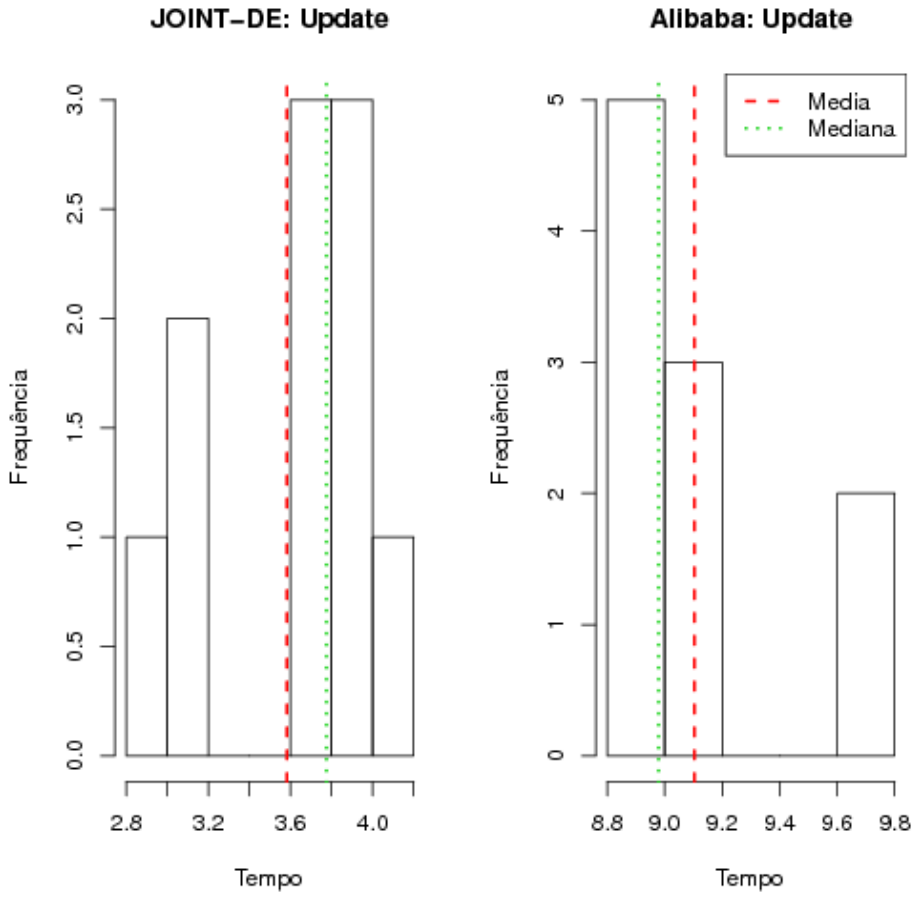


Fonte: Elaborado pelo autor.

apresentados na Figura 50. O histograma da ferramenta JOINT-DE para operação *update* na variável tempo não aponta nenhuma distribuição clara, ou seja, é necessário fazer os testes estatísticos para verificar a normalidade dos dados. Por outro lado, o histograma do Alibaba, para a mesma operação e variável, sugere uma distribuição de dados do tipo cauda longa.

A Tabela 9 sumariza os dados obtidos com relação ao tempo por iteração de cada ferramenta em cada operação.

Figura 50 – Histogramas da variável Tempo na operação *Update*.



Fonte: Elaborado pelo autor.

Tabela 9 – Sumarização dos dados relativos a variável Tempo por Iteração (T).

OOMS	Operação	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
JOINT-DE	Create	0.2765	0.2824	0.2924	0.3038	0.3312	0.3424
JOINT-DE	Retrieve	0.7442	0.7756	0.7836	0.8235	0.826	1.005
JOINT-DE	Update	2.952	3.205	3.775	3.582	3.852	4.01
Alibaba	Create	4.473	4.676	5.207	5.35	5.636	7.606
Alibaba	Retrieve	2.122	2.388	2.434	2.482	2.649	2.809
Alibaba	Update	8.896	8.91	8.978	9.104	9.014	9.781

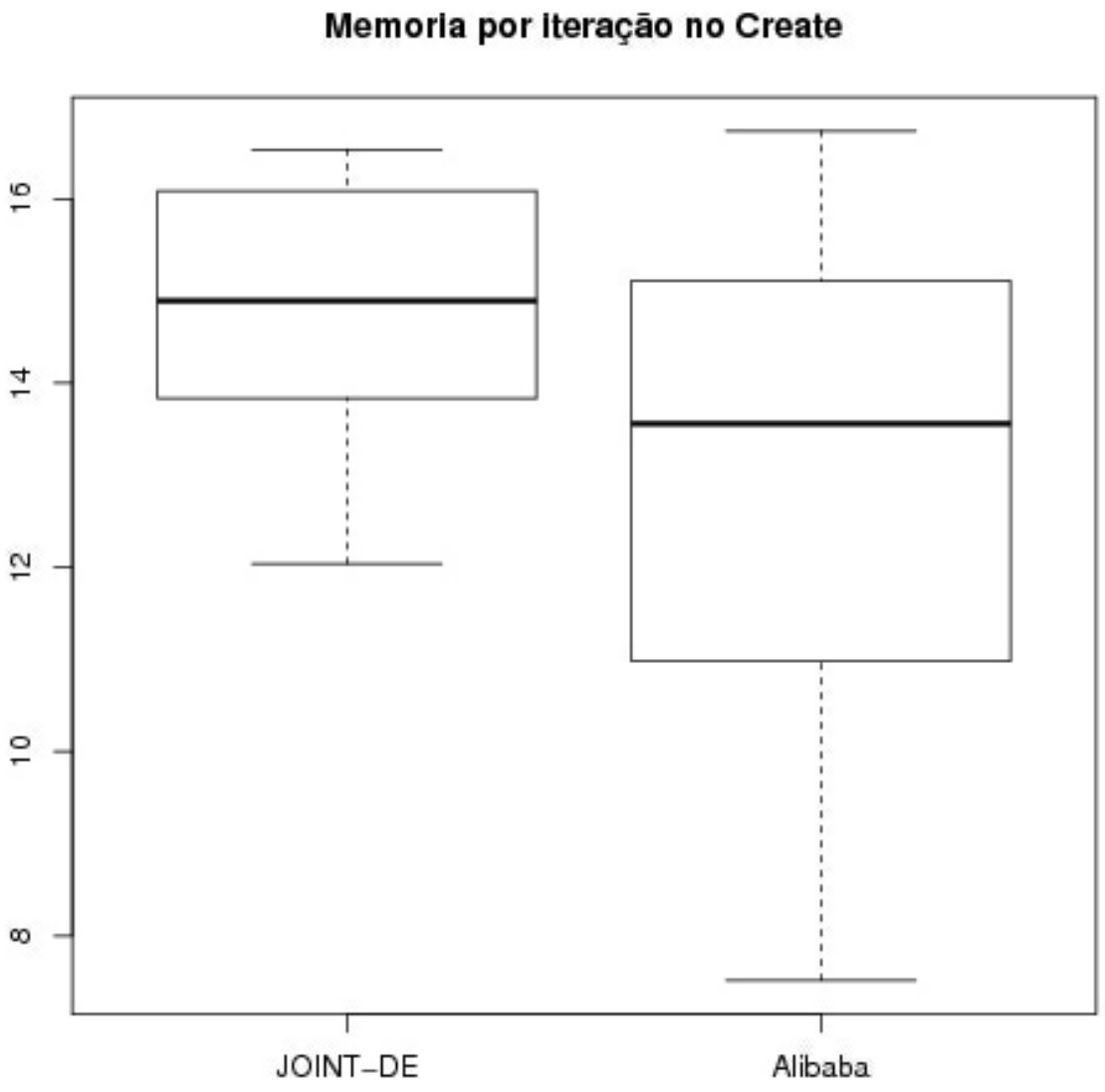
Fonte: Elaborado pelo autor.

6.3.2 Análise Descritiva da Média de Memória

Nesta subseção será analisada em termos de média de utilização de memória os OOMS JOINT-DE e o Alibaba nas três operações em questão. A Figura 51 mostra os diagramas de caixa para média de utilização de memória da operação de criação de instâncias. Diferentemente dos outros diagramas de caixa apresentados até então, este em particular,

mostra apenas uma pequena vantagem a favor da ferramenta Alibaba. Isso se deve ao fato de que não há muita diferença com relação ao uso de memória entre as ferramentas pois a instância é nova e não possui dados. Contudo, como os diagramas se interceptam, é necessário verificar o intervalo de confiança dos dados e a execução dos testes de hipóteses para ter uma conclusão estatisticamente válida.

Figura 51 – Diagramas de caixa com comparativo de Memória no *Create*.

