ObInject Query Language

Carlos Ferro

SERVIÇO DE PÓS-GRADUAÇÃO DA UNIFEI

Data de Depósito:

Assinatura: _____

ObInject Query Language [1]

Carlos Ferro

Orientador: *Prof. Dr. Enzo Seraphim*

Dissertação de Mestrado apresentada ao Instituto de Engenharia de Sistemas e Tecnologia da Informação (IESTI) da Universidade Federal de Itajubá (UNIFEI), para obtenção do título de Mestre em Ciência e Tecnologia da Computação.

UNIFEI - Itajubá - MG
Novembro de 2012

[1]Este trabalho conta com o apoio financeiro da CAPES

À minha amada família, que segue sempre ao meu lado.

Agradecimentos

Agradeço primeiramente aos meus pais, *Antônio Carlos Ferro* e *Rosa Emília Ferro*, pelo apoio incondicional e dedicação, por me proporcionarem a oportunidade da realização deste mestrado, por toda a paciência tida comigo e por me fazerem entender que o conhecimento é a melhor herança que se pode adquirir. Espero um dia poder retribuir pelo menos uma parcela do que vocês me deram.

Ao meu amigo, *Prof. Dr. Enzo Seraphim*, que me deu o privilégio de o ter como orientador no mestrado. A sua dedicação e paixão como professor e pesquisador me inspirou a continuar na areá acadêmica. Você é um exemplo a ser seguido e espero que um dia eu consiga ser um profissional tão bom quanto o senhor. Agradeço também à *Profª Drª Thatyana de Faria Piola Seraphim* por toda a paciência com as demoras e atrasos do *Enzo* por minha culpa.

A minha namorada, *Kátia Mendes de Barro*, pela compreensão, paciência e por todos puxões de orelha e palavras de ânimo ditas, sempre me ajudando a continuar.

A todos meus amigos, especialmente minha irmã *Aline Ferro* e meus companheiros de república, por estarem sempre presentes contribuindo para o meu crescimento pessoal e profissional, e por me fazerem companhia nas horas de diversão e de trabalho.

A *Profª M. Sc. Katia Cristina Lage dos Santos*, e a atual turma do 3º ano de Engenharia da Computação (ECO 2012), pela oportunidade do estágio de docência.

Ao *Prof. Dr. Otávio Augusto Salgado Carpinteiro* por sanar minhas dúvidas quanto ao curso e por sempre me atender de forma receptiva.

A todos os professores do programa de pós-graduação em Ciência e Tecnologia da Informação, por disseminarem seus conhecimentos entre nós alunos.

A todos os meus familiares, por sempre desejarem o melhor para mim e por todo o carinho que sempre me deram me ajudando ser quem sou hoje.

À CAPES, pelo apoio financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

RESUMO

O surgimento de bancos de dados revolucionou a maneira como dados são armazenados. Eles permitiram que uma enorme quantidade de dados fossem armazenadas em estruturas além de facilitar a sua manipulação. Junto aos bancos de dados, surgiram as linguagens de consulta. Estas linguagens transferiram ao banco de dados a tarefa de manipular as estruturas e consequentemente o desempenho da extração de dados. Mais recentemente, *frameworks* para persistência de dados se tornaram muito populares. Entre eles, o *framework* Object-Inject (CARVALHO et al., 2013) se mostrou bastante promissor para a persistência de objetos. Entretanto, este *framework* ainda não apresenta uma linguagem de consulta, sendo necessário a manipulação das estruturas para realizar a extração de dados. Este trabalho tem como objetivo definir uma linguagem de consulta para tal *framework*.

Abstract

The emergence of databases revolutionized the way data is stored. They allowed a huge amount of data to be stored in structures and facilitate their manipulation. Along the databases, query languages were created. These languages shift to the database the task of manipulating data structures and consequently the performance of data extraction. More recently, *frameworks* for data persistence have become very popular. Among them, the framework Object-Inject (CARVALHO et al., 2013) proved quite promising for object persistence. However, this *framework* does not present a query language, so the manipulation of structures to perform data extraction is required. This work aims to define a query language for such a *framework*.

Capítulo 1 Introdução

Até os primeiros anos da década de 60, a grande maioria dos dados eram guardadas em arquivos de texto. Estes arquivos geralmente eram formados por campos de tamanho fixo e o acesso a eles era simplesmente feito através da leitura e escrita. É claro que tal armazenamento é muito custoso e não proporciona quase nenhuma vantagem na organização e hierarquia destes dados.

Devido à crescente necessidade de otimização no armazenamento, o Departamento de Defesa dos EUA, em 1957, inaugurou a *Conference on Data Systems Languages* (Conferência sobre as Linguagens de Sistemas de Dados), ou CODASYL, para desenvolver linguagens de programação de computador. A CODASYL, famosa pela criação do COBOL, foi responsável pela criação do primeiro banco de dados moderno.

Em 1963, duas divisões do Departamento de Defesa dos EUA formaram a conferência intitulada "Desenvolvimento e Gerenciamento de um Banco de Dados para Computadores" onde o termo "banco de dados" foi concebido e definido como: Um conjunto de arquivos (tabelas), onde um arquivo é uma coleção ordenada de registros (linhas), e um registro consiste em uma ou mais chaves e dados.

Em 1965, a CODASYL formou um grupo chamado de *List Processing Task Force*, que tornou-se posteriormente o grupo *Data Base Task Group*. Este grupo emitiu um importante relatório em 1971 delineando o *Network Data Model*. Esse modelo de dados definiu vários conceitos importantes para bancos de dados, dentre eles uma linguagem de manipulação de dados.

Em 1966, membros da IBM, North American Rockwell e a Caterpillar Tractor juntaram-se para começar o projeto e desenvolvimento do Sistema de Controle de Informações - ICS da Linguagem de Dados/I (DL/I). O ICS era o bancos de dados, responsável pelo armazenamento e recuperação de dados, enquanto que o DL/I era responsável pela linguagem de consulta usada para acessar os dados.

Em 1970, Edgar Codd, publicou o artigo "Um modelo relacional de dados para grandes bancos de dados compartilhados", onde a ideia de modelo relacional foi introduzida. Nele, Codd apresentou uma linguagem simples de consulta de alto-nível para acesso a dados. Este modelo permitiria que

os desenvolvedores utilizassem operações em um conjunto completo de dados de uma única vez, ao invés de trabalhar com um registro de cada vez.

Alguns anos depois, a IBM desenvolveu o "System R", e juntamente criou a linguagem de consulta SQL, que se tornou a linguagem de consulta mais utilizada e um padrão para outras linguagens. Este sistema abriu portas para o desenvolvimento de vários outros banco de dados, que começaram a surgir na década de 80, dentre eles o SQL Server e MySQL.

Recentemente, o desenvolvimento de *frameworks* de persistência ganharam força para o armazenamento de dados. Entre eles está o *framework* Object-Inject (CARVALHO et al., 2013), criado para a indexação e persistência de objetos. Este *framework* faz uso de índices primários e secundários, indexando objetos em estruturas de dados. Três estrutura de dados estão presentes no *framework*, a *BTree*, a *RTree* e a *MTree*.

1.1 Motivação

Apesar do *framework* Object-Inject (CARVALHO et al., 2013) ter se mostrado bastante promissor, a aquisição dos dados armazenados é feita através da manipulação das estruturas de dados, tornando a consulta de dados uma tarefa difícil e ineficiente.

A criação de uma linguagem de consulta para o *framework* possibilitaria ao usuário uma forma simples e estruturada de obter dados da base.

1.2 Objetivo

O objetivo deste trabalho é a criação de uma linguagem de consulta para o *framework* Object-Inject. Esta linguagem transferirá a responsabilidade da manipulação de estruturas para a obtenção de dados para o próprio *framework*.

A criação desta linguagem também tornará o *framework* mais completo, facilitando o acesso aos dados, além de possibilitar a criação de consultas mais complexas de modo simplificado.

1.3 Organização do Trabalho

O restante deste trabalho é organizado da seguinte forma: revisão bibliográfica é apresentada no capítulo 2. O desenvolvimento do trabalho é detalhado no capítulo 3. Experimentos são descritos no capítulo 4. Conclusões, contribuições do trabalho e sugestões para trabalhos futuros são apresentadas no capítulo 5.

Capítulo 2
Revisão Bibliográfica

Neste capítulo serão apresentados tópicos relacionados execução de consultas, linguagens de consultas e o *framework* Object-Inject.

2.1 Execução de Consultas

O processador de consultas é o grupo de componentes de um Sistema de Gerenciamento de Banco de Dados (SGBD) que transforma as consultas do usuário e os comandos de modificação de dados em uma sequência de operações sobre o banco de dados e executa essas operações. O processador de consultas pode ser dividido em duas etapas, compilação de consultas e execução de consultas, porém apenas a execução de consultas, que é responsável pelos algoritmos para os operadores que manipulam os dados, será abordada.

2.1.1 Álgebra relacional

Consultas são construídas a partir de ações elementares chamadas de operadores. Tais operadores são descritos pela álgebra relacional, que ganhou atenção após Codd propô-la como base para linguagens de consulta em banco de dados em Codd (1970). Porém, como alguns recursos de SQL não são descritos nesta álgebra, é necessária a adição de alguns operadores. Além disso, a álgebra relacional foi projetada originalmente para conjuntos, entretanto as relações em SQL são sacolas ou conjuntos múltiplos. Por isso a álgebra relacional será introduzida com algumas modificações.

Os operadores de união ($\cup$), interseção ($\cap$) e diferença ($-$) são os operadores usuais de conjuntos. No entanto relações de SQL tem esquemas, isto é, conjuntos de atributos que denominam suas colunas. Então ao se aplicar algum destes operadores, os esquemas das relações dos dois argumentos devem ser os mesmos, resultando em um esquema igual ao de qualquer um dos argumentos. Também são necessárias novas definições para lidar com sacolas:

1. Em $R \cup S$, uma tupla t está no resultado tantas vezes quanto os número de vezes que ela está em R mais o número de vezes que ela está em S.

2. Em $R \cap S$, uma tupla t está no resultado no mínimo o número de vezes que ela está em R e S.

3. Em $R - S$, uma tupla t está no resultado o número de vezes que ela está em R menos o número de vezes que está em S, mas não um número de vezes menor que zero.

A versão destes operadores para conjuntos será indicada pelo subscrito S, e caso não haja subscrito trata-se de um operador para sacolas.

O operador de seleção (σ) produz uma nova relação a partir de uma antiga, selecionando algumas linhas da relação antiga com base em alguma condição ou algum predicado. A seleção $\sigma_C(R)$ toma uma relação R e uma condição C, envolvendo operadores aritméticos ou de strings sobre constantes e/ou atributos, comparações entre estes termos e conectivos booleanos *AND*, *OR* e *NOT* aplicados aos termos construídos. Esta seleção produz a sacola das tuplas de R que satisfazem à condição C. O esquema da relação resultante é igual ao esquema de R.

O operador de projeção (π) também produz uma nova relação a partir de uma antiga, selecionando algumas colunas. Será utilizado a versão estendida deste operador (GUPTA; HARINARAYAN; QUASS, 1995) onde é possível renomear atributos e construir atributos por cálculos com constantes e atributos da relação antiga. $\pi_L(R)$ é a projeção da relação R sobre a lista L. Uma lista de projeção pode conter: um atributo único de R; uma expressão $x \rightarrow y$, que renomeia o atributo x de R como y; uma expressão $E \rightarrow z$, onde o resultado da expressão E, que pode envolver atributos de R, constantes, operadores aritméticos e operadores de strings, se torna um novo atributo com o nome z.

