UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

OPERADORES DE PROJEÇÃO E SELEÇÃO PARA A LINGUAGEM DE CONSULTA DO OBINJECT

João Francisco Campos do Amaral Rennó Alckmin

UNIFEI
Itajubá
Agosto, 2017

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

João Francisco Campos do Amaral Rennó Alckmin

OPERADORES DE PROJEÇÃO E SELEÇÃO PARA A LINGUAGEM DE CONSULTA DO OBINJECT

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação.

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

UNIFEI
Itajubá
Agosto, 2017

Dados Internacionais de Catalogação na Publicação (CIP)

Campos do Amaral Rennó Alckmin, João Francisco.
A346p Operadores de Projeção e Seleção para a Linguagem de Consulta do ObInject / João Francisco Campos do Amaral Rennó Alckmin. – Itajubá: UNIFEI, 2017.
xvii + 89 f. : il. ; 29.7

Orientador: Enzo Seraphim.
Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2017.
1. ObInject. 2. Consulta. 3. Persistência. 4. Indexação. 5. Framework. I. Seraphim, Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

CDU 004.65

Exemplar correspondente à versão final do texto, após a realização da defesa, incluindo, portanto, as devidas considerações dos examinadores.

Typeset in pdfLaTeX

Folha de Aprovação

Dissertação **aprovada** pela Banca Examinadora em 1º de agosto de 2017, conferindo ao autor o título de **Mestre em Ciência e Tecnologia da Computação**.

Dr. Enzo Seraphim _____

Orientador

Universidade Federal de Itajubá

Dr. Edmilson Marmo Moreira _____

Membro da Banca

Universidade Federal de Itajubá

Dr. Marcelo Zanchetta do Nascimento _____

Membro da Banca

Universidade Federal de Uberlândia

Meu objetivo é dar o máximo e o melhor de mim.
Minha motivação é a busca da perfeição, a busca
de me aperfeiçoar sempre, de aprender sempre.

Ayrton Senna da Silva

Á memória de meu pai

Agradecimentos

Agradeço primeiramente *Àquele*, onipotente e onipresente, que me ampara, me guia e protege em meio à imensidão de dúvidas e incertezas que fazem parte da minha existência. *Àquele* que me dá forças para enfrentar o abatimento que por vezes me assola. *Àquele* que jamais duvidou de mim, mesmo quando eu duvidava. Que sempre acreditou, quando eu mesmo não acreditava. Que sempre insistiu, apesar de minhas incontáveis desistências. *Àquele* que me faz ter a certeza de que tudo é possível. Deixo aqui registrado meus mais sinceros agradecimentos por me ajudar a me tornar quem sou e, principalmente, por me fazer entender que a imperfeição e a volatilidade da vida são justamente o que a torna perfeita.

À minha mãe, *Maria Christina Campos do Amaral Rennó Alckmin*. Raríssimas são as pessoas com as quais podemos contar por toda a vida, e você é uma delas. Apesar de possuirmos personalidades completamente opostas, posso dizer com propriedade que minha índole veio de berço. Hoje conseguimos nos enxergar como dois humanos providos de qualidades e defeitos, como qualquer outro. Tal fato só me faz te admirar ainda mais, pois, apesar de todas as adversidades, você foi capaz de criar dois filhos se esforçando ao máximo para acertar. Obrigado por tudo, eu te amo.

À minha família, em especial à minha avó *Maria de Lourdes Ribeiro Capistrano de Alckmin*, cujo amor incondicional pelo autor muitas vezes ultrapassa os limites da razão. A senhora sempre foi uma segunda mãe para mim, fazendo tudo ao seu alcance para me ver feliz. Nossos encontros mensais, por mais breves que sejam, sempre renovam minhas energias. Eu lhe agradeço pelo amor doado ao longo de todos estes anos, e espero que a conclusão deste trabalho lhe traga muitas alegrias.

À memória de meu pai, *Carlos Alberto Capistrano de Alckmin*, e meu avô, *Paulo Capistrano de Alckmin*. Por vezes lamento a brevidade de nosso convívio, mas encontro consolo nas vívidas lembranças presentes em minha memória. Apesar de fisicamente separados, sinto vocês cada vez mais próximos. Sei que estão torcendo por mim, como sempre o fizeram em vida. Amo vocês.

À minha namorada, *Andreza de Oliveira Souza*, pela ajuda em cada aspecto da composição deste trabalho, desde o lado técnico ao lado emocional. Seu sorriso e otimismo me forneceram a energia necessária para a conclusão de mais esta etapa na minha formação acadêmica. Sua preocupação durante todo este ciclo superou por vezes a minha. Sua inquietação me permitiu dar o melhor de mim. Você foi e continua sendo meu alicerce nesta jornada denominada *vida*. Obrigado pela companheira fantástica que é, te amo.

Ao meu orientador e amigo, *Prof. Dr. Enzo Seraphim*. Um educador e ser humano fantástico, como poucos neste mundo. Aprendi muito com você ao longo desta nossa breve, mas marcante convivência, mais até que com o próprio trabalho desenvolvido. Nossas conversas, desde as mais técnicas às mais pessoais, agregaram muito em minha vida. Não tenho palavras para expressar minha admiração à pessoa íntegra, honesta, compreensiva, serena, inteligente, humilde e verdadeira que você sempre demonstrou ser. Espero que nossa amizade permaneça forte e que possamos trabalhar juntos novamente em um futuro não muito distante, seria uma honra para mim. Muito obrigado por tudo.

À amiga *Profª. Drª. Thatyana de Faria Piola Seraphim*, pelo carinho e atenção que sempre demonstrou, desde a época de graduação. Por me acolher em sua casa, fora do horário do expediente, por incontáveis vezes, para que fosse possível concluir este trabalho. Obrigado por ser esta pessoa gentil, bondosa e amiga. Também agradeço aos pequenos *Miguel* e *Raphael* pela paciência e por me proporcionarem tantos momentos de alegria.

Ao amigo *Prof. Dr. Edmilson Marmo Moreira*, um profissional exemplar, apaixonado pelo que faz. Nosso convívio teve início na graduação, e muitas das disciplinas ministradas por você foram essenciais para o desenvolvimento deste trabalho, em especial a disciplina de Matemática Discreta. Em 2011 tive o privilégio de participar da Maratona de Programação, e com isso pude presenciar de perto sua dedicação pela computação. Nossa viagem até Passos para participar da etapa regional continua sendo uma das recordações mais marcantes da minha passagem pela UNIFEI. Minha participação no programa Ciência sem Fronteiras só foi possível graças à sua carta de recomendação, e até mesmo meu ingresso no programa de Mestrado teve grande influência de seu curso preparatório ministrado durantes as férias. Obrigado por ter sido tão essencial na minha vida acadêmica como professor, coordenador, educador e amigo.

Aos meus amigos, dentro e fora da sala de aula, os quais contribuíram para que eu me tornasse a pessoa e o profissional que sou hoje. Não citarei nomes, pois são numerosos e não quero cometer injustiças, mas saibam que vocês foram fundamentais durante toda esta minha caminhada, cada um à sua maneira. Obrigado pelo companheirismo, incentivo, paciência, apoio, pelas risadas compartilhadas e os problemas divididos, pelo imenso aprendizado e por todos os momentos partilhados durante estes anos todos. Levo um pouco de cada um comigo, onde quer que eu esteja.

Aos membros da *Banca Examinadora*. Agradeço por disporem de seu tempo a fim de contribuir com a realização do trabalho proposto através de valorosos comentários e sugestões.

Por fim, agradeço a todos que, embora não citados nominalmente, colaboraram direta ou indiretamente com o desenvolvimento deste trabalho.

Resumo

Os avanços tecnológicos vêm tornando os sistemas de armazenamento de informações cada vez mais necessários. Conforme os sistemas evoluem e se tornam mais complexos, o número de objetos que precisam ser instanciados e persistidos aumenta consideravelmente. O *framework* ObInject surgiu a partir desta necessidade, realizando a persistência e indexação de uma larga quantidade de objetos de maneira transparente à aplicação com um desempenho superior aos *frameworks* comumente utilizados para este fim. A persistência dos objetos de uma aplicação Java de forma nativa sem a necessidade de mapeá-los em tabelas proporciona ao ObInject uma clara vantagem nos quesitos desempenho e escalabilidade em relação aos *frameworks* de persistência que fazem uso da técnica de mapeamento objeto-relacional, os chamados *frameworks* ORM. Contudo, o grau de usabilidade do *framework* ObInject era insuficiente para tornar sua adoção em massa uma realidade devido à complexidade inerente à manipulação de suas estruturas de dados, até que o surgimento da linguagem de consulta do ObInject (OIQL) tornou as operações de consulta mais acessíveis e intuitivas. A linguagem OIQL é uma linguagem de consulta baseada em métodos, porém muitos destes métodos ainda não foram implementados ou possuem uma implementação parcial apenas. Este trabalho tem por objetivo principal implementar os operadores de projeção e seleção através dos métodos da linguagem *select* e *where*, respectivamente. Objetivos secundários incluem a implementação dos operadores lógicos de conjunção e disjunção e a melhoria da estrutura interna do *framework* ObInject. Os experimentos realizados ao final do trabalho validaram as mudanças na estrutura interna do *framework* e em sua linguagem de consulta, confirmando um desempenho superior em relação às alternativas.

Palavras-chave: ObInject, Consulta, Persistência, Indexação, *Framework*.

Abstract

Projection and Selection Operators for ObInject Query Language

The technological advancements are making information storage systems increasingly necessary. As the systems evolve and become more complex, the number of objects that need to be instantiated and persisted increases considerably. The ObInject framework emerged because of this necessity, performing the persistence and indexing of a large number of objects in a transparent way for the application with a higher performance than the frameworks normally used for this end. The persistence of objects from a Java application natively without the need to map them in tables gives ObInject a clear advantage in the performance and scalability categories when compared to persistence frameworks that use the object-relational mapping technique, the so-called ORM frameworks. However, the ObInject framework degree of usability was insufficient to make his mass adoption a reality thanks to the inherent complexity of manipulating his data structures, until the appearance of ObInject Query Language (OIQL) made query operations more accessible and intuitive. The OIQL language is a query language based on methods, yet many of this methods are still unimplemented or are only partially implemented. The main objective of this work is to implement the projection and selection operators using the language methods select and where, respectively. Minor objectives include the implementation of conjunction and disjunction logic operators and the improvement of ObInject framework internal structure. The experiments performed at the end of the work validated the changes made to the framework internal structure and its query language, confirming a better performance than the alternatives.

Keywords: *ObInject, Query, Persistence, Indexing, Framework.*

Sumário

Lista de Figuras **xiii**

Lista de Tabelas **xiv**

Lista de Códigos **xv**

Abreviaturas e Siglas **xvii**

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivo 3

1.3 Organização do Trabalho 3

2 Revisão Bibliográfica **4**

2.1 Frameworks 4

2.1.1 Framework ObInject 5

2.1.1.1 Índices Primários e Secundários 5

2.1.1.2 Identificador Único Universal 5

2.1.1.3 Módulos 7

2.1.1.4 Metaprogramação 11

2.1.2 Framework Hibernate 14

2.1.2.1 Mapeamento Objeto-Relacional 14

2.1.2.2 Metaprogramação 15

2.1.3 Framework ActiveJDBC 18

2.1.3.1 Metaprogramação 19

2.2 Linguagens de Consulta 21

2.2.1 ObInject Query Language 21

2.2.2 Structured Query Language 23
2.2.3 Object Query Language 25
2.2.4 Hibernate Query Language 28
2.3 Operadores de Consulta 31

3 Operadores de Projeção e Seleção **33**
3.1 Aprimoramentos na ObInject Query Language 33
3.1.1 Operador de Projeção 35
3.1.2 Operador de Seleção 37
3.2 Aprimoramentos no Framework ObInject 39
3.3 Estudo de Caso das Contribuições Realizadas 41
3.4 Considerações Finais 49

4 Experimentos **50**
4.1 Metodologia 50
4.2 Resultados 53
4.2.1 Tempo 53
4.2.2 Memória 55
4.2.3 Processador 58
4.2.4 Disco 60
4.2.5 Medições Adicionais 61
4.3 Considerações Finais 63

5 Conclusão **64**
5.1 Trabalho Futuros 65

Referências Bibliográficas **67**

A Tabela de Funções da Linguagem HQL **70**

B Tabela de Elementos do Esquema XML do Framework Hibernate **71**

C Classe de Empacotamento de Persistência da Classe de Usuário Map 72

D Classe de Empacotamento de Persistência da Classe de Usuário Terrain 78

Lista de Figuras

2.1 Hierarquia de Módulos do *Framework* ObInject 7

2.2 Hierarquia de Serialização 8

2.3 Hierarquia do Módulo de Armazenamento 9

2.4 Estrutura do Bloco 9

2.5 Hierarquia da Classe Node 10

2.6 Hierarquia do Módulo de Dispositivos 11

2.7 Metaprogramação no *Framework* ObInject 12

2.8 Mapeamento Objeto-Relacional do *Framework* Hibernate 14

3.1 Diagrama UML das Classes de Consulta 34

3.2 Diagrama UML das Entidades Geradas 44

4.1 Diagrama UML do Sistema Eleitoral Americano 51

4.2 Tempo de Inserção dos *Frameworks* 54

4.3 Tempo de Consulta dos *Frameworks* 54

4.4 Consumo Médio de Memória dos *Frameworks* na Inserção 56

4.5 Consumo Médio de Memória dos *Frameworks* na Consulta 56

4.6 Consumo Máximo de Memória dos *Frameworks* na Inserção 57

4.7 Consumo Máximo de Memória dos *Frameworks* na Consulta 57

4.8 Consumo Médio do Processador pelos *Frameworks* na Inserção 59

4.9 Consumo Médio do Processador pelos *Frameworks* na Consulta 59

4.10 Espaço em Disco Ocupado pelos Objetos Persistidos 60

4.11 Média de Acessos a Disco do ObInject 62

4.12 Média de Verificações do ObInject 62

Lista de Tabelas

2.1 Anotações do ObInject 13

2.2 Anotações do Hibernate 17

2.3 Anotações do ActiveJDBC 20

2.4 Precedência dos Operadores em HQL 29

4.1 Distribuição dos Objetos para Inserção 51

4.2 Distribuição dos Objetos para Consulta 52

A.1 Funções da Linguagem HQL 70

B.1 Elementos do Esquema XML do *Framework* Hibernate 71

Lista de Códigos

2.1 Anotação @Persistent 12
2.2 Classe de Usuário com Anotações do ObInject 13
2.3 Mapeamento da Classe de Usuário via XML 15
2.4 Classe de Usuário com Anotações JPA 18
2.5 Classe de Usuário no Padrão Active Record 19
2.6 Criação de um Registro no ActiveJDBC 19
2.7 Busca de um Registro no ActiveJDBC 19
2.8 Uso do Método *from* em OIQL 22
2.9 Uso do Método *select* em OIQL 22
2.10 Uso do Método *where* em OIQL 22
2.11 Uso do método *groupBy* em OIQL 22
2.12 Uso do Método *orderBy* em OIQL 23
2.13 Uso do Método *having* em OIQL 23
2.14 Exemplo de Consulta SQL 24
2.15 Consulta OQL com Retorno `bag` 26
2.16 Consulta OQL com Retorno `set` 26
2.17 Consulta OQL com Retorno `list` 26
2.18 Consulta OQL com Operador de Conjunto 26
2.19 Consulta OQL com Quantificador 27
2.20 Inserção em OQL via Função Construtora 27
2.21 Consulta em HQL 28
2.22 Consulta Resultante em SQL 28
2.23 Uso da Cláusula SELECT em HQL 28
2.24 Uso da cláusula WHERE em HQL 29
2.25 Uso do Operador IS NULL em HQL 29
2.26 Uso dos Operadores de Coleção em HQL 30
2.27 Uso da Cláusula ORDER BY em HQL 30
2.28 Uso da Cláusula GROUP BY em HQL 30
2.29 Uso de Funções em HQL 30
3.1 Implementação do Operador de Projeção 36
3.2 Implementação do Operador de Seleção 38
3.3 Implementação dos Operadores de Conjunção e Disjunção 39
3.4 Inicialização Tardia de uma Associação de Multiplicidade Um 40
3.5 Inicialização Tardia de uma Associação Múltipla 40
3.6 Classe *Map* 41
3.7 Classe *Terrain* 42

3.8 Persistência de Objetos pelo Método *inject* 45
3.9 Atualização de Objetos Persistentes pelo Método *inject* 46
3.10 Exclusão de Objetos Persistentes pelo Método *reject* 47
3.11 Consulta aos Objetos Persistidos pela Linguagem OIQL 48
C.1 Classe de Empacotamento de Persistência $Map 72
D.1 Classe de Empacotamento de Persistência $Terrain 78

Abreviaturas e Siglas

API	–	Application Programming Interface
CoC	–	Convension over Configuration
DTD	–	Document Type Definition
GC	–	Garbage Collector
IDE	–	Integrated Development Environment
JDK	–	Java Development Kit
JPA	–	Java Persistence API
MAC	–	Media Access Control
MBR	–	Minimum Bounding Rectangle
ODMG	–	Object Data Management Group
OID	–	Object Identifier
OIQL	–	ObInject Query Language
OQL	–	Object Query Language
ORM	–	Object-Relational Mapping
POO	–	Programação Orientada a Objetos
SGBD	–	Sistema de Gerenciamento de Banco de Dados
SGBDOO	–	Sistema de Gerenciamento de Banco de Dados Orientado a Objetos
SGBDR	–	Sistema de Gerenciamento de Banco de Dados Relacional
SQL	–	Structured Query Language
SSD	–	Solid-State Drive
UTC	–	Universal Time Coordinated
UUID	–	Universally Unique Identifier
XML	–	Extensible Markup Language

Capítulo 1

Introdução

O paradigma de *programação orientada a objetos* (POO) é um dos paradigmas mais utilizado da atualidade (TIOBE, 2017). Grande parte das aplicações desenvolvidas atualmente através desse modelo de *software* necessita que parte de seus dados permaneça disponível mesmo após o término da aplicação. Para tal, é necessário armazenar os dados em um meio não-volátil, como discos rígidos ou unidades de estado sólido. A esse tipo de armazenamento dá-se o nome de *persistência*.

Muitas das soluções de persistência de dados utilizadas pelas aplicações orientadas a objeto ainda seguem o *modelo relacional* (CODD, 1970; ASTRAHAN et al., 1976). Este modelo surgiu na década de 70, em uma época onde o paradigma de programação procedural era imensamente popular. No modelo relacional, os dados são persistidos em tabelas, sendo cada linha uma tupla e cada coluna um atributo. Tal modelo é incompatível com o *modelo orientado a objetos*, visto que a serialização completa de um objeto e seus relacionamentos não pode ser feita de forma trivial através de tabelas.

A incompatibilidade entre os modelos relacional e orientado a objetos foi ficando cada vez mais evidente na medida em que o paradigma ia ganhando popularidade. Muitos dos problemas gerados por essa incompatibilidade foram resolvidos através do emprego de uma técnica conhecida como *mapeamento objeto-relacional* (*Object-Relational Mapping* - ORM) (BARRY; STANIENDA, 1998).

Diversos *frameworks* ORM surgiram a partir dos anos 90 com o intuito de abstrair o mapeamento dos objetos da aplicação em registros no banco de dados e facilitar a consulta aos objetos persistidos através de uma linguagem de consulta mais próxima ao modelo orientado a objetos. Apesar de bem-sucedida, a técnica de mapeamento objeto-relacional possui suas desvantagens, sendo a maior delas o impacto negativo causado no desempenho da camada de persistência devido à utilização de um *middleware* entre o sistema de gerenciamento de banco de dados relacional (SGBDR) e a aplicação orientada a objetos.

Paralelamente ao surgimento dos bancos de dados baseados no modelo relacional, os chamados bancos de dados relacionais, pesquisas vinham sendo realizadas com o intuito de desenvolver bancos de dados compatíveis com o modelo orientado a objeto (MAIER et al., 1986; GARZA; KIM, 1988). Os primeiros bancos de dados orientados a objeto comerciais surgiram somente na década de 80, o que dificultou sua adoção, visto que os bancos relacionais já estavam bem consolidados no mercado e existiam grandes empresas por trás dessa tecnologia, como a Oracle®. A confiabilidade de um produto e o nível de suporte oferecido costumam ter bastante relevância no mundo corporativo, e os bancos de dados relacionais comerciais tinham ambas as variáveis a seu favor.

Outros modelos de persistência de dados existem, como o modelo *NoSQL* (*Not Only SQL*) (CHANG et al., 2006; DECANDIA et al., 2007). Este modelo abrange todos os bancos de dados onde os dados não são modelados através de tabelas, incluindo aqueles que seguem os modelos de persistência par chave-valor, tabular por colunas, orientado a documentos, orientados a grafos e orientado a objetos. Muitos dos bancos de dados baseados nestes modelos costumam oferecer um maior desempenho para certos domínios específicos quando comparados aos bancos relacionais.

Atualmente, os *frameworks* de persistência que não utilizam a técnica de mapeamento objeto-relacional vêm ganhando cada vez mais espaço. Entre eles se encontra o ObInject (CARVALHO, 2013), um *framework* que apresenta seu próprio esquema de definição e armazenamento de dados. A principal característica deste *framework* é permitir que quaisquer objetos sejam indexados por estruturas de dados que podem estar armazenadas em diferentes dispositivos.

1.1 Motivação

O *framework* ObInject (CARVALHO, 2013) faz uso de sua própria linguagem de consulta, intitulada *ObInject Query Language* (OIQL) (FERRO, 2012). Ela é uma linguagem de consulta baseada em métodos cuja sintaxe foi inspirada na linguagem SQL.

Apesar da linguagem OIQL possuir uma definição de sintaxe consistente, muitos de seus operadores estão apenas parcialmente implementados, dentre eles os operadores de projeção e seleção.

A implementação total do operador de projeção permitiria ao usuário da linguagem retornar apenas os atributos da consulta relevantes para sua aplicação. Já a implementação do operador de seleção forneceria ao usuário a possibilidade de filtrar os dados retornados de acordo com critérios específicos relevantes para sua aplicação.

1.2 Objetivo

Este trabalho tem por objetivo geral implementar os operadores de projeção e seleção da linguagem de consulta do ObInject (*ObInject Query Language* - OIQL).

Objetivos específicos do trabalho incluem a implementação dos operadores lógicos de conjunção e disjunção para a composição de múltiplas condições através do método *where* presente na linguagem OIQL e o aprimoramento das estruturas internas do *framework* ObInject.

A comparação de desempenho entre o *framework* ObInject e os *frameworks* Hibernate e ActiveJDBC, após realizadas as mudanças em sua estrutura interna e linguagem de consulta, é também um objetivo deste trabalho.

1.3 Organização do Trabalho

Este trabalho está organizado da seguinte maneira:

- **Capítulo 2 - Revisão Bibliográfica**
  Apresenta os conceitos necessários para a realização e o devido entendimento do trabalho, incluindo detalhes sobre os *frameworks* utilizados no trabalho e suas linguagens de consulta, além de conceitos sobre os operadores de consulta utilizados em sistemas de gerenciamento de banco de dados.

- **Capítulo 3 - Operadores de Projeção e Seleção**
  Descreve as contribuições alcançadas pelo trabalho, incluindo detalhes sobre a implementação dos operadores de projeção, seleção, conjunção e disjunção da linguagem de consulta do ObInject, além de detalhar as mudanças estruturais realizadas no *framework* ObInject.

- **Capítulo 4 - Experimentos**
  Detalha a realização dos experimentos realizados no trabalho, incluindo a metodologia utilizada e os resultados obtidos.

- **Capítulo 5 - Conclusão**
  Apresenta as conclusões inferidas através da realização do trabalho e sugere tópicos para possíveis trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Este capítulo apresenta os conceitos utilizados no desenvolvimento dos operadores de projeção e seleção da linguagem de consulta do ObInject, além de outros tópicos de relevância na implementação das diversas contribuições secundárias ao *framework* ObInject apresentadas no trabalho. Conceitos relevantes à realização dos experimentos também estão presentes neste capítulo.

A seção 2.1 explica o conceito de *framework* e discorre sobre os *frameworks* utilizados no desenvolvimento do trabalho: ObInject, Hibernate e ActiveJDBC. A seção 2.2 introduz as linguagens de consulta utilizadas ao longo do trabalho pelos diferentes *frameworks* apresentados. Por fim, a seção 2.3 detalha os principais operadores de consulta utilizados pelos sistemas de gerenciamento de banco de dados, dentre eles os operadores de projeção e seleção.

2.1 Frameworks

Frameworks fornecem uma solução estruturada para um conjunto de problemas através de funcionalidades já implementadas e fornecidas na arquitetura do *framework* e de componentes customizáveis de acordo com a necessidade e individualidade de cada aplicação (MATTSSON; BOSCH, 2000).

Segundo Fayad e Schmidt (1997), *frameworks* possuem duas classificações distintas, definidas conforme sua aplicação: *Frameworks* verticais e horizontais. *Frameworks* verticais são restritos a um domínio específico, enquanto que *frameworks* horizontais não possuem essa restrição, podendo ser utilizados em diversos domínios. Os *frameworks* citados neste trabalho são classificados como horizontais e usados para persistência e mapeamento de objetos.

2.1.1 Framework ObInject

O *framework* ObInject (CARVALHO, 2013) tem por objetivo fornecer uma solução estruturada para a persistência e indexação de objetos de uma aplicação. Seu desenvolvimento é feito utilizando a linguagem de programação Java.

2.1.1.1 Índices Primários e Secundários

O ObInject utiliza índices primários e secundários para realizar a persistência e indexação dos objetos, respectivamente. Um índice primário determina a localização de um registro completo em um arquivo de dados, enquanto que um índice secundário é qualquer outro índice que não seja um índice primário (RAMAKRISHNAN; GEHRKE, 1999; GARCIA-MOLINA et al., 2008).

No *framework* ObInject, um índice primário está associado ao objeto completo persistido, sendo indexado através de seu Identificador Único Universal, ou UUID (Seção 2.1.1.2). Por sua vez, um índice secundário está associado ao Identificador Único Universal do objeto, sendo indexado por seus atributos. A principal função de um índice secundário no contexto do *framework* é a de otimizar a busca de um objeto através de seus atributos.

A estrutura de dados utilizada pelos índices primários é uma árvore B+ modificada de forma a armazenar os registros completos em suas folhas. No caso dos índices secundários, as estruturas disponíveis são: árvore B+ (COMER, 1979), árvore R (GUTTMAN, 1984) e árvore M (CIACCIA et al., 1997). A árvore B+ é utilizada em domínios ordenáveis, a árvore R em domínios espaciais e a árvore M em domínios métricos.

2.1.1.2 Identificador Único Universal

Um Identificador Único Universal (*Universally Unique Identifier* - UUID) é um número de 128 *bits* utilizado para identificar informações em sistemas computacionais. Uma das formas mais comuns de se representar um UUID textualmente é através de um conjunto de 32 dígitos hexadecimais, divididos em cinco grupos separados por hífen, na forma 8-4-4-4-12, como, por exemplo, o UUID `FE7BDD81-5151-4761-AAF1-B67B5AE0217A`.

As especificações de um Identificador Único Universal são regidas pela norma ISO 9834-8 (ISO/IEC, 2008), tecnicamente equivalente à norma RFC 4122 (LEACH et al., 2005). Estas normas definem três variantes de UUID, denominadas variante 0, variante 1 e variante 2. Todas as variantes possuem 128 *bits*, porém a forma como os *bits* são interpretados é distinta entre elas. A variante 0 existe apenas para garantir a compatibilidade retroativa com formatos de UUID obsoletos. As variantes 1 e 2 são as utilizadas de fato nas especificações atuais de UUID, sendo ambas idênticas, exceto no que diz respeito à forma de armazenamento e

transmissão de seu formato binário. A variante 1 utiliza o formato *big-endian* e a variante 2 utiliza o *little-endian*.

As normas também definem cinco versões para as variantes 1 e 2. As versões definem o algoritmo de geração dos UUIDs a ser utilizado. As versões 1 e 2 são ditas *time-based*, pois utilizam uma marca temporal em seu algoritmo. As versões 3 e 5 são *name-based*, pois utilizam *hash* de cadeias de caracteres em seu algoritmo. A versão 4 é gerada de forma randômica, através de um gerador de números pseudo-aleatórios.

A versão 1 gera UUIDs utilizando o endereço MAC ou outro identificador de 48 *bits* do nó com uma marca temporal de 60 *bits* e uma sequência de *clock* de 13 ou 14 *bits*, medida em intervalos de 100 nanossegundos a partir da meia-noite do dia 15 de Outubro de 1582 UTC. Essa marca temporal é estendida por uma sequência de *clock* de 13 ou 14 *bits*, para lidar com casos em que o *clock* do processador não é rápido o bastante ou casos onde existam múltiplos processadores e geradores de UUID por nó. A principal característica dos UUIDs gerados por essa versão é a possibilidade de rastreamento de seu sistema computacional gerador.