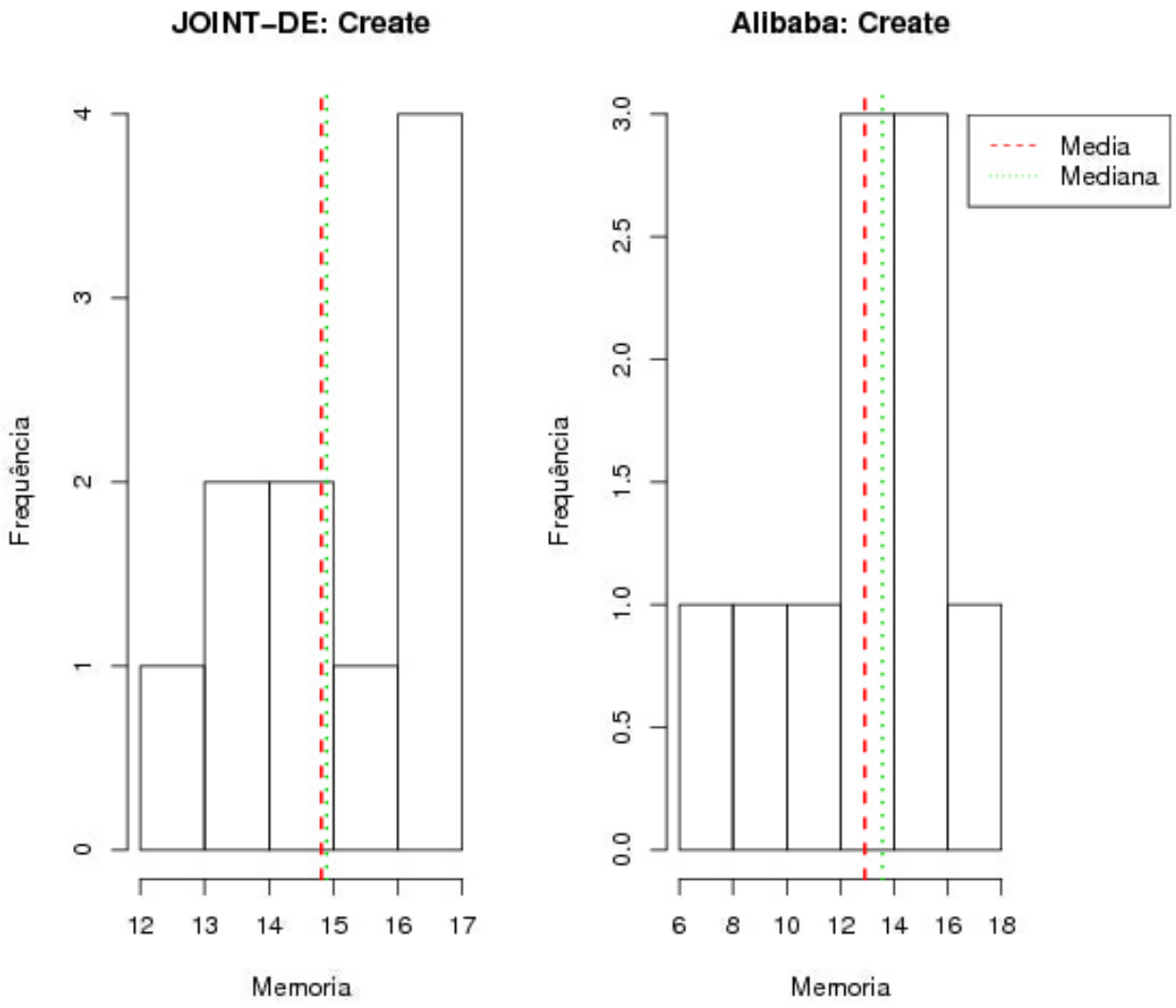


Fonte: Elaborado pelo autor.

Para verificar se os dados obtidos sobre memória na operação de criação de instâncias são normais ou não é necessário, primeiramente, analisar graficamente os histogramas. Os histogramas desta operação e métrica são apresentados na Figura 52. No caso do

JOINT-DE, o histograma sugere uma distribuição do tipo cauda longa a direita, já para o Alibaba a distribuição aparenta ser do tipo normal.

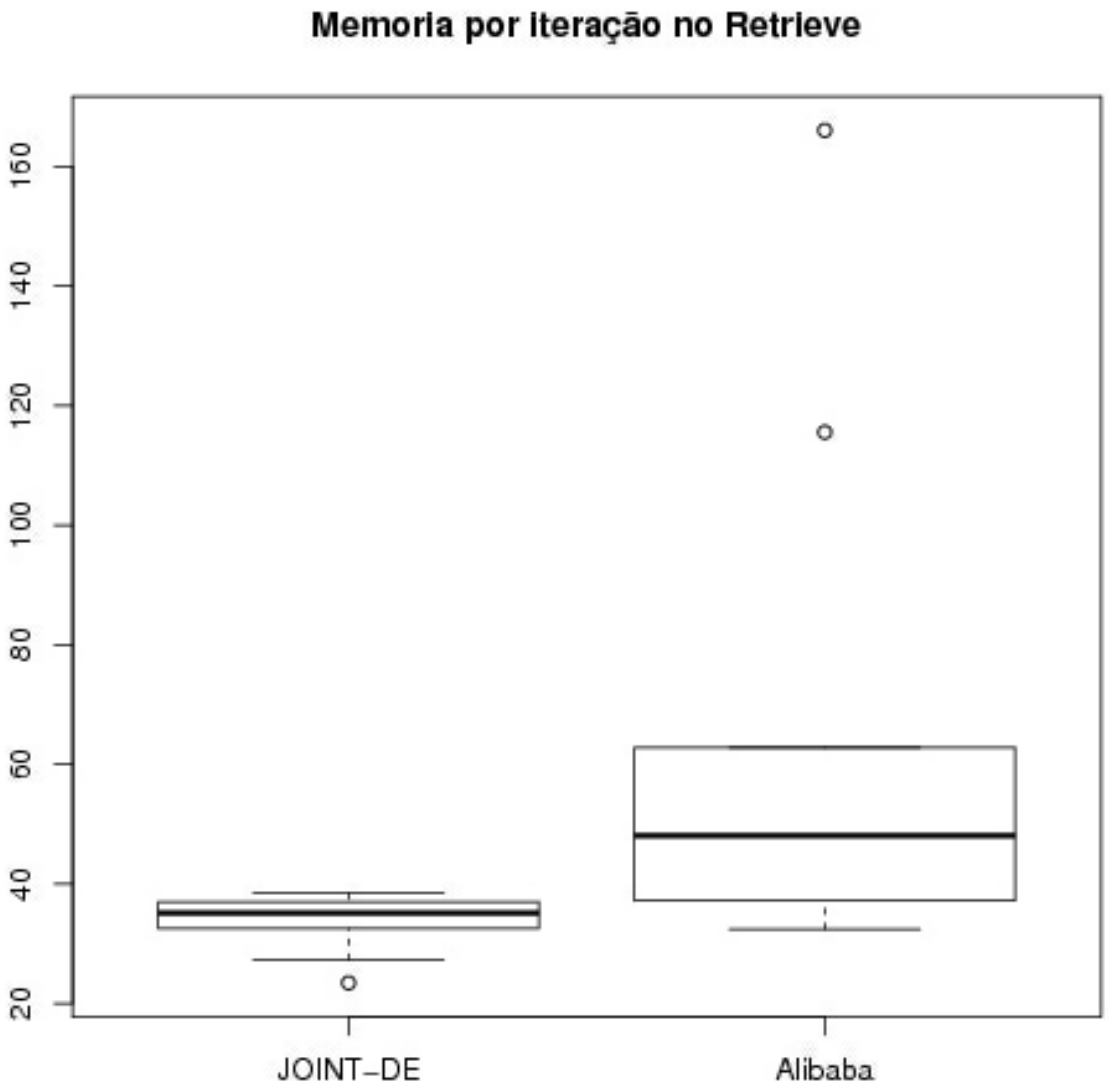
Figura 52 – Histogramas da variável Memória na operação *Create*.



Fonte: Elaborado pelo autor.

Para a operação de *retrieve*, o resultado já foi um pouco surpreendente. A operação de recuperação de instâncias, onde o JOINT-DE armazena todas as propriedades da instância na memória para uso *offline* e o Alibaba recupera as propriedades por demanda, deveria ser mais custosa quanto ao uso de memória para o JOINT-DE. Todavia, os diagramas de caixa para média de utilização de memória da operação de recuperação de instâncias apontam uma vantagem para o JOINT-DE, como pode ser visto na Figura 53. Porém, assim como no *create*, faz-se necessário verificar o intervalo de confiança com o intuito de analisar se esses intervalos se interceptam ou não.