O resultado da projeção é calculado considerando-se cada tupla de R por vez. A lista L é avaliada substituindo os atributos correspondentes mencionados em L pelos componentes da tupla e aplicando os operadores indicados por L a esses valores. O resultado é uma relação cujo esquema corresponde aos nomes dos atributos na lista L, com qualquer mudança de nome que a lista especificar.

O operador de produto de relações ($\times$) é o produto cartesiano da teoria de conjuntos, que constrói tuplas emparelhando as tuplas de duas relações de todas as maneiras possíveis. O produto $R \times S$ é uma relação cujo esquema consiste nos atributos de R e S. As tuplas do produto são aquelas que podem ser formadas tomando-se uma tupla de R e seguindo seus componentes de qualquer tupla de S.

Operadores de junção são construídos a partir de um produto seguido por uma seleção e uma projeção. O mais simples e comum é a junção natural (X). A junção natural das relações R e S é denotada por RXS. 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 de R e S que têm o mesmo nome e L é uma lista de todos os atributos 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) converte uma sacola em um conjunto. A operação $\delta(R)$ retornaria então o conjunto que consiste apenas em tuplas distintas em R.

Existe uma família de características em SQL que funcionam em conjunto para permitir consultas envolvendo "agrupamento e agregação". Dentre elas existem os operadores de agregação *AVG*, *SUM*, *COUNT*, *MIN* e *MAX* que produzem, respectivamente, os valores médios, total, de contagem, mínimo e máximo do atributo ao qual são aplicados. Esses operadores aparecem em cláusulas *SELECT*. Existe também a cláusula *GROUP BY*, que agrupa, de acordo com o valor do atributo ou dos atributos mencionados, a relação construída pelas cláusulas *FROM* e *WHERE*. Além destes, há a cláusula *HAVING*, que segue uma cláusula *GROUP BY* fornecendo uma condição envolvendo agregações ou atributos no grupo que contribuem para o resultado da consulta.

Como o agrupamento e a agregação precisam, em geral, serem implementados e otimizados juntos, será introduzido apenas um operador, γ, representando o efeito de ambos. A cláusula *HAVING* é implementada seguindo o operador γ com uma relação e uma projeção.

O operador γ é usado com o subscrito L, que é uma lista de elementos. Estes elementos podem ser atributos da relação à qual o operador γ é aplicado, que é chamado de atributo de agrupamento. Esse atributo é um dos atributos da lista GROUP BY da consulta. Estes elementos também podem ser operadores de agregação aplicados a um atributo da relação. Para se fornecer um nome ao atributo correspondente a essa agregação no resultado, uma seta e um novo nome são anexados à agregação. Esse elemento representa uma das agregações na cláusula SELECT da consulta. O atributo subjacente é chamado um atributo de agregação.

A relação retornada pela expressão $\gamma_L(R)$ é construída particionando-se as tuplas de R em grupos. Cada grupo consiste de todas as tuplas que têm uma atribuição de valores específica para atributos de agrupamento na lista L. Se não houver nenhum atributo de agrupamento, a relação R inteira será um grupo. Então para cada grupo, se produz uma tupla consistindo nos valores dos atributos de agrupamento para esse grupo e nas agregações sobre todas as tuplas desse grupo, especificados pelos atributos agregação na lista L.

Para classificar uma relação utiliza-se o operador de classificação (τ). A classificação também desempenha um papel como um operador de plano físico de consulta, pois muitos dos outros operadores de álgebra relacional podem ser facilitados classificando-se primeiro uma ou mais das relações dos argumentos. Se R é uma relação, a expressão $\tau_L(R)$, onde L é uma lista de alguns dos atributos de R, é a relação R, mas com as tuplas de R classificadas na ordem indicada por L.

É possível combinar vários operadores da álgebra relacional em uma expressão aplicando um

operador ao(s) resultado(s) de um ou mais operadores diferentes. Geralmente a combinação destes operadores é chamada de *plano lógico de consulta*. Assim, pode-se representar um plano lógico de consulta como uma árvore de expressões, onde as folhas são nomes de relações, e cada um dos interiores é identificado por um operador que faz sentido quando aplicado à(s) relação(ões) representada(s) por seu filho ou filhos.

2.1.2 Operadores de planos físicos de consultas

Os planos físicos de consulta são construídos a partir de operadores, cada um dos quais implementa uma etapa do plano. Na maioria, os operadores físicos são implementações particulares para um dos operadores da álgebra relacional, mas também são necessários operadores físicos para outras tarefas que não envolvem um operador da álgebra relacional, por exemplo para varredura de tabelas. Em outras palavras, o plano físico de consultas é a transformação de um plano lógico de consultas em algoritmos que realizaram a consulta de fato.

Em geral, uma consulta consiste em várias operações de álgebra relacional e o plano físico de consulta correspondente é composto de vários operadores físicos. A escolha sensata de operadores de planos físicos é um aspecto essencial de um bom processador de consultas, pois isso, deve-se ser capaz de estimar o "custo" de cada operador que é usado. Como obter dados do disco é mais demorado que realizar qualquer ação útil com eles, o número de operações de entrada e saída de disco será utilizado como a medida do custo de uma operação. O custo de entrada e saída de disco de saída será considerado como zero, pois se o operador produzir a resposta final a uma consulta, e esse resultado for de fato gravado no disco, então o custo de fazê-lo dependerá apenas do tamanho da resposta, e não de como a resposta foi calculada. Além disso, em muitos aplicativos, a resposta não fica de modo algum armazenada em disco, mas é enviada para alguma outra saída. O resultado de um operador será tratado da mesma forma, já que com frequência ele não é gravado em disco.

As estimativas de custo são essenciais se o otimizador tiver que determinar qual dos muitos planos de consulta provavelmente será executado com maior rapidez. Supondo que a memória principal esteja dividida em buffers, o número de buffers da memória principal disponíveis para uma execução de um operador específico será representado por M.

Para medir o custo de acesso a relações de argumentos, utilizam-se parâmetros que medem o tamanho e a distribuição dos dados em uma relação e que com frequência são calculados periodicamente para ajudar o otimizador de consultas a escolher operadores físicos. Para simplificar, será feita a suposição de que os dados são acessados no disco um bloco de cada vez. O parâmetro $B(R)$ representa número de blocos necessários para conter todas as tuplas de R, que normalmente presume-se

estar agrupado em clusters. Para representar o número de tuplas será utilizado o parâmetro $T(R)$. Por fim, se R é uma relação e um de seus atributos é a, então o parâmetro $V(R, a)$ é o número de valores distintos da coluna correspondente a a em R.

Talvez a varredura seja a ação mais básica do plano físico de consulta, ela simplesmente lê todo conteúdo de uma relação R. Geralmente, a relação R está armazenada em uma área da memória secundária, com suas tuplas organizadas em blocos. Os blocos que contêm tuplas de R são conhecidos pelo sistema, e são obtidos um a um através da operação chamada *varredura de tabela*. Quando há um índice sobre qualquer atributo de R, ele poderá ser usado para obter todas as tuplas de R pela operação de *varredura de índices*.

Uma relação também pode ser classificada a medida que é lida. A *varredura de classificação* toma uma relação R e uma especificação dos atributos em que a classificação deve ser feita e produz R nessa ordem classificada.

Se a relação R estiver agrupada em clusters, o número de operações de E/S de disco para o operador de varredura de tabela será aproximadamente B. Da mesma forma, se R couber na memória principal, a varredura de classificação poderá ser implementada transferindo R para a memória e executando uma classificação na memória, mais uma vez exigindo apenas B operações de E/S de disco.

Se R for agrupada em clusters e não couber na memória, então será necessário a utilização do método de ordenação como o bem conhecido *merge-sort*. Para este método são necessárias cerca de $3B$ operações de E/S de disco, divididas igualmente entre as etapas necessárias para a leitura de R em sub-listas, escrita das sub-listas e releitura das sub-listas.

Porém, se R não for agrupada em clusters, o número de operações de E/S de disco necessárias em geral será muito maior. Se R for distribuída entre tuplas de outras relações, então uma varredura de tabelas para R poderá exigir a leitura de tantos blocos quantas são as tuplas de R; isto é, o custo de E/S é T. De maneira semelhante, se R couber na memória, para classifica-la T operações de E/S de disco serão necessárias para inserir R na memória.

Finalmente, se R não estiver agrupada em clusters e exigir um merge-sort, ele demorará T operações de E/S de disco para ler inicialmente os subgrupos. Porém, pode-se armazenar e reler as sublistas na forma agrupada em clusters, e assim essas etapas exigirão apenas $2B$ operações de E/S de disco. Logo, o custo total para realização de uma varredura de classificação neste caso $T + 2(B)$.

Agora, considere o custo de uma varredura de índice. Em geral, um índice sobre uma relação R exige muitos menos de $B(R)$ blocos. Então, uma varredura da relação R inteira, que exige no mínimo B operações de E/S de disco, exigirá significativamente mais operações de E/S do que o exame de índice inteiro. Desse modo, ainda que a varredura de índice exija o exame da relação e de seu índice,

B ou T continuarão sendo usados como uma estimativa do custo para acessar uma relação agrupada em clusters ou não agrupada em clusters em sua totalidade, usando um índice.

2.1.3 Algoritmos de uma passagem

A escolha de um algoritmo para cada operador é parte essencial do processo para transformar um plano lógico de consulta em um plano físico de consulta. Embora muitos algoritmos para operadores tenham sido propostos, eles recaem amplamente em três classes: métodos baseados na classificação, métodos baseados em hash e métodos baseados em índices.

Além disso, pode-se dividir os algoritmos para operadores em três "graus" de dificuldade e custo. Os algoritmos de *uma passagem* realizam a leitura dos dados do disco apenas uma vez e, em geral, eles funcionam apenas quando pelo menos um dos argumentos da operação se encaixa na memória principal. Já os algoritmos de *duas passagens* são caracterizados por lerem dados do disco uma primeira vez, processá-los de alguma forma, gravar todos ou quase todos eles no disco, e depois fazerem uma segunda leitura para processamento adicional durante a segunda passagem. Estes algoritmos conseguem lidar com dados grandes demais para caber na memória principal disponível, mas não para os maiores conjuntos de dados imagináveis. Para lidar com dados sem um limite de tamanho, utilizam-se métodos que usam três ou mais passagens para fazer seu trabalho e são generalizações naturais e recursivas dos algoritmos de duas passagens.

Primeiramente serão apresentados os métodos de uma passagem, que serão classificados, assim como os métodos de mais passagens, em três grandes grupos:

1. *Operações unárias em uma tupla por vez.* Essas operações - seleção e projeção - não exigem uma relação inteira, ou mesmo uma grande parte dela, na memória ao mesmo tempo. Desse modo, pode-se ler um bloco de cada vez, usar um único buffer da memória principal e produzir a saída.

2. *Operações unárias na relação inteira.* Essas operações de um argumento exigem a visualização de todas ou da maioria das tuplas na memória ao mesmo tempo; assim, os algoritmos de uma passagem se limitam a relações que têm aproximadamente um tamanho M ou menor. As operações dessa classe que serão consideradas aqui são a *eliminação de duplicatas* e o *agrupamento*.

3. *Operações binárias em relações inteiras.* Todas as outras operações estão nesta classe: versões de conjuntos e sacolas de união, interseção e diferença, junções e produtos.

As operações em uma tupla por vez $\sigma(R)$ e $\pi(R)$ têm algoritmos óbvios. Lê-se os blocos de R um de cada vez para um buffer de entrada, se executa a operação sobre cada tupla e então move-se as tuplas selecionadas ou as tuplas projetadas para o buffer de saída. Desse modo, exige-se que $M \geq 1$ para o buffer de entrada, independente de B. O requisito de E/S de disco para esse processo só depende de como a relação do argumento R é fornecida. Se R estiver inicialmente no disco, então o custo será o tempo para executar uma varredura de tabelas ou uma varredura de índices de R.