A versão 2 é similar à versão 1, com exceção do fato de que os 8 *bits* menos significativos da sequência de *clock* são substituídos por um número que representa o domínio local, e os 32 *bits* menos significativos da marca temporal são substituídos por um identificador numérico que tenha algum significado para aquele determinado domínio local. A versão 2 produz um número significativamente menor de UUIDs que a versão 1 no mesmo intervalo de tempo, visto que os intervalos entre marcas temporais seriam de 429.49 segundos, bem maior que os 100 nanossegundos da versão 1. Por esse motivo, a versão 2 não é adequada para casos em que a taxa de geração de UUIDs por nó seja maior que 1 UUID a cada 7 segundos.

A versão 3 gera UUIDs utilizando um *hash* do identificador do *namespace* concatenado com um nome. Qualquer UUID pode ser usado como identificador do *namespace*, porém os identificadores de certos *namespaces* são fornecidos pela própria especificação, incluindo o *namespace* para identificadores de objetos (*Object Identifier* - OID). O algoritmo de *hash* utilizado é o MD5, sendo os *bits* correspondentes à identificação da versão e do variante substituídos no resultado do *hash*. A principal característica dos UUIDs gerados por essa versão é o fato de uma mesma combinação de nome e *namespace* produzir sempre o mesmo UUID.

A versão 5 é similar à versão 3, diferindo apenas no algoritmo de *hash* utilizado. Essa versão utiliza o algoritmo SHA1 truncado em 128 *bits*. A norma recomenda a utilização da versão 5 no lugar da versão 3.

A versão 4 gera UUIDs utilizando um gerador de números pseudoaleatórios. Os *bits* correspondentes à identificação da versão e do variante são pré-determinados, portanto, apenas 122 dos 128 *bits* são gerados de forma aleatória. É necessário garantir entropia o bastante para que os UUIDs produzidos sejam suficientemente pseudoaleatórios.

No *framework* ObInject, UUIDs são utilizados para identificar os objetos persistidos e suas classes de origem. Os UUIDs são gerados de acordo com a RFC 4122 (LEACH et al., 2005), utilizando a versão 4 da variante 1.

2.1.1.3 Módulos

O *framework* ObInject é conceitualmente dividido em 4 módulos, conforme a figura 2.1:

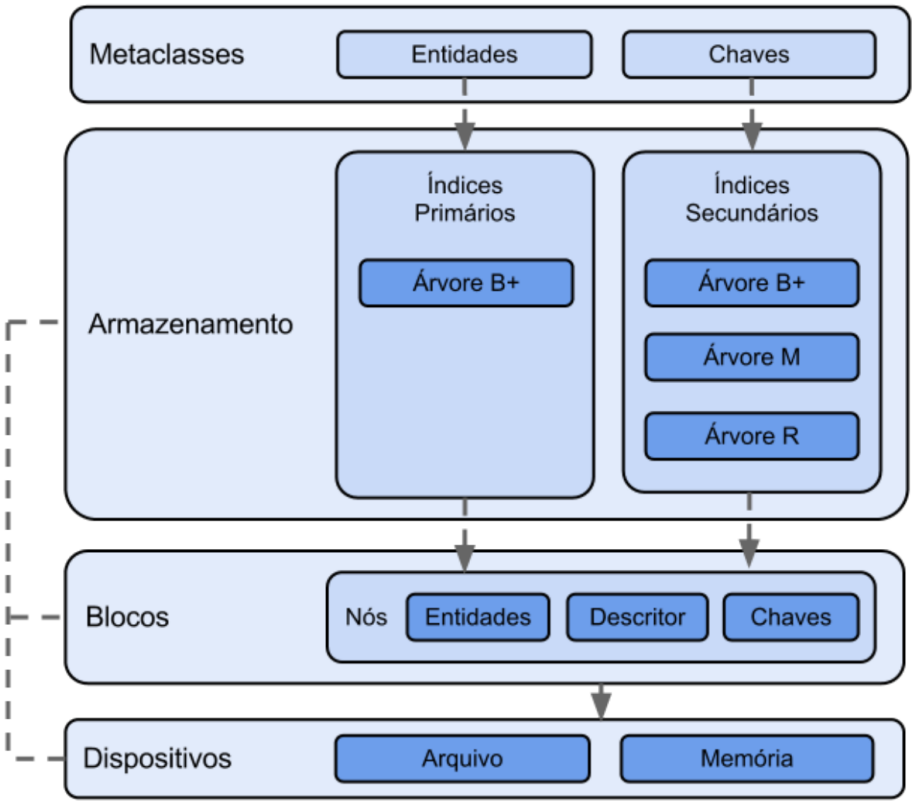

Figura 2.1: Hierarquia de Módulos do *Framework* ObInject
Fonte: (OLIVEIRA, 2012)

- **Metaclasses**: Define o vínculo entre aplicação e *framework* através de entidades (objetos persistentes), chaves (objetos indexáveis compostos por atributos de uma entidade), domínios de indexação e classes ajudantes de serialização e desserialização;
- **Armazenamento**: Define as estruturas de dados dos índices primários e secundários, responsáveis por indexar entidades e chaves, respectivamente;
- **Blocos**: Define as unidades de armazenamento para as estruturas de dados do *framework*;
- **Dispositivos**: Define abstrações para realizar a manipulação dos meios físicos de armazenamento.

Módulo de Metaclasses

O módulo de Metaclasses define o vínculo das classes de aplicação do usuário com o *framework* ObInject através de índices primários e secundários. Para realizar a indexação de entidades e chaves, a classe de usuário deve implementar as interfaces *Entity* e *Key*, respectivamente.

A implementação da interface *Entity* permite que as classes de usuário se tornem classes persistentes, também chamadas entidades. As entidades derivadas de *Entity* devem implementar as operações de serialização, desserialização e igualdade através dos métodos *pushEntity*, *pullEntity* e *isEqual*. As entidades devem ainda fornecer dois atributos do tipo *Uuid*: o atributo *uuid*, responsável por identificar uma instância da entidade de forma única e inequívoca, e o atributo estático *classId*, responsável por identificar a entidade à qual os dados retornados pertencem.

A implementação da interface *Key* deve realizar as operações de serialização e desserialização das chaves através da implementação dos métodos *pushKey* e *pullKey*. A forma como as chaves são indexadas ditam quais métodos adicionais devem ser implementados a fim de realizar as operações de comparação. Estes métodos são definidos por interfaces que estendem a interface *Key*, como a interface *Sort*, responsável por indexar os dados a partir de uma relação de ordem direta, e a interface *Metric*, responsável por indexar os dados através de uma medida de proximidade ou similaridade a partir de uma função de distância métrica.

Objetos persistentes e chaves indexadas são armazenados em índices primários e secundários, respectivamente, utilizando suas formas serializadas. Esta serialização é realizada através de duas classes derivadas da classe *Page*: *PushPage* e *PullPage*.

As especializações da classe *Page* fornecem as primitivas básicas para a serialização e desserialização dos objetos. A classe *PushPage* é utilizada para serializar o objeto, ou seja, transformar os valores de atributos deste objeto em um único vetor de *bytes*. Já a classe *PullPage* é utilizada para desserializar o objeto, que equivale a extrair do vetor de *bytes* as informações do objeto. A hierarquia destas classes é exibida pela figura 2.2.

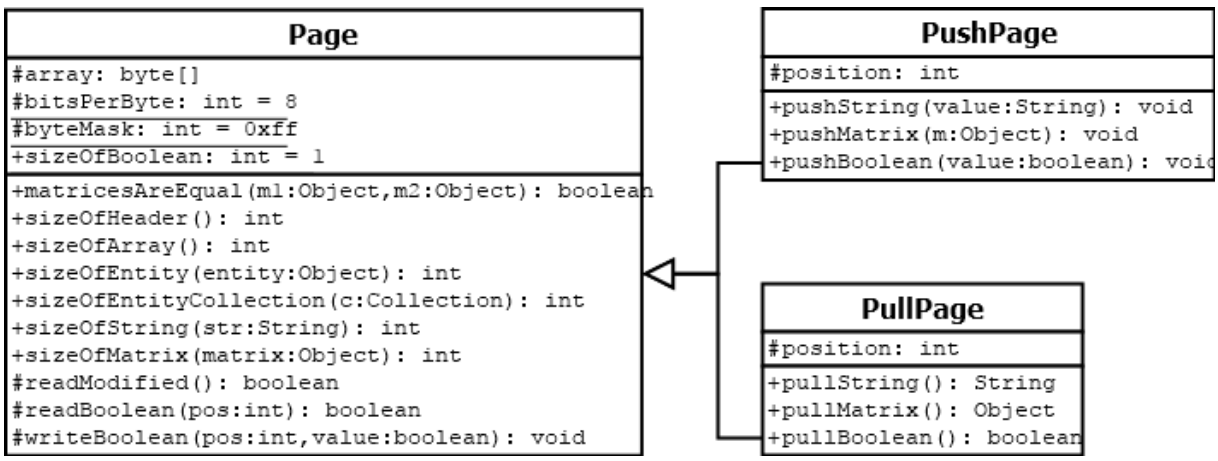

Figura 2.2: Hierarquia de Serialização

Módulo de Armazenamento

O módulo de Armazenamento especifica como as estruturas de índices primários e secundários são implementadas no *framework*. O modelo do módulo de Armazenamento é exemplificado na figura 2.3.

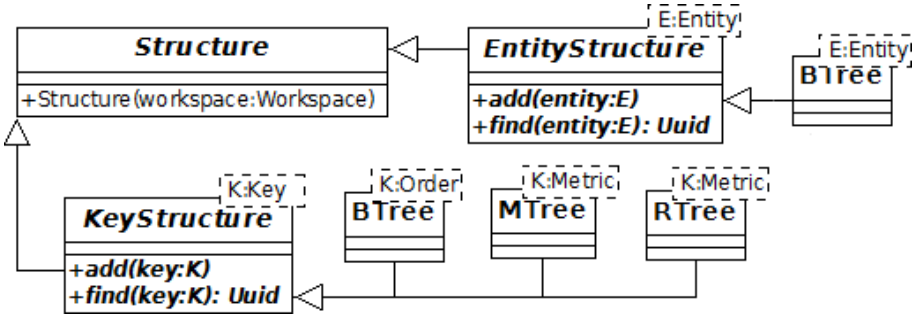

Figura 2.3: Hierarquia do Módulo de Armazenamento

A classe *Structure* depende da classe *Workspace* (Seção 2.1.1.3) para criar, recuperar e atualizar blocos. Assim sendo, para que uma *Structure* seja instanciada, é necessário que exista um *Workspace*. A classe *Structure* é especializada em *EntityStructure*, que é um índice primário e gerencia as entidades persistentes, e também em *KeyStructure*, que é um índice secundário, gerenciando as chaves indexadas.

A classe *EntityStructure* define as operações comuns às estruturas responsáveis pelo armazenamento dos objetos, como as operações de inserção, remoção e busca. Já a classe *KeyStructure* define as operações das estruturas de dados responsáveis pelo armazenamento dos índices secundários.

As estruturas de índices secundários são a árvore B+ (*BTree*), a árvore R (*RTree*) e a árvore M (*MTree*). Elas são as responsáveis por armazenar as chaves indexadas. As entidades persistentes são armazenadas em uma árvore B+ (*BTree*).

Módulo de Blocos

Um bloco é a unidade básica de transferência de dados entre as estruturas de dados presentes no *framework*, e é representado por um vetor de *bytes*. O tamanho do bloco é equivalente ao comprimento deste vetor, sendo variável conforme o tipo de *Workspace*. Cada bloco é montado de acordo com o esquema da figura 2.4.

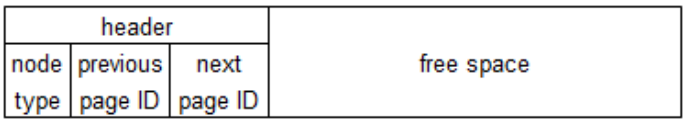

Figura 2.4: Estrutura do Bloco

O campo *header* contém os primeiros *bytes* do bloco, sendo tais blocos compostos de metadados definidos pelo *framework*. O campo *node type* é um valor utilizado para determinar de qual estrutura de dados o bloco faz parte. Os campos *previous page ID* e *next page ID* apontam para blocos vizinhos no mesmo nível, formando uma lista circular duplamente encadeada, a fim de facilitar a navegação entre os blocos. O campo de *bytes* livres é utilizado pelas estruturas de dados para alocar suas próprias informações.

O espaço livre de cada bloco, representado pelo termo *free space* na figura 2.4, é utilizado para armazenar as estruturas de dados, sendo organizado em três partes distintas:

- **features**: Contém dados gerais sobre o bloco;
- **entries**: Contém informações sobre os objetos contidos no bloco;
- **entities/keys**: Contém os objetos em si.

A classe *Node* encapsula um bloco e fornece os métodos para manipular o vetor de *bytes*, sendo utilizada para acessar e armazenar dados nos blocos. Diferentes estruturas de dados podem organizar um *Node* de diferentes formas. Para que as estruturas possam customizá-lo e identificar à qual estrutura um *Node* particular pertence, é necessário abstrair a classe *Node*, permitindo que ela possa ser especializada em *DescriptorNode*, *EntityNode* e *KeyNode*, como pode ser observado na figura 2.5.

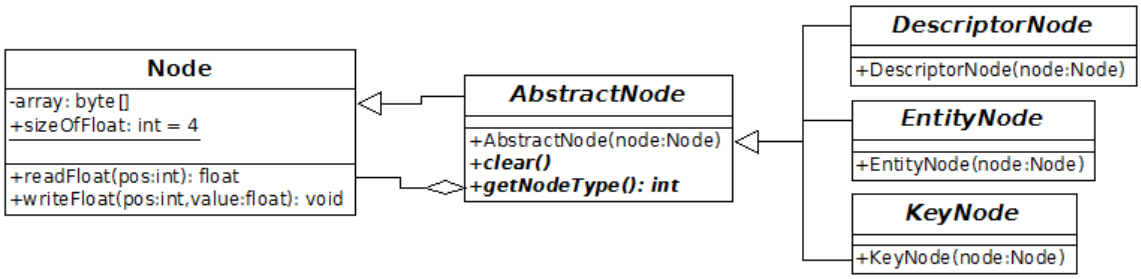

Figura 2.5: Hierarquia da Classe Node

Módulo de Dispositivos

O módulo de Dispositivos separa as estruturas de dados e as entidades persistentes das abstrações de armazenamento, ou seja, um objeto não sabe que existem dispositivos de armazenamento e desconhece o formato de dados usado quando é persistido.

Esta ideia permite que os dados sejam salvos em diferentes dispositivos, como, por exemplo, na memória ou em arquivo. A maneira mais conveniente de representar tais dispositivos é através de uma hierarquia de classes abstratas, como ilustrado na figura 2.6.

Na figura 2.6, a classe *AbstractWorkspace* é a interface comum da qual todos os tipos de dispositivos de armazenamento são especializados, e representa uma área virtual que pode ser usada pelas estruturas de dados para acessar e compartilhar dados.

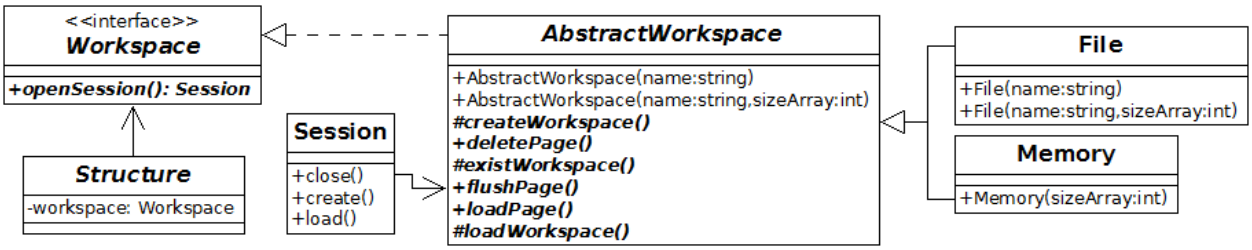

Figura 2.6: Hierarquia do Módulo de Dispositivos
Fonte: (CARVALHO, 2013)

A classe *AbstractWorkspace* é especializada nas classes *File* e *Memory*. A classe *File* manipula dados em um meio persistente, como o disco, e estes dados continuam salvos mesmo após o fim da aplicação. Já a classe *Memory* é responsável por dados voláteis, que são perdidos quando a aplicação termina.

A classe *Session* funciona como uma área temporária de salvamento, e mantém as páginas que foram utilizadas recentemente na memória para acelerar o acesso às mesmas. Ela é um intermediário entre as estruturas de dados e os métodos de acesso da classe *AbstractWorkspace*.

2.1.1.4 Metaprogramação

Metaprogramação é a criação de programas que escrevem ou manipulam outros programas utilizando a metalinguagem, os chamados metaprogramas. Os metaprogramas podem ser categorizados em compiladores e geradores (SHEARD, 2001), sendo os compiladores utilizados na tradução de códigos escritos em uma linguagem para outra linguagem distinta e os geradores utilizados na criação de programas que atendem soluções específicas.

O *framework* ObInject pode ser categorizado como sendo um metaprograma gerador, visto que gera outras classes com um escopo específico a partir dos metadados presentes nas classes de usuário. No contexto da metaprogramação, os metadados são dados que fornecem informações sobre outros dados. No caso do ObInject, os metadados são expressos através de anotações de classe que fornecem as informações necessárias ao *framework* para realizar a persistência das classes de usuário. A figura 2.7 ilustra o modelo de metaprogramação utilizado pelo *framework* ObInject.

A grande vantagem de se utilizar metaprogramação para auxiliar na persistência das classes é a redução no tamanho e na complexidade do código gerado pelo usuário do *framework*. Um código mais simples e compacto se traduz em um menor tempo de desenvolvimento da aplicação e, consequentemente, um menor custo de desenvolvimento. Algumas das desvantagens são a maior complexidade interna do *framework* em si e uma maior rigidez na forma de se realizar a persistência.

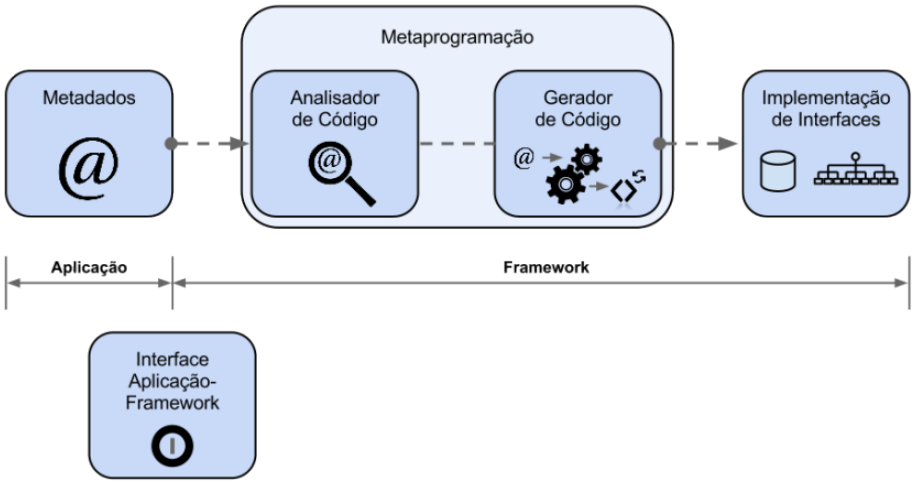

Figura 2.7: Metaprogramação no *Framework* ObInject
Fonte: (OLIVEIRA, 2012)

Na linguagem de programação Java, as anotações são parte integral da linguagem desde a versão 5. Anotações são um tipo especial de interface que estende implicitamente a interface *Annotation*, sendo declaradas através da palavra-chave `@interface`. As anotações não podem ser declaradas como tipo genérico, e sua declaração pode ocorrer em qualquer lugar onde uma declaração de interface é permitida.

As anotações possuem um nome e podem conter elementos, sendo eles os responsáveis por armazenar as informações pertencentes à anotação. Os elementos podem ter como retorno tipos primitivos, enumerados, anotações, classes, ou vetores de um dos tipos citados (ARNOLD et al., 2000). O código 2.1 define uma das classes de anotação do *framework*.

Código 2.1: Anotação @Persistent

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Persistent
{
    int blockSize() default 4096;
}
```

O *framework* ObInject utiliza apenas anotações proprietárias, pois possui uma filosofia bem diferente dos demais *frameworks* conformantes com a API de persistência Java (*Java Persistence API* - JPA) regulada pela especificação EJB 3.0 (JSR-220, 2012). As anotações de campo presentes no *framework* relativas aos domínios de indexação possibilitam a criação de múltiplos índices utilizando uma única anotação por domínio e a definição do posicionamento dos atributos na composição da chave de indexação através dos parâmetros *number* e *order*, respectivamente (CORREIA, 2017). As anotações presentes no *framework* estão listadas na tabela 2.1.

Anotação	Finalidade
`@Persistent`	Anotação de classe que define a classe anotada como sendo persistente
`@Unique`	Anotação de campo que define a chave de identificação primária do objeto, indexada através de uma árvore B+
`@Sort`	Anotação de campo que indexa um atributo ordenável utilizando uma árvore B+
`@Edition`	Anotação de campo que indexa um atributo que possui relação de semelhança utilizando uma árvore M
`@Point`	Anotação de campo que indexa um atributo que possui relação de semelhança no espaço euclidiano utilizando uma árvore M
`@Coordinate`	Anotação de campo que indexa um atributo que possui relação de semelhança no espaço esférico utilizando uma árvore M
`@Origin`	Anotação de campo que define um atributo numérico como sendo o ponto de origem no espaço euclidiano. O conjunto de atributos definidos pelas anotações `@Origin` e `@Extension` compõe a informação final que será indexada utilizando uma árvore R
`@Extension`	Anotação de campo que define um atributo numérico como sendo a distância a partir da origem no espaço euclidiano. O conjunto de atributos definidos pelas anotações `@Origin` e `@Extension` compõe a informação final que será indexada utilizando uma árvore R
`@Protein`	Anotação de campo que indexa um atributo que possui relação de semelhança entre proteínas utilizando uma árvore M
`@Feature`	Anotação de campo que indexa uma imagem utilizando uma árvore M

Tabela 2.1: Anotações do ObInject

O *framework* suporta o modelo de programação POJO (*Plain Old Java Object*) para as classes anotadas. Um POJO é um objeto Java que não implementa interfaces especiais e não chama nenhuma API do *framework* diretamente (RICHARDSON, 2006). Em outras palavras, o *framework* ObInject não impõe nenhuma restrição extra às classes de usuário além daquelas já definidas pela própria linguagem Java. Alguns dos benefícios de se utilizar POJO em conjunto com *frameworks* não-invasivos são uma maior separação de conceitos, maior portabilidade do código e desenvolvimento mais ágil. O código 2.2 ilustra uma classe de usuário anotada.

Código 2.2: Classe de Usuário com Anotações do ObInject

```
@Persistent
public class Bookstore {
    @Unique
    private String name;
    @Sort
    private String address;
    private List<Book> books;

    //gets() e sets()
}
```

2.1.2 Framework Hibernate

O *framework* Hibernate (BAUER; KING, 2004) tem por objetivo realizar a persistência e indexação de objetos de uma aplicação utilizando mapeamento objeto-relacional. Seu desenvolvimento é feito na linguagem de programação Java.

2.1.2.1 Mapeamento Objeto-Relacional

Mapeamento Objeto-Relacional (*Object-Relational Mapping* - ORM) é a persistência automática e transparente de objetos através de tabelas em um banco de dados relacional, utilizando metadados que descrevem o mapeamento entre os objetos e o banco de dados.

Algumas das vantagens desse modelo são: maior produtividade, visto que os códigos relacionados à persistência serão menos complexos de se implementar, maior manutenibilidade, já que menos linhas de código serão geradas e mantidas, e independência entre SGBDRs, pois o *framework* consegue lidar com diferentes tipos de bancos de dados relacionais de forma transparente à aplicação.

A grande desvantagem desse modelo quando comparado ao mapeamento manual de objetos em tabelas e uso de SQL nativo é o impacto no desempenho da aplicação. Porém, dependendo do tipo de aplicação, esse impacto pode ser negligível. Tal impacto se baseia no pressuposto de que a aplicação produzirá o mapeamento e as consultas mais otimizadas para sua situação, algo que muitas vezes não se verifica na prática.

A figura 2.8 ilustra o processo de mapeamento objeto-relacional efetuado pelo *framework* Hibernate do ponto de vista da aplicação do usuário.

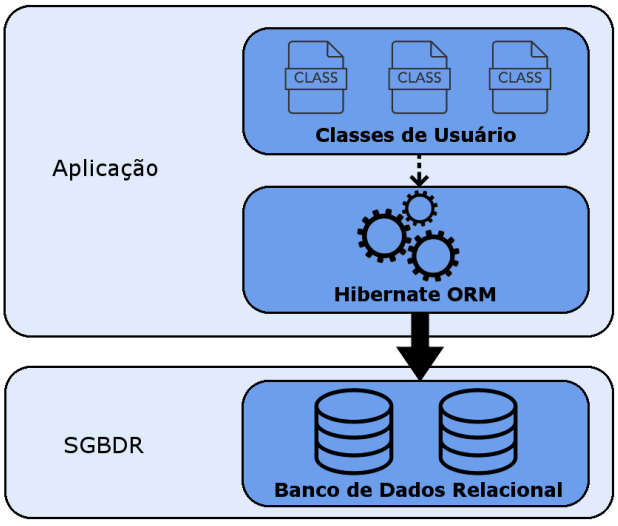

Figura 2.8: Mapeamento Objeto-Relacional do *Framework* Hibernate

2.1.2.2 Metaprogramação

O *framework* Hibernate faz uso de metaprogramação para gerar as classes de persistência a partir de metadados fornecidos pelo usuário do *framework*. Esses metadados especificam o mapeamento entre as classes de usuário e o banco de dados relacional, como, por exemplo, o mapeamento de classes em tabelas, propriedades em colunas e associações em chaves estrangeiras. Existem duas formas de se definir os metadados no Hibernate: arquivos XML e anotações.

Metadados via XML

A forma mais popular de definição de metadados nos *frameworks* de mapeamento objeto-relacional é através de arquivos XML (*Extensible Markup Language*). Tais arquivos são leves e facilmente manipulados através de editores de texto, seu formato é legível por humanos, podem ser customizados em tempo de implantação (ou mesmo em tempo de execução, através de geração programática de XML) e são compatíveis com sistemas de controle de versão.

A utilização de arquivos XML também traz suas desvantagens, entre elas a necessidade de gerenciar arquivos extras no projeto e o fato de muitos ambientes de desenvolvimento integrado (*Integrated Development Environment* - IDE) para Java não possuírem o mesmo tipo de suporte para arquivos XML, como a validação dos dados e a função de autocompletar.

O esquema XML especifica de maneira formal os elementos presentes em um documento XML utilizando restrições, normalmente expressas através de uma combinação de regras gramaticais que governam a ordem dos elementos, predicados booleanos, os quais devem ser satisfeitos pelo documento, tipos de dados que definem o conteúdo dos elementos e seus atributos, e regras mais especializadas como unicidade e integridade referencial. Ele pode ser utilizado para verificar se a sintaxe apresentada no documento XML está correta, ou seja, se os elementos presentes no documento e seus atributos aderem à especificação fornecida pelo esquema.

A tabela exposta no apêndice B lista os elementos mais comuns do esquema XML utilizado pelo *framework* Hibernate para definir os metadados.

O código 2.3 é um exemplo de como realizar o mapeamento de uma classe de usuário através de um arquivo XML. As informações presentes no arquivo XML indicam ao *framework* Hibernate que a classe *Message* deve ser persistida na tabela *MESSAGES* e sua chave primária deve ser gerada de forma automática e armazenada na coluna *MESSAGE_ID*. Além disso, a propriedade *text* deve ser mapeada pela coluna *MESSAGE_TEXT* e a associação com multiplicidade muitos-para-um definida na propriedade *nextMessage* deve ser mapeada através de uma chave estrangeira armazenada na coluna *NEXT_MESSAGE_ID*.

Código 2.3: Mapeamento da Classe de Usuário via XML

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class
name="msg.Message"
table="MESSAGES">
<id
name="id"
column="MESSAGE_ID">
<generator class="increment"/>
</id>
<property
name="text"
column="MESSAGE_TEXT"/>
<many-to-one
name="nextMessage"
cascade="all"
column="NEXT_MESSAGE_ID"
foreign-key="FK_NEXT_MESSAGE"/>
</class>
</hibernate-mapping>
```

Metadados via Anotações

As anotações são a forma recomendada de definição de metadados pelo *framework* Hibernate. A principal vantagem dessa técnica é a integração entre os metadados e os dados os quais eles representam, visto que ambos se encontram no mesmo arquivo. A integração entre os Ambientes de Desenvolvimento Integrado para Java e o conceito de anotações também pode ser considerada uma vantagem, visto que a função autocompletar e a validação de dados podem ser utilizadas tanto na aplicação como nas anotações.