Figura 53 – Diagramas de caixa com comparativo de Memória no *Retrieve*.

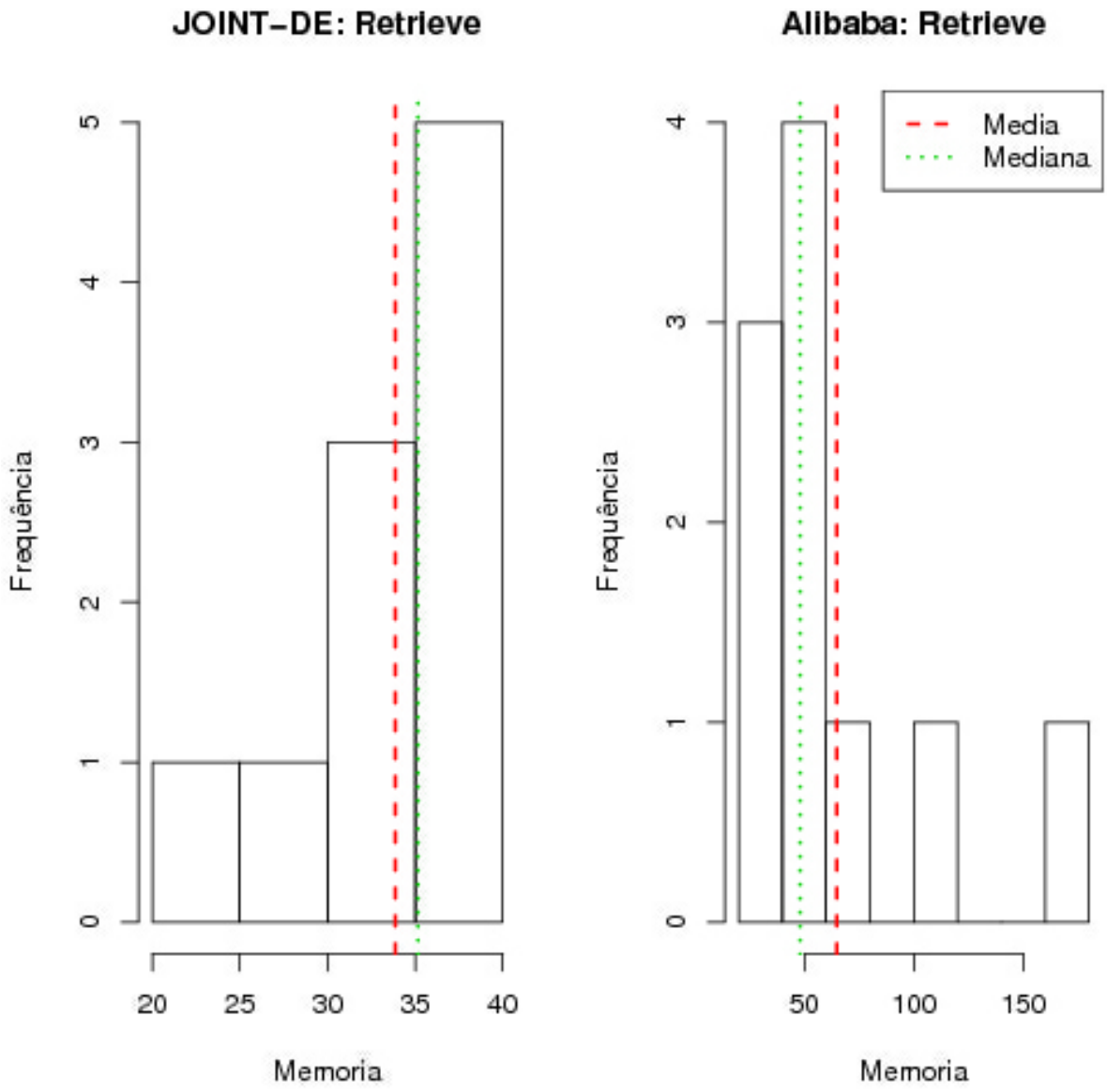


Fonte: Elaborado pelo autor.

Os histogramas de cada ferramenta com relação ao uso de memória na operação *retrieve* são apresentados na Figura 54, estes histogramas mostram que as distribuições dos dados de média de memória utilizada em ambas as ferramentas são do tipo cauda longa (para a direita no JOINT-DE e para a esquerda no Alibaba).

Finalizando a análise dos diagramas de caixa deste experimento, a Figura 55 apresenta os diagramas para a operação de atualização de instâncias com relação à média de memória utilizada. Nestes diagramas de caixa, fica evidente que o Alibaba possui uma boa vantagem quanto ao JOINT-DE no uso de memória na operação *update*. Os gráficos

Figura 54 – Histogramas da variável Memória na operação *Retrieve*.

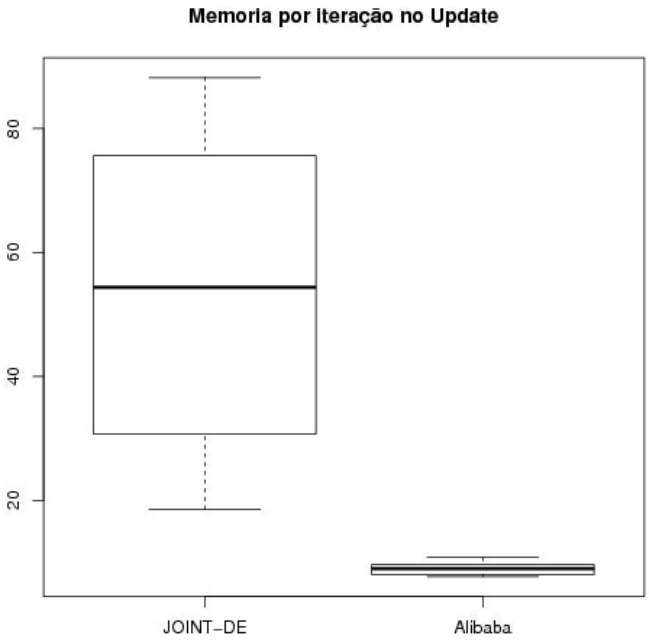


Fonte: Elaborado pelo autor.

de intervalo de confiança devem apenas confirmar essa vantagem. Os histogramas para a operação *update* e variável memória são apresentados na Figura 56. Pode-se observar que no caso do JOINT-DE, o histograma sugere uma distribuição do tipo bimodal (dois picos dispersos) e para o Alibaba a distribuição dos dados obtidos de memória utilizada aparenta ser normal.

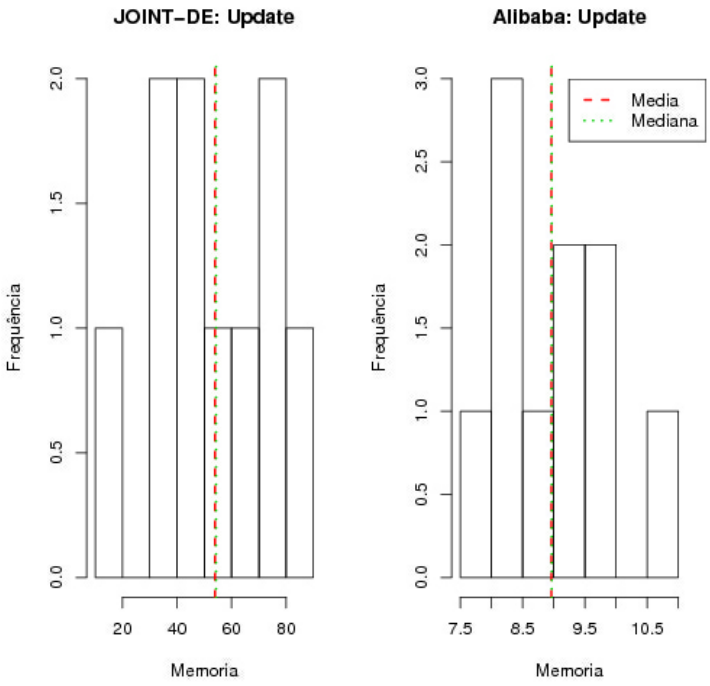
A Tabela 10 sumariza os dados obtidos com relação à média de memória utilizada durante a execução dos micro benchmarks em cada operação de cada ferramenta.

Figura 55 – Diagramas de caixa com comparativo de Memória no *Update*.



Fonte: Elaborado pelo autor.

Figura 56 – Histogramas da variável Memória na operação *Update*.



Fonte: Elaborado pelo autor.

Tabela 10 – Sumarização dos dados relativos a variável Média de Memória Utilizada (M).