Para a *eliminação de duplicatas*, lê-se um bloco de R de cada vez. Para cada tupla analisa-se se é a primeira vez que essa tupla é vista e, nesse caso, ela é copiada para a saída, caso o contrário ela é ignorada. Para esta análise é necessário manter na memória uma cópia de cada tupla vista. Um buffer de memória contém um bloco de tuplas de R, e os $M - 1$ buffers restantes podem ser usados para guardar uma única cópia de cada tupla vista até o momento. Quando uma nova tupla de R é considerada, ela é comparada a todas as tuplas armazenadas e, se ela não for igual a nenhuma dessas tuplas, ela é copiada para a saída e é incluída na lista de memória de tuplas que já foram vistas.

Entretanto, se houver n tuplas na memória principal, cada nova tupla ocupará um tempo de processador proporcional a n, e assim a operação completa ocupará o tempo do processador proporcional a n^2. Tendo em vista que n poderia ser muito grande, a hipótese de que apenas o tempo de E/S seria significativo se torna inválida. Desse modo, é necessário uma estrutura da memória principal que permita adicionar uma nova tupla e saber se uma dada tupla já está lá, exigindo um tempo próximo a uma constante para isto, independente do tamanho de n. Muitas dessas estruturas são conhecidas, por exemplo, uma tabela de hash com um grande número de depósitos. Essas estruturas exigem algum overhead, além do espaço necessário para armazenar as tuplas. Contudo, o espaço extra necessário tende a ser pequeno em comparação com o espaço exigido para armazenar as tuplas. Desse modo, será utilizada a hipótese simplificadora de que nenhum espaço extra é necessário, e toda atenção será focada no espaço exigido para armazenar as tuplas na memória principal.

Seguindo esta hipótese, pode-se armazenar nos $M - 1$ buffers disponíveis da memória principal tantas tuplas quantas cabem em $M - 1$ blocos de R. Para que uma cópia de cada tupla distinta de R caiba na memória principal, $B(\delta(R))$ não deverá ser maior que $M - 1$. Como espera-se que M seja muito maior que 1, uma aproximação mais simples para essa regra, e a única que será usada em geral, é $B(\delta(R)) \leq M$.

Na operação de agrupamento γ_L, é fornecido zero ou mais atributos de agrupamento e possivelmente um ou mais atributos de agregação. Se for criada uma entrada para cada grupo na memória principal, isto é, para cada valor dos atributos de agrupamento, então é possível varrer as tuplas de R, um bloco de cada vez. A entrada para um grupo consiste em valores dos atributos de agrupamento e um ou mais valores acumulados para cada agregação. Para um agregado $MIN(a)$ ou $MAX(a)$

registra-se o valor mínimo ou máximo, respectivamente, do atributo a visto por qualquer tupla do grupo até o momento. Para qualquer agregação de $COUNT$, adiciona-se uma unidade para cada tupla do grupo que for vista. Na operação $SUM(a)$, adiciona-se o valor do atributo a à soma acumulada até o momento. Por fim, para $AVG(a)$ deve-se manter duas totalizações: a contagem do número de tuplas no grupo e a soma dos valores a dessas tuplas, cada uma calculada da mesma forma que nas agregações *COUNT* e *SUM*, respectivamente. Depois que todas as tuplas de R forem vistas, toma-se o quociente da soma pela contagem para obter a média.

Quando todas as tuplas de R forem lidas para o buffer de entrada e contribuírem para a(s) agregação(ões) de seu grupo, pode-se produzir a saída gravando a cada tupla para cada grupo. O número de operações de E/S de disco necessárias para esse algoritmo de uma passagem é B. O número M de buffers de memória exigidos não está relacionado a B de forma simples, embora em geral M seja menor que B. O problema é que as entradas para os grupos poderiam ser mais longas ou mais curtas que as tuplas de R e o número de grupos poderia ser algo igual ou menor que o número de tuplas de R. Porém, na maioria dos casos, as entradas de grupos não serão maiores que as tuplas de R, e haverá muito menos grupos que tuplas.

A *união de sacolas* pode ser calculada por um algoritmo muito simples de uma passagem. Para calcular $R \cup_B S$, copiamos cada tupla de R para a saída e, em seguida, copiamos cada tupla de S. Para isso $M = 1$ é suficiente. Assim como todos os algoritmos binários de uma passagem, esta operação exige $B(R) + B(S)$ operações de E/S de disco.

Outras operações binárias exigem a leitura do menor dos operandos R e S para a memória principal e a construção de uma estrutura de dados adequada, de forma que as tuplas possam ser inseridas e encontradas com rapidez. Como antes, uma opção é a tabela de hash. A estrutura exige um espaço pequeno extra, que será desprezado. Desse modo, o requisito aproximado para uma operação binária sobre relações R e S a ser executada em uma passagem é $min(B(R), B(S)) \leq M$. Todas as outras operações serão detalhadas pressupondo-se que R é a maior das relações, alojando-se então S na memória principal.

Na *União de Conjuntos*, lê-se S para $M - 1$ buffers da memória principal e se constrói uma estrutura de pesquisa na qual a chave de pesquisa é a tupla inteira. Todas essas tuplas também são copiadas para a saída. Em seguida, lê-se cada bloco de R para o M-ésimo buffer, um de cada vez. Para cada tupla t de R, é verificado se t está em S. Se não estiver, t é copiado para a saída, caso contrário é ignorado.

Na *Interseção de Conjuntos* também lê-se S para $M-1$ buffers da memória principal e se constrói uma estrutura de pesquisa na qual a chave de pesquisa é a tupla inteira. Lê-se então cada bloco de R e, para cada tupla t de R, verifica-se se t também está em S. Se estiver, t é copiado para a saída, caso

contrário é ignorado.

Na *Interseção de sacola* lê-se S para $M-1$ buffers, porém várias cópias de uma tupla t não são armazenadas individualmente. Em vez disso, apenas uma cópia de t é armazenada e associada a uma contagem igual ao número de vezes que t ocorre. Em seguida, lê-se cada bloco de R e verifica-se cada uma de suas tupla t. Caso t ocorra em S e sua contagem associada seja é positiva, então é feita a saída de t e sua contagem é decrementada 1 unidade. Caso t apareça em S, mas sua contagem tenha alcançado 0, então a saída t não é feita.

Como a diferença não é um operador comutativo, devemos distinguir entra $R -_S S$ e $S -_S R$. Em cada caso, lê-se S para $M-1$ buffers e uma estrutura de pesquisa com tuplas completas como a chave de pesquisa é construída. Para calcular $R -_S S$, lê-se cada bloco de R e cada tupla t nesse bloco é examinada. Se t estiver em S, ela é ignorado. Se não estiver em S, ela é copiado para a saída. Para calcular $S -_S R$, lê-se cada bloco de R e cada tupla t desse bloco é examinada. Caso t esteja em S, ela é eliminada da cópia de S que está na memória principal, caso o contrário, nada é feito. Depois de considerada cada tupla de R, as tuplas restantes de S são copiadas para saída.

Para calcular $S - R$, lê-se as tuplas de S para a memória principal e é contado o número de ocorrências de cada tupla distinta. Quando R é lido, verifica-se se cada uma de suas tuplas t ocorre em S. Caso ocorra, sua contagem associada é decrementada. No final, cada tupla da memória principal cuja contagem é positiva é copiada para saída o número de vezes da sua contagem. Para calcular $R - S$, também lê-se as tuplas de S para a memória principal e é contado o número de ocorrências de cada tupla distinta. Quando R é lido, verifica-se se cada uma das suas tuplas t ocorre em S. Caso não ocorra, t é copiado para a saída. Caso ocorra, a contagem atual c associada a t é examinada. Se $c = 0$, t é copiado para a saída. Se $c > 0$, c é decrementado em 1 unidade.

No cálculo do *produto cartesiano*, S é lido para $M-1$ buffers na memória principal. Em seguida, lê-se cada bloco de R e cada tupla t de R é concatenada com cada tupla de S na memória principal, sendo feita a saída de cada tupla concatenada à medida que ela é formada.

No algoritmo da *junção natural* será adotada a convenção de que está sendo feita a junção de $R(K, Y)$ com $S(Y, Z)$, onde Y representa todos os atributos que R e S têm em comum, K representa todos os atributos de R que não estão no esquema de S e Z representa todos os atributos de S que não estão no esquema de R. Para calcular a junção natural, lê-se todas as tuplas de S e forma-se uma estrutura de pesquisa na memória principal com elas tendo os atributos de Y como a chave de pesquisa. Utiliza-se $M-1$ blocos da memória para esse propósito. Lê-se então cada bloco de R no buffer da memória principal restante. Para cada tupla t de R, encontra-se as tuplas de S que concordam com t em todos os atributos de Y, usando a estrutura de pesquisa. Finalmente, cada tupla correspondente de S é juntada a t e movida para a saída.

Este algoritmo funciona desde que $B(S) \leq M - 1$ ou, aproximadamente, $B(S) \leq M$. Além disso, assim como nos algoritmos anteriores, o espaço exigido pela estrutura de pesquisa na memória principal não é contado, mas pode levar a um pequeno requisito de memória adicional.

2.1.4 Junções de loops aninhados

Esses algoritmos são, de certo modo, de "uma e meia" passagem pois, a cada variação, um dos dois argumentos tem suas tuplas lidas somente uma vez, enquanto o outro argumento é lido repetidamente. As junções de loops aninhados podem ser usadas com relações de qualquer tamanho não sendo necessário que uma relação caiba na memória.

A versão mais simples para *junção de loops alinhados* é baseada em tuplas. Para calcular $R(K, Y) \boxtimes S(Y, Z)$, lê-se todas as tuplas de S, uma por uma. Para cada tupla lida de S, lê-se todas as tuplas de R, uma por vez, e, caso a tupla lida de S se junte a tupla lida de R para formar uma tupla da resposta, a tupla resultante é enviada para saída.

Esta junção pode ser melhorada organizando-se o acesso a ambas as relações de argumentos por blocos, garantindo assim que, quando examinadas as tuplas de R no loop interno, o número mínimo de operações de E/S de disco possível para ler R é usado. Também obtêm-se melhorias utilizando o máximo de memória principal possível para armazenar tuplas pertencentes à relação S, a relação de loop externo. Isto permite juntar cada tupla de R lida, não apenas com uma tupla de S, mas com tantas tuplas de S quantas couberem na memória.

Supondo que $B(S) \leq B(R)$ e que nenhuma ralação cabe inteiramente na memória principal. Lê-se repetidamente $M - 1$ blocos de S para buffers da memória principal. Em seguida, todos os blocos de R são percorridos, lendo-se um de cada vez para o último bloco de memória. Uma vez lá, todas as tuplas do bloco de R são comparadas a todas as tuplas em todos os blocos de S que estão atualmente na memória principal. As tuplas que se juntam formam a saída resultante.

Supondo que S seja a relação menor, o número de grupos, ou iterações loop externo é $B(S)/(M-1)$. Em cada iterações, lemos $M-1$ blocos de S e $B(R)$ blocos de R. O número de operações de E/S de disco é portanto: $(B(S)/(M-1))(M-1+B(R))$.

2.1.5 Algoritmos de duas passagens baseados em classificação

Agora, serão apresentados algoritmos de várias passagens para a execução de operações de álgebra relacional sobre relações que não cabem totalmente na memória, que foram inicialmente desenvolvido por Blasgen (BLASGEN; ESWARAN, 1977).

Nesta seção a classificação será considerada uma ferramenta para implementar operações relaci-

onais. A ideia básica é que, se há uma grande relação R, na qual $B(R)$ é maior que M, pode-se repetidamente ler M blocos de R para a memória principal, classificar esses M blocos na memória principal usando um algoritmo de classificação eficiente e gravar a lista classificada em M blocos no disco. O conteúdo desses blocos serão referidos como uma das *sublistas classificadas* de R. Todos os algoritmos que serão descritos utilizam então uma segunda passagem para "mesclar" as sublistas classificadas de algum modo para executar o operador desejado.