As anotações utilizadas pelo *framework* são um superconjunto das anotações descritas na API de persistência Java (*Java Persistence API* - JPA) pela especificação EJB 3.0 (JSR-220, 2012). As anotações JPA fazem parte do Kit de Desenvolvimento Java (*Java Development Kit - JDK*) a partir da versão 5.0 e se encontram no pacote `javax.persistence`.

Algumas das anotações presentes no *framework* não se encontram padronizadas pela especificação JPA. O escopo de grande parte dessas anotações é a otimização de desempenho, como configurações de busca no banco e *caching* dos resultados. A grande desvantagem de se utilizar essas anotações é a questão da portabilidade, visto que elas são de uso exclusivo do Hibernate. As anotações mais utilizadas estão listadas na tabela 2.2.

Anotação	**Finalidade**
`@Entity`	Anotação de classe que define a classe anotada como uma entidade
`@Table`	Anotação de classe que especifica a tabela utilizada para mapear a entidade
`@Index`	Anotação de classe e campo que especifica os índices da tabela utilizada para mapear a entidade
`@Column`	Anotação de campo que especifica a coluna utilizada para mapear a propriedade
`@Id`	Anotação de campo que especifica a chave primária da entidade
`@GeneratedValue`	Anotação de campo que provê a estratégia utilizada para a geração da chave primária
`@Version`	Anotação de campo que especifica a versão da entidade
`@OrderBy`	Anotação de campo que define a ordenação dos elementos retornados de uma coleção
`@Lob`	Anotação de campo que especifica que a propriedade deve ser armazenada como sendo do tipo *large object*
`@Transient`	Anotação de campo que define uma propriedade como sendo não-persistente
`@OneToOne`	Anotação de campo que define uma associação um-para-um
`@ManyToOne`	Anotação de campo que define uma associação muitos-para-um
`@OneToMany`	Anotação de campo que define uma associação um-para-muitos
`@ManyToMany`	Anotação de campo que define uma associação muitos-para-muitos
`@JoinColumn`	Anotação de campo que especifica uma coluna para realizar a junção entre associações ou coleções de elementos
`@JoinTable`	Anotação de campo que especifica o lado proprietário da associação
`@Inheritance`	Anotação de classe que define a estratégia de herança utilizada para uma hierarquia de entidades. As estratégias disponíveis são: `SINGLE_TABLE`, `TABLE_PER_CLASS` e `JOINED`
`@Cache`	Anotação de classe que adiciona uma estratégia de *caching* para a entidade
`@Synchronize`	Anotação de classe que garante que as consultas não retornarão dados obsoletos
`@Fetch`	Anotação de campo que define a estratégia de busca utilizada para a associação
`@Generated`	Anotação de campo que especifica que a propriedade é gerada de forma automática pelo banco de dados
`@Immutable`	Anotação de classe e campo que especifica que a entidade ou coleção é imutável
`@NotFound`	Anotação de campo que define a ação a ser tomada caso o elemento buscado não seja encontrado na associação
`@Filter`	Anotação de classe e campo que permite adicionar um filtro a uma entidade ou associação
`@Formula`	Anotação de campo que define um fragmento SQL como fórmula para a resolução de uma coluna
`@OnDelete`	Anotação de classe e campo que define a estratégia a ser utilizada quando ocorre uma remoção nas coleções e vetores

Tabela 2.2: Anotações do Hibernate

O código 2.4 representa uma classe anotada utilizando as anotações JPA, definidas no pacote `javax.persistence`. Os objetos da classe *Bookstore* serão persistidos no banco de dados relacional através da tabela *bookstore*, sendo a chave primária composta pelo atributo *name*. A associação de multiplicidade muitos-para-muitos entre a classe *Bookstore* e a classe *Book* também será mapeada pelo *framework* através da anotação *ManyToMany*.

Código 2.4: Classe de Usuário com Anotações JPA

```
@Entity
@Table(name = "bookstore")
public class Bookstore {
    @Id
    private String name;
    @Index
    @Column(name="address")
    private String address;
    @ManyToMany
    private List<Book> books;

    //gets() e sets()
}
```

2.1.3 Framework ActiveJDBC

ActiveJDBC é um *framework* de mapeamento objeto-relacional desenvolvido por Igor Polevoy, cujo objetivo é realizar a persistência e indexação de objetos de uma aplicação.

Uma das motivações de Polevoy ao criar o ActiveJDBC foi o excesso de complexidade encontrado nos *frameworks* ORM mais populares na linguagem Java. A poluição das classes de usuário com métodos *getters* e *setters* também era um ponto que não o agradava. A solução foi criar um *framework* ORM minimalista baseado em uma série de convenções pré-estabelecidas que não necessita de configuração para ser utilizado.

O *framework* ActiveJDBC é uma implementação do padrão de projeto *Active Record* (FOWLER, 2002) na linguagem de programação Java. O padrão Active Record dita uma maneira de acessar os dados em um banco de dados relacional através do mapeamento de tabelas em classes. Um objeto de tais classes está ligado a uma única linha da tabela. Quando um objeto é criado, uma nova linha na tabela é adicionada ao salvar o objeto. Caso o objeto já exista na tabela, a linha correspondente a ele é atualizada. Associações entre tabelas feitas através de chaves estrangeiras são mapeadas instanciando as propriedades do objeto com informações do tipo apropriado.

O *framework* infere as propriedades de uma classe conformante com o padrão *Active Record* utilizando a tabela apropriada no banco de dados, baseado em sua convenção de nomes. A

classe associada à tabela *bookstores* é mostrada através do código 2.5.

Código 2.5: Classe de Usuário no Padrão Active Record

```
import org.javalite.activejdbc.Model;

public class Bookstore extends Model {}
```

O código 2.6 demonstra a criação de uma instância da classe *Bookstore* definida acima. O método *set* é responsável por atribuir valores aos atributos do objeto, enquanto que o método *save* persiste o objeto na tabela *bookstores*.

Código 2.6: Criação de um Registro no ActiveJDBC

```
Bookstore bs = new Bookstore();
bs.set("name", "ASD");
bs.save();
```

A busca do objeto persistido pelo código 2.6 é feita através do código 2.7. O método *where* aceita como parâmetro a parte da consulta em SQL relativa à cláusula *WHERE*, sendo o restante da consulta deduzido de forma implícita pelo *framework*. O retorno da consulta também é inferido a partir da mesma, sendo neste caso uma coleção de objetos da classe *Bookstore*. As propriedades do objeto podem ser acessadas individualmente através do método *get*.

Código 2.7: Busca de um Registro no ActiveJDBC

```
List<Bookstore> bookstores = Bookstore.where("name = 'ASD'");
Bookstore bs = bookstores.get(0);
String name = bs.get("name");
```

2.1.3.1 Metaprogramação

O *framework* ActiveJDBC segue o modelo de desenvolvimento de software intitulado Convenção sobre Configuração (*Convension over Configuration* - CoC). Este modelo foi criado por David Heinemeier Hansson para descrever uma das características de seu *framework*, Ruby On Rails (THOMAS et al., 2006).

O modelo de Convenção sobre Configuração tem por objetivo diminuir o número de decisões tomadas pelos usuários do *framework* ActiveJDBC, sem necessariamente perder a flexibilidade. O número de decisões é reduzido através da utilização de padrões sensíveis que cobrem a grande maioria dos casos de uso, enquanto a flexibilidade é mantida pelo uso de metaprogramação para substituir os padrões a partir de metadados fornecidos pelo usuário do *framework*. Tais metadados são fornecidos através de anotações de classe definidas pelo ActiveJDBC.

A principal consequência do modelo de desenvolvimento utilizado pelo *framework* ActiveJDBC é a necessidade de um menor número de anotações nas classes de usuário quando comparado a outros *frameworks* ORM que não seguem este modelo, como o Hibernate (Seção 2.1.2). A principal vantagem, além do ganho de produtividade gerado pela necessidade de se produzir um menor número de anotações, é a diminuição da curva de aprendizagem do *framework*, visto que muitos dos padrões utilizados são baseados em convenções intuitivas que os usuários costumam seguir de forma automática. Uma das desvantagens é a maior dificuldade de personalizar um determinado comportamento da aplicação, já que, nestes casos, é necessário conhecer o padrão implícito utilizado para modelar tal comportamento e a forma explícita de modificá-lo.

O *framework* ActiveJDBC utiliza apenas anotações proprietárias, não seguindo a convenção especificada pela API de persistência Java. As anotações disponibilizadas pelo *framework* estão listadas na tabela 2.3.

Anotação	**Finalidade**
`@Table`	Anotação de classe que define a tabela utilizada no mapeamento
`@DbName`	Anotação de classe que define o banco de dados utilizado no mapeamento
`@IdName`	Anotação de classe que define o nome da coluna responsável por armazenar a chave primária
`@IdGenerator`	Anotação de classe que permite utilizar uma sequência armazenada no SGBDR para gerar as chaves primárias de forma automática
`@CompositePK`	Anotação de classe que define uma chave primária composta de múltiplas colunas
`@Many2Many`	Anotação de classe que especifica as informações necessárias para mapear uma associação de multiplicidade muitos-para-muitos
`@BelongsTo`	Anotação de classe que especifica as informações necessárias para mapear uma associação de multiplicidade um-para-um
`@BelongsToParents`	Anotação de classe que especifica as informações necessárias para mapear uma associação de multiplicidade um-para-muitos
`@BelongsToPolymorphic`	Anotação de classe que define uma associação polimórfica entre classes
`@UnrelatedTo`	Anotação de campo que cancela uma associação criada de forma automática pelo *framework*
`@VersionColumn`	Anotação de classe que define o nome da coluna utilizada para armazenar a versão de um dado registro
`@Cached`	Anotação de classe que permite a utilização de *caching* nas consultas referentes à classe anotada

Tabela 2.3: Anotações do ActiveJDBC

2.2 Linguagens de Consulta

2.2.1 ObInject Query Language

A linguagem *ObInject Query Language* (OIQL) (FERRO, 2012) é uma linguagem de consulta baseada em métodos utilizada pelo *framework* ObInject (Seção 2.1.1) para realizar operações de consulta aos dados persistidos.

Uma linguagem de consulta baseada em métodos utiliza a sintaxe de chamada de função presente em grande parte das linguagens de programação para definir os operadores e parâmetros da consulta. No caso da linguagem OIQL, os parâmetros passados aos métodos das classes de consulta definem os parâmetros da consulta, enquanto que os métodos das classes em si definem os operadores. As classes responsáveis por realizar as consultas no *framework* ObInject estão presentes no pacote *queries*.

A principal vantagem de se utilizar uma linguagem de consulta baseada em métodos implementada em uma linguagem estaticamente tipada, como é o caso da linguagem de programação Java, é a segurança de tipos garantida em tempo de compilação. Tal característica garante uma maior confiabilidade nas consultas criadas. Algumas das desvantagens dessa abordagem são a maior verbosidade e a necessidade de identificar os tipos de forma explícita através de *casting* em alguns casos isolados onde o tipo não consegue ser inferido corretamente pela linguagem.

A consulta é definida pelo usuário do *framework* através dos métodos *select*, *from*, *where*, *groupBy*, *orderBy* e *having* presentes na classe *Query*. Os nomes dos métodos são propositalmente semelhantes aos nomes das cláusulas utilizadas pela linguagem de consulta SQL (Seção 2.2.2) para melhorar a usabilidade da linguagem, visto que muitos usuários estão familiarizados com a sintaxe da linguagem SQL.

Apesar das assinaturas dos métodos utilizados pela sintaxe da linguagem OIQL já estarem definidas, muitas de suas implementações se encontram incompletas. Portanto, os códigos de consulta utilizados nesta seção possuem como objetivo apenas ilustrar a utilização teórica dos métodos da linguagem, não devendo os códigos serem considerados plenamente funcionais.

Uma classe deve ser referenciada nas consultas através de sua entidade, ou seja, dada uma classe de nome *CLASS*, todas as referências a ela devem utilizar o nome *$CLASS*. Os atributos devem ser referenciados de forma direta, independentemente dos modificadores de acesso do atributo original, visto que os atributos da entidade são sempre públicos e possuem escopo de classe.

O método *from* define de quais entidades os dados serão recuperados. Este método é obrigatório em uma operação de consulta. O código 2.8 ilustra seu uso em uma consulta que retorna os livros cadastrados e imprime seus nomes e preços na tela.

Código 2.8: Uso do Método *from* em OIQL

```
Query q = new Query();
q.from($Book.class);
Collection<Book> books = q.execute();
```

O método *select* realiza a operação de projeção no conjunto retornado pela consulta. Em outras palavras, o método *select* define o retorno da consulta. Este método é opcional e, caso omitido, o retorno da consulta não sofre alterações. O código 2.9 faz uso do método *select* em uma consulta que retorna os nomes e preços dos livros e os imprime na tela.

Código 2.9: Uso do Método *select* em OIQL

```
Query q = new Query();
q.select($Book.name, $Book.price);
q.from($Book.class);
Collection<Object[]> results = q.execute();
```

O método *where* é um método opcional utilizado para restringir a consulta. Esse método recebe como parâmetro uma condição, que são instâncias das classes filhas de *Conditional*. A classe *Conditional* permite a concatenação de condições através dos métodos *and* e *or*. O código 2.10 utiliza o método *where* em conjunto com a condição *Equal* para realizar uma consulta que retorna os livros de um determinado autor e imprime o nome e o preço dos livros na tela.

Código 2.10: Uso do Método *where* em OIQL

```
Query q = new Query();
q.from($Book.class);
q.where(new Equal($Book.author, "ASD"));
Collection<Book> books = q.execute();
```

O método *groupBy* é utilizado para realizar o agrupamento dos resultados retornados pela consulta. Ele recebe como parâmetro um ou mais atributos da entidade e realiza o agrupamento de acordo com tais atributos. O código 2.11 faz uso do método *groupBy* para retornar os livros que custam 10 unidades monetárias agrupados por autor. Após realizada a consulta, o nome dos autores que possuem ao menos um livro que custa 10 unidades monetárias é impresso na tela.

Código 2.11: Uso do método *groupBy* em OIQL

```
Query q = new Query();
q.from($Book.class);
q.where(new Equal($Book.price, 10));
q.groupBy($Book.author);
Collection<Book> books = q.execute();
```

O método *orderBy* é responsável pela ordenação dos resultados de uma consulta. Este método recebe um ou mais atributos da entidade como parâmetro e realiza a ordenação de acordo com tais atributos. O código 2.12 ilustra seu uso em uma consulta que retorna os livros cadastrados ordenados pelo nome e imprime o nome e o autor dos livros na tela.

Código 2.12: Uso do Método *orderBy* em OIQL

```
Query q = new Query();
q.from($Book.class);
q.orderBy($Book.name);
Collection<Book> books = q.execute();
```

O método *having* restringe o resultado de um agrupamento e, por esse motivo, deve ser utilizado em conjunto com o método *groupBy*. Assim como o método *where*, o método *groupBy* recebe como parâmetro uma condição. Concatenações de múltiplas condições são permitidas através dos métodos *and* e *or* da classe *Conditional*. O código 2.13 utiliza o método *having* em conjunto com a condição *Equal* para realizar uma consulta que retorna os livros cadastrados agrupados por nome, cujo nome pertence a uma lista de três elementos. O nome dos livros pertencentes à lista é impresso na tela após a consulta.

Código 2.13: Uso do Método *having* em OIQL

```
Query q = new Query();
q.from($Book.class);
q.groupBy($Book.name);
q.having(new Equal($Book.name, "ASD").or(new Equal($Terrain.register, "ZXC
    ").or(new Equal($Terrain.register, "QWE"));
Collection<Book> books = q.execute();
```

2.2.2 Structured Query Language

A linguagem SQL, ou *Structured Query Language*, é uma linguagem declarativa utilizada para gerenciar dados em um Sistema de Gerenciamento de Banco de Dados Relacional (SGBDR). SQL é a linguagem de banco de dados mais utilizada atualmente, e influenciou no *design* de diversas outras linguagens.

SQL foi baseada nas ideias de Codd (CODD, 1970), utilizando diversos conceitos de álgebra relacional e cálculo relacional baseado em tuplas. Ela se tornou um padrão ANSI (*American National Standards Institute*) em 1986 e um padrão ISO (*International Organization for Standardization*) em 1987. Apesar de ser padronizada, muitos códigos escritos em SQL não são completamente portáveis entre diferentes sistemas de gerenciamento de banco de dados.

A linguagem SQL é composta por quatro tipos de linguagens: Linguagem de Definição de Dados, Linguagem de Manipulação de Dados, Linguagem de Controle de Dados e Linguagem

de Consulta de Dados. O escopo da linguagem incluiu operações de inserção, busca, atualização e remoção de dados, além de modificações no esquema de banco de dados e controle de acesso aos dados.

A Linguagem de Definição de Dados (*Data Definition Language*) é a responsável pela definição das estruturas de dados. Ela permite a criação, modificação e remoção de tabelas e índices, utilizando os comandos CREATE, ALTER e DROP.

A Linguagem de Manipulação de Dados (*Data Manipulation Language*) tem como função a inclusão, remoção e modificação de dados. Os comandos INSERT, UPDATE e DELETE fazem parte do dialeto dessa linguagem.

A Linguagem de Controle de Dados (*Data Control Language*) controla o acesso aos dados do banco. Ela é composta pelos comandos GRANT e REVOKE, sendo que o primeiro inclui e o segundo restringe permissões.

A Linguagem de Consulta de Dados (*Data Query Language*) é a linguagem mais utilizada e tem como função realizar a busca de dados no banco. Suas operações são não-destrutivas, isto é, não possuem efeitos persistentes no banco de dados. Esta linguagem é baseada no comando SELECT e as diversas cláusulas que o acompanham, como FROM, WHERE, HAVING, GROUP BY e ORDER BY.

O código 2.14 ilustra uma consulta utilizando SQL. Esta consulta tem por finalidade retornar os nomes dos livros cadastrados em uma determinada livraria, juntamente com o número de livros de mesmo nome escritos por autores distintos presentes naquela loja, desde que o número de livros de mesmo nome seja superior a um. Apenas os livros com preço de ao menos 10 unidades monetárias serão considerados na consulta. A cláusula SELECT especifica que a consulta retornará apenas os nomes dos livros e o número de livros de mesmo nome. A cláusula FROM especifica as tabelas que serão utilizadas. A cláusula WHERE relaciona as tabelas através das chaves primárias e especifica que apenas os livros de uma livraria específica com preço superior a 10 unidades monetárias serão considerados. A cláusula GROUP BY agrupa os livros por nome. A cláusula HAVING especifica que, após serem agrupados, apenas os livros com múltiplos autores serão contabilizados. Por fim, a cláusula ORDER BY ordena o resultado pelo nome dos livros.

Código 2.14: Exemplo de Consulta SQL

```
SELECT b.name, COUNT(b.author)
FROM Book b, Bookstore bs
WHERE bs.book_id = b.id AND bs.name = "ASD" AND b.price >= 10
GROUP BY b.name
HAVING COUNT(b.author) > 1
ORDER BY b.name;
```

2.2.3 Object Query Language

A linguagem OQL, ou *Object Query Language*, é uma linguagem de consulta utilizada para gerenciar dados em um Sistema de Gerenciamento de Banco de Dados Orientado a Objetos (SGBDOO). A principal diferença entre um SGBDR e um SGBDOO é a forma como a informação é representada: o primeiro representa os dados na forma de tabelas, enquanto que o segundo os representa na forma de objetos.

Essa linguagem foi desenvolvida pelo *Object Data Management Group* (ODMG), e se encontra atualmente na versão 3.0 (CATTELL; BARRY, 2000). A linguagem OQL é um superconjunto da linguagem SQL, agregando a ela conceitos provenientes do paradigma orientado a objetos, como polimorfismo, invocação de métodos e primitivas para lidar com conjuntos de objetos.

Não existe diferença na forma de se endereçar atributos, relações ou mesmo métodos através da linguagem OQL em relação às linguagens de programação orientadas a objeto. Todos estes componentes são acessados utilizando a notação de ponto ou a notação de seta, sendo ambas equivalentes. É possível formar expressões com múltiplos pontos, desde que nenhum dos elementos intermediários seja uma coleção.

OQL possui suporte para a manipulação de objetos mutáveis (registros que possuem um OID) e literais (tipos básicos como `string`, `integer`, `float`, `boolean`, `character`...). Além disso, é possível manipular estruturas compostas através do tipo `struct` e coleções através dos tipos `set`, `bag`, `list` e `array`.

O tipo do resultado de uma consulta em OQL é inferido a partir dos operadores que fazem parte da expressão de consulta. Os tipos de retorno possíveis são:

- **Objeto**: A consulta `ELEMENT(SELECT b FROM Books b WHERE b.name="ABC")` retorna um objeto do tipo Book;
- **Coleção de objetos**: A consulta `SELECT b FROM Books b WHERE b.category="ZXC"` retorna uma coleção de objetos do tipo Book;
- **Literal**: A consulta `ELEMENT(SELECT b.author FROM Books b WHERE b.name="ABC")` retorna uma *string*;
- **Coleção de literais**: A consulta `SELECT b.author FROM Books b WHERE b.category="ZXC"` retorna uma coleção de *strings*.

O resultado de uma consulta em OQL na forma *SELECT-FROM-WHERE* (SFW) é sempre uma coleção. O tipo padrão do resultado de um SFW é uma `bag`. O cógido 2.15 representa uma consulta com retorno padrão.

Código 2.15: Consulta OQL com Retorno `bag`

```
SELECT b.name, b.price
FROM Bookstore bs, bs.books b
WHERE b.price < 20.00
```

Nem sempre uma `bag` é o tipo mais apropriado, visto que ela pode conter elementos repetidos. É possível mudar o tipo de retorno para um `set`, bastando inserir a palavra-chave DISTINCT após a cláusula SELECT, conforme o código 2.16.

Código 2.16: Consulta OQL com Retorno `set`

```
SELECT DISTINCT b.name, b.price
FROM Bookstores bs, bs.books b
WHERE b.price < 20.00
```

Também é possível mudar o tipo de retorno para uma `list`, bastando utilizar a cláusula ORDER BY. A `list` é uma coleção ordenada, diferentemente das coleções `bag` e `set`. O cógido 2.17 utiliza a cláusula ORDER BY para retornar uma coleção ordenada do tipo *list*.

Código 2.17: Consulta OQL com Retorno `list`

```
SELECT b.name, b.price
FROM Bookstores bs, bs.books b
WHERE b.price < 20.00
ORDER BY b.price ASC
```

Os conjuntos `bag` e `set` podem ser manipulados através de operadores de conjunto. O tipo do resultado está diretamente relacionado ao tipo dos operandos — o resultado é uma `bag` se ao menos um dos operandos for também uma `bag`. Caso contrário, o resultado é um `set`. Os operadores de conjunto são listados abaixo, e seu uso é exemplificado através do código 2.18.

- **UNION**: Retorna a união dos conjuntos;
- **INTERSECT**: Retorna a intersecção dos conjuntos;
- **EXCEPT**: Retorna a diferença dos conjuntos.

Código 2.18: Consulta OQL com Operador de Conjunto

```
( SELECT b.name FROM Bookstores bs, bs.books b WHERE bs.name = "ABC" )
INTERSECT
( SELECT b.name FROM Bookstores bs, bs.books b WHERE bs.name = "XYZ" )
```

A linguagem OQL possui operadores chamados quantificadores, os quais permitem compor expressões booleanas que aplicam uma condição a todos os elementos de uma coleção. A lista a seguir descreve os operadores quantificadores, enquanto o código 2.19 apresenta o uso do operador EXISTS em uma consulta.

- **FOR ALL**: A expressão é verdadeira caso todos os elementos da coleção satisfaçam a condição. Do contrário, a expressão é falsa;
- **EXISTS**: A expressão é verdadeira caso ao menos um elemento da coleção satisfaça a condição. Do contrário, a expressão é falsa;
- **UNIQUE**: A expressão é verdadeira caso apenas um elemento da coleção satisfaça a condição. Do contrário, a expressão é falsa.

Código 2.19: Consulta OQL com Quantificador

```
SELECT bs.name
FROM Bookstores bs
WHERE EXISTS b IN bs.books : b.price < 20.00
```

Existe um valor literal especial para representar o acesso a uma propriedade de um objeto nulo na linguagem OQL: `UNDEFINED`. As regras de manipulação desse valor estão descritas a seguir:

- A expressão `UNDEFINED` sempre retorna verdadeiro;
- A expressão `UNDEFINED` sempre retorna falso;
- `UNDEFINED` é sempre tratado como falso caso seja retornado pela cláusula WHERE;
- Uma coleção pode conter `UNDEFINED`;
- `UNDEFINED` é uma expressão válida para a operação de agregação COUNT;
- Qualquer outra operação com qualquer operando `UNDEFINED` resulta em `UNDEFINED`.

OQL não provê operadores específicos para criar e atualizar os objetos persistidos. Para criar um objeto com um OID, uma função construtora deve ser utilizada, conforme ilustrado pelo código 2.20. Os parâmetros que não forem inicializados recebem um valor padrão.

Código 2.20: Inserção em OQL via Função Construtora

```
newBook = Book(name: "ABC", author: "ZXC")
```

2.2.4 Hibernate Query Language

A linguagem HQL, ou *Hibernate Query Language*, é uma linguagem de consulta desenvolvida para ser utilizada pelo *framework* Hibernate (Seção 2.1.2). A sintaxe da linguagem é muito semelhante à utilizada pela linguagem SQL, porém HQL é orientada a objetos e como tal possui suporte a polimorfismo, herança e associações. HQL influenciou diretamente a definição da linguagem de consulta JPQL, ou *Java Persistence Query Language*, sendo esta um subconjunto daquela.

Apesar da linguagem HQL ser orientada a objetos, o *framework* Hibernate utiliza bancos de dados relacionais. Por esse motivo, consultas escritas em HQL são traduzidas para SQL pelo *framework* antes de serem enviadas ao banco de dados. O código 2.21 apresenta uma consulta simples em HQL e o código 2.22 representa a consulta resultante gerada pelo *framework* na linguagem SQL.

Código 2.21: Consulta em HQL

```
from Book
```

⇒

Código 2.22: Consulta Resultante em SQL

```
select b.BOOK_ID, b.NAME, b.
    AUTHOR, ... from BOOK b
```

A cláusula SELECT é uma cláusula opcional que permite especificar com precisão quais objetos ou propriedades de objetos serão retornados pela consulta. Quando omitida, o retorno da consulta será definido de forma implícita através da cláusula FROM. Em ambos os casos, os elementos retornados pela consulta não possuem garantia de unicidade, isto é, o resultado da consulta pode conter elementos repetidos. Caso esse comportamento não seja o desejado, pode-se utilizar a palavra-chave DISTINCT logo após a cláusula SELECT para garantir apenas a existência de elementos únicos no resultado. O código 2.23 ilustra uma consulta em HQL que utiliza a cláusula SELECT para retornar apenas os atributos *name* e *author* dos objetos persistidos do tipo *Book*.

Código 2.23: Uso da Cláusula SELECT em HQL

```
SELECT b.name, b.author FROM Book b
```

Assim como em SQL, a cláusula WHERE pode ser utilizada seguida de expressões condicionais para restringir a quantidade de objetos retornados pela consulta. As expressões condicionais em HQL são avaliadas utilizando lógica ternária, podendo resultar em *verdadeiro*, *falso* ou *nulo*, em contraste com a lógica binária utilizada em SQL. Os seguintes operadores podem ser utilizados para compor uma expressão condicional válida na cláusula WHERE: operadores de sinalização, operadores aritméticos, operadores lógicos, operadores de comparação e operadores de coleção.

A precedência dos operadores em HQL e suas respectivas descrições estão representadas na tabela 2.4. Parênteses podem ser utilizados para agrupar expressões, enquanto que os operadores lógicos podem combiná-las.

Operador	Descrição
`.`	Operador de navegação
`+, -`	Operadores unários de sinalização positiva ou negativa
`*, /`	Operadores binários de multiplicação e divisão de valores numéricos
`+, -`	Operadores binários de adição e subtração de valores numéricos
`=, <>, <, >, >=, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL`	Operadores binários de comparação
`IS [NOT] EMPTY, [NOT] MEMBER [OF]`	Operadores binários para coleções
`NOT, AND, OR`	Operadores lógicos

Tabela 2.4: Precedência dos Operadores em HQL
Fonte: (BAUER, 2006)

O código 2.24 executa uma consulta na linguagem HQL utilizando a cláusula WHERE composta por um operador binário de comparação. Seu objetivo é retornar apenas os livros que possuam um preço maior ou igual a 10 unidades monetárias.

Código 2.24: Uso da cláusula WHERE em HQL

```
SELECT b.name FROM Book b WHERE b.price >= 10
```

Qualquer comparação em HQL na qual um dos operandos seja nulo resulta em nulo, portanto, não é possível testar a nulidade de um operando utilizando uma comparação simples da forma `OPERANDO=null`. Tal comparação deve ser realizada utilizando o operador IS NULL, conforme ilustrado pelo código 2.25.