OOMS	Operação	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
JOINT-DE	Create	12.04	13.9	14.89	14.81	16.08	16.54
JOINT-DE	Retrieve	23.43	33.1	35.14	33.86	36.93	38.4
JOINT-DE	Update	18.53	33.84	54.35	53.91	72.97	88.21
Alibaba	Create	7.516	11.25	13.56	12.9	14.91	16.74
Alibaba	Retrieve	32.44	39.13	48.1	64.72	61.69	166
Alibaba	Update	7.707	8.131	8.968	8.964	9.576	10.78

Fonte: Elaborado pelo autor.

6.3.3 Verificação das Hipóteses

Após a realização da análise descritiva dos dados coletados durante a execução do experimento, é necessário fazer uma análise estatística para verificar as hipóteses apresentas na subseção 6.1.1. Em todas as hipóteses levantadas, é importante determinar qual sistema de mapeamento objeto-ontologia é melhor de acordo com a variável e operação específica. Para responder tais hipóteses, os intervalos de confiança de cada ferramenta em cada operação/métrica serão analisados.

Um intervalo de confiança dá um alcance estimado de valores que é provável que inclua um parâmetro populacional desconhecido, este intervalo estimado é calculado a partir de um determinado conjunto de dados de amostra. Se essas amostras independentes são obtidas repetidamente da mesma população, então, uma determinada percentagem (nível de confiança) dos intervalos irá incluir o parâmetro populacional desconhecido (JAIN, 2008).

Para verificar as hipóteses faz-se necessário conhecer se as distribuições são normais ou não. Se a distribuição for normal, o teste de hipótese adequado é um teste paramétrico, caso contrário o teste escolhido deve ser um não paramétrico. Para verificar normalidade, os estatísticos sugerem o uso do teste de Shapiro-Wilk (SHAPIRO; FRANCIA, 1972). Para os casos de distribuição não normal foi escolhido o teste de Wilcoxon (WILCOXON, 1945) e para as distribuições normais o T-Test (WELCH, 1938), estas escolhas foram motivadas pela simplicidade dos testes e pela implementação dos mesmos pela ferramenta R, que foi utilizada para analisar este experimento. Vale ressaltar também, que o nível de confiança escolhido nas análises deste experimento é de 95%, ou seja, considerando um alfa (nível de significância) de 5% ou 0,05.

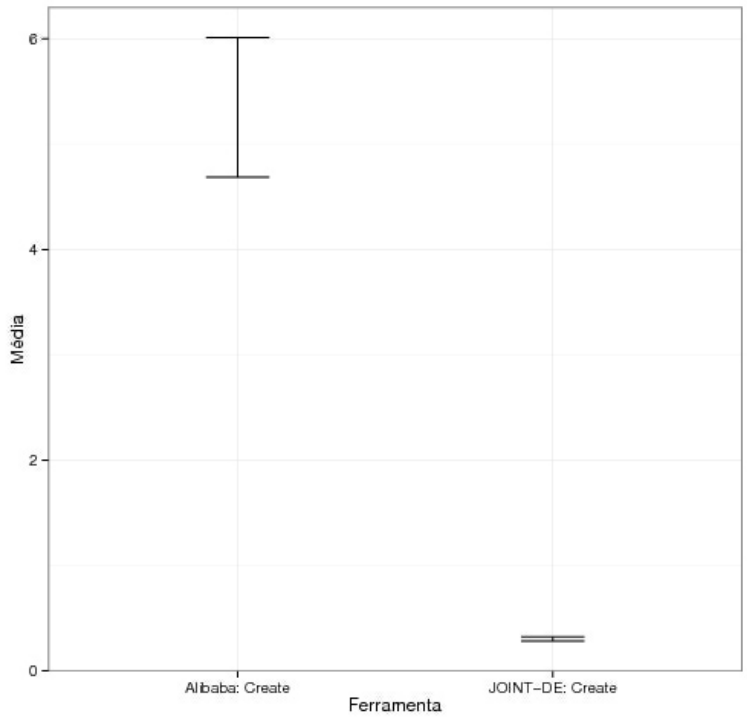
A primeira verificação de hipóteses será feita com relação ao tempo na operação de criação de instâncias. As hipóteses nula e alternativa estão numeradas abaixo, respectivamente:

- **H1-0:** O desempenho para as ferramentas no método de criação de objetos é igual.

- **H1-1:** O desempenho para as ferramentas no método de criação de objetos é diferente.

A primeira análise a ser feita é observando os intervalos de confiança tanto para o JOINT-DE como para o Alibaba. A Figura 57 ilustra os intervalos de confiança de cada ferramenta comparando o tempo por iteração na operação *create* com 5% de significância. Como pode se observar, não há sobreposição entre os intervalos e o intervalo do JOINT-DE está inferior ao do Alibaba. Dessa forma, pode-se afirmar apenas visualmente com 95% de confiança que o OOMS JOINT-DE é melhor do que o Alibaba com relação ao tempo por iteração na operação de criação de instâncias, refutando, assim, a hipótese nula **H1-0**.

Figura 57 – Intervalos de Confiança da métrica Tempo na operação *Create*.



Fonte: Elaborado pelo autor.

Mesmo a análise gráfica dos intervalos de confiança sendo suficientes para refutar a hipótese nula, um teste de hipótese comparativo pode ser realizado para confirmar o resultado. Para saber qual teste utilizar, primeiro é executado o teste de Shapiro com o objetivo de saber se os dados obtidos envolvendo esta métrica e operação possuem uma distribuição normal para ambas as ferramentas. O teste de normalidade de Shapiro apontou um p-valor para os dados do JOINT-DE de 0,03822 e um p-valor para os dados do Alibaba de 0,03098. Nos dois casos o p-valor é menor que o alfa (0,05, representando um nível de significância de 5%), dessa forma, ambas as distribuições aparentam não ser

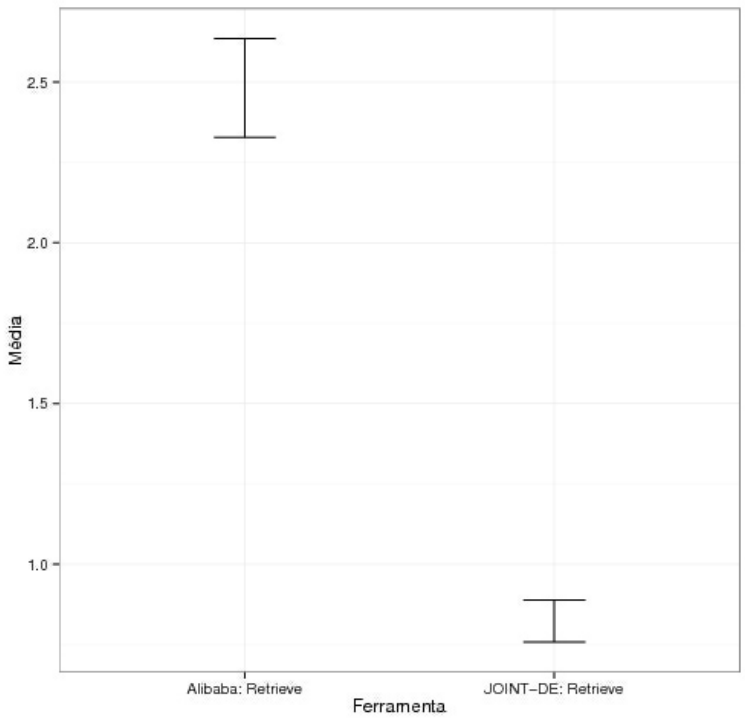
normais. Diante disto, o teste de hipótese executado para comparar estes dados foi o de Wilcoxon retornando um p-valor de 1,083e-05 (menor que 0,05), confirmando que o tempo para as ferramentas no método de criação de objetos é diferente.

A próxima verificação de hipótese será feita com relação à operação de recuperação de instâncias, também na variável tempo. As hipóteses nula e alternativa estão numeradas abaixo, respectivamente:

- **H2-0:** O desempenho para as ferramentas no método de recuperação de objetos é igual.
- **H2-1:** O desempenho para as ferramentas no método de recuperação de objetos é diferente.

A Figura 58 ilustra os intervalos de confiança de cada ferramenta comparando o tempo por iteração na operação *retrieve* com 5% de significância. Assim como nos intervalos de confiança da operação *create*, não há sobreposição entre os intervalos e o intervalo do JOINT-DE está inferior ao do Alibaba. Portanto, pode-se afirmar apenas visualmente que o OOMS JOINT-DE é melhor do que o Alibaba com relação ao tempo por iteração na operação de recuperação de instâncias, refutando, assim, a hipótese nula **H2-0**.

Figura 58 – Intervalos de Confiança da métrica Tempo na operação *Retrieve*.