Para executar a operação $\delta(R)$ em duas passagens, classificam-se as tuplas de R em sublistas da maneira descrita. Em seguida, utiliza-se a memória principal disponível para conter um bloco de cada sublista classificada. Então, copia-se repetidamente uma tupla para a saída e ignora-se todas a tuplas idênticas a ela. Mais precisamente, examina-se a primeira tupla não considerada de cada bloco e encontra-se entre elas a primeira em ordem classificada, digamos t. Faz-se uma cópia de t na saída e remove-se do início dos diversos blocos de entrada todas as cópias de t. Se um bloco se esgotar, o próximo bloco da mesma sublista é trazido para seu buffer e, se houver algum t nesse bloco, ele também é removido. Este algoritmo utiliza $B(R)$ operações de E/S de disco para ler cada bloco de R ao criar as sublistas classificadas, $B(R)$ para escrever para cada uma das sublistas classificadas no disco e $B(R)$ para ler cada bloco das sublistas no tempo apropriado, totalizando $3B(R)$.

Supondo que M blocos de memória estejam disponíveis, cria-se sublistas classificadas de M blocos cada uma. Para a segunda passagem é preciso um bloco de cada sublista na memória principal, e assim não pode haver mais de M sublistas, cada qual com M blocos. Desse modo, $B \leq M^2$ é necessário para o algoritmo de duas passagens ser possível.

O algoritmo de duas passagens para $\gamma_L(R)$ é muito semelhante ao algoritmo para $\delta(R)$. Primeiro lê-se as tuplas de R para a memória, M blocos de cada vez. Classifica-se cada M bloco, usando os atributos de agrupamento de L como chave de classificação e grava-se cada sublista classificada no disco. Então usa-se um buffer da memória principal para carregar, inicialmente, o primeiro bloco de cada sublista. A partir dai, encontra-se repetidamente o menor valor da chave de classificação (atributos de agrupamento) presente entre as primeiras tuplas disponíveis nos buffers. Esse valor, v, se torna o próximo grupo, para o qual é preparado o cálculo de todos os agregados na lista L desse grupo, utilizando uma contagem e uma soma em lugar de uma média. Examina-se então cada uma das tuplas com a chave de classificação v e acumula-se os agregados necessários e, caso um buffer se esvazie, ele é substituído pelo bloco seguinte da mesma sublista. Quando não há mais tuplas com a chave de classificação v disponíveis, se faz a saída de uma tupla consistindo nos atributos de agrupamento de L e nos valores associados das agregações calculadas para o grupo.

Como no algoritmo δ, esse algoritmo de duas passagens para γ demora $3B(R)$ operações de E/S de disco e funcionará, desde que $B(R) \leq M^2$.

O uso do algoritmo de duas passagens é apenas considerado para a união de conjuntos, já que o algoritmo de uma passagem para união de sacolas funciona independentemente do tamanho dos argumentos. Para calcular $R \cup S$, M blocos de R são trazidos repetidamente para a memória principal, classificam-se suas tuplas e grava-se a sublista classificada resultante de novo em disco. O mesmo é feito para S, a fim de criar sublistas classificadas para a relação S. Inicializa-se então um buffer da memória principal para cada sublista de R e S com o primeiro bloco da sublista correspondente. A partir dai encontra-se repetidamente a primeira tupla t restante entre todos os buffers, copia-se t para a saída e todas suas cópias são removidas dos buffers (se R e S forem conjuntos, deverá haver no máximo duas cópias). Se um buffer ficar vazio, próximo bloco de sua sublista é carregado.

É necessário que cada tupla de R e S seja lida duas vezes para a memória principal, uma vez quando as sublistas estão sendo criadas, e a segunda vez como parte de uma das sublistas. A tupla também é gravada em disco uma vez, como parte de uma sublista recém-formada. Desse modo, o custo em operações de E/S de disco é $3(B(R) + B(S))$.

O algoritmo funciona desde que o número total de sublistas entre as duas relações não exceda M, porque é necessário um buffer para cada sublista. Tendo em vista que cada sublista tem M blocos, o tamanho das duas relações não devem exceder M^2, isto é, $B(R) + B(S) \leq M^2$.

Tanto para conjuntos ou sacolas, os algoritmos de *interseção* e *diferença* são essencialmente os mesmos dos de uma passagem, a não ser pelo modo como as cópias de uma tupla t são tratadas no início das sublistas classificadas. Em geral, cria-se as sublistas classificadas de M blocos cada uma para ambas as relações dos argumentos R e S. Usamos um buffer da memória principal para cada sublista, carregado inicialmente com o primeiro bloco da sublista.

Em seguida, considera-se repetidamente a menor tupla t entre as tuplas restantes em todos os buffers. Conta-se o número de tuplas de R que são idênticas a t e além do número de tuplas de S idênticas a t. Isso exige que os buffers sejam recarregados a partir de quaisquer sublistas cujo bloco atualmente bufferizado se esgotou. Então, caso a operação seja a interseção de conjuntos, se t aparecer em R e S, ele é enviado para saída. Caso a operação seja a interseção de sacolas, t é enviado para saída o um número de vezes igual ao menor número de vezes que ela aparece em R e em S. Observe que t não será colocado na saída se qualquer dessas contagens for 0, isto é, se t não aparecer em ambas as relações. Caso a operação seja a diferença de conjuntos, $R - S$, é feita saída de t se e somente se ele aparecer em R, mas não em S. Por fim, se a operação for a diferença de sacolas, $R - S$, é feita a saída de t o número de vezes em que ele aparece em R menos o número de vezes que ela aparece em S. É claro que se t aparecer em S pelo menos tantas vezes quantas aparece em R, t não é incluído na saída.

Do mesmo modo que o algoritmo de união de conjuntos de uma passagem, para este algoritmo é

necessário $3(B(R) + B(S))$ operações de E/S de disco e de aproximadamente $B(R) + B(S) \leq M^2$ para o algoritmo funcionar.

Antes de examinar o algoritmo de junção, é necessário observar um problema que pode ocorrer quando se calcula uma junção, mas que não foi um problema importante no caso das operações binárias consideradas até agora. Quando uma junção é efetuada, o número de tuplas das duas relações que compartilham um valor comum do(s) atributo(s) de junção, e que portanto precisam estar na memória principal simultaneamente, pode exceder o número que caberia na memória. O exemplo extremo ocorre quando existe apenas um valor dos atributos de junção e cada tupla de uma relação se junta a cada tupla da outra relação. Nessa situação, não há realmente nenhuma escolha além de fazer uma junção de loops aninhados dos dois conjuntos de tuplas com um valor comum no(s) atributos(s) de junção.

Para evitar essa situação, pode-se tentar reduzir o uso da memória principal para outros aspectos do algoritmo, e assim tornar disponível um grande número de buffers para conter as tuplas com um dado valor de atributo de junção. Será discutido então o algoritmo que torna o maior número possível de buffers disponível para juntar tuplas com um valor comum.

Dadas as relações $R(K, Y)$ e $S(Y, Z)$ para junção e dados M blocos da memória principal para buffers, classifica-se R e S, utilizando merge-sort, com Y como chave de classificação. Em seguida, R e S são mescladas e classificadas. Para isto, geralmente utiliza-se apenas dois buffers, um para o bloco atual de R e o outro para o bloco atual de S. Então repetidamente encontra-se o menor valor de y dos atributos de junção Y, que atualmente está no inicio dos blocos de R e S. Caso y não apareça no início da outra relação, remove-se a(s) tupla(s) com a chave de classificação y. Caso contrário, identifica-se todas as tuplas de ambas as relações que têm a chave de classificação y. Se necessário, lê-se os blocos de R e/ou S classificados, até se ter a certeza de que não há mais nenhum y em uma ou outra relação. M buffers estão disponíveis para esse propósito. Finalmente, se faz a saída de todas as tuplas que podem ser formadas pela junção de tuplas de R e S com um valor y de Y comum e, se uma das relações não tiver mais tuplas não consideradas na memória principal, o buffer para essa relação é recarregado.

Se houver um valor y de Y para o qual o número de tuplas com esse valor não cabe em M buffers, é necessário modificar o algoritmo anterior. Caso as tuplas de uma das relações, digamos R, que têm o valor y de Y, couberem em $M-1$ buffers, carrega-se esses blocos de R nos buffers e lê-se os blocos de S que contêm tuplas com y, um de cada vez, nos buffers restantes. Na realidade, a junção de uma passagem é feita somente sobre as tuplas com o valor y de Y. Caso nenhuma das relações tenha um número suficientemente pequeno de tuplas com o valor y de Y tal que todas elas caibam em $M-1$ buffers, usa-se os M buffers para executar uma junção de loops aninhados sobre tuplas com o valor y

de Y de ambas as relações.

Este algoritmo executa cinco operações de E/S de disco para cada bloco da relação do argumento. A exceção ocorreria se houvesse tantas tuplas com um valor de Y comum que fosse preciso fazer uma das junções especializadas sobre essas tuplas.

Também é preciso considerar o quanto M precisa ser grande para que a simples junção de classificação possa funcionar. A principal restrição é que necessita-se ser capaz de executar as classificações utilizando merge-sort sobre R e S. Para executar essas classificações é necessário $B(R) \leq M^2$ e $B(S) \leq M^2$. Depois de realizá-las, os buffers não serão esgotados, embora como vimos antes, talvez seja necessário desviar-se da mesclagem simples, se as tuplas com um valor de Y comum não couberem em M buffers. Em suma, supondo que não seja necessário nenhum desvio, a junção de classificação simples usa $5(B(R) + B(S))$ operações de E/S de disco e exige $B(R) \leq M^2$ e $B(S) \leq M^2$ para funcionar.

2.1.6 Algoritmos de duas passagens baseados em hash

A família de algoritmos baseados em hash foi inicialmente desenvolvida pelo japonês Kitsuregawa (KITSUREGAWA; TANAKA; MOTO-OKA, 1983) e tem como base a seguinte ideia: se os dados são grandes demais para serem armazenados em buffers da memória principal, mistura-se todas as tuplas do argumento ou argumentos usando uma chave de hash apropriada. Para todas as operações comuns, existe um modo de selecionar a chave de hash de tal forma que todas as tuplas que precisem ser consideradas em conjunto quando a operação for executada tenham o mesmo valor de hash.

Em seguida, executa-se a operação atuando sobre um depósito de cada vez (ou sobre um par de depósitos com o mesmo valor de hash, no caso de uma operação binária). Na realidade, se reduz o tamanho do(s) operandos(s) por um fator igual ao número de depósitos. Se houver M buffers disponíveis, pode-se escolher M como o número de depósitos, ganhando assim um fator M no tamanho das relações que se pode manipular.

Para realizar a *eliminação de duplicatas* $\delta(R)$, inicialmente mistura-se R a $M - 1$ depósitos através de uma função hash h, o que consequentemente faz que duas cópias da mesma tupla t sejam misturadas ao mesmo depósito. Desse modo, se cria a propriedade essencial necessária: é possível examinar um depósito de cada vez, executar δ sobre esse depósito isolado e tomar como resposta a união de $\delta(R_i)$, onde R_i é a porção de R que se mistura ao i-ésimo depósito.

Caso R_i seja suficientemente pequeno para caber na memória, pode-se usar o algoritmo de eliminação de duplicatas de uma passagem e escrever as tuplas exclusivas resultantes. Supondo que a função hash h particione R em depósitos de tamanho igual, cada R_i terá aproximadamente o tamanho

$B(R)/(M-1)$. Logo, é necessário que este número não seja maior que M para o algoritmo funcionar, isto é, $B(R) \leq M(M-1)$. Como espera-se que M seja muito maior que 1, é possível fazer uma aproximação, necessitando assim que $B(R) \leq M^2$ para o algoritmo funcionar.