Código 2.25: Uso do Operador IS NULL em HQL

```
SELECT b.name FROM Book b WHERE b.price IS NOT NULL
```

É possível utilizar coleções como operandos em uma cláusula WHERE, desde que o operador utilizado seja próprio para coleções. Tais operadores estão descritos a seguir:

- **IS EMPTY**: Operador unário que retorna *verdadeiro* caso a coleção esteja vazia. Do contrário, retorna *falso*;
- **MEMBER OF**: Operador binário que retorna *verdadeiro* caso o elemento faça parte da coleção. Do contrário, retorna *falso*.

A consulta executada pelo código 2.26 faz uso dos operadores de coleção descritos anteriormente para retornar o nome das livrarias que possuem um determinado livro.

Código 2.26: Uso dos Operadores de Coleção em HQL

```
SELECT bs.name FROM Bookstore bs, Book b WHERE bs.books IS NOT EMPTY AND b
    .name="ASD" AND b MEMBER OF bs.books
```

A ordenação dos resultados na linguagem HQL é feita através da cláusula ORDER BY, de forma similar ao SQL. É possível ordenar de forma ascendente ou descendente através das palavras-chave ASC e DESC. Também é possível especificar múltiplos atributos na ordenação, conforme ilustrado pelo código 2.27.

Código 2.27: Uso da Cláusula ORDER BY em HQL

```
FROM Book b ORDER BY b.name ASC, b.price DESC
```

A cláusula GROUP BY realiza o agrupamento dos resultados. Em uma consulta onde existe agrupamento, a cláusula WHERE é aplicada sobre os valores não agrupados e a cláusula HAVING é aplicada após o agrupamento ocorrer. O código 2.28 utiliza as cláusulas GROUP BY e HAVING para buscar o nome e o preço médio de cada um dos livros de um determinado autor, desde que o preço médio seja maior que 5.

Código 2.28: Uso da Cláusula GROUP BY em HQL

```
SELECT b.name, AVG(b.price) FROM Book b WHERE b.author="ASD" GROUP BY b.
    name HAVING AVG(b.price) > 5
```

Uma das características que mais se destacam na linguagem HQL é a possibilidade de invocar funções SQL nas cláusulas SELECT e WHERE. Caso o banco de dados utilizado permita a criação de funções, elas também podem ser utilizadas. O código 2.29 utiliza as funções COUNT() e SIZE() para realizar uma busca pelo número de livrarias com mais de 10 livros.

Código 2.29: Uso de Funções em HQL

```
SELECT COUNT(bs) FROM Bookstore bs WHERE SIZE(bs.books) > 10
```

Existe uma classe especial de funções, as chamadas funções de agregação. Tais funções são usadas na cláusula SELECT e permitem que os resultados sejam agrupados e transformados em um único valor. Por consequência, qualquer cláusula SELECT que possui uma função de agregação deve conter apenas funções de agregação na ausência da cláusula GROUP BY.

A tabela anexada ao apêndice A lista as funções nativas em HQL. A maioria das funções HQL são traduzidas para funções SQL pelo *framework*, sendo possível adicionar novas traduções através da classe `org.hibernate.Dialect`.

2.3 Operadores de Consulta

A execução de consultas em um Sistema de Gerenciamento de Banco de Dados, ou SGBD, é realizada através do processador de consultas. O processador de consultas é responsável por transformar as consultas e os comandos de modificação de dados fornecidos pelo usuário em uma sequência lógica de operações realizadas sobre o banco de dados e executá-las.

Consultas são construídas a partir de operadores de consulta, os quais representam ações elementares realizadas sobre conjuntos ou multiconjuntos. Ambos podem ser definidos como uma coleção de elementos onde a ordem não é relevante, porém apenas o multiconjunto permite a repetição de elementos. É possível combinar vários operadores de consulta em uma mesma expressão, aplicando um operador aos resultados de um ou mais operadores diferentes.

Os operadores da álgebra relacional foram usados como base para a construção dos operadores de consulta para bancos de dados propostos por Edgar Frank Codd (CODD, 1970). Como os operadores da álgebra relacional lidam apenas com conjuntos, foi necessário redefinir certos operadores para lidarem com multiconjuntos. Operadores que conseguem atuar sobre multiconjuntos também possuem suporte intrínseco a conjuntos, visto que o primeiro é a generalização do segundo. Codd também criou novos operadores para certas operações que não estão descritas na álgebra relacional.

O operador de união ($\cup$) é um operador de consulta que realiza a união entre multiconjuntos, produzindo um novo multiconjunto composto pelas tuplas que pertencem a pelo menos um destes multiconjuntos. Em $R \cup S$, uma tupla está no resultado tantas vezes quanto o número de vezes que ela se encontra em R mais o número de vezes que ela se encontra em S.

O operador de interseção ($\cap$) é um operador de consulta que atua sobre multiconjuntos, produzindo um novo multiconjunto que contém apenas as tuplas pertencentes a todos os multiconjuntos. Em $R \cap S$, uma tupla está no resultado tantas vezes quanto o número de vezes que ela se encontra em ambos R e S.

O operador de diferença ($-$) é um operador de consulta que calcula o complemento relativo entre dois multiconjuntos. Em outras palavras, a operação $R - S$ retorna um multiconjunto composto pelas tuplas de R que não se encontram em S. Em $R - S$, uma tupla está no resultado tantas vezes quanto o número de vezes que ela se encontra em R menos o número de vezes que se encontra em S. Valores negativos são considerados zero.

O operador de projeção (π) é um operador de consulta que atua sobre um multiconjunto, selecionando determinadas colunas de suas tuplas e produzindo um novo multiconjunto contendo tuplas compostas apenas pelas colunas selecionadas. A projeção $\pi_L(R)$ é a projeção da relação R sobre a lista de atributos L. O resultado da projeção é o multiconjunto composto pelas tuplas pertencentes a R com seus componentes compostos apenas pela lista de

atributos L. O esquema do resultado corresponde aos nomes dos atributos na lista L.

O operador de seleção (σ) é um operador de consulta que atua sobre um multiconjunto, selecionando determinadas linhas baseado em uma condição ou predicado e produzindo um novo multiconjunto como resultado. A seleção $\sigma_C(R)$ é composta por um multiconjunto R e uma condição C. A condição pode ser composta por expressões envolvendo operadores aritméticos, operadores de cadeias de caracteres e operadores booleanos (AND, OR e NOT). O resultado da seleção é o multiconjunto composto pelas tuplas pertencentes a R que satisfazem a condição C, sendo o esquema resultante igual ao de R.

O operador de produto de relações ($\times$) é um operador de consulta que realiza o produto cartesiano entre dois multiconjuntos R e S, sendo o resultado composto pelo multiconjunto de todos os pares ordenados cujo primeiro termo pertence a R e o segundo pertence a S. O esquema do produto $R \times S$ consiste nos atributos das tuplas de R e S.

Os operadores de junção são operadores de consulta construídos a partir de um produto cartesiano entre dois multiconjuntos seguido por uma operação de seleção e uma operação de projeção. Um dos operadores de junção é a junção natural ($\bowtie$). A junção natural dos multiconjuntos R e S é representada pela expressão $R \bowtie S$. Essa expressão é uma abreviação de $\pi_L(\sigma_C(R \times S))$, onde C é uma condição que compara todos os pares de atributos das tuplas de R e S que possuem o mesmo nome e L é uma lista de todos os atributos das tuplas de R e S, exceto pelo fato de uma cópia de cada par de atributos comparados ser omitida.

O operador de eliminação de duplicatas (δ) é um operador de consulta que converte um multiconjunto em um conjunto. Em outras palavras, a operação $\delta(R)$ retorna um conjunto composto por apenas uma cópia de cada tupla que aparece uma ou mais vezes em R.

O operador de agrupamento (γ) é um operador de consulta responsável por realizar operações de agrupamento em um multiconjunto R a partir de uma lista de atributos de agrupamento L. O multiconjunto retornado pela expressão $\gamma_L(R)$ é construído particionando-se as tuplas de R em grupos. Cada grupo consiste de todas as tuplas que possuem uma atribuição de valores específica para atributos de agrupamento na lista L. Caso não haja atributos de agrupamento, o próprio multiconjunto R será um grupo. Para cada grupo, se produz uma tupla consistindo nos valores dos atributos de agrupamento para esse grupo e nas agregações sobre todas as tuplas do grupo.

O operador de classificação (τ) é um operador de consulta utilizado para classificar um multiconjunto. A expressão $\tau_L(R)$, onde R é um multiconjunto e L é uma lista de atributos de R, retorna um multiconjunto composto pelas tuplas de R classificadas na ordem indicada por L.

Capítulo 3

Operadores de Projeção e Seleção

Este capítulo apresenta as contribuições deste trabalho ao *framework* ObInject e sua linguagem de consulta *ObInject Query Language* (OIQL).

A seção 3.1 detalha as melhorias implementadas na linguagem de consulta OIQL, incluindo as duas principais contribuições deste trabalho: os operadores de projeção e seleção. A seção 3.2 trata das modificações estruturais implementadas no *framework* ObInject ao longo deste trabalho, como a persistência de associações entre objetos e os novos métodos *inject* e *reject*. A seção 3.3 apresenta um estudo de caso que utiliza as informações apresentadas nas seções anteriores de forma a retratar a interdependência entre as contribuições do trabalho e seus efeitos no usuário final do *framework*. Por fim, a seção 3.4 traz um balanço de todas as contribuições apresentadas neste capítulo e suas consequências.

3.1 Aprimoramentos na ObInject Query Language

A forma mais indicada de recuperar os objetos persistidos pelas estruturas de dados do *framework* ObInject é através da linguagem *ObInject Query Language* (OIQL). Esta linguagem de consulta baseada em métodos torna mais simples a consulta aos dados persistidos pelo *framework*. Apesar da linguagem já possuir uma definição formal de sua sintaxe, alguns de seus operadores de consulta não se encontram implementados ou possuem apenas uma implementação parcial. Entre eles, estão os operadores de *projeção* e de *seleção*.

A figura 3.1 apresenta um diagrama UML que retrata as classes presentes no pacote *queries* e seus relacionamentos. Tais classes são as responsáveis por processar e executar as consultas feitas através da linguagem OIQL.

As consultas em OIQL são definidas pelo usuário do *framework* através dos métodos *select*, *from*, *where*, *groupBy*, *having* e *orderBy* presentes na classe *Query*. Os métodos *select*,

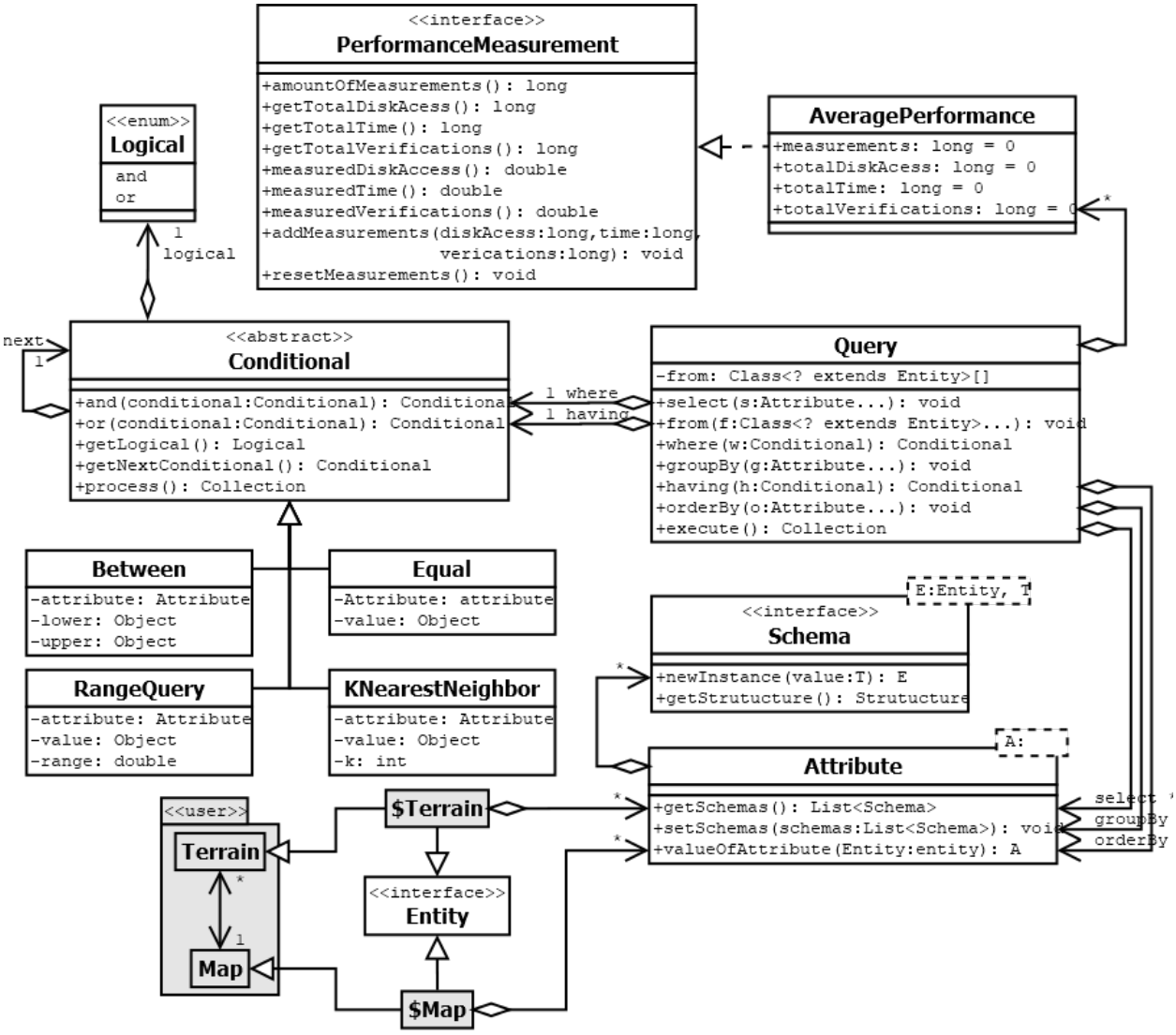

Figura 3.1: Diagrama UML das Classes de Consulta

groupBy e *orderBy* recebem um ou mais elementos da classe *Attribute* como parâmetro, enquanto os métodos *where* e *having* recebem como parâmetro uma instância da classe *Conditional*. O método *from* recebe uma ou mais classes de empacotamento de persistência, que são as classes derivadas de *Entity*. No diagrama 3.1, as classes de empacotamento estão representadas pelas classes *$Terrain* e *$Map*.

A classe *Attribute* encapsula os atributos privados da classe de usuário em atributos públicos de escopo de classe na classe de empacotamento de persistência. Em outras palavras, um atributo privado da classe de usuário é mapeado em sua classe de empacotamento de persistência correspondente através de um atributo público de mesmo nome, porém do tipo *Attribute*. Cada objeto do tipo *Attribute* possui uma lista de objetos do tipo *Schema*, que indicam as estruturas de dados nas quais o atributo se encontra indexado. A classe *Schema* é responsável pela recuperação dos atributos tanto nos índices primários como nos secundários, sendo capaz de buscá-los nas árvores B+, M e R.

A classe *Conditional* é responsável por implementar os algoritmos que fazem uso das informações contidas nas estruturas de dados para responder um determinado tipo de consulta. Por ser uma classe abstrata, não é possível utilizá-la diretamente nos métodos *where* e *having*, sendo necessário utilizar instâncias de suas classes filhas. A principal delas é a classe *Equal*, responsável por implementar a condição de igualdade entre dois elementos através de um atributo indexado em uma árvore B+ ou uma busca sequencial sobre toda a base de dados.

O método *execute* da classe *Query* deve ser chamado manualmente pelo usuário do *framework* para processar todas as operações definidas pela aplicação através dos métodos citados anteriormente. Sua execução retorna uma coleção contendo o resultado da consulta.

3.1.1 Operador de Projeção

O operador de projeção (π) é um operador de consulta unário cuja função é produzir um multiconjunto onde exista um elemento para cada elemento do multiconjunto de entrada, sendo a estrutura dos membros do multiconjunto resultante definida através dos argumentos passados ao operador. No escopo de bancos relacionais, pode ser entendida como a operação responsável por filtrar as colunas de uma tabela.

O método *select* é o responsável pela operação de projeção na linguagem de consulta do ObInject. Na versão da linguagem anterior a este trabalho, apenas a assinatura do método *select* havia sido definida, não existindo uma implementação para este método. Uma das contribuições principais deste trabalho é a implementação do método *select* e, consequentemente, da operação de projeção na linguagem de consulta do ObInject.

Caso uma consulta seja executada sem que antes ocorra uma chamada ao método *select*, a operação de projeção não é realizada e, portanto, o resultado da consulta será composto por uma coleção de instâncias de uma determinada classe de usuário. Do contrário, o tipo de retorno varia conforme o número de parâmetros passados ao método *select*, como descrito a seguir:

- **Um parâmetro**: uma coleção de elementos de mesmo tipo do atributo mapeado pelo parâmetro especificado é retornada;
- **Múltiplos parâmetros**: uma coleção composta por vetores de elementos é retornada, onde cada elemento do vetor possui o tipo do atributo mapeado pelo parâmetro relativo à posição daquele elemento no vetor.

A implementação deste comportamento foi feita através da classe *Query*. A chamada ao método *select* apenas define os atributos que serão utilizados pelo algoritmo de projeção, que se encontra implementado através do método *execute*. O algoritmo de projeção é o

último dos algoritmos a ser executado pelo método *execute*, tratando todos os casos citados anteriormente. Uma coleção vazia é retornada caso a consulta não produza nenhum elemento.

O código 3.1 apresenta a implementação do operador de projeção, omitindo os códigos irrelevantes ao algoritmo do mesmo.

Código 3.1: Implementação do Operador de Projeção

```
1  public class Query {
2      protected Attribute[] selectColumns;
3      ...
4
5      public void select(Attribute... selectColumns) {
6          this.selectColumns = selectColumns;
7      }
8
9      public Collection execute() {
10         TreeSet <? extends Entity> resultWhere = new TreeSet(new
               EntityComparable());
11         Collection<Object> result = new LinkedList<>();
12         ...
13         if (selectColumns != null) {
14             if (selectColumns.length == 1) {
15                 for (Entity entity : resultWhere) {
16                     Object attrib = this.selectColumns[0].valueOfAttribute
                           (entity);
17                     if (attrib != null) {
18                         result.add(attrib);
19                     }
20                 }
21             } else if (selectColumns.length > 1) {
22                 for (Entity entity : resultWhere) {
23                     Object[] vector = new Object[this.selectColumns.length
                           ];
24                     int i = 0;
25                     for (Attribute att : this.selectColumns) {
26                         vector[i++] = att.valueOfAttribute(entity);
27                     }
28                     result.add(vector);
29                 }
30             }
31             return result;
32         } else {
33             return resultWhere;
34         }
35     }
36 }
```

3.1.2 Operador de Seleção

O operador de seleção (σ) é um operador de consulta unário cuja função é produzir um multiconjunto estruturalmente idêntico ao multiconjunto de entrada, porém contendo apenas os elementos que atendam a uma determinada condição fornecida ao operador. No escopo de bancos relacionais, pode ser entendida como a operação responsável por filtrar as linhas de uma tabela.

O método *where* é o responsável pela operação de seleção na linguagem de consulta do ObInject. Assim como no caso do operador de projeção (Seção 3.1.1), apenas a assinatura do método *where* havia sido definida na versão da linguagem anterior a este trabalho. A implementação da operação de seleção na *ObInject Query Language* através do método *where* e suas cláusulas condicionais é também uma das principais contribuições deste trabalho.

A chamada ao método *where* em uma consulta é opcional. Caso seja omitida, o resultado da consulta passa a ser composto por todos os elementos pertencentes às estruturas de dados navegadas. Caso contrário, apenas os elementos que satisfaçam a condição equivalente passada como parâmetro ao método *where* farão parte do resultado final.

A condição equivalente pode ser gerada a partir de múltiplas condições concatenadas através de operadores lógicos binários. Ao final da operação de seleção, uma coleção de instâncias de uma determinada classe de usuário que satisfaçam a condição equivalente é retornada. Uma coleção vazia é retornada caso não existam elementos que satisfaçam a dada condição.

De forma análoga ao operador de projeção, o operador de seleção teve seu algoritmo implementado através do método *execute* da classe *Query*, enquanto seus parâmetros foram tratados na chamada do método *where* da mesma classe. O algoritmo de seleção foi inicialmente projetado para tratar apenas uma condição, sendo mais tarde aprimorado para permitir suporte à concatenação de múltiplas condições.

Para realizar o tratamento de múltiplas condições, foi necessária a implementação dos operadores lógicos *and* e *or*, também chamados operadores de conjunção e disjunção. Ambos foram implementados através da classe *Conditional*, pois, deste modo, todas as condições podem ser devidamente concatenadas. O comportamento destes operadores é descrito abaixo:

- **Operador Lógico de Conjunção**: Implementado através do método *and*, atua sobre duas condições A e B, realizando a interseção entre os resultados obtidos através da consulta caso apenas a condição A seja aplicada com os resultados obtidos caso apenas a condição B seja aplicada;
- **Operador Lógico de Disjunção**: Implementado através do método *or*, realiza a união entre os resultados obtidos pelas consultas executadas com as condições A e B separadamente.

O operador de seleção, implementado através do método *where*, processa múltiplas condições de forma distinta à maneira como o operador de projeção processa múltiplos atributos. O primeiro possui suporte para apenas um parâmetro, enquanto o segundo pode receber múltiplos parâmetros. Múltiplas condições são passadas de forma indireta ao método *where* através da aplicação do padrão de projeto *Chain of Responsibility* (GAMMA et al., 1995) para compor condições em cascata através dos métodos *and* e *or*.

Os códigos 3.2 e 3.3 apresentam a implementação do operador de seleção juntamente com os operadores lógicos de conjunção e disjunção.

Código 3.2: Implementação do Operador de Seleção

```
1  public class Query {
2      private Conditional whereConditional;
3      ...
4      public Conditional where(Conditional conditional) {
5          this.whereConditional = conditional;
6          return this.whereConditional;
7      }
8
9      public Collection execute() {
10         Conditional actual = whereConditional;
11         Collection resultProcess = null;
12         TreeSet <? extends Entity> resultWhere = new TreeSet(new
               EntityComparable());
13         Collection<Object> result = new LinkedList<>();
14         if (actual == null) {
15             ...
16         } else {
17             Conditional.Logical logical = null;
18             while (actual != null) {
19                 resultProcess = actual.process();
20                 if (logical == null)
21                     resultWhere.addAll(resultProcess);
22                 else if (logical == Conditional.Logical.and)
23                     resultWhere.retainAll(resultProcess);
24                 else if (logical == Conditional.Logical.or)
25                     resultWhere.addAll(resultProcess);
26                 resultProcess = null;
27                 logical = actual.getLogical();
28                 actual = actual.getNextConditional();
29             }
30         }
31         ...
32         return result;
33     }
34 }
```

Código 3.3: Implementação dos Operadores de Conjunção e Disjunção

```
1  public abstract class Conditional {
2
3      private Conditional nextConditional;
4      private Logical logical = null;
5      public enum Logical {
6          and,
7          or
8      };
9
10     public Conditional and(Conditional conditional) {
11         this.nextConditional = conditional;
12         this.logical = Logical.and;
13         return this.nextConditional;
14     }
15
16     public Conditional or(Conditional conditional) {
17         this.nextConditional = conditional;
18         this.logical = Logical.or;
19         return this.nextConditional;
20     }
21
22     public Logical getLogical() {
23         return logical;
24     }
25
26     protected Conditional getNextConditional() {
27         return nextConditional;
28     }
29
30     ...
31 }
```

3.2 Aprimoramentos no Framework ObInject

O *framework* ObInject não possuía nenhum tipo de suporte ao armazenamento e eventual busca de associações unidirecionais e bidirecionais com um ou mais objetos. Foi necessária a modificação das estruturas internas do *framework* a fim de acomodar este comportamento.

A solução implementada serializa o UUID dos objetos que fazem parte de um relacionamento juntamente com os demais atributos do objeto persistido. Também é persistida a quantidade de UUIDs serializados no momento da persistência. Tal comportamento pode ser verificado através dos métodos *pushEntity*, *pullEntity* e *sizeOfEntity*, presentes nos códigos das classes de empacotamento de persistência dos apêndices C e D.

Uma das características desta implementação é o fato de apenas o UUID dos objetos pertencentes às associações ser desserializado. Esse fato faz com que seja necessário buscar nos índices primários pelos objetos da associação através de seu UUID. Contudo, buscar todos os objetos de uma associação durante uma consulta pode prejudicar o desempenho da mesma, visto que os objetos recuperados podem nem ser utilizados pela aplicação.

Para evitar uma perda de desempenho nas operações de consulta, foi utilizada a técnica de inicialização tardia (*Lazy Initialization*) (FOWLER, 2002). Esta técnica consiste em inicializar os objetos pertencentes às relações apenas quando eles forem requisitados pela aplicação. A requisição aos objetos pertencentes às relações é processada pela invocação do método de leitura (*get*) do atributo de relacionamento apenas uma única vez. Desta maneira, a classe de empacotamento de persistência realiza a sobrecarga de todos os métodos de leitura de associações.

O código 3.4 ilustra a aplicação dessa técnica na associação de multiplicidade de grau um da classe *$Terrain* e o código 3.5 mostra a inicialização tardia sendo aplicada na associação múltipla da classe *$Map*. Ambas as classes são explicadas com mais detalhes na seção 3.3.

Código 3.4: Inicialização Tardia de uma Associação de Multiplicidade Um

```
1  public Map getMap() {
2      Map superMap = super.getMap();
3      if (superMap == null && uuidMap != null) {
4          superMap = $Map.entityStructure.find(uuidMap);
5          this.setMap(superMap);
6      }
7      return superMap;
8  }
```

Código 3.5: Inicialização Tardia de uma Associação Múltipla

```
1  public java.util.List<obinject.terrain.Terrain> getTerrains() {
2      java.util.List<obinject.terrain.Terrain> superTerrains = super.
           getTerrains();
3      if ((superTerrains.isEmpty()) && (!uuidTerrains.isEmpty())) {
4          for (Uuid uuid : uuidTerrains) {
5              superTerrains.add($Terrain.entityStructure.find(uuid));
6          }
7      }
8      return superTerrains;
9  }
```

As sobrecargas dos métodos de leitura das associações nos códigos 3.4 e 3.5 invocam inicialmente os métodos de leitura da classe mãe através da palavra-chave *super*. Caso eles não retornem os objetos da associação, verifica-se a existência de UUIDs que possam ser utilizados na busca pelos objetos através das estruturas de dados de seus índices primários.

Outra contribuição do trabalho refere-se à alteração da sintaxe das operações de inserção, alteração e exclusão de objetos persistidos. O novo modelo é composto de apenas dois métodos: *inject* e *reject*. A mudança na sintaxe foi feita para tornar a utilização do *framework* mais simples e intuitiva. As implementações de ambos os métodos podem ser encontradas nos apêndices C e D.

O método *inject* é responsável pelas operações de inserção e alteração. Ele verifica se o objeto já foi persistido através de seu atributo único e, em caso negativo, realiza uma operação de inserção. Caso o objeto já tenha sido persistido, o método *inject* realiza uma operação de alteração no objeto.

Já o método *reject* realiza a operação de remoção. Ele busca o objeto persistido através de seu atributo único e o exclui de todas as estruturas de dados do *framework*. Caso a aplicação tente remover um objeto que não se encontre persistido, o método *reject* retorna falso e não realiza ação alguma.

A alteração de um objeto persistente não levava em consideração a possibilidade de existirem outros objetos persistentes associados a ele, visto que a persistência de associações ainda não havia sido implementada. Antes da realização deste trabalho, a alteração de um objeto persistente era implementada em duas etapas: na primeira, o objeto que se encontrava persistido com o mesmo atributo único era excluído das estruturas de dados, enquanto na segunda etapa o novo objeto era inserido. Todavia, o objeto excluído possui um UUID diferente do objeto inserido e, portanto, qualquer associação com o objeto antigo era automaticamente perdida. A solução encontrada foi fazer com que o objeto persistido herdasse o UUID do objeto excluído. Desta forma, as associações anteriores são mantidas intactas.

3.3 Estudo de Caso das Contribuições Realizadas

Nesta seção é apresentado um caso de uso simplificado do *framework* ObInject e sua linguagem de consulta OIQL. O *framework* é utilizado para realizar a persistência, alteração e remoção dos objetos gerados a partir das classes de usuário *Terrain* e *Map* enquanto a linguagem OIQL é utilizada para executar uma série de consultas aos dados persistidos. Através dos exemplos apresentados, é possível entender melhor o impacto das contribuições realizadas por este trabalho na evolução do *framework* ObInject.