Fonte: Elaborado pelo autor.

Objetivando confirmar a análise gráfica dos intervalos de confiança, o teste de hipótese comparativo também foi executado para estes dados. O teste de normalidade de Shapiro

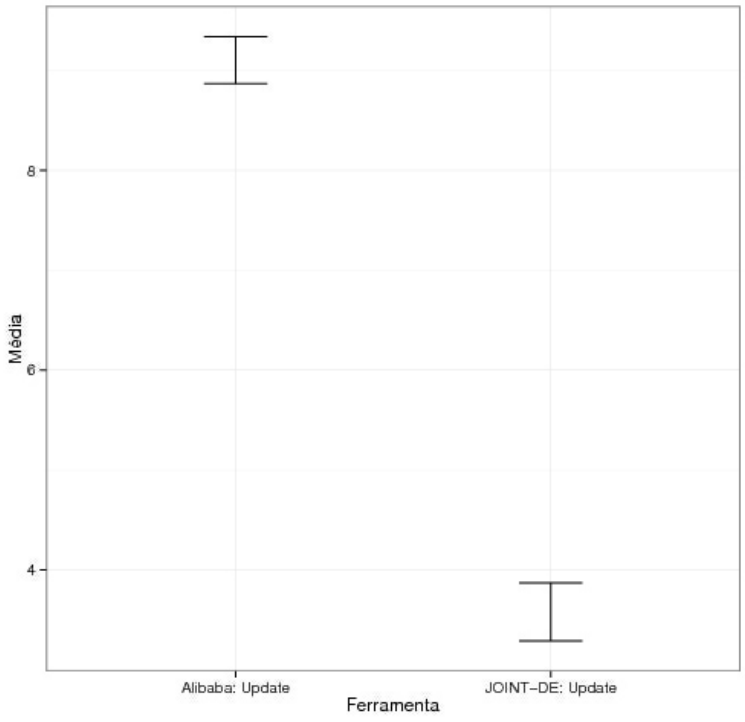
apresentou um p-valor de 0,002626 para os dados do JOINT-DE e um p-valor de 0,486 para os dados do Alibaba, apontando assim, uma normalidade nos dados do Alibaba e uma distribuição não normal nos dados do JOINT-DE. Para executar o T-Test é necessário que ambas as distribuições sejam normais, logo, o teste de Wilcoxon foi executado e apresentou um p-valor de 1,083e-05 menor que 0,05, confirmando que o tempo para as ferramentas no método de recuperação de objetos é diferente.

A última verificação de hipótese com relação ao tempo é na operação de atualização de instâncias. As hipóteses nula e alternativa estão numeradas abaixo, respectivamente:

- **H3-0:** O desempenho para as ferramentas no método de atualização de objetos é igual.
- **H3-1:** O desempenho para as ferramentas no método de atualização de objetos é diferente.

A Figura 59 ilustra os intervalos de confiança de cada ferramenta comparando o tempo por iteração na operação *update* com 5% de significância. Seguindo o mesmo resultado das outras operações na variável tempo, não há sobreposição entre os intervalos e o intervalo do JOINT-DE também está inferior ao do Alibaba. Portanto, nesta operação, pode-se afirmar visualmente que o OOMS JOINT-DE é melhor do que o Alibaba com relação ao tempo por iteração, refutando, assim, a hipótese nula **H3-0**.

Figura 59 – Intervalos de Confiança da métrica Tempo na operação *Update*.



Fonte: Elaborado pelo autor.

A execução do teste de normalidade de Shapiro, apontou um p-valor de 0,008859 para os dados do JOINT-DE e um p-valor de 0,0002056 para os dados do Alibaba. Como os dois são menores que 0.05 (distribuições não normais), foi executado também o teste de hipótese de Wilcoxon, retornando um p-valor de 0,0001817 (menor que 0.05). Portanto, pelo teste de hipótese comparativo de Wilcoxon, o tempo para as ferramentas no método de atualização de objetos é diferente.

Após analisar as hipóteses relativas ao tempo por iteração de cada operação, a próxima verificação de hipóteses será realizada envolvendo a métrica de média de memória utilizada, começando com as hipóteses acerca da operação de criação de instâncias. As hipóteses nula e alternativa estão numeradas abaixo, respectivamente:

- **H4-0:** A utilização de memória para as ferramentas no método de criação de objetos é igual.
- **H4-1:** A utilização de memória para as ferramentas no método de criação de objetos é diferente.

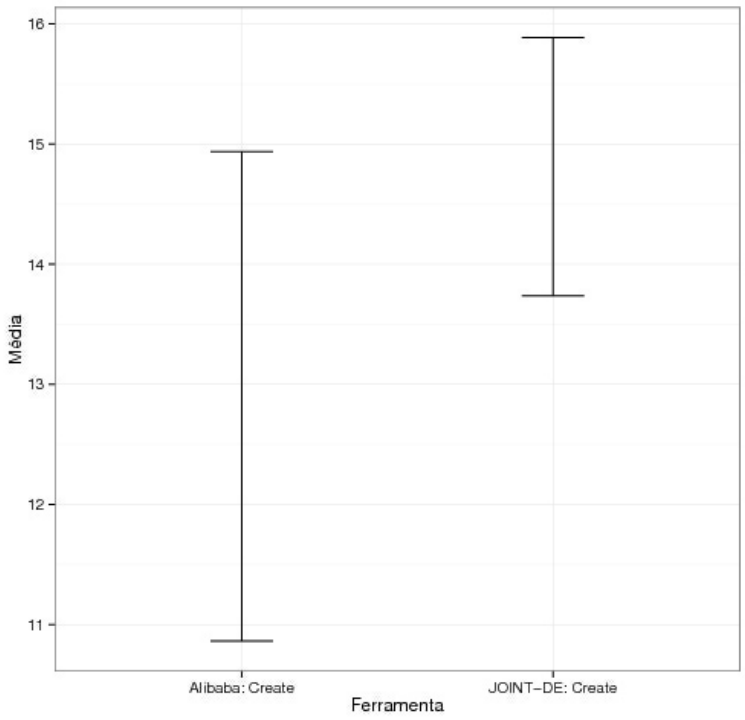
A Figura 60 apresenta os intervalos de confiança de cada ferramenta comparando a média de memória utilizada na operação *create* com 5% de significância. Diferentemente dos outros intervalos de confiança, o intervalo de memória no *create* para o JOINT-DE se sobrepõe quase que totalmente com o intervalo de memória do Alibaba. Dada esta sobreposição, não há como fazer qualquer afirmação sobre qual ferramenta possui menor utilização de memória neste cenário (apesar da aparente vantagem da ferramenta Alibaba). Dessa forma, há uma necessidade real de verificar as hipóteses por meio de um teste de hipótese comparativo.

O teste de Shapiro para os dados de memória do JOINT-DE na operação de criação de instâncias resultou em um p-valor de 0,3286, ou seja, p-valor maior que 0,05 (distribuição normal). Para os dados de memória do Alibaba, o teste de Shapiro retornou um p-valor de 0,8086, sendo também maior que 0,05. Dessa forma, assumindo que as duas distribuições são normais aplicou-se o T-Test resultando em um p-valor de 0,08198, sendo maior que 0,05. Portanto, com 95% de confiança não há evidência estatística para refutar a hipótese nula **H4-0**, i.e., estatisticamente não pode-se afirmar que a utilização de memória para as ferramentas no método de criação de objetos é diferente.

A próxima verificação de hipótese será feita com relação à operação de recuperação de instâncias, também na variável memória. As hipóteses nula e alternativa estão numeradas abaixo, respectivamente:

- **H5-0:** A utilização de memória para as ferramentas no método de recuperação de objetos é igual.
- **H5-1:** A utilização de memória para as ferramentas no método de recuperação de objetos é diferente.

Figura 60 – Intervalos de Confiança da métrica Memória na operação *Create*.