O número de operações de E/S de disco é semelhante ao do algoritmo baseado em classificação. Lê-se cada bloco de R uma vez à medida que suas tuplas são misturadas e cada bloco de cada depósito é gravado em disco. Em seguida, lê-se novamente cada bloco de cada depósito no algoritmo de uma passagem que se concentra nesse depósito. Desse modo, o número total de operações de E/S de disco é $3B(R)$.

Para executar a operação $\gamma_L(R)$, novamente todas as tuplas de R são misturadas $M-1$ depósitos. Porém, para se ter certeza de que todas as tuplas no mesmo grupo acabarão no mesmo depósito, é necessário escolher uma função de hash que dependa apenas dos atributos de agrupamento da lista L.

Tendo particionado R em depósitos, assim como no algoritmo anterior pode-se então usar o algoritmo de uma passagem para γ, a fim de processar um depósito de cada vez, desde que $B(R) \leq M^2$. Porém, na segunda passagem, é necessário apenas um registro por grupo à medida que cada depósito é processado. Portanto, se o tamanho de um depósito for maior que M, pode-se tratar o depósito em uma passagem, desde que os registros para todos os grupos no depósito não ocupem mais de M buffers. Normalmente, o registro de um grupo não será maior que uma tupla de R. Nesse caso, um limite superior melhor sobre $B(R)$ é M^2 vezes o número médio de tuplas por grupo.

Como consequência, se houver poucos grupos, é possível realmente tratar relações R muito maiores de que é indicado pela regra $B(R) \leq M^2$. Por outro lado, se M exceder o número de grupos, não é possível preencher todos os depósitos. Assim, a limitação real sobre o tamanho de R como uma função de M é complexa, mas $B(R) \leq M^2$ é uma estimativa conservadora. Finalmente, o número de operações de E/S de disco para γ, como também para δ, é $3B(R)$.

Quando a operação é binária, é preciso ter certeza de que foi usada a mesma função de hash para misturar tuplas de ambos os argumentos. Por exemplo, para calcular $R \cup_S S$, R e S são misturadas a $M-1$ depósitos cada, digamos R_1, R_2, ..., R_{M-1} e S_1, S_2, ..., S_{M-1}. Em seguida, toma-se a união de conjuntos de R_i com S_i. Para $\cup$, o algoritmo simples de união de sacolas apresentado é preferível a qualquer outra abordagem relativa a essa operação.

Para tomar a interseção ou a diferença de R ou S, são criados $2(M-1)$ depósitos exatamente como para a união de conjuntos, e aplica-se o algoritmo de uma passagem apropriado a cada par de depósitos correspondentes. Note que todos esses algoritmos exigem $B(R) + B(S)$ operações de E/S de disco. A essa quantidade é preciso adicionar as duas operações de E/S de disco por bloco necessários para misturar as tuplas das duas relações e armazenar os depósitos em disco, dando um total de $3(B(R) + B(S))$ operações de E/S de disco.

Para que os algoritmos funcionem, é necessário a capacidade de se tomar a união, a interseção ou a diferença de uma passagem de R_i e S_i, cujos tamanhos serão aproximadamente $B(R)/(M-1)$ e $B(S)/(M-1)$, respectivamente. Como os algoritmos de uma passagem para essas operações exigem que o menor operando ocupe no máximo $M-1$ blocos, os algoritmos de duas passagens baseados em hash exigem que $min(B(R), B(S)) \leq M^2$, aproximadamente.

O cálculo da junção $R(K,Y) \boxtimes S(Y,Z)$ é feito quase da mesma forma que em outras operações binárias de hash. A única diferença é que deve-se usar como chave de hash apenas os atributos de junção, Y. Com isso, é possível ter certeza de que, se as tuplas de R e S se juntarem, elas ocuparão depósitos correspondentes R_i e S_i para algum i. Uma junção de uma passagem de todos os pares de depósitos correspondentes completa esse algoritmo. Da mesma forma do algoritmo anterior, a junção de hash exige $3(B(R) + B(S))$ operações de E/S de disco e funcionará desde que $min(B(R), B(S)) \leq M^2$, aproximadamente.

2.1.7 Algoritmos baseados em índices

Algoritmos baseados em índices são especialmente úteis para o operador de seleção, mas os algoritmos para junção e outros operadores binários também usam índices com bastante proveito.

Primeiramente, é necessário trazer a ideia de *índices agrupados em clusters*. Uma relação está "agrupada em clusters" se suas tuplas estão reunidas em tão poucos blocos quantos poderiam conter essas tuplas. Todas as análises feitas até agora pressupõem que as relações estão agrupadas em clusters. Estendendo este conceito, também é possível falar em *índices agrupados em clusters*, que são índices sobre um ou mais atributos, tais que todas as tuplas com um valor fixo para a chave de pesquisa desse índice aparecem em aproximadamente tão poucos blocos quantos podem contê-las. Observe que uma relação que não está agrupada em clusters não pode ter um índice agrupado em clusters, mas até mesmo uma relação agrupada em clusters pode ter índices não agrupados em clusters.

Para implementação de uma seleção $\sigma_C(R)$, suponha que a condição C seja a forma $a = v$, onde a é um atributo para o qual existe um índice, e v é um valor. Então, seria possível pesquisar o índice com o valor v e obter ponteiros para exatamente as tuplas de R que têm um valor v de a. Essas tuplas constituem o resultado de $\sigma_{a=v}(R)$, e então o que resta fazer é recuperá-las.

Se o índice sobre $R.a$ está agrupado em clusters, então o número de operações de E/S de disco para recuperar o conjunto $\sigma_{a=v}(R)$ será cerca de $B(R)/V(R,a)$. O número real pode ser um pouco mais alto, porque, com frequência, o índice não é mantido inteiramente na memória principal, e então são necessárias algumas operações de E/S de disco para admitir a pesquisa de índices. Também é

possível que, ainda que todas as tuplas com $a = v$ possam caber em b blocos, elas podem estar espalhadas por $b + 1$ blocos, porque elas não começam no início de um bloco. Além disso, embora o índice esteja agrupado em clusters, as tuplas com $a = v$ podem estar espalhadas por vários blocos extras. Este ultimo ocorre, pois não é possível compactar blocos de R da forma mais firme possível, afim de deixar espaço para o crescimento de R, além de que R pode estar armazenada com algumas outras tuplas que não pertencem a R, como em uma organização de arquivo agrupado em clusters.

Além disso, deve-se arredondar os valores se a razão $B(R)/V(R, a)$ não for inteira. O detalhe mais significativo é que, no caso de a ser uma chave para R, então $V(R, a) = T(R)$, que é presumivelmente muito maior que $B(R)$, ainda que sem dúvida seja exigida uma operação de E/S de disco para recuperar a tupla com o valor de chave V, além das operações de E/S de disco necessárias para acessar o índice.

Agora, considere o que ocorre quando o índice sobre $R.a$ não está agrupado em clusters. Em uma primeira aproximação, cada tupla que recuperada estará em um bloco diferente, e é preciso acessar $T(R)/V(R, a)$ tuplas. Desse modo, $T(R)/V(R, a)$ é uma estimativa do número de operações de E/S de disco necessário. O número poderia ser maior, porque também é possível existir a necessidade de ler alguns blocos de índices do disco; ele poderia ser mais baixo, porque pode acontecer de algumas tuplas recuperadas aparecerem no mesmo bloco, e esse bloco permanecer bufferizado na memória.

A varredura de índices como um método de acesso, também pode ajudar em vários outros tipos de operações de seleção. Um índice, como uma árvore B, por exemplo, permite o acesso aos valores da chave de pesquisa em um dado intervalo de forma eficiente. Se existir tal índice sobre o atributo a da relação R, é pos usar o índice para recuperar apenas as tuplas de R no intervalo desejado para seleções como $\sigma_{a\geq 0}(R)$, ou mesmo $\sigma_{a\geq 10} AND \sigma_{a\leq 20}(R)$. Uma seleção com uma condição complexa C também pode, às vezes, ser implementada por uma varredura de índices, seguida por outra seleção apenas sobre as tuplas recuperadas pela varredura de índices. Se C for da forma $a = v AND C'$, onde C' é qualquer condição, é possível dividir a seleção em uma cascata de duas seleções, a primeira verificando apenas as $a = v$, e a segunda verificando a condição C'. A primeira é uma candidata para uso do operador de varredura de índices.

Agora considere junção natural $R(K, Y) \boxtimes S(Y, Z)$, podendo K, Y e Z corresponder a conjuntos de atributos. Como primeiro algoritmo de junção baseado em índices, suponha que S seja um índice sobre o(s) atributo(s) Y. Então, uma forma de calcular a junção é examinar cada bloco de R e, dentro de cada bloco, considerar cada tupla t. Seja t_Y o componente ou componentes de t que corresponde(m) ao(s) atributo(s) Y. Use o índice para encontrar todas as tuplas de S que têm t_Y em seu(s) componente(s) de Y. Essas serão exatamente as tuplas de S que participarão da junção com a tupla t de R, e assim é feita a saída da junção de cada dessas tuplas com t.

O número de operações de E/S de disco depende de vários fatores. Primeiro, supondo que R esteja agrupada em clusters, deve-se ler $B(R)$ blocos para obter todas as tuplas de R. Se R não estiver agrupada em clusters, então poderão ser necessárias até $T(R)$ operações de E/S de disco.

Para cada tupla t de R deve-se ler em média $T(S)/V(S,Y)$ tuplas de S. Se S tiver um índice não agrupado em clusters sobre Y, então o número exigido de operações de E/S de disco será $T(R)T(S)/V(S,Y)$, mas se o índice for agrupado em clusters, somente $T(R)T(S)/V(S,Y)$ operações de E/S de disco bastarão. Em qualquer caso, deve-se de adicionar algumas operações de E/S de disco para cada valor de Y, a fim de levar em conta a leitura do próprio índice.

Independente do fato de R estar ou não agrupada em clusters, o custo de acessar tuplas de S dominará, e assim é possível tomar $T(R)T(S)/V(S,Y)$ ou $T(R)(max(1, B(S)/V(S,Y)))$ como o custo desse método de junção, para os casos de índices não agrupados em clusters e agrupados em clusters sobre S, respectivamente.

Quando um índice for uma árvore B ou outra estrutura da qual se pode extrair facilmente as tuplas de uma relação em ordem classificada, existirão uma série de outras oportunidades para usar o índice. Talvez a mais simples surja quando se deseja calcular $R(K,Y) \boxtimes S(Y,Z)$ e existe um índice classificado sobre Y para R ou S. Então é possível executar uma junção de classificação comum, mas não é necessário executar a etapa intermediária de classificar uma das relações sobre Y.

Como um caso extremo, se existirem índices de classificação sobre Y para R e S, será necessário executar apenas a etapa final da junção simples baseada classificação. Às vezes, esse método é chamado *junção de ziguezague*, porque salta-se de um lado para outro entre os índices, encontrando valores de Y que eles compartilham em comum. Note que as tuplas de R com um valor de Y que não aparecem em S nunca precisam ser recuperadas e, de modo semelhante, tuplas de S cujo valor de Y não aparece em R não precisam ser recuperadas.

Se os índices são árvores B, então é possível examinar as folhas das duas árvores B em ordem a partir da esquerda, usando os ponteiros de uma folha para outra que estão incorporados na estrutura. Se R e S estão agrupadas em clusters, a recuperação de todas as tuplas com um dada chave resultará em uma série de operações de E/S de disco proporcional às frações lidas dessas duas relações. Observe que, em casos extremos, onde existem tantas tuplas de R e S que nenhuma delas se encaixa na memória principal disponível, deve-se de usar uma correção como a descrita no algoritmo de junção simples baseado em classificação. Porém, em casos típicos, a etapa de junção de todas as tuplas com um valor comum de Y poderá ser executada apenas com o número de operações de E/S de disco exigidas para a leitura dessas tuplas.