As figuras 3.6 e 3.7 permitem visualizar o código das classes de usuário *Map* e *Terrain*, respectivamente. Foram adicionadas diversas anotações nas classes de usuário para que fosse possível torná-las entidades persistentes de forma transparente por meio da metaprogramação presente no *framework* ObInject. As declarações de pacote, importações e implementações dos métodos *getters* e *setters* foram omitidas.

Código 3.6: Classe *Map*

```
1  @Persistent
2  public class Map {
3      @Unique
4      private String name;
5      private String description;
6
7          private List<Terrain> terrains = new ArrayList<>();
8
9      //gets() e sets()
10 }
```

Código 3.7: Classe *Terrain*

```
1  @Persistent
2  public class Terrain {
3      @Unique
4      private long register;
5      @Sort
6      @Edition
7      private String owner;
8      private String city;
9      private String district;
10     private String street;
11     private float propertyValue;
12     private int number;
13     @Point
14     @Origin
15     private float[] originCoordinate = new float[2];
16     @Extension
17     private float[] extensionCoordinate = new float[2];
18
19     private Map map;
20
21     //gets() e sets()
22 }
```

As classes *Map* e *Terrain* foram anotadas com a anotação `@Persistent` para declarar ao *framework* ObInject que ambas desejam se tornar entidades persistentes através de suas respectivas classes de empacotamento de persistência.

A anotação `@Unique` foi utilizada em ambas as classes para identificar um atributo com valores que jamais se repetirão entre as instâncias das classes, para que o usuário seja capaz de buscar de forma inequívoca um determinado objeto. A estrutura utilizada para indexar o atributo é uma árvore B+.

A anotação `@Sort`, presente no atributo *owner* da classe *Terrain*, tem por objetivo agilizar as consultas de atributos que possuem relação de ordem através da criação de um índice secundário que utiliza o atributo na composição de sua chave de indexação. A estrutura utilizada para a criação desse tipo de índice é uma árvore B+.

A anotação `@Edition`, também presente no atributo *owner*, tem por objetivo agilizar as consultas de cadeias de caracteres através da criação de um índice secundário que utiliza uma relação de semelhança de palavras entre seus elementos. A estrutura utilizada neste índice é uma árvore M.

A anotação `@Point` se encontra no atributo *originCoordinate* da classe *Terrain* e seu objetivo é indexar atributos que possuem uma relação de semelhança no espaço euclidiano com o objetivo de agilizar consultas envolvendo estes atributos. A árvore M é a estrutura utilizada na criação deste índice.

As anotações `@Origin` e `@Extension`, ambas presentes na classe *Terrain*, indicam os atributos que irão compor um retângulo indexado por um índice secundário que utiliza a relação de semelhança no espaço euclidiano. A estrutura utilizada para esta indexação é uma árvore R.

Após as classes de usuário estarem devidamente anotadas, é possível gerar as classes auxiliares responsáveis pela persistência e indexação dos dados através da classe *Wrapper* do *framework* ObInject. Essa classe possui o método estático *create*, responsável pelo processamento das anotações presentes nas classes de usuário e consequente criação das classes auxiliares utilizando os metadados coletados.

As seguintes classes foram geradas pelo *framework* ObInject:

- **$Map**: Gerada a partir da anotação `@Persistent` presente na classe *Map* (Apêndice C);
- **$Terrain**: Gerada a partir da anotação `@Persistent` presente na classe *Terrain* (Apêndice D);
- **UniqueOneMap**: Gerada a partir da anotação `@Unique` presente na classe *Map*;
- **UniqueOneTerrain**: Gerada a partir da anotação `@Unique` presente na classe *Terrain*;
- **SortOneTerrain**: Gerada a partir da anotação `@Sort` presente na classe *Terrain*;
- **EditionOneTerrain**: Gerada a partir da anotação `@Edition` presente na classe *Terrain*;
- **PointOneTerrain**: Gerada a partir da anotação `@Point` presente na classe *Terrain*;
- **RectangleOneTerrain**: Gerada a partir das anotações `@Origin` e `@Extension` presentes na classe *Terrain*.

O diagrama UML representado pela figura 3.2 contém a relação entre todas as classes geradas pelo *framework* e as classes de usuário *Terrain* e *Map*.

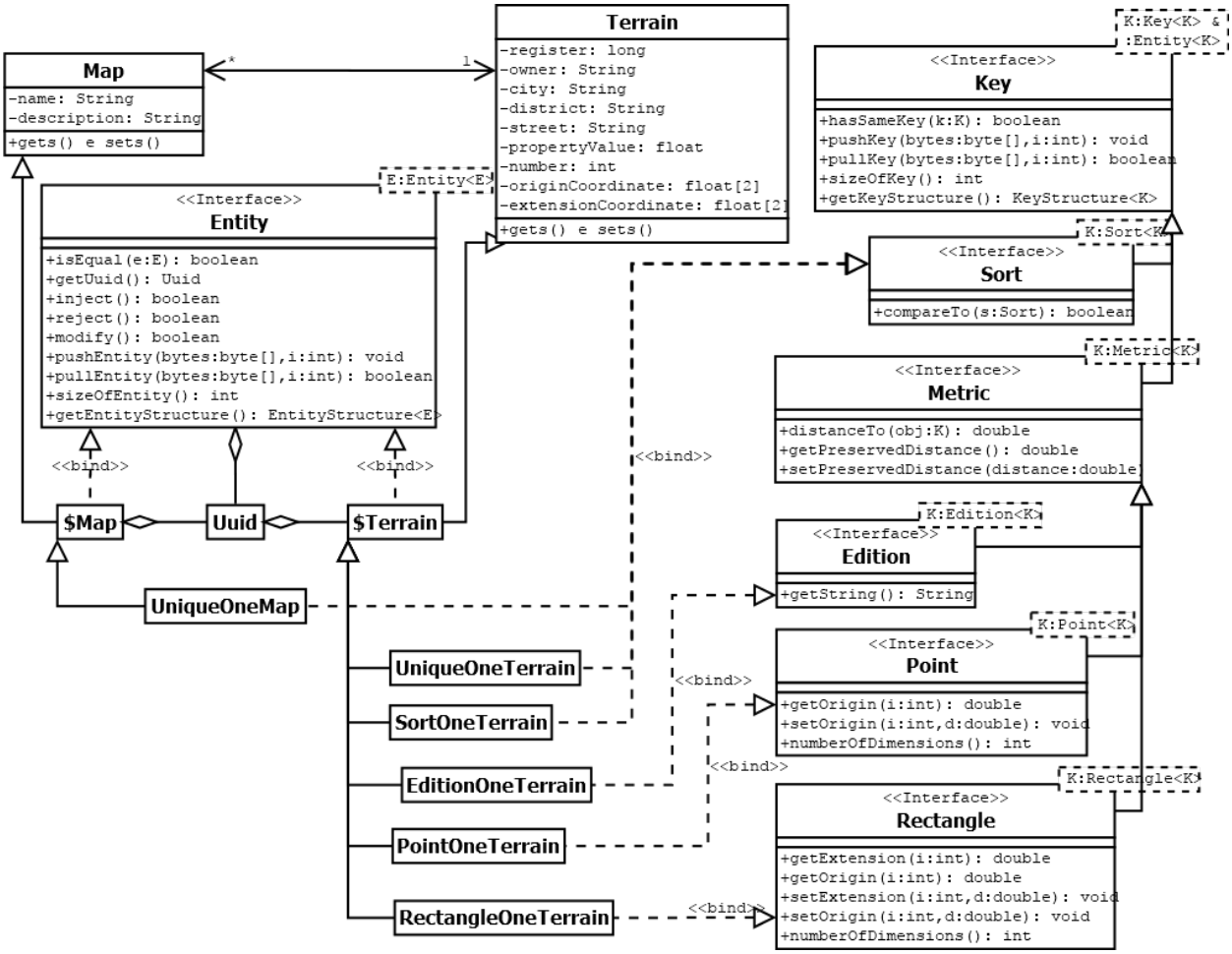

Figura 3.2: Diagrama UML das Entidades Geradas

Foram criadas quatro aplicações com propósitos distintos utilizando a modelagem apresentada no diagrama 3.2. Uma breve descrição dos objetivos de cada aplicação segue abaixo:

- **AppInsert**: Exemplifica a utilização do método *inject* para armazenar os objetos nas estruturas de dados do *framework* ObInject através do código 3.8;
- **AppUpdate**: Exemplifica a utilização do método *inject* para atualizar os objetos e seus índices nas estruturas de dados do *framework* através do código 3.9;
- **AppDelete**: Exemplifica a utilização do método *reject* para remover os objetos e seus índices das estruturas de dados do *framework* através do código 3.10;
- **AppQuery**: Exemplifica a utilização dos operadores de projeção e seleção da linguagem de consulta OIQL para realizar uma busca aos objetos armazenados nas estruturas de dados do *framework* através do código 3.11.

O código 3.8 utiliza o método *inject* para persistir três objetos do tipo *Terrain* e um objeto do tipo *Map*. Todos os três objetos da classe *Terrain* se encontram associados ao mesmo objeto da classe *Map*, de forma bidirecional.

Código 3.8: Persistência de Objetos pelo Método *inject*

```
1  public class AppInsert {
2      public static void main(String[] args) {
3          PersistentManager pm = PersistentManagerFactory.
4                                  createPersistentManager();
5          pm.getTransaction().begin();
6
7          Map m = new Map();
8          m.setName("Itajuba");
9          m.setDescription("Municipio de Itajuba");
10         pm.inject(m);
11
12         Terrain t1 = new Terrain();
13         t1.setRegister(1);
14         t1.setDistrict("Centro");
15         t1.setExtensionCoordinate(new float[]{2, 2});
16         t1.setMap(m);
17         pm.inject(t1);
18
19         Terrain t2 = new Terrain();
20         t2.setRegister(2);
21         t2.setDistrict("BPS");
22         t2.setExtensionCoordinate(new float[]{123, 456});
23         t2.setMap(m);
24         pm.inject(t2);
25
26         Terrain t3 = new Terrain();
27         t3.setRegister(3);
28         t3.setDistrict("Pinheirinho");
29         t3.setExtensionCoordinate(new float[]{987, 654});
30         t3.setMap(m);
31         pm.inject(t3);
32
33         m.getTerrains().add(t1);
34         m.getTerrains().add(t2);
35         m.getTerrains().add(t3);
36         pm.inject(m);
37
38         pm.getTransaction().commit();
39         pm.close();
40     }
41 }
```

As linhas 10, 17, 24, 31 e 36 do código 3.8 indicam que os objetos m, t1, t2 e t3 são persistidos, nesta ordem, durante a execução da aplicação. A linha 5 isola as quatro ações de persistência em uma única transação, implementada sem controle de concorrência, enquanto a linha 38 indica o momento de realizar as ações de persistência contidas na transação.

O código 3.9 busca e atualiza um dos objetos da classe *Terrain* através do método *inject*, realizando uma nova busca após a atualização. Na linha 9 é realizada uma busca por um dos objetos da classe *Terrain* cujo atributo *register* possui o valor um. Na linha 16, o valor do atributo *district* é alterado para "Boa Vista". A linha 17 indica que o objeto do tipo *Terrain* deve ser persistido e, na linha 18, ocorre de fato a persistência. Por fim, o objeto da classe *Terrain* é buscado novamente na linha 23 e seus atributos impressos para confirmar que houve efetivamente uma atualização.

Código 3.9: Atualização de Objetos Persistentes pelo Método *inject*

```
1  public class AppUpdate {
2      public static void main(String[] args) {
3          PersistentManager pm = PersistentManagerFactory.
4                                  createPersistentManager();
5
6          Query q1 = new Query();
7          q1.from($Terrain.class);
8          q1.where(new Equal($Terrain.register, 1L));
9          Collection<Terrain> res1 = q1.execute();
10         Terrain t1 = res1.iterator().next();
11         System.out.println("uuid:" + (($Terrain) t1).getUuid());
12         System.out.println("register: " + t1.getRegister());
13         System.out.println("district: " + t1.getDistrict());
14
15         pm.getTransaction().begin();
16         t1.setDistrict("Boa Vista");
17         pm.inject(t1);
18         pm.getTransaction().commit();
19
20         Query q2 = new Query();
21         q2.from($Terrain.class);
22         q2.where(new Equal($Terrain.register, 1L));
23         Collection<Terrain> res2 = q2.execute();
24         Terrain t2 = res2.iterator().next();
25         System.out.println("uuid:" + (($Terrain) t2).getUuid());
26         System.out.println("register: " + t2.getRegister());
27         System.out.println("district: " + t2.getDistrict());
28
29         pm.close();
30     }
31 }
```

O código 3.10 busca por um dos objetos da classe *Terrain* persistidos pelo código 3.8 e, em seguida, efetua a remoção deste objeto através dos seguintes passos:

1. **Remoção da Associação**: A linha 16 do código 3.10 remove da associação com o objeto m1 da classe *Map* o objeto da classe *Terrain* cujo atributo *register* possui valor um. Na linha 17, o objeto m1 é marcado para atualização e, na linha 18, a atualização é concretizada;

2. **Remoção do Objeto**: Na linha 21 do código 3.10, o objeto da classe *Terrain* removido da associação no passo anterior é marcado para remoção das estruturas de dados do *framework* ObInject. A linha 22 concretiza a remoção deste objeto.

Código 3.10: Exclusão de Objetos Persistentes pelo Método *reject*

```
1  public class AppDelete {
2      public static void main(String[] args) {
3          PersistentManager pm = PersistentManagerFactory.
4                                  createPersistentManager();
5
6          Query q1 = new Query();
7          q1.from($Map.class);
8          q1.where(new Equal($Map.name, "Itajuba"));
9          Collection<Map> res1 = q1.execute();
10         Map m1 = res1.iterator().next();
11         Terrain tDel = null;
12         for (Terrain t : m1.getTerrains())
13             if (t.getRegister() == 1) tDel = t;
14
15         pm.getTransaction().begin();
16         m1.getTerrains().remove(tDel);
17         pm.inject(m1);
18         pm.getTransaction().commit();
19
20         pm.getTransaction().begin();
21         pm.reject(tDel);
22         pm.getTransaction().commit();
23
24         Query q2 = new Query();
25         q2.from($Terrain.class);
26         q2.where(new Equal($Terrain.register, 1L));
27         Collection<Terrain> res3 = q2.execute();
28         if (res3.isEmpty())
29             System.out.println("Terrain 1 not found");
30         pm.close();
31     }
32 }
```

Por fim, o código 3.11 realiza consultas aos objetos persistidos pelas aplicações anteriores através da linguagem OIQL. A primeira consulta utiliza o operador de seleção, através do método *where*, para retornar apenas objetos da classe *Terrain* cujo campo *register* possua um dos valores especificados, e o operador de projeção, através do método *select*, para retornar apenas os atributos *register* e *number*.

A segunda consulta do código 3.11 retorna objetos completos da classe *Map*, incluindo suas associações com outros objetos da classe *Terrain*. Os objetos associados são buscados de fato somente ao serem impressos na tela.

Código 3.11: Consulta aos Objetos Persistidos pela Linguagem OIQL

```
1  public class AppQuery {
2      public static void main(String[] args) {
3          int n = 0;
4          Query q1 = new Query();
5          q1.select($Terrain.register, $Terrain.number);
6          q1.from($Terrain.class);
7          q1.where(new Equal($Terrain.register, 1L)).or(new Equal($Terrain.
               register, 2L));
8          Collection<Object[]> res1 = q1.execute();
9          for (Object o : res1) {
10             System.out.println("===== query 1 : tuple " + n + " ====");
11             System.out.println("Register: " + obj[0]);
12             System.out.println("Number: " + obj[1]);
13             n++;
14         }
15
16         Query q2 = new Query();
17         q2.from($Map.class);
18         q2.where(new Equal($Map.name, "Itajuba"));
19         Collection<Map> res2 = q2.execute();
20         n = 0;
21         for (Map map : res2) {
22             System.out.println("===== query 2 : tuple " + n + " ====");
23             System.out.println("Register: "+map.getName());
24             System.out.println("Description: "+map.getDescription());
25             System.out.print("Terrain: ");
26             for (Terrain t : map.getTerrains()) {
27                 System.out.print("[register:" + t.getRegister());
28                 System.out.print(", district:" + t.getDistrict() + "]");
29             }
30             System.out.println();
31             n++;
32         }
33     }
34 }
```

3.4 Considerações Finais

A principal contribuição deste trabalho foi a implementação dos operadores de consulta de projeção (π) e seleção (σ) através dos métodos *select* e *where*, respectivamente. O impacto da implementação destes operadores na linguagem é extensivo, pois muitas das consultas mais rotineiras em uma aplicação costumam utilizar ambos os operadores.

Os operadores lógicos de conjunção e disjunção também foram implementados para possibilitar a utilização de múltiplas condições no método *where*. O operador de conjunção foi implementado através do método *and*, enquanto o operador de disjunção é representado pelo método *or*, ambos presentes na classe *Conditional*.

Diversas modificações no próprio *framework* também foram efetuadas ao longo deste trabalho, como a mudança na forma de serialização e desserialização dos objetos de modo a fornecer suporte à persistência de associações entre objetos. A recuperação das associações persistidas é feita através da técnica de *Lazy Initialization*, ou seja, as associações são recuperadas somente no momento de seu primeiro uso.

Outra contribuição secundária deste trabalho foi a definição e implementação dos métodos *inject* e *reject*, ambos relacionados à operação de persistência do *framework*. O método *inject* realiza as operações de inserção e atualização de um objeto nas estruturas de dados, enquanto o método *reject* é o responsável pela operação de remoção nestas mesmas estruturas.

Por fim, é possível concluir que as contribuições apresentadas neste capítulo consistem em uma evolução natural e progressiva do *framework* ObInject e sua linguagem de consulta, a *ObInject Query Language*.

Capítulo 4

Experimentos

Este capítulo descreve os experimentos realizados neste trabalho, com o objetivo de testar a persistência de objetos e consultas utilizando o *framework* ObInject (Seção 2.1.1) em conjunto com a linguagem de consulta *ObInject Query Language* (Seção 2.2.1), ambos atualizados com as contribuições descritas no capítulo 3.

Os experimentos descritos a seguir foram realizados com os seguintes *frameworks* de persistência desenvolvidos na linguagem de programação Java: ObInject (Seção 2.1.1), Hibernate (Seção 2.1.2) e ActiveJDBC (Seção 2.1.3). O principal objetivo da realização dos experimentos propostos neste capítulo é assegurar que o *framework* ObInject continua mantendo um grau de desempenho equiparável ao medido por Ferro (2012) mesmo após as mudanças estruturais sofridas pelo mesmo.

4.1 Metodologia

Os experimentos foram realizados em um MacBook Air™ com sistema operacional macOS Sierra™ versão 10.12.5, sistema de arquivos HFS+ com *journaling* e blocos de 4 KB, 4 GB de memória RAM de 1600 MHz, unidade de estado sólido (*Solid-State Drive* - SSD) modelo APPLE SSD TS128E com capacidade de 128 GB, e processador Intel® Core™ i5-3317u com 2 núcleos, 4 *threads*, *clock* de 1.7 GHz e 3 MB de *cache* L3.

Os aplicativos utilizados na realização dos experimentos foram implementados na linguagem de programação Java, utilizando o kit de desenvolvimento Java (*Java Development Kit* - JDK) versão 1.8.0_77-b03. A máquina virtual Java (*Java Virtual Machine* - JVM) responsável pela execução dos aplicativos é a HotSpot™ Server VM 64 *bits* versão 25.77-b03. Os parâmetros de invocação da JVM estabelecidos pelos aplicativos do experimento são `-Xms2048m`, que especifica a alocação inicial de memória para a JVM em 2 GB, e `-Xmx2048m`, que especifica a alocação máxima de memória para a JVM também em 2 GB.

Os *frameworks* utilizados nos experimentos, além do próprio ObInject, são o Hibernate ORM versão 4.3.1.Final e o ActiveJDBC versão 1.4.10. O sistema de gerenciamento de banco de dados relacional (SGBDR) utilizado por ambos os *frameworks* é o MySQL versão 5.6.21. A conexão entre os *frameworks* e o SGBD é feita através do *driver* JDBC MySQL Connector/J versão 5.1.23. O mecanismo de armazenamento utilizado pelo MySQL nesses experimentos é o InnoDB. Os testes realizados com o *framework* ObInject foram conduzidos utilizando um tamanho de bloco de 4 KB.

O modelo de dados ilustrado pela figura 4.1 representa o sistema eleitoral americano e foi utilizado na realização dos experimentos.

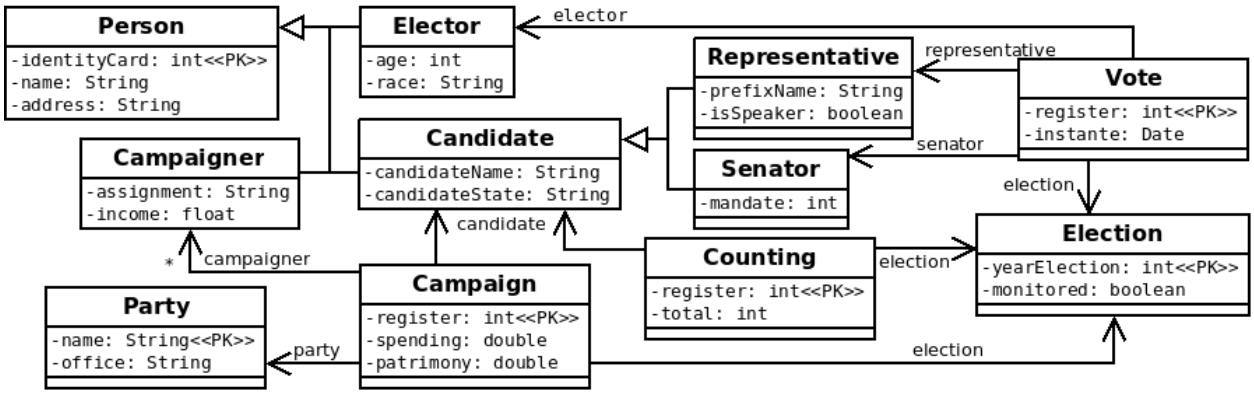

Figura 4.1: Diagrama UML do Sistema Eleitoral Americano

Os experimentos de inserção utilizam cinco conjuntos de dados distintos gerados programaticamente de forma pseudo-aleatória. Esses dados são baseados no diagrama 4.1 e os objetos gerados a partir deles obedecem uma proporção pré-definida. A distribuição dos objetos criados para inserção a partir dos dados gerados pode ser verificada na tabela 4.1. As classes *Person* e *Candidate* não possuem objetos, pois sua função na modelagem é somente servir como superclasse das classes concretas.

Experimento	**1**	**2**	**3**	**4**	**5**
`Election`	1	1	1	1	1
`Party`	10	20	30	40	50
`Senator`	14	39	112	172	264
`Representative`	98	264	1025	1476	2792
`Campaigner`	162	768	4036	7301	16793
`Campaign`	112	303	1137	1648	3056
`Counting`	112	303	1137	1648	3056
`Elector`	100000	200000	300000	400000	500000
`Vote`	100000	200000	300000	400000	500000
Total	**200509**	**401698**	**607478**	**8122286**	**1026012**

Tabela 4.1: Distribuição dos Objetos para Inserção

Da mesma forma que os experimentos de inserção, os experimentos de consulta utilizam cinco conjuntos de dados distintos gerados a partir do diagrama 4.1. A diferença está no algoritmo de geração: os dados de consulta são gerados a partir dos dados de inserção, sendo aqueles um subconjunto destes. Aproximadamente 10% dos dados de inserção são selecionados de forma pseudo-aleatória para compor os dados de consulta, sendo garantida a geração de ao menos uma consulta por classe concreta. A distribuição dos objetos criados para consulta a partir dos dados gerados pode ser verificada na tabela 4.2.

Experimento	**1**	**2**	**3**	**4**	**5**
`Election`	1	1	1	1	1
`Party`	1	2	5	4	7
`Senator`	1	6	14	19	25
`Representative`	9	26	97	155	266
`Campaigner`	14	77	383	782	1649
`Campaign`	10	32	111	174	291
`Counting`	9	38	124	176	291
`Elector`	10106	20234	29679	40477	49839
`Vote`	10106	20234	29679	40477	49839
Total	**20257**	**40650**	**60093**	**82265**	**102208**

Tabela 4.2: Distribuição dos Objetos para Consulta

Os experimentos propostos foram realizados a partir das especificações técnicas e dos conjuntos de dados apresentados acima. A lista abaixo define as características que foram quantificadas durante realização dos experimentos de inserção e consulta pelos diferentes *frameworks* listados anteriormente.

- **Tempo de Inserção**: Tempo gasto para persistir todos os objetos definidos no experimento;
- **Tempo de Consulta**: Tempo gasto para realizar a busca de todos os objetos definidos no experimento;
- **Consumo de Memória na Inserção**: Memória utilizada durante a persistência de todos os objetos definidos no experimento;
- **Consumo de Memória na Consulta**: Memória utilizada durante a busca de todos os objetos definidos no experimento;
- **Consumo do Processador na Inserção**: Porcentagem do processador utilizada durante a persistência de todos os objetos definidos no experimento;
- **Consumo do Processador na Consulta**: Porcentagem do processador utilizada durante a busca de todos os objetos definidos no experimento;
- **Tamanho em Disco**: Quantidade de *bytes* ocupada pelos objetos persistidos em disco.

Os tempos de inserção e consulta foram coletados através de medições internas feitas pelas próprias aplicações Java, utilizando o método `System.nanoTime()`. Este método possui uma precisão de nanossegundos, mas não necessariamente uma resolução de nanossegundos, pois a resolução é dependente do sistema operacional utilizado. Todavia, como os valores de tempo medidos são da ordem de segundos, os valores retornados podem ser considerados suficientemente precisos.

A ferramenta de monitoramento Java *JConsole* foi utilizada para realizar as medições de consumo de memória e processador. As medições foram possíveis graças à instrumentação extensiva presente na JVM, da qual o JConsole faz uso. A versão utilizada nos experimentos é a 1.8.0_77-b03.

O tamanho em disco ocupado pelos objetos persistidos pelo *framework* ObInject foi medido através do comando *ls*, visto que todos os objetos persistidos e suas estruturas de dados se encontram em um único arquivo com extensão *dbo*. No caso dos *frameworks* Hibernate e ActiveJDBC, essa medição foi feita através do comando *du*, pois o SGBDR MySQL armazena as informações pertinentes ao banco de dados em um diretório com múltiplos arquivos.

4.2 Resultados

Os resultados foram divididos em quatro categorias principais: Tempo (Seção 4.2.1), Memória (Seção 4.2.2), Processador (Seção 4.2.3) e Disco (Seção 4.2.4). Cada categoria representa uma característica distinta quantificada durante a realização dos experimentos de inserção e consulta. Os *frameworks* ObInject, Hibernate e ActiveJDBC foram testados de maneira independente e os resultados agrupados em gráficos de acordo com a categoria medida. Uma quinta categoria existe com o intuito de apresentar as medidas relacionadas somente ao *framework* ObInject: Medições Adicionais (Seção 4.2.5).

4.2.1 Tempo

As figuras 4.2 e 4.3 indicam o tempo de execução das aplicações desenvolvidas com os *frameworks* ObInject, Hibernate e ActiveJDBC após realizadas as inserções e consultas dos objetos.

Analisando os tempos totais de inserção e consulta para cada um dos cinco experimentos propostos, é possível notar uma clara vantagem do *framework* ObInject em relação aos demais. Ele foi em média 216% mais rápido em relação ao Hibernate e 169% mais rápido em relação ao ActiveJDBC nas operações de inserção. Nas operações de consulta, o ObInject foi em média 607% mais rápido em relação ao Hibernate e 251% mais rápido em relação ao ActiveJDBC.