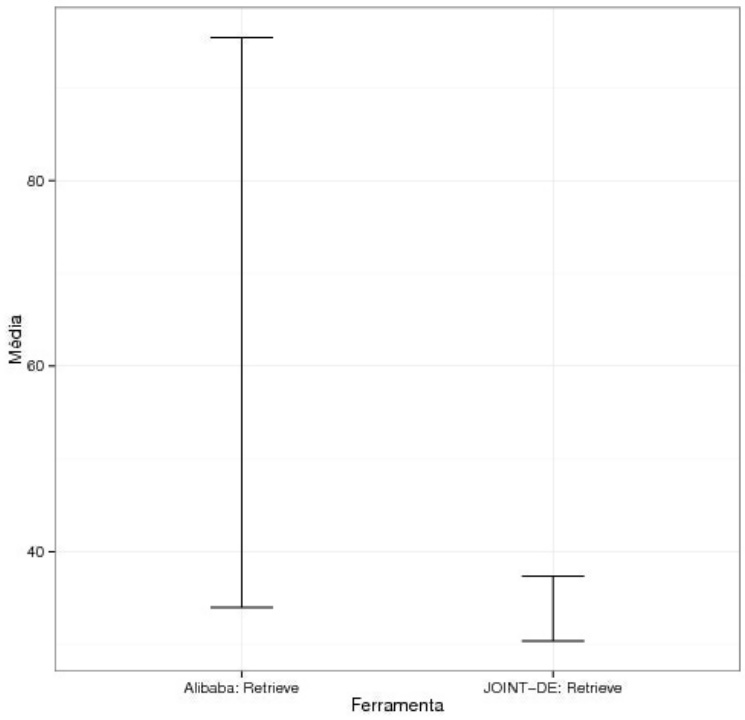
Fonte: Elaborado pelo autor.

A Figura 61 mostra os intervalos de confiança de cada ferramenta envolvendo a média de memória utilizada na operação *retrieve*, também com 5% de significância. Assim como nos intervalos de memória da operação *create*, há uma sobreposição entre os intervalos, embora, neste caso, o intervalo do JOINT-DE está ligeiramente inferior ao do Alibaba. Seguindo a mesma análise dos intervalos de memória da operação de criação de objetos, não tem como fazer qualquer afirmação apenas com esta análise gráfica, faz-se necessário executar um teste de hipótese comparativo.

Conforme o teste de normalidade de Shapiro, os dados de memória na operação *retrieve* para o JOINT-DE obtiveram um p-valor de 0,03369 e os dados do Alibaba obtiveram um p-valor de 0,002715, ambos com valor inferior a 0,05, caracterizando distribuições não normais. Em virtude da natureza destas distribuições, o teste Wilcoxon foi executado com estes dados e resultou em um p-valor de 0,0115, sendo inferior que 0,05. Logo, pelo teste de hipótese comparativo de Wilcoxon, a utilização de memória para as ferramentas no método de recuperação de objetos é diferente, refutando a hipótese nula **H5-0**. Vale ressaltar, que ao refutar a hipótese nula, pode-se concluir, com 95% de confiança, que o JOINT-DE utiliza menos memória do que o Alibaba na operação *retrieve*.

Finalizando a verificação de hipóteses, serão analisadas as hipóteses que envolvem o uso de memória na operação de atualização de instâncias. As hipóteses nula e alternativa estão numeradas abaixo, respectivamente:

Figura 61 – Intervalos de Confiança da métrica Memória na operação *Retrieve*.



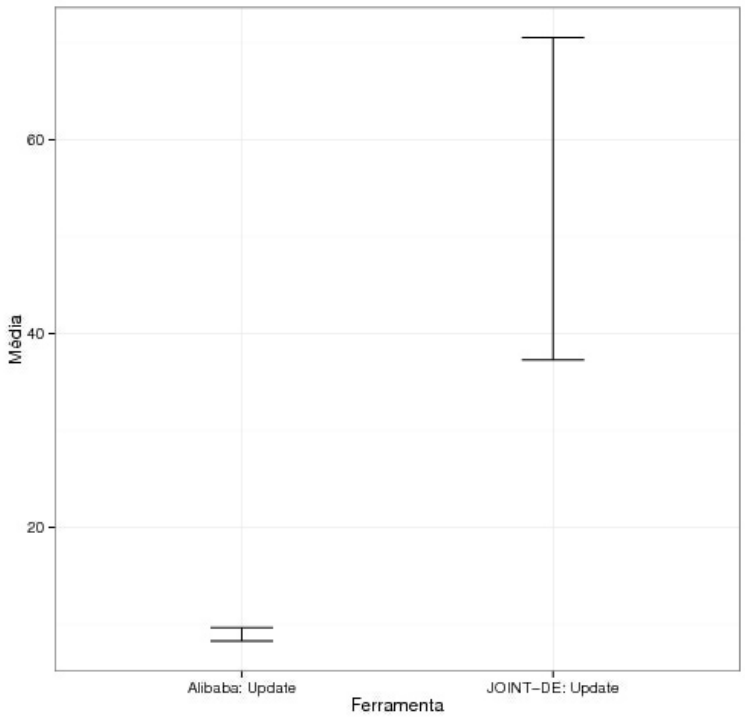
Fonte: Elaborado pelo autor.

- **H6-0:** A utilização de memória para as ferramentas no método de atualização de objetos é igual.
- **H6-1:** A utilização de memória para as ferramentas no método de atualização de objetos é diferente.

Os intervalos de confiança de cada ferramenta comparando o uso médio de memória na operação *update* com 5% de significância são apresentados na Figura 62. Ao contrário das outras duas análises envolvendo memória, nesta, o intervalo de memória do Alibaba está totalmente abaixo do intervalo de memória do JOINT-DE. Dessa forma, independente de realizar qualquer teste de hipótese, pode-se afirmar que o OOMS Alibaba utiliza menos memória do que o JOINT-DE na operação de atualização, refutando, assim, a hipótese nula **H6-0**.

Com o objetivo de confirmar a análise gráfica dos intervalos de confiança, o teste de hipótese comparativo também foi executado para estes dados. O teste de normalidade de Shapiro apresentou um p-valor de 0,7863 para os dados do JOINT-DE e um p-valor de 0,6572 para os dados do Alibaba. Assim, assumindo que as duas distribuições são normais (p-valor maior que 0.05) aplicou-se o T-Test resultando em um p-valor de 0,0001755, sendo menor que 0,05. Portanto, com 95% de confiança, pelo teste de hipótese comparativo T-

Figura 62 – Intervalos de Confiança da métrica Memória na operação *Update*.



Fonte: Elaborado pelo autor.

Test, a utilização de memória para as ferramentas no método de atualização de objetos é diferente.

6.4 Principais Conclusões

O experimento conduzido neste capítulo, visou avaliar em termos de desempenho e de utilização de memória o JOINT-DE em comparação com outro OOMS bastante conhecido na literatura: Alibaba. Essas variáveis foram avaliadas separadamente em cada uma das três operações em objetos: *create*, *retrieve* e *update* (operações do CRUD, com exceção do delete não presente na ferramenta Alibaba).

Pelas análises gráficas e em alguns casos pelos testes de hipóteses comparativos, foi possível verificar que: i) o JOINT-DE obteve um melhor desempenho (menor tempo por iteração) em comparação com o Alibaba nas três operações, talvez devido ao fato de que ele utiliza classes pré-compiladas o que torna seus algoritmos mais rápido; ii) no que tange ao uso de memória, não houve diferença estatística na operação de criação de instâncias entre as ferramentas, o que também era esperado, pois a criação de objetos não requer um consumo alto de memória para nenhuma das ferramentas; iii) o JOINT-DE teve uma pequena vantagem na utilização de memória na operação de recuperação de instâncias, esse foi o único resultado que de fato foi surpreendente, pois teoricamente o JOINT-DE era pra consumir mais memória ao carregar, além do objeto, todas as suas

propriedades para uso *offline*; iv) o Alibaba utiliza bem menos memória do que o JOINT-DE na atualização de instâncias, o que também era de se esperar pois o JOINT-DE está com todas as propriedades do objeto na memória e o Alibaba não.

7 CONCLUSÃO E TRABALHOS FUTUROS

O trabalho apresentou o JOINT-DE, um sistema de mapeamento objeto-ontologia que suporta o uso de objetos *desconectados*. Através de diagramas e algoritmos foi descrito o mecanismo interno da ferramenta que a possibilitou prover objetos *desconectados*, consolidados em sistemas com bancos de dados relacionais, no contexto de aplicações baseadas em ontologias. O JOINT-DE se faz necessário justamente pela falta de OOMS estáveis que permitam o desenvolvimento de aplicações baseadas em ontologias através de objetos *desconectados*. O sistema foi construído utilizando, principalmente, a API do Sesame para conexão com o banco de dados RDF e a Java *Reflection* API para carregar os objetos e suas propriedades em memória. Neste sentido, foi possível a implementação das quatro operações CRUD em instâncias, além de métodos para execução de consultas SPARQL. Um gerador de código Java a partir de ontologias também foi apresentado, permitindo a criação automática de interfaces e classes Java correspondentes as entidades presentes nas ontologias.