2.1.8 Gerenciamento de buffers

Apesar de ter sido suposto que operadores têm um número M de buffers disponíveis na memória principal para armazenar dados necessários, na prática, esses buffers são raramente alocados com antecedência e o valor de M pode variar. É o *gerenciador de buffers* que é responsável por distribuir a memória à processos, como consultas, que a necessitam, assim como minimizar o atraso e solicitações não atendidas.

Há duas arquiteturas gerais: Na primeira, o gerenciamento de buffers controla diretamente a memória principal. Na segunda, o gerenciador de buffers aloca buffers na memória virtual, permitindo que o sistema operacional decida quais buffers estão realmente na memória principal em qualquer momento e quais estão no "espaço de troca" em disco que o sistema operacional gerencia.

Em ambas abordagens surge um mesmo problema: o gerenciador de buffers deve limitar o número de buffers em uso, de forma que eles caibam na memória principal disponível. No primeiro caso, se as solicitações excedem o espaço disponível, ele tem de selecionar um buffer para esvaziar, retornando seu conteúdo ao disco. Caso seu conteúdo tenha sido alterado, ele é gravado novamente no disco, caso contrário, ele é simplesmente apagado. Já no segundo caso, ele consegue alocar mais buffers do que é possível encaixar na memória principal. Porém, se todos esses buffers realmente estiverem em uso, haverá "lixo", um problema onde muitos blocos são movidos para dentro e para fora do espaço de troca do disco, resultando em muito tempo gasto com trocas de blocos e pouco trabalho útil realizado.

Em geral, o número de buffers é um parâmetro definido quando o SGBD é inicializado. Por isso, não nos preocuparemos com o modo de bufferização usado, e simplesmente faremos a suposição de que há um *poll de buffers* de tamanho fixo disponível para consultas e outras ações do banco de dados.

A escolha crítica que o gerenciador de buffers deve fazer é qual bloco retirar do pool de buffers quando for necessário um buffer para um bloco recém-solicitado. As estratégias de substituição de buffers incluem:

- *Least-Recently Used* (LRU). Esta regra exclui o bloco que não foi acessado a mais tempo. Para isso é necessário que o gerenciador de buffers mantenha uma tabela indicando a última vez que o bloco de cada buffer foi acessado e que seja criada uma entrada nessa tabela cada vez que o banco de dados for acessado.

- *First-In-First-Out* (FIFO). Nesta regra o buffer ocupado há mais tempo por um mesmo bloco é esvaziado. Esse método exige que o gerenciador de buffers armazene uma tabela com a ordem que os blocos foram lidos do disco.

- *Clock*. Neste algoritmo, existe uma lista circular para os buffers com uma "mão" (iterador)

apontando para o ultimo buffer examinado. Cada buffer tem um sinalizador associado indicando 0 ou 1. Quando um bloco é lido ou acessado, seu sinalizador é definido como 1. Quando o gerenciador precisa de um buffer a "mão" roda a lista procurando pelo primeiro buffer com sinalizador 0 e redefinindo para 0 buffers com sinalizador 1.

- *Controle de sistema*. O processador de consultas ou outros componentes de um SGDB podem fornecer indicações ao gerenciador de buffers, a fim de evitar alguns equívocos que ocorreriam com uma norma estrita como LRU, FICO ou Clock. Às vezes existem razões técnicas que impedem que um bloco seja movido para o disco. Estes blocos são ditos "fixados" e qualquer gerenciador de buffers precisa modificar sua estratégia de substituição a fim de evitar expulsar blocos fixados. Isso nos dá a oportunidade de forçar alguns blocos a permanecerem na memória principal, declarando-os "fixados" a fim de evitar alguns problemas que alguns dos algoritmos ocasionaria.

O otimizador de consultas selecionará eventualmente um conjunto de operadores físicos que serão usados para executar uma dada consulta. Essa seleção de operadores pode pressupor que um certo número de buffers M está disponível para a execução de cada um desses operadores. Porém, como visto, o gerenciador de buffers pode não estar disposto ou não ser capaz de garantir a disponibilidade desses M buffers quando a consulta for executada. Desse modo, há duas questões inter-relacionadas sobre os operadores físicos. O algoritmo pode se adaptar a mudanças no valor de M. Quando os M buffers esperados não estão disponíveis, e alguns blocos que deveriam estar na memória tiverem realmente sido movidos para o disco pelo gerenciador de buffers, como a estratégia de substituição de buffers usada pelo gerenciador de buffers influenciará o número de operações de E/S de disco adicionais que terão de ser executadas.

Sobre estas questões, é possível observar que caso seja utilizado um algoritmo baseado em classificação para algum operador, então será possível adaptar-se a mudanças em M. Se M se contrair, é possível mudar o tamanho de uma sublista, pois os algoritmos baseados em classificação discutidos não dependem do fato das sublistas terem o mesmo tamanho. A principal limitação é que, à medida que M se contrai, pode haver a necessidade de se criar tantas sublistas que não será possível alocar em seguida um buffer para cada sublista no processo de mesclagem. Também pode-se observar que a classificação de sublistas na memória principal pode ser executada por vários algoritmos diferentes. Tendo em vista que algoritmos como merge-sort e quicksort são recursivos, a maior parte do tempo será gasta em regiões de memória bastante pequenas. Desse modo, a estratégia LRU ou FIFO funcionará bem para essa parte de um algoritmo baseado em classificação. Outra observação, é que se o algoritmo for baseado em hash, é possível reduzir o número de depósitos se M se contrair, desde que

os depósitos não fiquem tão grandes que não possam caber na memória principal alocada. Porém, diferente do que ocorre nos algoritmos baseados em classificação, não é possível responder a mudanças em M enquanto o algoritmo é executado. Em vez disso, depois de escolhido o número de depósitos, ele permanecerá fixo durante a primeira passagem e, se buffers se tornarem indisponíveis, os blocos pertencentes a alguns depósitos terão de ser trocados.

2.1.9 Algoritmos que utilizam mais de duas passagens

Embora duas passagens serem normalmente suficientes para todas as operações, é possível generalizar os algoritmos baseados em classificação e hash para usarem tantas passagens quantas necessárias.

Algoritmos de várias passagens baseados em classificação

Suponha que há M buffers da memória principal disponíveis para classificar uma relação R armazenada, que será considerada agrupada em clusters. Então, se faz o seguinte:

BASE: Se R couber na memória principal, leia R para a memória principal, classifique-a usando seu algoritmo de classificação da memória principal favorito e grave a relação classificada em disco..

INDUÇÃO: Se R não couber na memória principal, particione os blocos que contêm R em M grupos, que chamaremos R_1, R_2, ..., R_M. Classifique R_i recursivamente para cada $i = 1, 2, ..., M$. Em seguida, faça a mesclagem das M sublistas classificadas.

Se não estiver apenas classificando R, mas executando uma operação unária, o procedimento anterior será modificado de forma que na mesclagem final, seja executada a operação sobre as tuplas do início das sublistas classificadas. Isto é:

- Para o operador δ, faça a saída uma cópia de cada tupla distinta e ignore as cópias da tupla.
- Para o operador γ, classifique apenas sobre os atributos de agrupamento e combine as tuplas com um dado valor desses atributos de agrupamento da maneira apropriada.

Para executar uma operação binária, utiliza-se essencialmente a mesma idéia, exceto pelo fato de que as duas relações serão primeiro divididas em um total de M sublistas. Depois, cada sublista será classificada pelo algoritmo recursivo anterior. Finalmente, lê-se cada uma das M sublistas, cada uma em um buffer, e a operação é executada de alguma maneira descrita seção 2.1.5.

É possível dividir os M buffers entre as relações R e S de qualquer maneira. Porém, para minimizar o número total de passagens, em geral os buffers são divididos proporcionalmente ao número de blocos usados pelas relações. Isto é, R receberá $M \times B(R)/(B(R) + B(S))$ dos buffers, e S receberá os restantes.

Seja $s(M, k)$ o tamanho máximo de uma relação que se pode classificar usando M buffers e k passagens. Então, é possível calcular $s(M, k)$ desta forma:

BASE: Se $k = 1$, é necessário ter $B(R) \leq M$. Em outras palavras $s(M, 1) = M$.

INDUÇÃO: Suponha que $k > 1$. Então, R é particionado em M fragmentos, cada um dos quais deverá ser classificado em $k - 1$ passagens. Se $B(R) = s(M, k)$, então $s(M, k)/M$, que é o tamanho de cada um dos M fragmentos de R, não poderá exceder $s(M, k-1)$. Isto é: $s(M, k) = Ms(M, k-1)$.

Expandindo a recursão anterior, se tem:

$$s(M, k) = Ms(M, k-1) = M^2 s(M, k-2) = ... = M^{k-1} s(M, 1)$$

Tendo em vista que $s(M, 1) = M$, se concluí que $s(M, k) = M^k$. Ou seja, usando k passagens, é possível classificar uma relação R se $B(R) \leq s(M, k)$, o que significa que $B(R) \leq M^k$. Em outras palavras, para classificar R em k passagens, o número mínimo de buffers que se pode usar será $M = (B(R))^{1/k}$.

Cada passagem de um algoritmo de classificação lê todos os dados do disco e os grava outra vez. Desse modo, um algoritmo de classificação de k passagens exige $2kB(R)$ operações de E/S de disco.

Agora, considerando o custo de uma junção de várias passagens $R(K, Y) \boxtimes S(Y, Z)$ como representante de uma operação binária sobre relações, seja $j(M, k)$ o maior número de blocos tais que, em k passagens e usando M buffers, é possível fazer a junção de relações de $j(M, k)$ blocos ou menos no total. Isto é, a junção pode ser realizada desde que $B(R) + B(R) \leq j(M, k)$.

Na última passagem, M sublistas classificadas das duas relações são mescladas. Cada uma das sublistas é classificada com o uso de $k-1$ passagens, e assim elas não podem ser maiores que $s(M, k-1) = M^{k-1}$ cada uma, ou um total de $Ms(M, k-1) = M^k$. Ou seja, $B(R) + B(S)$ não poderão ser maiores que M^k ou, em outras palavras, $j(M, k) = M^k$. Invertendo o papel dos parâmetros, também se pode afirmar que o cálculo da junção em k passagens exigirá $(B(R) + B(S))^{1/k}$ buffers.

Para calcular o número de operações de E/S de disco necessárias nos algoritmos de várias passagens, devemos lembrar que, diferente da classificação, não é contado o custo da gravação do resultado final em disco para junções ou outras operações relacionais. Desse modo, é utilizado $2(k-1)(B(R)+B(S))$ operações de E/S de disco para classificar as sublistas, e outras $B(R)+B(S)$ operações de E/S de disco para ler as sublistas classificadas na passagem final. O resultado é um total de $(2k-1)(B(R) + B(S))$ operações de disco.

Algoritmos de várias passagens baseados em hash

Há uma abordagem recursiva correspondente para o uso do hash em operações sobre grandes relações. É feito o hash da relação ou das relações em $M - 1$ depósitos, onde M é o número de buffers da memória disponíveis. Em seguida, aplica-se a operação a cada depósito individualmente, no caso de uma operação unária. Se a operação for binária, é aplicada a operação a cada par de depósitos correspondentes, como se eles fossem as relações inteiras. Para as operações relacionais comuns que consideradas - eliminação de duplicatas, agrupamento, união, interseção, diferença, junção natural - o resultado da operação sobre a(s) relação(ões) inteira(s) será a união dos resultados sobre o(s) depósito(s). É possível descrever esta abordagem recursivamente como:

BASE: Para uma operação unária, se a relação couber em M buffers, lê-se a relação para a memória e executa-se a operação. Para uma operação binária, se uma outra relação couber em $M - 1$ buffers, a operação será executada lendo essa relação para a memória principal e, em seguida, a segunda relação será lida, um bloco de cada vez, para o M-ésimo buffer.