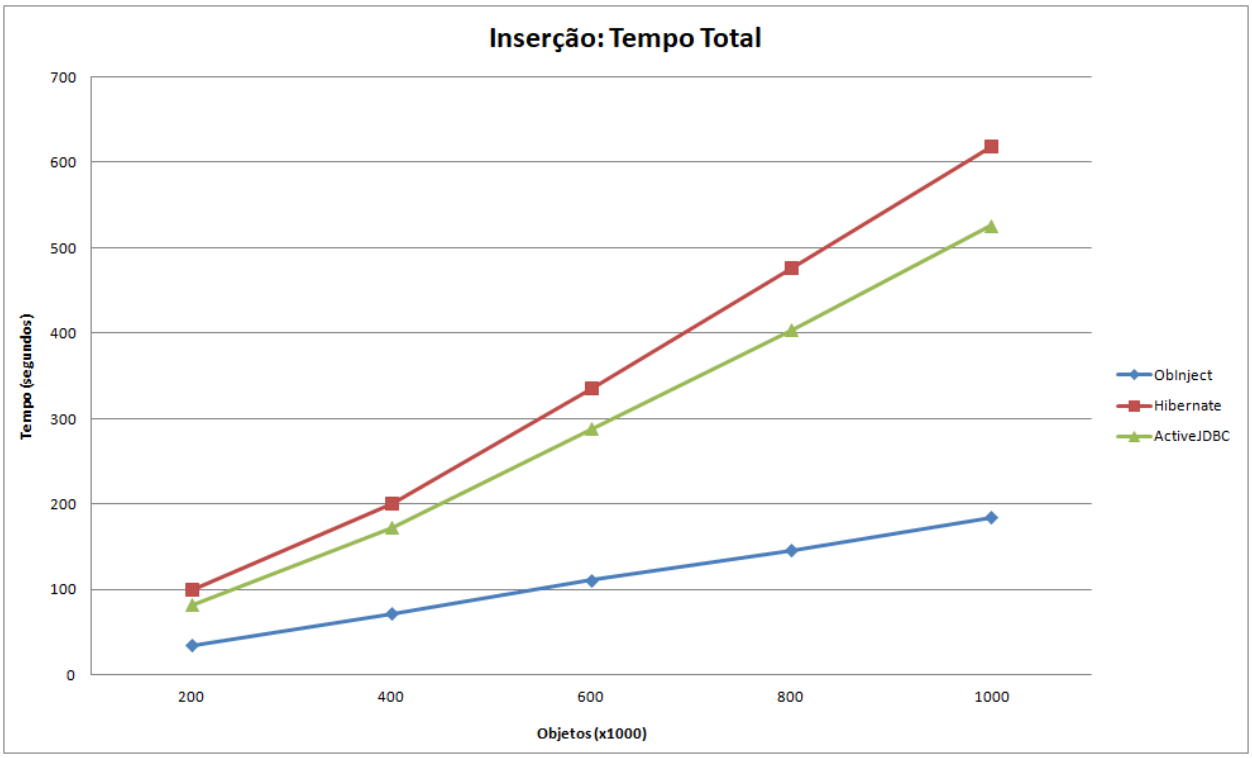

Figura 4.2: Tempo de Inserção dos *Frameworks*

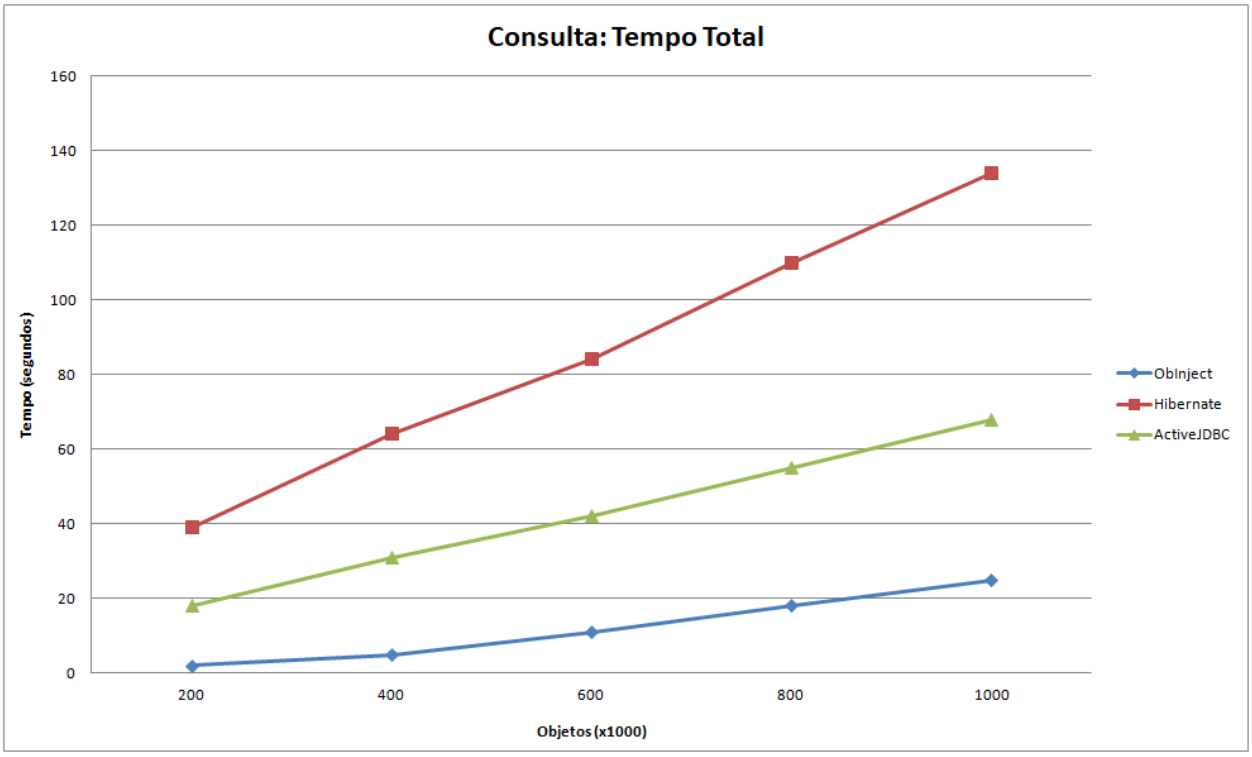

Figura 4.3: Tempo de Consulta dos *Frameworks*

4.2.2 Memória

As figuras 4.4 e 4.6 indicam o consumo médio e máximo de memória dos *frameworks* ObInject, Hibernate e ActiveJDBC ao realizar as inserções dos objetos descritos na tabela 4.1. Já as figuras 4.5 e 4.7 indicam o consumo médio e máximo de memória ao consultar os objetos referenciados pela tabela 4.2.

O consumo médio de memória foi calculado através da média aritmética dos valores de consumo de memória medidos ao longo da vida da aplicação, sendo os pontos iniciais e finais descartados. Os pontos iniciais possuem um valor artificialmente alto pois estão sob influência da inicialização da máquina virtual Java, enquanto que os pontos finais possuem um valor artificialmente baixo visto que a aplicação se encontra ociosa.

O consumo máximo de memória, também chamado consumo de pico, é o maior entre os valores de consumo de memória medidos ao longo da vida da aplicação. Este valor representa a quantidade mínima de memória necessária para executar a aplicação com sucesso.

É importante salientar a influência do coletor de lixo (*Garbage Collector* - GC) da linguagem Java nos experimentos de medição de memória consumida. O coletor de lixo é um processo utilizado para a automação do gerenciamento de memória. Este processo é responsável por desalocar recursos de referências que não serão mais acessadas no futuro pela aplicação, sendo acionado sempre que a JVM julga necessário. O coletor de lixo é executado múltiplas vezes durante a execução das aplicações relativas aos experimentos, influenciando diretamente nos valores de consumo de memória medidos.

A quantidade de memória utilizada pelo *framework* Hibernate foi bem superior à quantidade utilizada pelos outros dois *frameworks* nos experimentos relativos à inserção. Um fator contribuinte para o ocorrido é a forma como os objetos são inseridos pela aplicação de inserção desenvolvida para o Hibernate: a transação é aberta logo no início da aplicação e sofre um *commit* apenas ao final de todas as inserções. Isso faz com que as inserções aconteçam de forma mais rápida, influenciando diretamente no gráfico de tempo de inserção (Figura 4.2). Todavia, uma das consequências negativas é o maior consumo de memória por parte da aplicação, visto que os objetos permanecem na memória à espera do final da transação.

O *framework* ActiveJDBC teve um consumo de memória semelhante ao do ObInject, sendo o consumo de ambos inferior ao do Hibernate. O ActiveJDBC não trabalha com o conceito de transação, o que se traduz em uma menor quantidade de objetos simultâneos em memória secundária, visto que eles não precisam esperar o fim de uma transação e, por isso, podem ser salvos em disco logo após serem instanciados.

É importante ressaltar que os valores apresentados não levam em consideração a memória alocada pelo SGBD MySQL. Caso estes valores fossem considerados, o consumo de memória dos *frameworks* Hibernate e ActiveJDBC seria ainda maior.

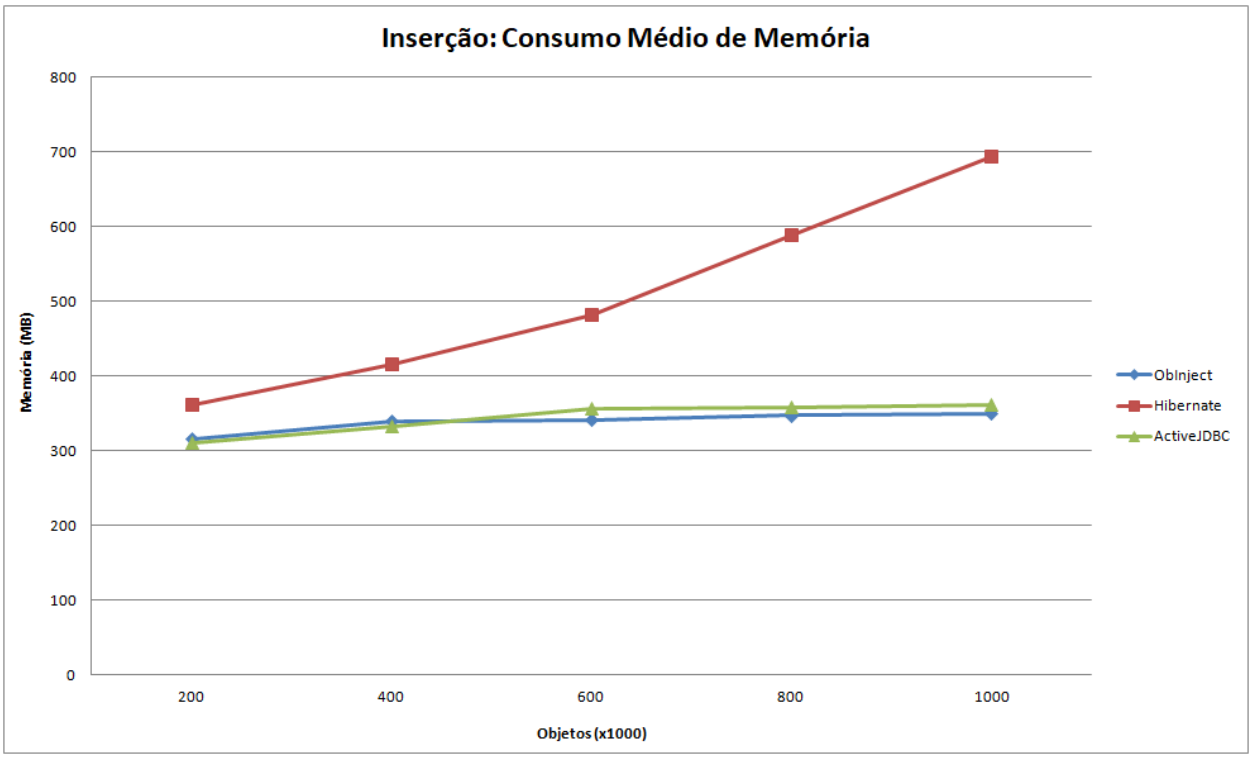

Figura 4.4: Consumo Médio de Memória dos *Frameworks* na Inserção

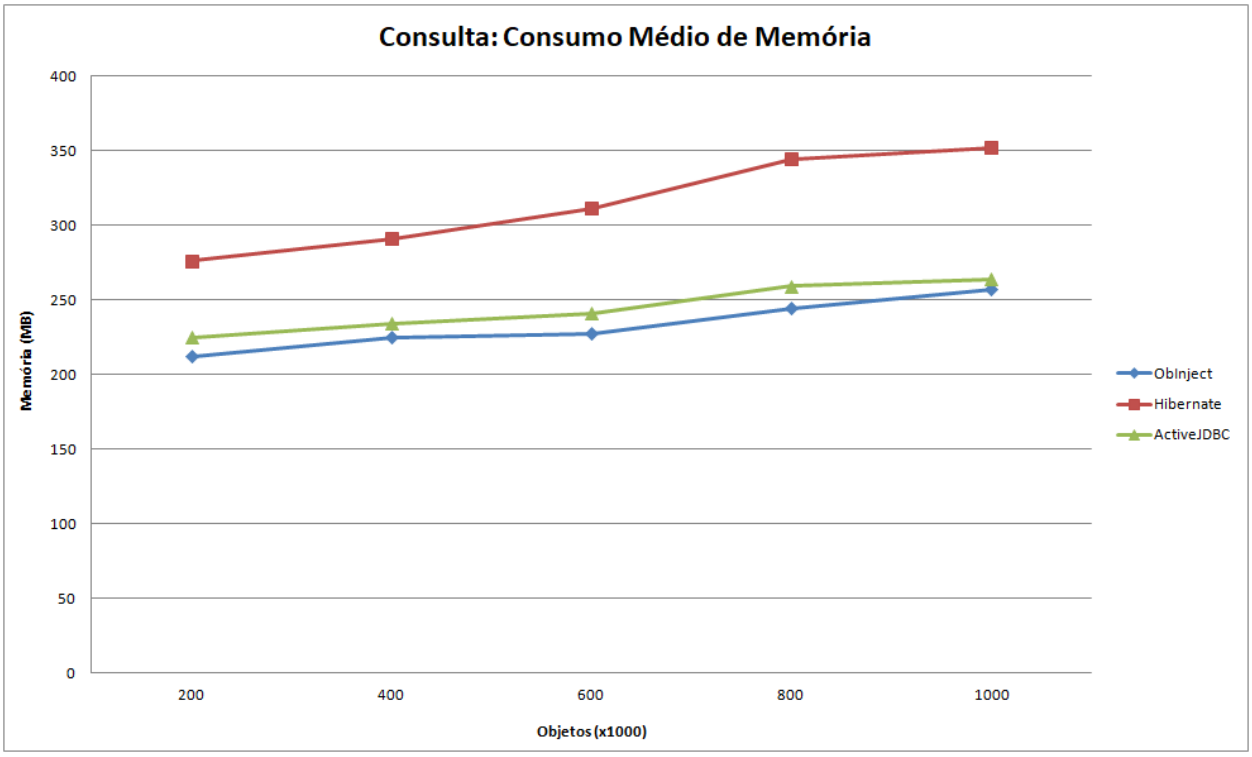

Figura 4.5: Consumo Médio de Memória dos *Frameworks* na Consulta

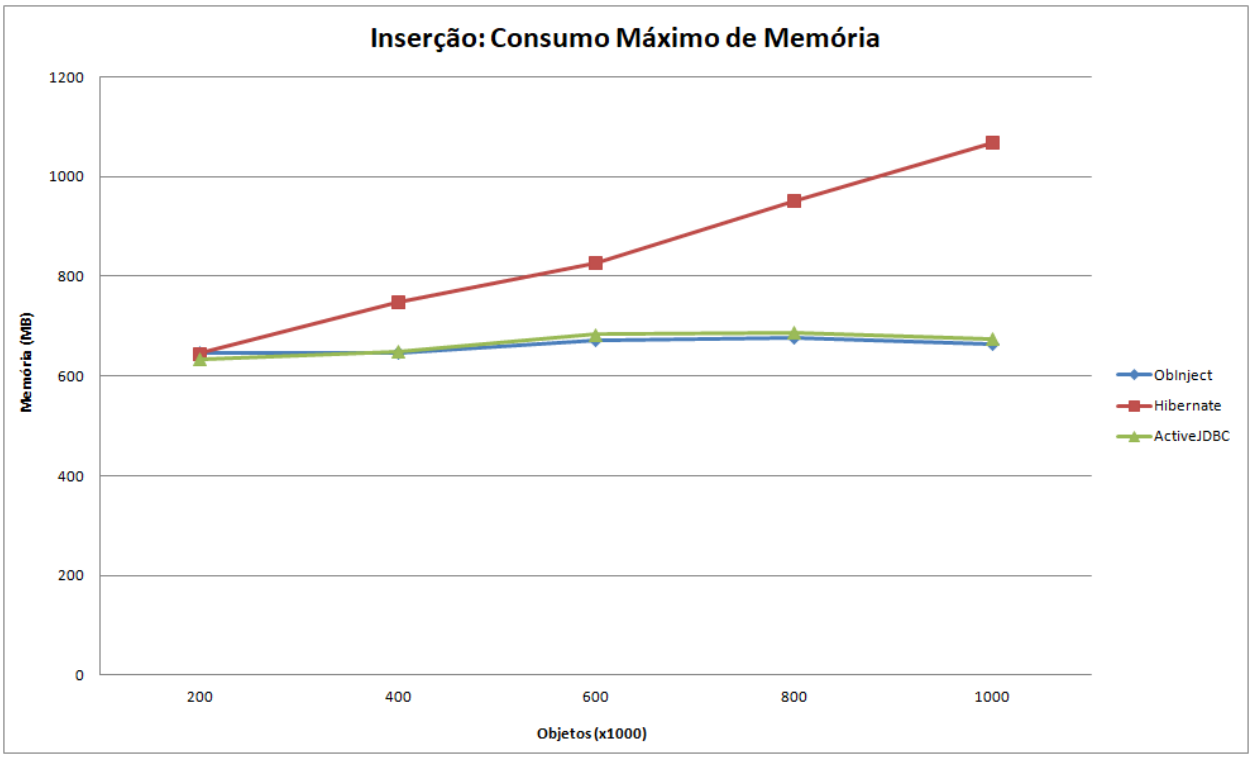

Figura 4.6: Consumo Máximo de Memória dos *Frameworks* na Inserção

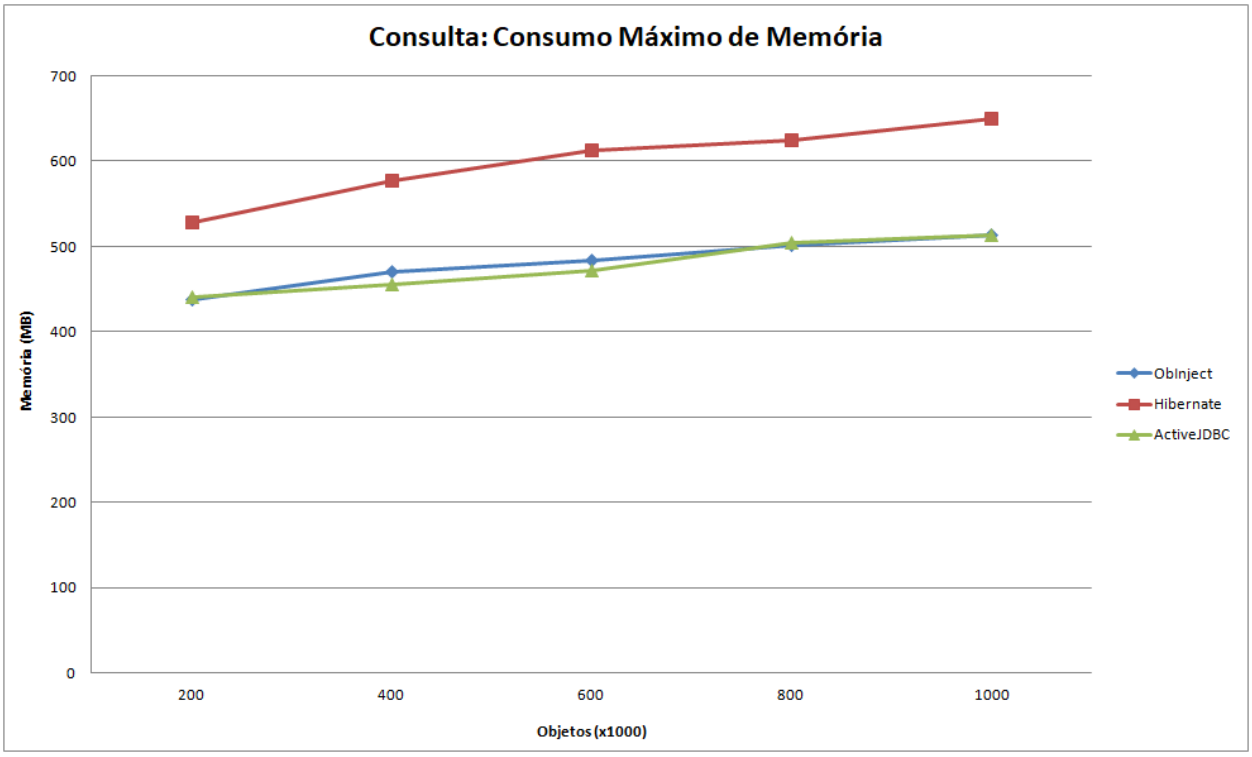

Figura 4.7: Consumo Máximo de Memória dos *Frameworks* na Consulta

4.2.3 Processador

As figuras 4.8 e 4.9 indicam o consumo percentual médio do processador pelas aplicações desenvolvidas com os *frameworks* ObInject, Hibernate e ActiveJDBC durante as operações de inserção e consulta dos objetos. As especificações do processador utilizado no experimento se encontram na seção 4.1.

O consumo percentual médio do processador foi calculado através da média aritmética dos valores percentuais medidos durante a execução da aplicação, sendo os pontos iniciais e finais descartados. Os pontos iniciais possuem um valor artificialmente alto pois estão sob influência da inicialização da máquina virtual Java, enquanto que os pontos finais possuem um valor artificialmente baixo visto que a aplicação se encontra ociosa.

Analisando os dados coletados durante a operação de inserção para cada um dos cinco experimentos (Figura 4.8), é possível concluir que o ObInject demanda mais poder de processamento que os demais *frameworks* testados ao realizar uma operação de inserção. Um fator contribuinte para o resultado é o fato do processamento adicional utilizado pelo sistema de gerenciamento de banco de dados dos *frameworks* Hibernate e ActiveJDBC não ter sido contabilizado nos resultados. O *framework* ObInject não exige nenhum processamento adicional de *softwares* externos para seu funcionamento, uma vez que é o único responsável pela persistência dos objetos.

Os dados coletados durante a operação de consulta revelam um menor consumo do processador pelo *framework* ObInject quando comparados aos *frameworks* ActiveJDBC e Hibernate. Este comportamento é o oposto do observado na operação de inserção, e o motivo é o fato da operação de busca demandar em média um menor processamento quando comparada à operação de inserção em uma árvore B+, visto que um número menor de operações é executado durante a busca.

Um comportamento curioso observado em todos os *frameworks* testados durante a consulta (Figura 4.9) é a diminuição do consumo médio de processamento na medida em que mais objetos são buscados. Entretanto, o delta das diminuições entre dois experimentos consecutivos se torna cada vez menor, sendo o consumo praticamente estável entre os dois últimos experimentos. Tal comportamento deve-se, sobretudo, ao fato da configuração inicial de ambos os *frameworks* ORM necessitar de uma maior quantidade de processamento para executar certas funções como negociar uma conexão com o SGBD MySQL. No caso do ObInject, este comportamento é explicado pela influência da inicialização da JVM nas medições, já que os experimentos 1 e 2 foram realizados em poucos segundos e, por este motivo, não havia dados o bastante para inutilizar uma quantidade suficiente de medidas iniciais.

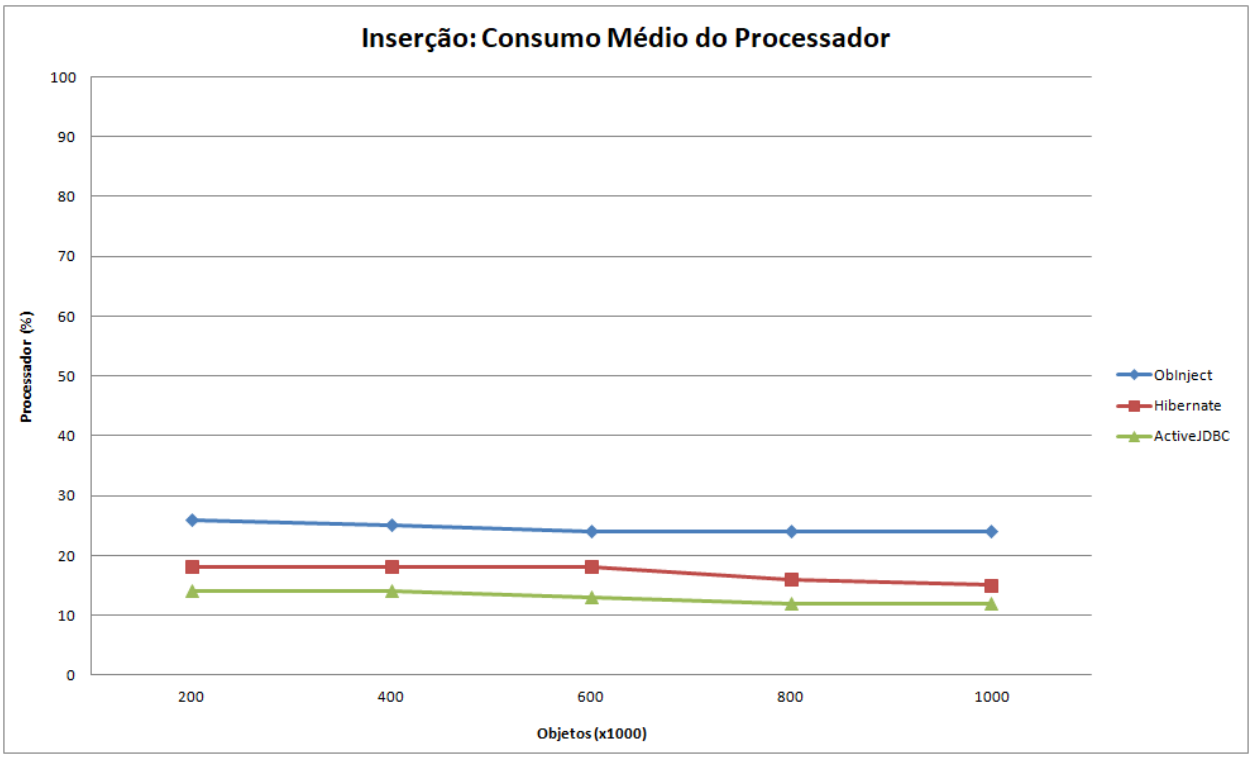

Figura 4.8: Consumo Médio do Processador pelos *Frameworks* na Inserção

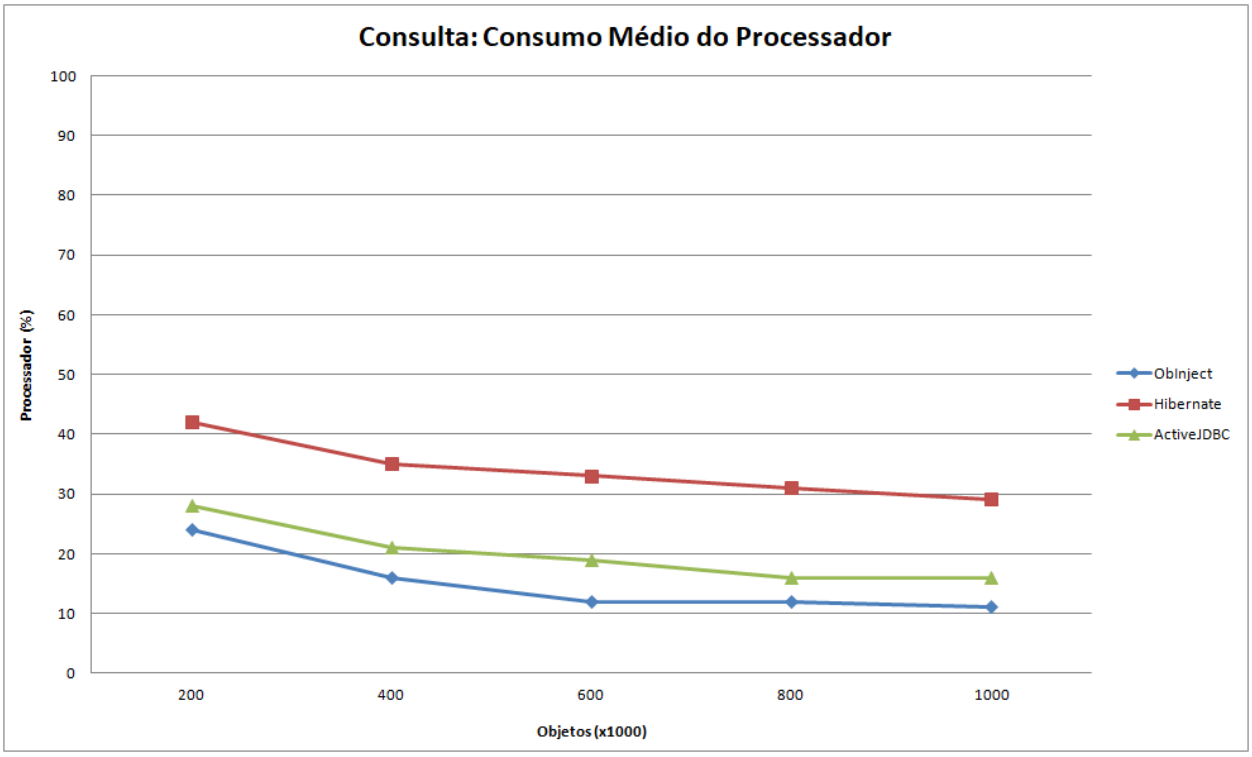

Figura 4.9: Consumo Médio do Processador pelos *Frameworks* na Consulta

4.2.4 Disco

A modelagem das classes de usuário e tabelas dos *frameworks* ActiveJDBC e Hibernate foi pensada com o objetivo de garantir que o mapeamento objeto-relacional de ambos fosse equivalente. Por consequência, o tamanho em disco ocupado pelos objetos persistidos por ambos os *frameworks* é idêntico. Por esse motivo, ambos estão representados no gráfico 4.10 pelo elemento *MySQL*.