Visando avaliar a ferramenta proposta em um cenário real de aplicações multicamadas orientadas a objetos, um estudo de caso foi desenvolvido implantando o JOINT-DE no MeuTutor-ENEM: um sistema web inteligente que prepara alunos do Ensino Médio para a prova do ENEM. O uso do JOINT-DE como OOMS no MeuTutor-ENEM, supriu algumas limitações da aplicação, como, por exemplo, permitir que os objetos provenientes do banco de dados RDF pudessem ser usados como objetos do modelo de negócio da aplicação, mantendo a transparência entre as camadas arquiteturais. Além disso, a implantação do JOINT-DE no MeuTutor-ENEM reduziu, consideravelmente, o número de conexões abertas com o banco de dados RDF. Nesse estudo de caso, não somente foi mostrado como o JOINT-DE beneficiou em termos arquiteturais o sistema do MeuTutor-ENEM, mas também foram apresentados resultados de testes comparativos que apontaram uma melhora significativa no tempo de resposta dos serviços da aplicação, além de uma leve redução no consumo de memória.

Por fim, um experimento foi conduzido a fim de avaliar e comparar em termos de desempenho e utilização de memória o JOINT-DE com o Alibaba. Essas duas características foram avaliadas em três operações em objetos: criação, recuperação e atualização de instâncias. O experimento apresentou resultados preliminares bastante satisfatórios, principalmente com relação ao desempenho onde o JOINT-DE obtive melhores resultados nas três operações. Quanto à utilização de memória, os resultados favoreceram ambos os lados: estatisticamente igual na criação; melhor uso de memória para o JOINT-DE na recuperação e; melhor uso de memória do Alibaba na atualização. Dessa forma, embora o JOINT-DE possua uma sobrecarga por carregar o objeto em memória para seu uso *offline*, ela não é impactante diante das necessidades arquiteturais supridas pelo uso de

objetos *desconectados*.

7.1 Principais Contribuições

As principais contribuições deste trabalho são apresentadas a seguir:

- Implementação de operações de criação, recuperação, atualização e remoção de instâncias em ontologias através do paradigma de orientação a objetos, permitindo que os desenvolvedores possam manipular os dados em uma ontologia de maneira mais simples;
- Construção de um gerador automático de código Java a partir de ontologias (mais especificamente arquivos OWL), reduzindo o esforço dos desenvolvedores. Uma vez que os desenvolvedores não precisam criar manualmente classes que representem as entidades nas ontologias;
- Viabilização da execução de consultas em SPARQL com o retorno convertido em objetos Java correspondentes, seja esses objetos representações de instâncias das ontologias ou tipos de dados primitivos;
- Utilização da abordagem de objetos desconectados, presentes em ORMS tradicionais, em aplicações baseadas em ontologias. Usando esta abordagem, objetos gerados pelo JOINT-DE podem ser usados como objetos do modelo de negócio da aplicação, mesmo que esta, exija uma transparência arquitetural entre suas camadas;
- Criação de experimentos e um estudo de caso real para avaliar as vantagens e desvantagens de um OOMS com objetos desconectados e de um OOMS com objetos persistentes;
- Publicação de dois artigos durante esses dois anos de mestrado:
  - JOINT: Java ontology integrated toolkit, Expert Systems with Applications, 2013 (HOLANDA et al., 2013b);
  - Towards an Agent-Based Approach for Automatic Generation of Researcher Profiles Using Multiple Data Sources, 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) (HOLANDA et al., 2013a).

7.2 Limitações e Trabalhos Futuros

Algumas questões que não foram foco deste trabalho, mas que devem ser consideradas em trabalhos futuros, são a inferência e a validação automática de construtores

OWL durante a manipulação dos objetos gerados pelo JOINT-DE pelos desenvolvedores. Atualmente, o OOMS proposto processa alguns construtores do OWL na geração de código Java apenas como anotações para fins de documentação de código. Por exemplo, propriedades que tem restrições de valores definidos por meio do construtor *owl:OneOf*, só deveriam aceitar como parâmetros no código Java esses valores específicos. Outro exemplo são as restrições de cardinalidade, as quais limitariam a quantidade de valores que uma propriedade deve receber como parâmetro. A complexidade de raciocinar sobre estes construtores a nível de orientação a objetos está, justamente, em como não perder desempenho na manipulação de instâncias.

Outra questão que não está dentro do escopo deste trabalho, mas que também deve ser discutida, é a segurança dos objetos retornados pelo JOINT-DE. Uma vez que os objetos são carregados na memória e retornados para a aplicação, o controle que o JOINT-DE possui sobre estes objetos se perde. Portanto, faz-se necessário estudar mecanismos que segurem estes *JavaBeans* para que intrusos não acessem os dados contidos neles. Por exemplo, a utilização de mecanismos de autenticação no acesso de propriedades específicas dos *JavaBeans*.

Como trabalhos futuros, pretende-se flexibilizar a ferramenta no que se refere a configuração do carregamento de propriedades no primeiro nível. Em alguns casos, é interessante permitir que o desenvolvedor configure quais objetos devem ser carregados de forma tardia e quais devem ser carregados inicialmente. Dessa forma, o JOINT-DE pode se adequar as necessidades especificas de negócio da maioria das aplicações.

Ainda, faz-se necessário uma análise cuidadosa de todos os algoritmos da ferramenta proposta com vistas a otimizá-la tanto em desempenho como em utilização de memória. Embora o JOINT-DE tenha sido desenvolvido com esta questão em mente, há muitas peculiaridades da própria linguagem Java que podem ser utilizadas para otimização, como, por exemplo, implementações de listas que consomem menos memória em um determinado cenário.

Finalmente, pretende-se estender o experimento realizado neste trabalho para um número maior de repetições e com a utilização de outros modelos. Portanto, a partir deste experimento espera-se generalizar os resultados obtidos para cenários semelhantes aos do experimento. O JOINT-DE também será disponibilizado para que a comunidade use com o objetivo de que outras aplicações, que demandem sistemas de mapeamento objeto-ontologia com suporte a objetos *desconectados*, se tornem outros estudos de casos.

REFERÊNCIAS

AASMAN, J. *Allegro graph: RDF triple database*. 2006. Technical report. Franz Incorporated.

BARD, J. B.; RHEE, S. Y. Ontologies in biology: design, applications and future challenges. *Nature Reviews Genetics*, Nature Publishing Group, v. 5, n. 3, p. 213–222, 2004.

BAUER, C.; KING, G. *Hibernate in action*. 1. ed. Greenwich: Manning Greenwich, 2005.

BAUER, C.; KING, G. *Java Persistence with Hibernate*. 1. ed. Greenwich: Manning Publications, 2006.

BERNERS-LEE, T. et al. The semantic web. *Scientific american*, New York, NY, USA:, v. 284, n. 5, p. 28–37, 2001.

BITTENCOURT, I. I. et al. A computational model for developing semantic web-based educational systems. *Knowledge-Based Systems*, Elsevier, v. 22, n. 4, p. 302–315, 2009.

BOLEY, H.; TABET, S.; WAGNER, G. Design rationale for ruleml: A markup language for semantic web rules. In: *International Semantic Web Working Symposium (SWWS)*. Stanford University, California, USA: IOS Press, 2001. v. 1, p. 381–401.

BORGES, A. M. et al. Towards a study opportunities recommender system in ontological principles-based on semantic web environment. *WSEAS Transactions on Computers*, World Scientific and Engineering Academy and Society (WSEAS), v. 8, n. 2, p. 279–291, 2009.

BOZSAK, E. et al. Kaon - towards a large scale semantic web. In: BAUKNECHT, K.; TJOA, A.; QUIRCHMAYR, G. (Ed.). *E-Commerce and Web Technologies*. Springer Berlin Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2455). p. 304–313. ISBN 978-3-540-44137-3. Disponível em: <http://dx.doi.org/10.1007/3-540-45705-4_32>.

BROEKSTRA, J.; KAMPMAN, A.; HARMELEN, F. van. Sesame: A generic architecture for storing and querying rdf and rdf schema. In: HORROCKS, I.; HENDLER, J. (Ed.). *The Semantic Web - ISWC 2002*. Springer Berlin Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2342). p. 54–68. ISBN 978-3-540-43760-4. Disponível em: <http://dx.doi.org/10.1007/3-540-48005-6_7>.

BUSCHMANN, F.; HENNEY, K.; SCHIMDT, D. *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. 1. ed. Chichester, England: John Wiley & Sons, 2007.

CARDOSO, J.; HEPP, M.; LYTRAS, M. D. Real-world applications from industry. In: JAIN, R.; SHETH, A. (Ed.). *The Semantic Web*. 1. ed. New York: Springer Science & Business Media, 2007, (Semantic Web and Beyond, v. 6).

CHEYER, A.; GRUBER, T. *Siri: A virtual personal assistant for iphone, an ontology-driven application for the masses*. 2010. Presentation at Open, International, Virtual Community of Practice on Ontology, Ontological Engineering and Semantic Technology.