INDUÇÃO: Se nenhuma relação couber na memória principal, é feito o hash de cada relação em $M - 1$ depósitos. Se executa recursivamente a operação sobre cada depósito ou par de depósitos correspondente, e é acumulada a saída de cada depósito ou par.

Será feita a suposição de que, quando executado o hash de uma relação, as tuplas se dividem da maneira mais uniforme possível entre os depósitos. Na prática, essa suposição é satisfeita de forma razoável se escolheda uma função de hash verdadeiramente aleatória, mas sempre haverá alguma desigualdade na distribuição de tuplas entre depósitos.

Primeiro, considere uma operação unária sobre a relação R usando M buffers. Seja $u(M, k)$ o número de blocos na maior relação que um algoritmo de hash de k passagens pode manipular. É possível definir u recursivamente por:

BASE: $u(M, 1) = M$, pois a relação R deve se encaixar em M buffers; isto é, $B(R) \leq M$.

INDUÇÃO: Supondo que a primeira etapa divide a relação R em $M - 1$ depósitos de igual tamanho, é possível calcular $u(M, k)$ como a seguir. Os depósitos para a próxima passagem deverão ser suficientemente pequeno para que seja possível tratá-los em $k - 1$ passagens, isto é, os depósitos terão o tamanho $u(M, k - 1)$. Tendo em vista que R está dividida em $M - 1$ depósitos, é necessário ter $u(M, k) = (M - 1)u(M, k - 1)$.

Se a recorrência anterior for expandida, se chega a $u(M, k) = M(M - 1)^{k-1}$, ou aproximadamente, supondo-se que M seja grande, $u(M, k) = M^k$. De forma equivalente, é possível executar uma das operações relacionais unárias sobre a relação R em k passagens com M buffers, desde que $M \leq (B(R))^{1/k}$.

Pode-se fazer uma análise semelhante para operações binárias. Considere a junção. Seja $j(M, k)$ um limite superior sobre o tamanho da menor das duas relações R e S envolvidas em $R(K, Y) \boxtimes S(Y, Z)$. Aqui, como antes, M é o número de buffers disponíveis e k é o número de passagens que se pode usar.

BASE: $j(M, 1) = M - 1$; isto é, se for utilizado o algoritmo de uma passagem para junção, R ou S deverá caber em $M - 1$ blocos.

INDUÇÃO: $j(M, k) = (M - 1)j(M, k - 1)$; ou seja, na primeira de k passagens, é possível dividir cada relação em $M - 1$ depósitos e se esperar que cada depósito seja $1/(M - 1)$ de sua relação inteira, mas deve-se então ser capaz de fazer a junção de cada par de depósitos correspondentes em $M - 1$ passagens.

Expandindo a recorrência para $j(M, k)$, se concluí que $j(M, k) = (M - 1)^k$. Mais uma vez supondo que M seja grande, se pode afirmar que $j(M, k) = M^k$, aproximadamente. Isto é, a junção de $R(K, Y) \boxtimes S(Y, Z)$ pode ser feita usando k passagens e M buffers, desde que $M^k \geq min(B(R), B(S))$.

2.2 Linguagens de consulta

Diversas linguagens de consulta surgiram juntamente ao modelo relacional de banco de dados. Estas linguagens se baseiam no modelo relacional proposto por Codd (1970) e têm como precursora a linguagem SQL. A partir desta primeira linguagem, várias outras surgiram, geralmente se baseando no padrão estabelecido por ela. Porém, com o advento da orientação a objetos, várias linguagens de consulta passaram a incorporar os conceitos deste paradigma, como herança e polimorfismo.

Nesta seção serão apresentas as linguagem de consulta SQL, OQL, HQL e Criteria.

2.2.1 SQL

A linguagem SQL, Structured Query Language, foi a primeira linguagem comercial baseada no modelo relacional de Codd (CODD, 1970). Ela foi desenvolvida inicialmente por Chamberlin (CHAMBERLIN; BOYCE, 1974) em 1974 na IBM com o nome SEQUEL e tinha como objetivo manipular e recuperar dados do banco de dados System R (CHAMBERLIN et al., 1981). Rapidamente surgiram variações criadas por outras empresas, o que fez com que, em 1986, o SQL se tornasse um padrão da American National Standards Institute (ANSI) e, em 1987, do International Organization for Standards (ISO).

SQL é uma linguagem declarativa de definição e manipulação de dados. Suas instruções são dadas

na forma de declarações, consistindo de declarações específicas do SQL e parâmetros e operadores adicionais que se aplicam a estas declarações.

A linguagem SQL pode ser dividida de acordo com a operação que desejada. A Linguagem de Definição de Dados (*Data Definition Language* ou DDL) é o conjunto de comandos responsável pela definição das estruturas de dados. Ela possui as instruções que permitem a criação, modificação e remoção das tabelas, assim como criação de índices. Os comandos *CREATE*, *ALTER* e *DROP* compõem esta linguagem e através deles pode ser definida a estrutura de uma base de dados, incluindo as linhas, colunas, tabelas, índices, e outros metadados.

Para a inclusão, remoção e modificação de informações em um banco de dados, utiliza-se o grupo de comandos presentes na Linguagem de Manipulação de Dados (*Data Manipulation Language* ou DML). Estes comandos são os comandos *INSERT*, *UPDATE* e *DELETE*.

O controle de acesso aos dados do banco é realizado pelos comandos presentes na Linguagem de Controle de Dados (*Data Control Language* ou DCL). Dentro deste grupo de comandos temos o comando *GRANT*, que é utilizado para permitir que usuários especificados realizem tarefas especificadas, e o comando *REVOKE*, que é utilizado para cancelar permissões previamente concedidas ou negadas.

A operação mais comum em SQL é a consulta e para esta tarefa utiliza-se a Linguagem de Consulta de Dados (*Data Query Language* ou DQL). Esta linguagem é baseada no comando *SELECT* que recupera dados de uma ou mais tabelas ou expressões. O comando *SELECT* padrão não tem efeitos persistentes no banco de dados e ele permite ao usuário descrever os dados desejados, encarregando o sistema de gerenciamento do banco de dados (DBMS) de planejar, otimizar e executar as operações físicas necessárias para produzir o resultado requisitado.

Uma consulta possui uma lista de colunas a serem incluídas no resultado final, que seguem a palavra chave *SELECT*. Também é possível utilizar um asterisco (*) para especificar que a consulta deve retornar todas as colunas das tabelas consultadas. Para a descrição dos dados pelo usuário, o comando *SELECT* faz uso de palavras chaves e cláusulas opcionais.

Para indicar a(s) tabela(s) de onde os dados serão recuperados, utiliza-se a cláusula *FROM*. Esta cláusula pode incluir subcláusulas *JOIN*, que são utilizadas para especificar regras para unir tabelas que estão relacionadas.

Pode-se restringir o resultado retornado pela consulta através da cláusula *WHERE*. Esta cláusula é composta por um ou mais predicados comparativos, que eliminam do resultados as linhas que não resultam em verdadeiro.

A cláusula *GROUP BY* é usada para projetar linhas com valores comuns em conjuntos menores de linhas. *GROUP BY* é geralmente usado em conjunto com funções SQL de agregação ou para eliminar

linhas duplicadas do conjunto resultante. A cláusula *WHERE* é aplicada antes da cláusula *GROUP BY*.

Assim como a cláusula *WHERE*, a cláusula *HAVING* restringe o resultado da consulta, porém ele é aplicado às linhas resultantes da cláusula *GROUP BY*. Ela também é composta por predicados comparativos, porém utiliza funções de agregação para tal. É possível a utilização da função *SUM*, que permite a soma de um campo numérico, da função *AVG*, que retorna o valor médio entre do conjunto de valores de um campo numérico, da função *COUNT*, que conta a quantidade de dados de um campo dado, da função *MAX*, que retorna o maior valor encontrado entre os dados de um campo dado, e da função *MIN*, que retorna o menor valor encontrado entre os dados de um campo dado.

Também é possível ordenar o resultado da consulta através da cláusula *ORDER BY*, que identifica quais colunas serão utilizadas para se realizar a ordenação. Nesta cláusula também é possível indicar a direção que o resultado deve ser ordenado (ascendente ou decrescente).

A figura 2.1 é um exemplo de uma consulta. A cláusula *SELECT* especifica que a consulta retornará nomes de alunos juntamente a soma de suas quantidades de crédito. A cláusula *FROM* especifica que as tabelas disciplina, aula e aluno serão utilizadas. A cláusula *WHERE* relaciona estas 3 tabelas através das chaves primárias e especifica que apenas os créditos de matérias onde os alunos obtiveram uma nota superior a 7.0 serão contabilizados. A cláusula HAVING especifica que apenas alunos que já completaram mais de 70 créditos serão retornados pela consulta. A cláusula GROUP BY agrupa os alunos por nome. Por fim, a cláusula ORDER BY ordena o resultado pelo nome dos alunos. Desta forma a consulta recupera, em ordem alfabética, o nome e a quantidade de créditos dos alunos que já completaram mais de 70 créditos, levando em conta apenas as matérias em que foram aprovados (média 7.0).

```
1  select al.nome, sum(di.QuantidadeCreditos)
2  from disciplina di, aula au, aluno al
3  where di.numdisp = au.numdisp and
4  au.numaluno = al.numaluno and
5  au.note >= 7.0
6  group by al.nome;
7  having sum(di.quantidadecreditos) >=70
8  order by al.nome;
```

Figura 2.1: Exemplo de consulta *SQL*

2.2.2 OQL

A linguagem de consulta OQL, Object Query Language, foi introduzida pelo padrão ODMG-93 em 1993 (CATTELL, 1993) e é baseada no modelo de objetos ODMG, que teve como versão final a

versão 3.0 (BERLER et al., 2000).

Esta linguagem é bem próxima ao SQL, porém, se estende as noções de orientação a objetos, como por exemplo polimorfismo e invocação de métodos, além de fornecer primitivas de alto nível para lidar com conjuntos de objetos. A linguagem também fornece primitivas para lidar com structs, listas e vetores e trata tais estruturas com a mesma eficiência.

OQL é uma linguagem funcional onde os operadores podem ser compostos livremente, contando que eles respeitem o sistema de tipos. Isto é uma consequência do fato que o resultado de qualquer consulta tem um tipo que pertence ao modelo de tipos ODMG e portanto pode ser consultado novamente. Porém, a linguagem OQL não é computacionalmente completa, ela é uma linguagem de consulta de fácil uso e é invocada dentro de outras linguagens onde um ODMG binding é definido. Com isto a linguagem é restrita ao mesmo sistema de tipos e consequentemente pode invocar operações programadas nestas linguagens.

Como uma linguagem embutida, OQL permite consultar objetos denotáveis que são suportados pela linguagem nativa através de expressões produzindo objetos atômicos, estruturas, coleções e literais. Uma consulta OQL é uma função que retorna um objeto. O tipo deste objeto pode ser inferido a partir dos operadores que fazem parte da expressão de consulta. O exemplo abaixo ilustra este ponto.

Imagine um esquema que defina os tipos *Pessoa* e *Empregado*. O tipo *Pessoa* define *nome*, *dataDeNascimento* e *sexo* como atributos e a operação *idade* e o tipo *Empregado*, um subtipo de *Pessoa*, define *salario* como atributo, o relacionamento *subordinados* e a operação *precedencia*. Estes tipos tem as extensões *Pessoas* e *Empregados*, respectivamente.

```
1  select distinct p.idade
2  from Pessoas p
3  where p.nome = "João"
```

Figura 2.2: Inferindo o tipo de retorno de uma consulta

A consulta da figura 2.2 retorna conjunto de idades de todas pessoas chamadas João, retornando, desta forma, um literal do tipo set<integer>.