O *framework* ObInject armazena todas as informações persistentes em um único arquivo, de extensão *dbo*. O sistema de gerenciamento de banco de dados relacional MySQL, por sua vez, armazena as informações persistentes de um determinado banco de dados em um diretório composto por diferentes arquivos e extensões.

A quantidade de espaço em disco alocada pelo *framework* ObInject para armazenar os objetos persistidos foi sempre superior à quantidade alocada pelo SGBDR MySQL em todos os cinco experimentos realizados. O espaço em disco alocado pelo ObInject foi em média 47% maior em relação ao espaço alocado pelo MySQL.

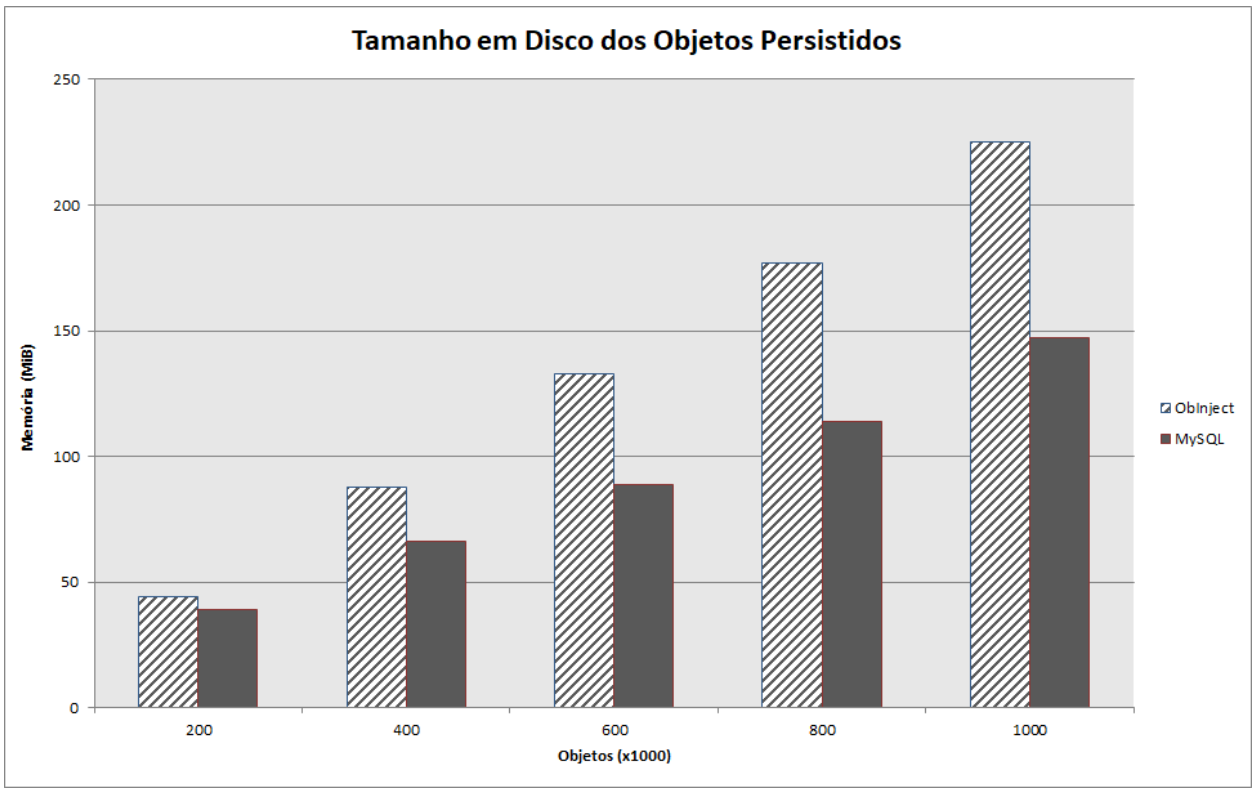

Figura 4.10: Espaço em Disco Ocupado pelos Objetos Persistidos

4.2.5 Medições Adicionais

Os resultados apresentados nesta seção têm por objetivo traçar uma comparação de desempenho entre as operações de consulta e inserção do *framework* ObInject. As medidas foram coletadas utilizando a instrumentação presente no *framework*, exposta através da classe *AveragePerformance*.

As operações de inserção realizadas no experimento persistem todos os objetos descritos na tabela 4.1. As operações de consulta foram realizadas utilizando a mesma tabela como referência, ou seja, todos os objetos inseridos são também buscados.

A média de acessos a disco é uma medida de desempenho que representa o número de vezes em que um bloco deve ser recuperado do disco pelas estruturas do *framework* dividido pelo número total de medições realizadas. As medições de acessos a disco foram realizadas utilizando um tamanho de bloco de 4 KB, assim como todos os outros experimentos deste trabalho.

A média de verificações é uma medida de desempenho que representa o número de vezes em que foi realizada uma operação de encaminhamento nas estruturas do *framework* dividido pelo número total de medições realizadas. A operação de encaminhamento é normalmente a operação de maior custo computacional da estrutura. Apenas a árvore B+ está presente nos experimentos deste trabalho, sendo sua operação de encaminhamento uma comparação entre atributos ordenáveis e índices.

A figura 4.11 indica a média de acessos a disco do *framework* ObInject nas operações de inserção e consulta em cada um dos cinco experimentos realizados. Ao analisar os dados presentes na figura, é possível concluir que a operação de consulta do *framework* ObInject realiza menos acessos a disco que a operação de inserção.

Já a figura 4.12 representa a média de verificações realizadas pelo *framework* ObInjejct nas operações de consulta e inserção. Os dados representados na figura indicam que a operação de consulta do ObInject executa menos verificações que a operação de inserção quando ambas são realizadas em um mesmo conjunto de objetos.

A análise dos dados apresentados ajuda a confirmar a explicação oferecida na seção 4.2.3 sobre a discrepância entre as medidas de consumo percentual médio do processador nas operações de inserção e consulta. Os dados comprovam que a operação de inserção executa um número maior de operações computacionalmente custosas se comparada à operação de consulta.

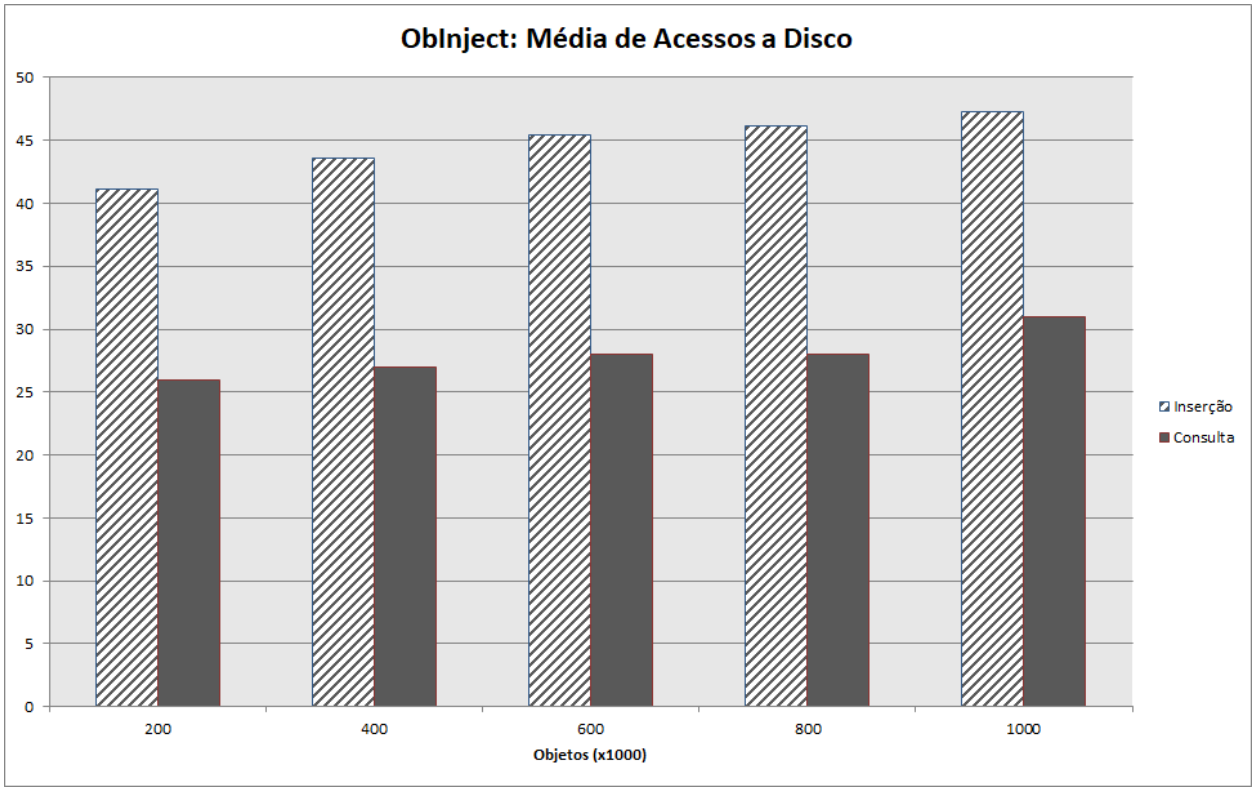

Figura 4.11: Média de Acessos a Disco do ObInject

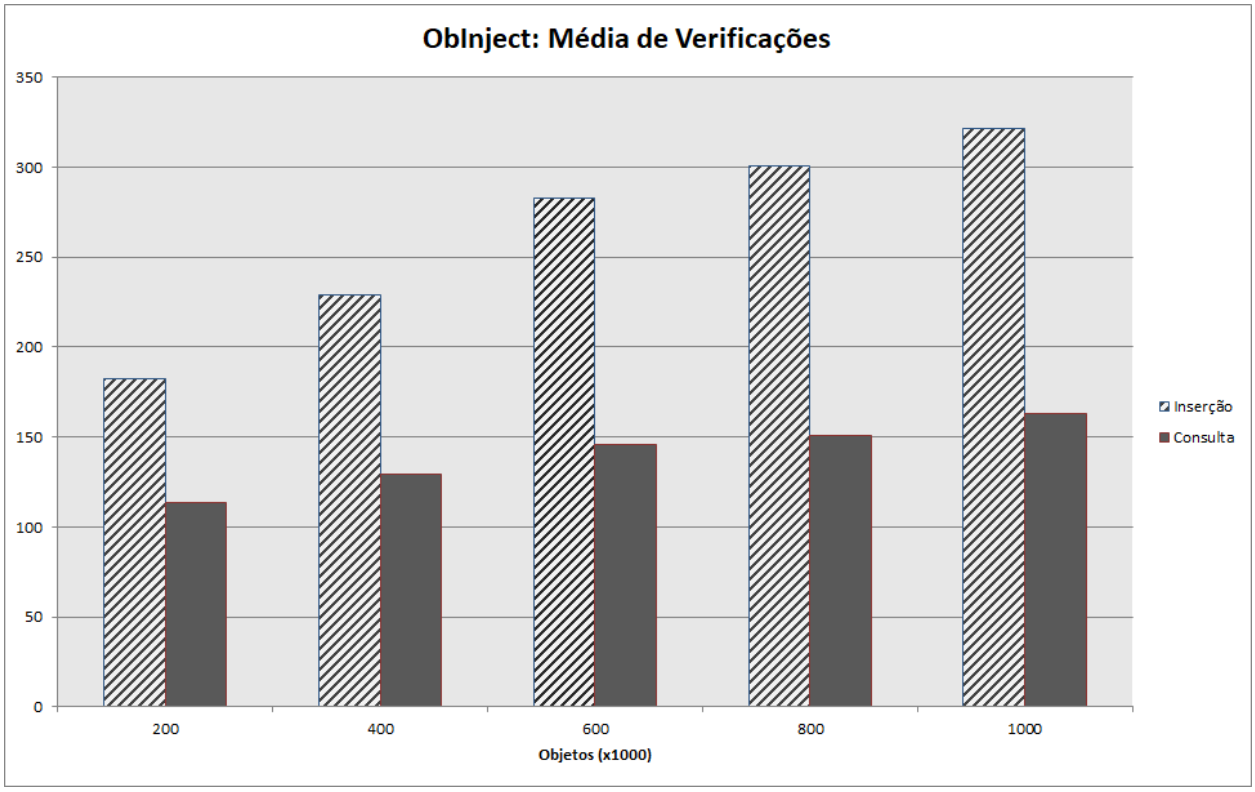

Figura 4.12: Média de Verificações do ObInject

4.3 Considerações Finais

Neste capítulo foram apresentados os resultados dos experimentos de inserção e comparação realizados com os *frameworks* ObInject, Hibernate e ActiveJDBC. Os experimentos tiveram como objetivo realizar a comparação de desempenho entre os diferentes *frameworks*, com foco em quatro características principais: Tempo, Memória, Processador e Disco.

O *framework* ObInject se mostrou superior aos demais nos experimentos relativos ao tempo, possuindo os menores tempos de execução por uma margem significativa tanto nas operações de inserção como nas operações de consulta.

Os experimentos relativos ao consumo de memória retrataram um comportamento muito semelhante entre os *frameworks* ObInject e ActiveJDBC em todas as operações de inserção e consulta realizadas. O *framework* Hibernate teve o pior desempenho nessa categoria, especialmente no experimentos finais onde mais objetos foram utilizados.

O consumo médio do processador pelo *framework* ObInject foi o mais alto nos experimentos de inserção e o mais baixo nos experimentos de consulta. Isso se deve à diferença de complexidade entre os algoritmos de busca e inserção de suas árvores B+. O *framework* ActiveJDBC demonstrou um menor consumo do processador em relação ao *framework* Hibernate em todos os experimentos realizados nessa categoria.

O espaço em disco alocado para armazenar os objetos persistidos pelos *frameworks* Hibernate e ActiveJDBC foi menor que o espaço alocado pelo *framework* ObInject em todos os cinco experimentos de inserção realizados.

É importante salientar que os recursos alocados pelo sistema de gerenciamento de banco de dados MySQL, utilizado pelos *frameworks* Hibernate e ActiveJDBC, não foram contabilizados nestes experimentos. Caso a utilização de tais recursos pelo SGBD fosse interpretada como recursos utilizados pelos próprios *frameworks*, o consumo de memória e processador de ambos, medidos pelos experimentos deste capítulo, se tornaria ainda maior.

Por fim, as informações adicionais coletadas sobre o *framework* ObInject permitem traçar uma comparação direta entre as operações de consulta e inserção. As medições obtidas podem ser referenciadas por trabalhos futuros para verificar o impacto de uma mudança estrutural do *framework* em seu desempenho.

Capítulo 5

Conclusão

A linguagem de consulta do ObInject (*ObInject Query Language* - OIQL) é uma linguagem baseada em métodos que representou um grande avanço na usabilidade do *framework*, visto que os detalhes internos das estruturas de dados responsáveis pela persistência e indexação dos dados são abstraídos ao se realizar uma consulta através da linguagem. O fato da linguagem utilizar muitos dos verbos SQL na sua própria sintaxe diminui consideravelmente a barreira de entrada para a adoção do *framework* em larga escala. Todavia, grande parte dos operadores da linguagem OIQL estão apenas parcialmente implementados, o que dificulta sua adoção imediata.

Este trabalho apresentou como contribuição principal a implementação dos operadores de projeção e seleção da linguagem de consulta OIQL. O operador de projeção foi implementado através do método *select* e seu uso permite ao usuário da linguagem retornar apenas os atributos da consulta relevantes à aplicação. Já o operador de seleção foi implementado através do método *where* e suas cláusulas condicionais, tornando possível a filtragem dos dados retornados por uma consulta de acordo com um critério especificado pela aplicação.

As contribuições secundárias realizadas por este trabalho incluem, mas não se limitam a:

- **Operador de Conjunção**: O operador lógico de conjunção foi implementado através do método *and* da linguagem de consulta OIQL. Seu uso torna possível realizar a interseção entre os resultados parciais das cláusulas condicionais na função *where*;
- **Operador de Disjunção**: O operador lógico de disjunção foi implementado através do método *or* da linguagem de consulta OIQL. Seu uso torna possível realizar a união entre os resultados parciais das cláusulas condicionais na função *where*;
- **Suporte à Alteração de Objetos Persistentes**: A alteração de um objeto e seus índices nas estruturas de dados do *framework* ObInject passou a ser possível através do método *inject*;

- **Suporte à Remoção de Objetos Persistentes**: A remoção de um objeto e seus índices das estruturas de dados do *framework* ObInject passou a ser possível através do método *reject*;
- **Associações entre Objetos Persistidos**: As entidades foram modificadas para acomodar a persistência e a busca de associações entre objetos de diferentes classes.

Os experimentos realizados ao final do trabalho traçaram uma comparação de desempenho entre o ObInject e os *frameworks* Hibernate e ActiveJDBC através de medições relativas ao tempo, memória, processador e disco em operações de inserção e consulta de objetos. Os recursos alocados pelo SGBD MySQL não foram contabilizados nos resultados. O ObInject mostrou ser o mais rápido dos *frameworks* analisados por uma margem significativa tanto nas operações de inserção como nas operações de consulta. Seu consumo de memória foi bem próximo ao consumo do ActiveJDBC, sendo o consumo de ambos inferior ao do Hibernate em ambas as operações. No quesito processador, o ObInject teve o pior desempenho na inserção e o melhor desempenho na busca. Por fim, o *framework* ObInject apresentou o maior consumo de disco entre os *frameworks* para armazenar os objetos persistidos.

As informações adicionais coletadas sobre o ObInject durante os experimentos em conjunto com as comparações realizadas sugerem que as aplicações que mais irão se beneficiar com a adoção do *framework* são aquelas em que as consultas ocorrem com muito mais frequência que as inserções, dado o maior desempenho de suas operações de consulta nos experimentos.

5.1 Trabalho Futuros

Os seguintes tópicos relacionados a este trabalho podem ser explorados em possíveis trabalhos futuros:

- **Testes de usabilidade da linguagem OIQL.** Testes de usabilidade da linguagem de consulta do *framework* ObInject podem ser realizados através de experimentos com um ou mais grupos de controle que comparem diversos parâmetros no uso da linguagem OIQL com o uso de outras linguagens de consulta populares, como as linguagens SQL e HQL;
- **Otimização no tratamento de múltiplas expressões condicionais.** O tratamento de múltiplas expressões condicionais foi uma das contribuições deste trabalho, porém é feito de forma pouco eficiente. Uma nova estratégia para a concatenação de expressões condicionais diretamente nas estruturas de dados do *framework* pode ser implementada para otimizar as buscas que envolvem mais de uma expressão condicional;
- **Implementação do operador de junção na linguagem OIQL.** O operador de junção deve ser implementado através do método *from* da linguagem de consulta OIQL.

Tal operador é essencial para se obter uma linguagem de consulta capaz de realizar consultas mais avançadas;

- **Implementação do operador de agrupamento na linguagem OIQL.** O operador de agrupamento é o responsável por agrupar resultados parciais de uma consulta, devendo ser implementado através do método *groupBy* da linguagem de consulta OIQL. É preciso que exista suporte a uma nova operação de seleção após o agrupamento, através da implementação do método *having*;

- **Implementação da ordenação na linguagem OIQL.** A ordenação dos resultados de uma consulta gerada pela linguagem OIQL deve ser definida através do método *orderBy*.

Referências Bibliográficas

ARNOLD, K.; GOSLING, J.; HOLMES, D. **The Java Programming Language**. 3rd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. 624 p.

ASTRAHAN, M. M.; BLASGEN, M. W.; CHAMBERLIN, D. D.; ESWARAN, K. P.; GRAY, J. N.; GRIFFITHS, P. P.; KING, W. F.; LORIE, R. A.; MCJONES, P. R.; MEHL, J. W.; PUTZOLU, G. R.; TRAIGER, I. L.; WADE, B. W.; WATSON, V. System R: relational approach to database management. **ACM Transactions on Database System**, ACM, New York, NY, USA, v. 1, n. 2, p. 97–137, jun. 1976.

BARRY, D.; STANIENDA, T. Solving the java object storage problem. **IEEE Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 31, n. 11, p. 33–40, nov. 1998.

BAUER, C. **Java Persistence with Hibernate**. Greenwich, CT, USA: Manning Publications Co., 2006. 904 p.

BAUER, C.; KING, G. **Hibernate in action**. Greenwich, CT, USA: Manning Publications, 2004. 408 p. (In Action Series).

CARVALHO, L. O. **Object-Injection: Um Framework de Indexação e Persistência**. Dissertação (Mestrado) — Univeridade Federal de Itajubá, Brazil, 2013.

CATTELL, R. G. G.; BARRY, D. K. **The object data standard: ODMG 3.0**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann, 2000. 288 p.

CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, R. E. Bigtable: A distributed storage system for structured data. In: **Proceedings of 7th Symposium on Operating System Design and Implementation**. Seattle, WA, USA: USENIX Association, 2006. (OSDI '06), p. 1–14.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: **Proceedings of 23rd International Conference on Very Large Data Bases**. Athens, Greece: Morgan Kaufmann Publishers Inc., 1997. (VLBD '97), p. 426–435.

CODD, E. F. A relational model of data for large shared data banks. **Communications of the ACM**, ACM, New York, NY, USA, v. 13, n. 6, p. 377–387, jun. 1970.

COMER, D. The ubiquitous B-tree. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 11, n. 2, p. 121–137, jun. 1979.

CORREIA, H. A. **Encadeamento de Anotações e Indexação de Imagens**. Dissertação (Mestrado) — Universidade Federal de Itajubá, Brazil, 2017.

DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; VOGELS, W. Dynamo: amazon's highly available key-value store. In: **Proceedings of 21st ACM Symposium on Operating Systems Principles**. Stevenson, Washington, USA: ACM, 2007. (SOSP '07), p. 205–220.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented application frameworks. **Communications of the ACM**, ACM, New York, NY, USA, v. 40, n. 10, p. 32–38, out. 1997.

FERRO, C. **ObInject Query Language**. Dissertação (Mestrado) — Universidade Federal de Itajubá, Brazil, 2012.

FOWLER, M. **Patterns of Enterprise Application Architecture**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. 576 p.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Boston, MA, USA: Addison-Wesley, 1995. 364 p.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book**. 2nd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. 1241 p.

GARZA, J. F.; KIM, W. Transaction management in an object-oriented database system. **SIGMOD Rec.**, ACM, New York, NY, USA, v. 17, n. 3, p. 37–45, jun. 1988.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. **ACM SIGMOD Record**, ACM, New York, NY, USA, v. 14, n. 2, p. 47–57, jun. 1984.

ISO/IEC. Norm 9834-8, **Procedures for the operation of OSI Registration Authorities**. Genève, Switzerland: International Organization for Standardization / International Electrotechnical Commission, 2008.

JSR-220. **Java Persistence API: Enterprise JavaBeans**. 2012. Disponível em: <`www.jcp.org/en/jsr/detail?id=220`>. Acesso em: 11 de fevereiro de 2012.

LEACH, P.; MEALLING, M.; SALZ, R. **A Universally Unique Identifier (UUID) URN Namespace**. Internet Engineering Task Force - IETF, 2005. RFC 4122. Disponível em: <`http://www.ietf.org/rfc/rfc4122.txt`>. Acesso em: 25 de maio de 2012.

MAIER, D.; STEIN, J.; OTIS, A.; PURDY, A. Development of an object-oriented dbms. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 21, n. 11, p. 472–482, jun. 1986.

MATTSSON, M.; BOSCH, J. Stability assessment of evolving industrial object-oriented frameworks. **Journal of Software Maintenance**, v. 12, n. 2, p. 79–102, 2000.

OLIVEIRA, M. F. **Metaprogramação e Metadados de Persistência e Indexação para o Framework Object-Injection**. Dissertação (Mestrado) — Universidade Federal de Itajubá, Brazil, 2012.

RAMAKRISHNAN, R.; GEHRKE, J. **Database Management Systems**. 2nd. ed. USA: McGraw-Hill, 1999. 931 p.

RICHARDSON, C. Untangling enterprise java. **Queue**, ACM, New York, NY, USA, v. 4, n. 5, p. 36–44, jun. 2006.

SHEARD, T. Accomplishments and research challenges in meta-programming. In: **Proceedings of the 2nd International Conference on Semantics, Applications, and Implementation of Program Generation**. Berlin, Heidelberg: Springer-Verlag, 2001. (SAIG'01), p. 2–44.

THOMAS, D.; HANSSON, D.; BREEDT, L.; CLARK, M.; DAVIDSON, J. D.; GEHTLAND, J.; SCHWARZ, A. **Agile Web Development with Rails**. Raleigh, NC, USA: Pragmatic Bookshelf, 2006. 720 p.

TIOBE. **TIOBE Programming Community index**. 2017. Disponível em: <`http://www.tiobe.com/tiobe-index`>. Acesso em: 25 de fevereiro de 2017.