CORBETT, A. T.; ANDERSON, J. R. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, Springer, v. 4, n. 4, p. 253–278, 1994.

DAS, A.; WU, W.; MCGUINNESS, D. L. Industrial strength ontology management. In: *International Semantic Web Working Symposium (SWWS)*. Stanford University, California, USA: IOS Press, 2001. v. 1, p. 17–38.

DEVEDŽIC, V. *Semantic web and education*. 1. ed. New York: Springer Science & Business Media, 2006. (Integrated Series in Information Systems, v. 12).

ERLING, O.; MIKHAILOV, I. Rdf support in the virtuoso dbms. In: PELLEGRINI, T. et al. (Ed.). *Networked Knowledge - Networked Media*. Springer Berlin Heidelberg, 2009, (Studies in Computational Intelligence, v. 221). p. 7–24. ISBN 978-3-642-02183-1. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02184-8_2>.

ESPOSITO, D. *Building web solutions with ASP. NET and ADO. NET*. 1. ed. Redmond, Washington: Microsoft Press, 2002.

FLATLA, D. R. et al. Calibration games: making calibration tasks enjoyable by adding motivating game elements. In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. Santa Barbara, California, USA: ACM, 2011. p. 403–412.

FORMAN, I. R.; FORMAN, N. *Java reflection in action*. Greenwich, CT, USA: Manning Publications, 2004. ISBN 1932394184.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GOSLING, J. *Java: An Overview*. 1995. Sun Microsystems Laboratories The First Ten Years.

GROVE, M. *Empire: RDF and SPARQL Meet Java and JPA*. 2010. 2010 Semantic Technology Conference.

HAARSLEV, V. et al. The racerpro knowledge representation and reasoning system. *Semantic Web*, IOS Press, v. 3, n. 3, p. 267–277, 2012.

HAASE, P. et al. The neon ontology engineering toolkit. In: W3C. *WWW 2008 Developers Track*. Beijing, China, 2008.

HEPP, M. et al. *semantic web, semantic web services, and business applications*. 1. ed. New York: Springer Science & Business Media, 2007. (Semantic Web and Beyond, v. 7).

HOLANDA, O. et al. Towards an agent-based approach for automatic generation of researcher profiles using multiple data sources. In: *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on*. Atlanta, GA: IEEE, 2013. v. 3, p. 163–166.

HOLANDA, O. et al. Joint: Java ontology integrated toolkit. *Expert Systems with Applications*, Elsevier, v. 40, n. 16, p. 6469–6477, 2013.

HUANG, N.; DIAO, S. Ontology-based enterprise knowledge integration. *Robotics and Computer-Integrated Manufacturing*, Elsevier, v. 24, n. 4, p. 562–571, 2008.

HUNT, C.; JOHN, B. *Java performance*. 1. ed. New Jersey, US: Addison-Wesley Longman Publishing Co., Inc., 2011. (The Java Series).

ISOTANI, S. et al. An ontology engineering approach to the realization of theory-driven group formation. *International Journal of Computer-Supported Collaborative Learning*, Springer, v. 4, n. 4, p. 445–478, 2009.

JAIN, R. *The art of computer systems performance analysis*. 2. ed. New Delhi Area, India: Wiley India Pvt. Limited, 2008.

JURISTO, N.; MORENO, A. M. *Basics of software engineering experimentation*. 1. ed. New York, US: Springer Verlag, 2001.

KAPP, K. M. *The gamification of learning and instruction: game-based methods and strategies for training and education*. 1. ed. San Francisco, CA, US: John Wiley & Sons, 2012.

KIRYAKOV, A. et al. The features of bigowlim that enabled the bbc's world cup website. In: VLDB. *Workshop on Semantic Data Management*. Singapore, 2010. p. 9–12.

KIRYAKOV, A.; OGNYANOV, D.; MANOV, D. Owlim – a pragmatic semantic repository for owl. In: DEAN, M. et al. (Ed.). *Web Information Systems Engineering - WISE 2005 Workshops*. Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3807). p. 182–192. ISBN 978-3-540-30018-2. Disponível em: <http://dx.doi.org/10.1007/11581116_19>.

KUPERBERG, M.; KROGMANN, M.; REUSSNER, R. Timermeter: Quantifying properties of software timers for system analysis. In: IEEE. *Quantitative Evaluation of Systems, 2009. QEST'09. Sixth International Conference on the*. Budapest, 2009. p. 85–94.

MCBRIDE, B. Jena: A semantic web toolkit. *IEEE Internet computing*, IEEE Computer Society, v. 6, n. 6, p. 55–59, 2002.

MIKA, P. Ontologies are us: A unified model of social networks and semantics. In: GIL, Y. et al. (Ed.). *The Semantic Web - ISWC 2005*. Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3729). p. 522–536. ISBN 978-3-540-29754-3. Disponível em: <http://dx.doi.org/10.1007/11574620_38>.

MITCHELL, M. *An introduction to genetic algorithms*. 1. ed. Cambridge, Massachusetts: MIT press, 1998.

NOY, N. F.; FERGERSON, R. W.; MUSEN, M. A. The knowledge model of protege-2000: Combining interoperability and flexibility. In: DIENG, R.; CORBY, O. (Ed.). *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*. Springer Berlin Heidelberg, 2000, (Lecture Notes in Computer Science, v. 1937). p. 17–32. ISBN 978-3-540-41119-2. Disponível em: <http://dx.doi.org/10.1007/3-540-39967-4_2>.

PAIVA, R. et al. Mineração de dados e a gestão inteligente da aprendizagem: Desafios e direcionamentos. In: CSBC. *Anais do Workshop de Desafios da Computação Aplicada a Educação*. Curitiba, PR: RBIE, 2012. p. 158–167.

PAVLUS, J. The game of life. *Scientific American*, Nature Publishing Group, v. 303, n. 6, p. 43–44, 2010.

POPOV, B. et al. Kim-a semantic platform for information extraction and retrieval. *Natural language engineering*, Cambridge Univ Press, v. 10, n. 3-4, p. 375–392, 2004.

SHAPIRO, S. S.; FRANCIA, R. An approximate analysis of variance test for normality. *Journal of the American Statistical Association*, Taylor & Francis Group, v. 67, n. 337, p. 215–216, 1972.

SHIRAZI, J. *Java performance tuning*. 2. ed. Cambridge: O'Reilly Media, Inc., 2003.

SILVA, A. P. da et al. An ontology-based model for driving the building of software product lines in an its context. In: DICHEVA, D.; MARKOV, Z.; STEFANOVA, E. (Ed.). *Third International Conference on Software, Services and Semantic Technologies S3T 2011*. Springer Berlin Heidelberg, 2011, (Advances in Intelligent and Soft Computing, v. 101). p. 155–159. ISBN 978-3-642-23162-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-23163-6_22>.

SILVA, A. P. da et al. Semantic web-based software product line for building intelligent tutoring systems. In: *Proceedings of the Software, Services, and Semantic Technologies: S3T*. Varna, Bulgaria: S3T 2010, 2010. p. 127–136.

SIRIN, E. et al. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, Elsevier, v. 5, n. 2, p. 51–53, 2007.

SZEKELY, B.; BETZ, J. *Jastor: Typesafe, ontology driven rdf access from java*. 2006. Jastor Web Site.

TSARKOV, D.; HORROCKS, I. Fact++ description logic reasoner: System description. In: FURBACH, U.; SHANKAR, N. (Ed.). *Automated Reasoning*. Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4130). p. 292–297. ISBN 978-3-540-37187-8. Disponível em: <http://dx.doi.org/10.1007/11814771_26>.

WADIA, Z. et al. *The Definitive Guide to Apache MyFaces and Facelets*. 1. ed. New York, US: Springer, 2008.

WELCH, B. L. The significance of the difference between two means when the population variances are unequal. *Biometrika*, v. 29, n. 3-4, p. 350–362, 1938. Disponível em: <http://biomet.oxfordjournals.org/content/29/3-4/350.short>.

WENZEL, K. Komma: An application framework for ontology-based software systems. *Semantic Web–Interoperability, Usability, Applicability*, v. 0, n. 1, p. 1–10, 2010.

WILCOXON, F. Individual comparisons by ranking methods. *Biometrics bulletin*, JSTOR, v. 1, n. 6, p. 80–83, Dezembro 1945.

WITTEN, I. H.; FRANK, E. *Data Mining: Practical machine learning tools and techniques*. 2. ed. San Franciso, CA, US: Elsevier, 2005.

WOHLIN, C. et al. *Experimentation in Software Engineering*. 1. ed. New York: Springer-Verlag, 2012. ISBN 0-7923-8682-5.

ZIMMERMANN, M. *Owl2Java-A Java Code Generator for OWL*. 2010.