Apêndice A

Tabela de Funções da Linguagem HQL

Função	**Finalidade**
`COUNT(r)`	Retorna o número total de valores da agregação `r`
`MIN(r)`	Retorna o menor valor da agregação `r`
`MAX(r)`	Retorna o maior valor da agregação `r`
`SUM(r)`	Retorna a soma dos valores da agregação `r`
`AVG(r)`	Retorna a média dos valores da agregação `r`
`UPPER(s), LOWER(s)`	Retorna a cadeia de caracteres `s` em caixa alta e caixa baixa, respectivamente
`CONCAT(s1, s2)`	Retorna as cadeias de caracteres `s1` e `s2` concatenadas
`SUBSTRING(s, offset, length)`	Retorna parte da cadeia de caracteres `s` especificada pelo deslocamento `offset` e tamanho `length`
`TRIM( [[BOTH|LEADING|TRAILING] char [FROM]] s)`	Remove todos os caracteres *char* nos lados especificados da cadeia de caracteres `s`. Remove espaços em branco de ambos os lados por padrão
`LENGTH(s)`	Retorna o número de caracteres da cadeia `s`
`LOCATE(s1, s2, o)`	Localiza a cadeia de caracteres `s1` dentro da cadeia de caracteres `s2`, buscando a partir da posição `o`
`ABS(n)`	Retorna o valor absoluto do número `n`
`SQRT(n)`	Retorna a raiz quadrada do número `n`
`MOD(n1, n2)`	Retorna o resto da divisão entre `n1` e `n2`
`SIZE(c)`	Retorna o número de elementos da coleção `c`
`BIT_LENGTH(s)`	Retorna o número de *bits* em `s`
`CAST(t as Type)`	Realiza o *casting* de um dado tipo `t` para o tipo `Type`
`CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP()`	Retorna a data e/ou a hora como reportados pelo Sistema de Gerenciamento de Banco de Dados
`SECOND(d), MINUTE(d), HOUR(d), DAY(d), MONTH(d), YEAR(d)`	Extrai a hora e/ou a data do argumento temporal `d`
`MAXELEMENT(c), MINELEMENT(c), ELEMENTS(c)`	Retorna os elementos e/ou os índices da coleção indexada c
`MAXINDEX(c), MININDEX(c), INDEX(c), INDICES(c)`	Retorna os índices da coleção indexada c

Apêndice B

Tabela de Elementos do Esquema XML do Framework Hibernate

Elemento	**Finalidade**
`<Doctype>`	Define a sintaxe utilizada pelo arquivo XML através de uma Definição de Tipo de Documento (*Document Type Definition - DTD*)
`<Hibernate-mapping>`	Permite configurar algumas opções do mapeamento através de seus atributos, como o prefixo de pacote
`<Class>`	Declara classes persistentes
`<Subclass>`	Declara classes polimórficas persistentes, onde tanto a classe mãe quanto as classes filhas são mapeadas por uma única tabela
`<Joined-subclass>`	Declara classes polimórficas persistentes, onde ambas as classes mãe e filhas são mapeadas por suas próprias tabelas
`<Union-subclass>`	Declara classes polimórficas persistentes, onde apenas as classes filhas são mapeadas em tabelas
`<id>`	Declara a propriedade que representa a chave primária da tabela
`<Discriminator>`	Define uma coluna que contém valores de marcação responsáveis em dizer à camada de persistência qual subclasse instanciar quando se utiliza o elemento `<subclass>`
`<Property>`	Declara uma propriedade persistente de uma classe
`<Properties>`	Declara o agrupamento de propriedades de uma classe
`<One-to-one>`	Define uma associação um-para-um
`<Many-to-one>`	Define uma associação muitos-para-um
`<Join>`	Mapeia propriedades de uma classe para tabelas que possuem associação um-para-um
`<Key>`	Define a chave estrangeira da tabela de junção
`<Import>`	Importa classes e interfaces para serem utilizadas na linguagem de consulta
`<Any>`	Define uma associação polimórfica entre classes de múltiplas tabelas
`<Version>`	Elemento opcional. Indica que a tabela possui dados versionados
`<Timestamp>`	Elemento opcional. Indica que a tabela possui dados temporais

Apêndice C

Classe de Empacotamento de Persistência da Classe de Usuário Map

```
1  public class $Map extends Map implements Entity<$Map> {
2
3      protected Uuid uuid;
4      private static Uuid classId;
5
6      public static Uuid getClassId() {
7          if ($Map.classId == null) {
8              $Map.classId = Uuid.fromString("9F5DC12A-33F3-7B26-3A5F-49071
                   B1BC396");
9          }
10         return $Map.classId;
11     }
12     public static final BTreeEntity<$Map> entityStructure = new
           BTreeEntity<$Map>(new File("build/classes/obinject/terrain/terrain.
           dbo", 4096)) {
13     };
14     public static final BTree<UniqueOneMap> uniqueOneMapStructure = new
           BTree<UniqueOneMap>(new File("build/classes/obinject/terrain/
           terrain.dbo", 4096)) {
15     };
16     public static final Attribute<java.lang.String> name = new Attribute<
           java.lang.String>() {
17         @Override
18         public java.lang.String valueOfAttribute(Entity entity) {
19             return ((Map) entity).getName();
20         }
21     };
22     public static final Attribute<java.lang.String> description = new
           Attribute<java.lang.String>() {
23         @Override
```

```
24          public java.lang.String valueOfAttribute(Entity entity) {
25              return ((Map) entity).getDescription();
26          }
27      };
28      public static final Attribute<java.util.List> terrains = new Attribute
            <java.util.List>() {
29          @Override
30          public java.util.List valueOfAttribute(Entity entity) {
31              return ((Map) entity).getTerrains();
32          }
33      };
34
35      public $Map() {
36          this.uuid = Uuid.generator();
37      }
38
39      public $Map(Map obj) {
40          this.setName(obj.getName());
41          this.setDescription(obj.getDescription());
42          this.setTerrains(obj.getTerrains());
43          this.uuid = Uuid.generator();
44      }
45
46      public $Map(Map obj, Uuid uuid) {
47          this.setName(obj.getName());
48          this.setDescription(obj.getDescription());
49          this.setTerrains(obj.getTerrains());
50          this.uuid = uuid;
51      }
52
53      public $Map($Map obj) {
54          this.setName(obj.getName());
55          this.setDescription(obj.getDescription());
56          this.setTerrains(obj.getTerrains());
57          this.uuid = obj.getUuid();;
58      }
59
60      public $Map(Uuid uuid) {
61          this.uuid = uuid;
62      }
63      protected java.util.List<Uuid> uuidTerrains = new java.util.ArrayList
            <>();
64
65      @Override
66      public java.util.List<obinject.terrain.Terrain> getTerrains() {
67          java.util.List<obinject.terrain.Terrain> superTerrains = super.
                getTerrains();
68          if ((superTerrains.isEmpty()) && (!uuidTerrains.isEmpty())) {
```

```
69              for (Uuid uuid : uuidTerrains) {
70                  superTerrains.add($Terrain.entityStructure.find(uuid));
71              }
72          }
73          return superTerrains;
74      }
75
76      private void resetUuidTerrains() {
77          uuidTerrains.clear();
78          if (this.getTerrains() != null) {
79              for (Terrain obj : this.getTerrains()) {
80                  if (obj instanceof Entity) {
81                      uuidTerrains.add(((Entity) obj).getUuid());
82                  } else {
83                      $Terrain entity = new $Terrain(obj);
84                      uuidTerrains.add($Terrain.find(entity));
85                      if (uuidTerrains == null) {
86                          throw new TransientObjectException("Map", "
                                terrains", "Terrain");
87                      }
88                  }
89              }
90          }
91      }
92
93      @Override
94      public boolean isEqual($Map obj) {
95          return (((this.getName() == null) && (obj.getName() == null)) ||
                ((this.getName() != null) && (obj.getName() != null) && (this.
                getName().equals(obj.getName())))) && (((this.getDescription()
                == null) && (obj.getDescription() == null)) || ((this.
                getDescription() != null) && (obj.getDescription() != null) &&
                (this.getDescription().equals(obj.getDescription())))));
96      }
97
98      @Override
99      public Uuid getUuid() {
100         return this.uuid;
101     }
102
103     @Override
104     public EntityStructure<$Map> getEntityStructure() {
105         return entityStructure;
106     }
107
108     @Override
109     public boolean inject() {
110         Uuid uuidInject = $Map.find(this);
```

```
111         if (uuidInject == null) {
112             resetUuidTerrains();
113             $Map.entityStructure.add(this);
114             UniqueOneMap.uniqueOneMapStructure.add(new UniqueOneMap(this,
                    this.getUuid()));
115             return true;
116         } else {
117             this.uuid = uuidInject;
118             return false;
119         }
120     }
121
122     @Override
123     public boolean reject() {
124         Uuid uuidReject = $Map.find(this);
125         if (uuidReject != null) {
126             UniqueOneMap.uniqueOneMapStructure.remove(new UniqueOneMap(
                    this, this.getUuid()));
127             $Map.entityStructure.remove(this);
128             return true;
129         } else {
130             return false;
131         }
132     }
133
134     @Override
135     public boolean modify() {
136         Uuid uuidOld = $Map.find(this);
137         $Map entityOld = $Map.entityStructure.find(uuidOld);
138         if (entityOld != null) {
139             resetUuidTerrains();
140             UniqueOneMap uniqueOneMapOld = new UniqueOneMap(entityOld,
                    entityOld.getUuid());
141             UniqueOneMap uniqueOneMapNew = new UniqueOneMap(this,
                    entityOld.getUuid());
142             if (uniqueOneMapOld.hasSameKey(uniqueOneMapNew)) {
143                 $Map.uniqueOneMapStructure.remove(uniqueOneMapOld);
144                 $Map.uniqueOneMapStructure.add(uniqueOneMapNew);
145             }
146             this.uuid = uuidOld;
147             $Map.entityStructure.remove(entityOld);
148             $Map.entityStructure.add(this);
149             return true;
150         } else {
151             return false;
152         }
153     }
154
```

```
155     public static Uuid find($Map entity) {
156         UniqueOneMap unique = new UniqueOneMap(entity, entity.getUuid());
157         return $Map.uniqueOneMapStructure.find(unique);
158     }
159
160     @Override
161     public boolean pullEntity(byte[] array, int position) {
162         PullPage pull = new PullPage(array, position);
163         Uuid storedClass = pull.pullUuid();
164         if ($Map.classId.equals(storedClass) == true) {
165             uuid = pull.pullUuid();
166             this.setName(pull.pullString());
167             this.setDescription(pull.pullString());
168             int totalTerrains = pull.pullInteger();
169             for (int i = 0; i < totalTerrains; i++) {
170                 this.uuidTerrains.add(pull.pullUuid());
171             }
172             return true;
173         }
174         return false;
175     }
176
177     @Override
178     public void pushEntity(byte[] array, int position) {
179         PushPage push = new PushPage(array, position);
180         push.pushUuid($Map.classId);
181         push.pushUuid(uuid);
182         push.pushString(this.getName());
183         push.pushString(this.getDescription());
184         push.pushInteger(this.uuidTerrains.size());
185         for (Uuid uuidPush : this.uuidTerrains) {
186             push.pushUuid(uuidPush);
187         }
188     }
189
190     @Override
191     public int sizeOfEntity() {
192         return Page.sizeOfUuid + Page.sizeOfUuid + Page.sizeOfString(this.
                getName()) + Page.sizeOfString(this.getDescription()) + Page.
                sizeOfEntityCollection(this.uuidTerrains);
193     }
194
195     static {
196         name.getSchemas().add(new Schema<$Map, UniqueOneMap, java.lang.
                String>() {
197             @Override
198             public $Map newEntity(java.lang.String value) {
199                 $Map obj = new $Map();
```

```
200                 obj.setName(value);
201                 return obj;
202             }
203
204             @Override
205             public UniqueOneMap newKey(java.lang.String value) {
206                 UniqueOneMap obj = new UniqueOneMap();
207                 obj.setName(value);
208                 return obj;
209             }
210
211             @Override
212             public EntityStructure<$Map> getEntityStructure() {
213                 return $Map.entityStructure;
214             }
215
216             @Override
217             public KeyStructure<UniqueOneMap> getKeyStructure() {
218                 return UniqueOneMap.uniqueOneMapStructure;
219             }
220         });
221     }
222 }
```

Apêndice D

Classe de Empacotamento de Persistência da Classe de Usuário Terrain

```
1  public class $Terrain extends Terrain implements Entity<$Terrain> {
2
3      protected Uuid uuid;
4      private static Uuid classId;
5
6      public static Uuid getClassId() {
7          if ($Terrain.classId == null) {
8              $Terrain.classId = Uuid.fromString("6DA4D3C7-A5C5-ACE1-E666-
                   BFDC8CE8579A");
9          }
10         return $Terrain.classId;
11     }
12     public static final BTreeEntity<$Terrain> entityStructure = new
           BTreeEntity<$Terrain>(new File("build/classes/obinject/terrain/
           terrain.dbo", 4096)) {
13     };
14     public static final BTree<SortOneTerrain> sortOneTerrainStructure =
           new BTree<SortOneTerrain>(new File("build/classes/obinject/terrain/
           terrain.dbo", 4096)) {
15     };
16     public static final BTree<UniqueOneTerrain> uniqueOneTerrainStructure
           = new BTree<UniqueOneTerrain>(new File("build/classes/obinject/
           terrain/terrain.dbo", 4096)) {
17     };
18     public static final RTree<RectangleOneTerrain>
           rectangleOneTerrainStructure = new RTree<RectangleOneTerrain>(new
           File("build/classes/obinject/terrain/terrain.dbo", 4096)) {
19     };
20     public static final MTree<EditionOneTerrain>
           editionOneTerrainStructure = new MTree<EditionOneTerrain>(new File(
```

```
        "build/classes/obinject/terrain/terrain.dbo", 4096)) {
21      };
22      public static final MTree<PointOneTerrain> pointOneTerrainStructure =
            new MTree<PointOneTerrain>(new File("build/classes/obinject/terrain
            /terrain.dbo", 4096)) {
23      };
24      public static final Attribute<Long> register = new Attribute<Long>() {
25          @Override
26          public Long valueOfAttribute(Entity entity) {
27              return ((Terrain) entity).getRegister();
28          }
29      };
30      public static final Attribute<java.lang.String> owner = new Attribute<
            java.lang.String>() {
31          @Override
32          public java.lang.String valueOfAttribute(Entity entity) {
33              return ((Terrain) entity).getOwner();
34          }
35      };
36      public static final Attribute<java.lang.String> city = new Attribute<
            java.lang.String>() {
37          @Override
38          public java.lang.String valueOfAttribute(Entity entity) {
39              return ((Terrain) entity).getCity();
40          }
41      };
42      public static final Attribute<java.lang.String> district = new
            Attribute<java.lang.String>() {
43          @Override
44          public java.lang.String valueOfAttribute(Entity entity) {
45              return ((Terrain) entity).getDistrict();
46          }
47      };
48      public static final Attribute<java.lang.String> street = new Attribute
            <java.lang.String>() {
49          @Override
50          public java.lang.String valueOfAttribute(Entity entity) {
51              return ((Terrain) entity).getStreet();
52          }
53      };
54      public static final Attribute<Float> propertyValue = new Attribute<
            Float>() {
55          @Override
56          public Float valueOfAttribute(Entity entity) {
57              return ((Terrain) entity).getPropertyValue();
58          }
59      };
```

```
60      public static final Attribute<Integer> number = new Attribute<Integer
            >() {
61          @Override
62          public Integer valueOfAttribute(Entity entity) {
63              return ((Terrain) entity).getNumber();
64          }
65      };
66      public static final Attribute<float[]> originCoordinate = new
            Attribute<float[]>() {
67          @Override
68          public float[] valueOfAttribute(Entity entity) {
69              return ((Terrain) entity).getOriginCoordinate();
70          }
71      };
72      public static final Attribute<float[]> extensionCoordinate = new
            Attribute<float[]>() {
73          @Override
74          public float[] valueOfAttribute(Entity entity) {
75              return ((Terrain) entity).getExtensionCoordinate();
76          }
77      };
78      public static final Attribute<obinject.terrain.Map> map = new
            Attribute<obinject.terrain.Map>() {
79          @Override
80          public obinject.terrain.Map valueOfAttribute(Entity entity) {
81              return ((Terrain) entity).getMap();
82          }
83      };
84
85      public $Terrain() {
86          this.uuid = Uuid.generator();
87      }
88
89      public $Terrain(Terrain obj) {
90          this.setRegister(obj.getRegister());
91          this.setOwner(obj.getOwner());
92          this.setCity(obj.getCity());
93          this.setDistrict(obj.getDistrict());
94          this.setStreet(obj.getStreet());
95          this.setPropertyValue(obj.getPropertyValue());
96          this.setNumber(obj.getNumber());
97          this.setOriginCoordinate(obj.getOriginCoordinate());
98          this.setExtensionCoordinate(obj.getExtensionCoordinate());
99          this.setMap(obj.getMap());
100         this.uuid = Uuid.generator();
101     }
102
103     public $Terrain(Terrain obj, Uuid uuid) {
```

```
104             this.setRegister(obj.getRegister());
105             this.setOwner(obj.getOwner());
106             this.setCity(obj.getCity());
107             this.setDistrict(obj.getDistrict());
108             this.setStreet(obj.getStreet());
109             this.setPropertyValue(obj.getPropertyValue());
110             this.setNumber(obj.getNumber());
111             this.setOriginCoordinate(obj.getOriginCoordinate());
112             this.setExtensionCoordinate(obj.getExtensionCoordinate());
113             this.setMap(obj.getMap());
114             this.uuid = uuid;
115         }
116 
117         public $Terrain($Terrain obj) {
118             this.setRegister(obj.getRegister());
119             this.setOwner(obj.getOwner());
120             this.setCity(obj.getCity());
121             this.setDistrict(obj.getDistrict());
122             this.setStreet(obj.getStreet());
123             this.setPropertyValue(obj.getPropertyValue());
124             this.setNumber(obj.getNumber());
125             this.setOriginCoordinate(obj.getOriginCoordinate());
126             this.setExtensionCoordinate(obj.getExtensionCoordinate());
127             this.setMap(obj.getMap());
128             this.uuid = obj.getUuid();;
129         }
130 
131         public $Terrain(Uuid uuid) {
132             this.uuid = uuid;
133         }
134         protected Uuid uuidMap;
135 
136         @Override
137         public Map getMap() {
138             Map superMap = super.getMap();
139             if (superMap == null && uuidMap != null) {
140                 superMap = $Map.entityStructure.find(uuidMap);
141                 this.setMap(superMap);
142             }
143             return superMap;
144         }
145 
146         private void resetUuidMap() {
147             if (this.getMap() != null) {
148                 if (this.getMap() instanceof Entity) {
149                     uuidMap = ((Entity) this.getMap()).getUuid();
150                 } else {
151                     $Map entity = new $Map(this.getMap());
```

```
152                 uuidMap = $Map.find(entity);
153                 if (uuidMap == null) {
154                     throw new TransientObjectException("Terrain", "map", "
                        Map");
155                 }
156             }
157         }
158     }
159
160     @Override
161     public boolean isEqual($Terrain obj) {
162         return (this.getRegister() == obj.getRegister()) && (((this.
            getOwner() == null) && (obj.getOwner() == null)) || ((this.
            getOwner() != null) && (obj.getOwner() != null) && (this.
            getOwner().equals(obj.getOwner())))) && (((this.getCity() ==
            null) && (obj.getCity() == null)) || ((this.getCity() != null)
            && (obj.getCity() != null) && (this.getCity().equals(obj.
            getCity()))))) && (((this.getDistrict() == null) && (obj.
            getDistrict() == null)) || ((this.getDistrict() != null) && (
            obj.getDistrict() != null) && (this.getDistrict().equals(obj.
            getDistrict()))))) && (((this.getStreet() == null) && (obj.
            getStreet() == null)) || ((this.getStreet() != null) && (obj.
            getStreet() != null) && (this.getStreet().equals(obj.getStreet
            ()))))) && (this.getPropertyValue() == obj.getPropertyValue())
            && (this.getNumber() == obj.getNumber());
163     }
164
165     @Override
166     public Uuid getUuid() {
167         return this.uuid;
168     }
169
170     @Override
171     public EntityStructure<$Terrain> getEntityStructure() {
172         return entityStructure;
173     }
174
175     @Override
176     public boolean inject() {
177         Uuid uuidInject = $Terrain.find(this);
178         if (uuidInject == null) {
179             resetUuidMap();
180             $Terrain.entityStructure.add(this);
181             SortOneTerrain.sortOneTerrainStructure.add(new SortOneTerrain(
                this, this.getUuid()));
182             UniqueOneTerrain.uniqueOneTerrainStructure.add(new
                UniqueOneTerrain(this, this.getUuid()));
```

```
183             RectangleOneTerrain.rectangleOneTerrainStructure.add(new
                    RectangleOneTerrain(this, this.getUuid()));
184             EditionOneTerrain.editionOneTerrainStructure.add(new
                    EditionOneTerrain(this, this.getUuid()));
185             PointOneTerrain.pointOneTerrainStructure.add(new
                    PointOneTerrain(this, this.getUuid()));
186             return true;
187         } else {
188             this.uuid = uuidInject;
189             return false;
190         }
191     }
192
193     @Override
194     public boolean reject() {
195         Uuid uuidReject = $Terrain.find(this);
196         if (uuidReject != null) {
197             SortOneTerrain.sortOneTerrainStructure.remove(new
                    SortOneTerrain(this, this.getUuid()));
198             UniqueOneTerrain.uniqueOneTerrainStructure.remove(new
                    UniqueOneTerrain(this, this.getUuid()));
199             RectangleOneTerrain.rectangleOneTerrainStructure.remove(new
                    RectangleOneTerrain(this, this.getUuid()));
200             EditionOneTerrain.editionOneTerrainStructure.remove(new
                    EditionOneTerrain(this, this.getUuid()));
201             PointOneTerrain.pointOneTerrainStructure.remove(new
                    PointOneTerrain(this, this.getUuid()));
202             $Terrain.entityStructure.remove(this);
203             return true;
204         } else {
205             return false;
206         }
207     }
208
209     @Override
210     public boolean modify() {
211         Uuid uuidOld = $Terrain.find(this);
212         $Terrain entityOld = $Terrain.entityStructure.find(uuidOld);
213         if (entityOld != null) {
214             resetUuidMap();
215             SortOneTerrain sortOneTerrainOld = new SortOneTerrain(
                    entityOld, entityOld.getUuid());
216             SortOneTerrain sortOneTerrainNew = new SortOneTerrain(this,
                    entityOld.getUuid());
217             if (sortOneTerrainOld.hasSameKey(sortOneTerrainNew)) {
218                 $Terrain.sortOneTerrainStructure.remove(sortOneTerrainOld)
                        ;
219                 $Terrain.sortOneTerrainStructure.add(sortOneTerrainNew);
```

```
220             }
221             UniqueOneTerrain uniqueOneTerrainOld = new UniqueOneTerrain(
                    entityOld, entityOld.getUuid());
222             UniqueOneTerrain uniqueOneTerrainNew = new UniqueOneTerrain(
                    this, entityOld.getUuid());
223             if (uniqueOneTerrainOld.hasSameKey(uniqueOneTerrainNew)) {
224                 $Terrain.uniqueOneTerrainStructure.remove(
                        uniqueOneTerrainOld);
225                 $Terrain.uniqueOneTerrainStructure.add(uniqueOneTerrainNew
                        );
226             }
227             RectangleOneTerrain rectangleOneTerrainOld = new
                    RectangleOneTerrain(entityOld, entityOld.getUuid());
228             RectangleOneTerrain rectangleOneTerrainNew = new
                    RectangleOneTerrain(this, entityOld.getUuid());
229             if (rectangleOneTerrainOld.hasSameKey(rectangleOneTerrainNew))
                     {
230                 $Terrain.rectangleOneTerrainStructure.remove(
                        rectangleOneTerrainOld);
231                 $Terrain.rectangleOneTerrainStructure.add(
                        rectangleOneTerrainNew);
232             }
233             EditionOneTerrain editionOneTerrainOld = new EditionOneTerrain
                    (entityOld, entityOld.getUuid());
234             EditionOneTerrain editionOneTerrainNew = new EditionOneTerrain
                    (this, entityOld.getUuid());
235             if (editionOneTerrainOld.hasSameKey(editionOneTerrainNew)) {
236                 $Terrain.editionOneTerrainStructure.remove(
                        editionOneTerrainOld);
237                 $Terrain.editionOneTerrainStructure.add(
                        editionOneTerrainNew);
238             }
239             PointOneTerrain pointOneTerrainOld = new PointOneTerrain(
                    entityOld, entityOld.getUuid());
240             PointOneTerrain pointOneTerrainNew = new PointOneTerrain(this,
                     entityOld.getUuid());
241             if (pointOneTerrainOld.hasSameKey(pointOneTerrainNew)) {
242                 $Terrain.pointOneTerrainStructure.remove(
                        pointOneTerrainOld);
243                 $Terrain.pointOneTerrainStructure.add(pointOneTerrainNew);
244             }
245             this.uuid = uuidOld;
246             $Terrain.entityStructure.remove(entityOld);
247             $Terrain.entityStructure.add(this);
248             return true;
249         } else {
250             return false;
251         }
```

```
252     }
253
254     public static Uuid find($Terrain entity) {
255         UniqueOneTerrain unique = new UniqueOneTerrain(entity, entity.
                getUuid());
256         return $Terrain.uniqueOneTerrainStructure.find(unique);
257     }
258
259     @Override
260     public boolean pullEntity(byte[] array, int position) {
261         PullPage pull = new PullPage(array, position);
262         Uuid storedClass = pull.pullUuid();
263         if ($Terrain.classId.equals(storedClass) == true) {
264             uuid = pull.pullUuid();
265             this.setRegister(pull.pullLong());
266             this.setOwner(pull.pullString());
267             this.setCity(pull.pullString());
268             this.setDistrict(pull.pullString());
269             this.setStreet(pull.pullString());
270             this.setPropertyValue(pull.pullFloat());
271             this.setNumber(pull.pullInteger());
272             this.setOriginCoordinate((float[]) pull.pullMatrix());
273             this.setExtensionCoordinate((float[]) pull.pullMatrix());
274             int totalMap = pull.pullInteger();
275             if (totalMap > 0) {
276                 this.uuidMap = pull.pullUuid();
277             }
278             return true;
279         }
280         return false;
281     }
282
283     @Override
284     public void pushEntity(byte[] array, int position) {
285         PushPage push = new PushPage(array, position);
286         push.pushUuid($Terrain.classId);
287         push.pushUuid(uuid);
288         push.pushLong(this.getRegister());
289         push.pushString(this.getOwner());
290         push.pushString(this.getCity());
291         push.pushString(this.getDistrict());
292         push.pushString(this.getStreet());
293         push.pushFloat(this.getPropertyValue());
294         push.pushInteger(this.getNumber());
295         push.pushMatrix(getOriginCoordinate());
296         push.pushMatrix(getExtensionCoordinate());
297         if (this.uuidMap != null) {
298             push.pushInteger(1);
```

```
299             push.pushUuid(this.uuidMap);
300         } else {
301             push.pushInteger(0);
302         }
303     }
304
305     @Override
306     public int sizeOfEntity() {
307         return Page.sizeOfUuid + Page.sizeOfUuid + Page.sizeOfLong + Page.
                sizeOfString(this.getOwner()) + Page.sizeOfString(this.getCity
                ()) + Page.sizeOfString(this.getDistrict()) + Page.sizeOfString
                (this.getStreet()) + Page.sizeOfFloat + Page.sizeOfInteger +
                Page.sizeOfMatrix(this.getOriginCoordinate()) + Page.
                sizeOfMatrix(this.getExtensionCoordinate()) + Page.sizeOfEntity
                (this.uuidMap);
308     }
309
310     static {
311         register.getSchemas().add(new Schema<$Terrain, UniqueOneTerrain,
                Long>() {
312             @Override
313             public $Terrain newEntity(Long value) {
314                 $Terrain obj = new $Terrain();
315                 obj.setRegister(value);
316                 return obj;
317             }
318
319             @Override
320             public UniqueOneTerrain newKey(Long value) {
321                 UniqueOneTerrain obj = new UniqueOneTerrain();
322                 obj.setRegister(value);
323                 return obj;
324             }
325
326             @Override
327             public EntityStructure<$Terrain> getEntityStructure() {
328                 return $Terrain.entityStructure;
329             }
330
331             @Override
332             public KeyStructure<UniqueOneTerrain> getKeyStructure() {
333                 return UniqueOneTerrain.uniqueOneTerrainStructure;
334             }
335         });
336         owner.getSchemas().add(new Schema<$Terrain, SortOneTerrain, java.
                lang.String>() {
337             @Override
338             public $Terrain newEntity(java.lang.String value) {
```

```
339                 $Terrain obj = new $Terrain();
340                 obj.setOwner(value);
341                 return obj;
342             }
343 
344             @Override
345             public SortOneTerrain newKey(java.lang.String value) {
346                 SortOneTerrain obj = new SortOneTerrain();
347                 obj.setOwner(value);
348                 return obj;
349             }
350 
351             @Override
352             public EntityStructure<$Terrain> getEntityStructure() {
353                 return $Terrain.entityStructure;
354             }
355 
356             @Override
357             public KeyStructure<SortOneTerrain> getKeyStructure() {
358                 return SortOneTerrain.sortOneTerrainStructure;
359             }
360         });
361         owner.getSchemas().add(new Schema<$Terrain, EditionOneTerrain,
                java.lang.String>() {
362             @Override
363             public $Terrain newEntity(java.lang.String value) {
364                 $Terrain obj = new $Terrain();
365                 obj.setOwner(value);
366                 return obj;
367             }
368 
369             @Override
370             public EditionOneTerrain newKey(java.lang.String value) {
371                 EditionOneTerrain obj = new EditionOneTerrain();
372                 obj.setOwner(value);
373                 return obj;
374             }
375 
376             @Override
377             public EntityStructure<$Terrain> getEntityStructure() {
378                 return $Terrain.entityStructure;
379             }
380 
381             @Override
382             public KeyStructure<EditionOneTerrain> getKeyStructure() {
383                 return EditionOneTerrain.editionOneTerrainStructure;
384             }
385         });
```

```
386         originCoordinate.getSchemas().add(new Schema<$Terrain,
                RectangleOneTerrain, float[]>() {
387             @Override
388             public $Terrain newEntity(float[] value) {
389                 $Terrain obj = new $Terrain();
390                 obj.setOriginCoordinate(value);
391                 return obj;
392             }
393
394             @Override
395             public RectangleOneTerrain newKey(float[] value) {
396                 RectangleOneTerrain obj = new RectangleOneTerrain();
397                 obj.setOriginCoordinate(value);
398                 return obj;
399             }
400
401             @Override
402             public EntityStructure<$Terrain> getEntityStructure() {
403                 return $Terrain.entityStructure;
404             }
405
406             @Override
407             public KeyStructure<RectangleOneTerrain> getKeyStructure() {
408                 return RectangleOneTerrain.rectangleOneTerrainStructure;
409             }
410         });
411         originCoordinate.getSchemas().add(new Schema<$Terrain,
                PointOneTerrain, float[]>() {
412             @Override
413             public $Terrain newEntity(float[] value) {
414                 $Terrain obj = new $Terrain();
415                 obj.setOriginCoordinate(value);
416                 return obj;
417             }
418
419             @Override
420             public PointOneTerrain newKey(float[] value) {
421                 PointOneTerrain obj = new PointOneTerrain();
422                 obj.setOriginCoordinate(value);
423                 return obj;
424             }
425
426             @Override
427             public EntityStructure<$Terrain> getEntityStructure() {
428                 return $Terrain.entityStructure;
429             }
430
431             @Override
```

```
432             public KeyStructure<PointOneTerrain> getKeyStructure() {
433                 return PointOneTerrain.pointOneTerrainStructure;
434             }
435         });
436         extensionCoordinate.getSchemas().add(new Schema<$Terrain,
                RectangleOneTerrain, float[]>() {
437             @Override
438             public $Terrain newEntity(float[] value) {
439                 $Terrain obj = new $Terrain();
440                 obj.setExtensionCoordinate(value);
441                 return obj;
442             }
443
444             @Override
445             public RectangleOneTerrain newKey(float[] value) {
446                 RectangleOneTerrain obj = new RectangleOneTerrain();
447                 obj.setExtensionCoordinate(value);
448                 return obj;
449             }
450
451             @Override
452             public EntityStructure<$Terrain> getEntityStructure() {
453                 return $Terrain.entityStructure;
454             }
455
456             @Override
457             public KeyStructure<RectangleOneTerrain> getKeyStructure() {
458                 return RectangleOneTerrain.rectangleOneTerrainStructure;
459             }
460         });
461     }
462 }
```