A linguagem OQL suporta objetos (ou seja, possuindo um OID) e literais (identidade igual ao seu valor), dependendo do modo que estes objetos são construídos ou selecionados.

Uma consulta em OQL pode retornar quatro tipos de resultados. Ela pode retornar uma coleção de objetos com identidade; um objeto com identidade; uma coleção de literais; ou um literal. Portanto, o resultado de uma consulta é um objeto com ou sem OID sendo que alguns objetos são gerados pelo interpretador da linguagem de consulta, enquanto outros são produzidos a partir do próprio banco de dados.

Quando um objeto é obtido, é necessário navegar sobre ele para alcançar o dado desejado. Para fazer isto em OQL, utiliza-se a notação "." (ou simplesmente " ->"), que possibilita entrar em objetos complexos, assim como em relacionamentos simples. Por exemplo, há uma *Pessoa p* e deseja-se saber o nome da cidade onde a cônjuge desta pessoa vive. Utiliza-se simplesmente *p.conjuge.endereco.cidade.nome*. Esta consulta começa em Pessoa, obtêm seu cônjuge, que é uma Pessoa novamente, navega dentro do atributo complexo do tipo Endereço para acessar o objeto Cidade, que assim tem o nome acessado.

Este exemplo trata uma relação 1-1. Porém, para uma relação n-p, é necessário o uso da cláusula select-from-where para acessar o dado desejado. Para acessar os nomes dos filhos da Pessoa p, por exemplo, não se pode escrever p.filhos.nome porque filhos é uma lista de referências. Para isto, deve-se utilizar o comando *select*, demonstrado na figura 2.3. O resultado desta consulta é um valor do tipo *bag<string>*.

```
1  select f.nome
2  from p.filhos f
```

Figura 2.3: Acessando uma relação n-p

Deste modo, OQL oferece um meio para navegar a partir de um objeto para qualquer objeto seguindo qualquer relacionamento e entrar em qualquer subvalores complexos de um objeto. Por exemplo, deseja-se que o conjunto de endereços das crianças de cada *Pessoa* do banco de dados. Sabe-se que a coleção nomeada *Pessoas* contém todas as pessoas do banco de dados. Agora tem-se que atravessar duas coleções: *Pessoas* e *Pessoa.filhos*. Como em SQL, o operador *select-from* permite consultar mais de uma coleção. Essas coleções aparecem então na cláusula *from*. Em OQL, uma coleção na cláusula *from* pode ser derivada a partir de uma cláusula anterior, seguindo um caminho que se inicia a partir dela.

```
1  select f.endereco
2  from Pessoas p, p.filhos f
```

Figura 2.4: Utilizando uma coleção derivada

A consulta da figura 2.4 acessa todas as crianças de todas as pessoas. O resultado é um valor do tipo bag<Endereço>.

Quando deseja-se restringir os resultado de uma consulta, utiliza-se a cláusula *where*, onde qualquer predicado pode ser definido. Estes predicados podem ser atômicos ou complexos. Predicados complexos são construídos através da combinação de predicados atômicos com operadores booleanos *and*, *or* e *not*. OQL suporta as versões especiais de *and* e *or*, ou seja, *andthen* e *orelse*. Estes

dois operadores permitem a avaliação condicional de seu segundo operando e também ditam que o primeiro operando deve ser avaliado primeiro.

O processamento de *joins* entre estas coleções que não são diretamente relacionadas é realizado na cláusula *from*. A consulta da figura 2.5 retorna pessoas que possuem o nome de uma flor, assumindo a existência de um conjunto de todas as flores chamado Flores.

```
1  select p
2  from Pessoas p, Flores f
3  where p.nome = f.nome
```

Figura 2.5: Realização de *join* através da cláusula *from*

A linguagem OQL também trás um valor literal especial para tratar o acesso a uma propriedade de um objeto nulo. Este é o valor *UNDEFINED*, que é um valor valido para qualquer tipo literal ou objeto dentro da linguagem OQL. Para se manipular este literal utiliza-se uma série de regras:

- is_undefined(UNDEFINED) retorna verdadeiro; is_defined(UNDEFINED) retorna falso.
- Se o predicado que é definido pela cláusula where retorna UNDEFINED, ele é tratado como se o predicado retornasse falso.
- UNDEFINED é um valor valido para cada expressão de construção explicita ou cada construção de coleção implícita, isto é, uma coleção construída implicitamente (por select-from-where) pode conter UNDEFINED.
- UNDEFINED é uma expressão válida para a operação de agregação count.
- Qualquer outra operação com qualquer operando UNDEFINED resulta em UNDEFINED.

Devido ao OQL ser orientado a objeto, ele permite a invocação de um método com ou sem parâmetros em qualquer caso em que o tipo de retorno do método corresponde ao tipo esperado na consulta. A notação para invocar um método é exatamente a mesma para acessar um atributo ou percorrer um relacionamento, no caso onde o método não possuir parâmetros. Se ele possuir parâmetros, estes são dados entre parenteses. Porém, se houver um conflito de nomes entre um atributo e um método, então o método pode ser chamado com parenteses para resolver este conflito.

Uma das principais contribuições da orientação a objetos é a possibilidade de manipular coleções polimórficas e, graças ao mecanismo de *late binding*, executar ações genéricas sobre os elementos destas coleções. Por exemplo, o conjunto *Pessoas* contêm objetos das classes *Pessoa*, *Empregado* e *Estudante*. Portanto, todas as consultas contra *Pessoas* lidam com as três possíveis classes de elementos da coleção.

Uma consulta é uma expressão na qual operadores operam sobre operandos tipificados. Uma consulta está correta se os tipos dos operandos correspondem àqueles requeridos pelos operadores. Neste sentido, OQL é uma linguagem de consulta tipificada. Está em uma condição necessária para um otimizador de consultas eficiente. Quando uma coleção polimórfica é filtrada (por exemplo, *Pessoas*), seus elementos são estaticamente conhecidos por serem daquela classe (por exemplo *Pessoa*). Isto significa que uma propriedade de uma subclasse (atributo ou método) não pode ser aplicada a tal elemento, exceto em dois casos importantes: *late binding* para um método ou indicação explicita de classe.

No *late bind*, uma operação pode ser invocada de todas as subclasses de uma classe mãe. Por exemplo, dado as atividade de cada *pessoa* na figura 2.6, onde *atividades* é um método que tem três personificações. Dependendo do tipo de pessoa do *p* atual, a personificação correta é invocada. Se *p* é um *Empregado*, OQL invoca a operação atividades definida sobre este objeto, ou se *p* é um *Estudante*, OQL invoca a operação atividades da tipo *Estudante*, ou se *p* é uma *Pessoa*, OQL invoca o método atividades do tipo *Pessoa*.

```
1  select p.atividades
2  from Pessoas p
```

Figura 2.6: *Late bindind* sobre *Pessoas*

Também é possível fazer a indicação de classe, onde, para navegar na hierarquia de classes, um usuário pode explicitamente declarar a classe de um objeto que não pode ser inferido estaticamente. O *evaluator* então tem de checar em tempo de execução que o este objeto realmente pertence à classe indicada (ou à alguma de suas subclasses). Por exemplo, assumindo que se sabe que apenas *Estudantes* utilizam seu tempo seguindo um plano de estudo, podemos selecionar estas *Pessoas* e obter suas notas. Nos indicamos explicitamente na consulta que estas *Pessoas* são da classe *Estudante*.

```
1  select ((Estudante)p). nota
2  from Pessoas p
3  where "plano de estudo" in p.atividades
```

Figura 2.7: Indicação explicita da classe *Estudante*

OQL é uma linguagem puramente funcional. Todos os operadores podem ser compostos livremente contanto que o sistema de tipo seja respeitado. Este é o motivo pelo qual a linguagem é tão simples. Esta filosofia é diferente da do SQL, o qual é uma linguagem *ad hoc* cuja regras de composição não são ortogonais. A adoção de uma ortogonalidade completa permite que o OQL não restrinja o poder das expressões e torna a linguagem mais simples de se aprender, sem perder a sintaxe SQL para consultas mais simples.

Entre os operadores oferecidos por OQL, mas ainda não introduzidos, é possível mencionar os operadores de conjunto (*union*, *intersect*, *except*), os qualificadores universais (*for all*) e existenciais (*exists*), a ordenação e agrupamento por operadores e os operadores de agregação (*count*, *sum*, *min*, *max*, e *avg*).

2.2.3 HQL

A linguagem Hibernate Query Language, ou simplesmente HQL, é a linguagem utilizada pelo *framework* Hibernate. Assim como a maioria das linguagens de consulta, ela é semelhante ao SQL. No entanto, HQL é uma linguagem totalmente orientada à objetos, compreendendo assim as noções de herança, polimorfismo e associações.

O Hibernate é um *framework* para o mapeamento objeto-relacional escrito na linguagem Java. Ele facilita o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação, mediante o uso de arquivos XML ou anotações Java. Devido a isso, consultas HQL são realizadas sobre objetos, porém, o Hibernate as traduz para consultas SQL sobre tabelas.

Como uma linguagem orientada a objetos, as consultas em HQL são polimórficas, isto é, elas também retornam instâncias de todas as classes persistentes que estendam a determinada classe ou implemente a determinada interface.

A cláusula *FROM* é responsável por definir o escopo de tipos de objeto do modelo para o resto da consulta. Ela também é responsável por definir as variáveis de identificação que serão válidas para o resto da consulta. Variáveis de identificação são associações às classes do modelo de objetos feitas na cláusula *FROM* que podem então ser utilizadas para se referir àquele tipo no resto da consulta.

A cláusula *FROM* também pode conter junções de relações explicitas utilizando a palavra chave *join*. Estas junções podem ser *inner join* ou *left outer join*. A consulta abaixo faz a junção das relações *Item* e *Oferta* através do atributo *item_id*. Como se pode perceber, a palavra chave *inner* não é necessária, assim como a palavra chave *outer*. Além disso, a palavra chave *on* é utilizada para estabelecer por qual atributo será feito a junção.

```
1   from Item i join Oferta O on i.item_id = b.item_id
```

Figura 2.8: Realização de *join* entre *Item* e *Oferta*

Um importante caso de uso para junções explicitas é a definição de *FETCH JOINS*. Quando um objeto possui uma coleção e/ou associação, por padrão, elas não são inicializadas juntamente ao objeto que elas pertencem. O Hibernate gera um *proxy* e carrega a coleção associada lentamente e somente sobre demanda. Um *FETCH JOIN* faz com que coleções e associações sejam inicializadas

juntamente ao objeto que elas pertencem, tornando assim a consulta mais rápida. Para se usar este tipo de junção se adiciona a palavra chave *fetch* ao lado direito da palavra chave *join*.

Assim como em SQL, a cláusula *WHERE* é utilizada para restringir consultas. Esta restrição é expressa usando a lógica ternária. A cláusula *WHERE* é uma expressão lógica que pode resultar em verdadeiro, falso ou nulo para cada tupla dos objetos. É possível construir expressões lógicas, comparando as propriedades dos objetos à outras propriedades ou valores literais usando operadores de comparação.

O exemplo da figura 2.9 a seguir restringe todos usuários, de acordo com seu e-mail.

```
1  from Usuario u
2  where u.email = 'joao@gmail.com'
```

Figura 2.9: Utilização da cláusula *WHERE*

HQL suporta os mesmos operadores básicos de SQL. Porém, devido a lógica ternária, testar valores nulos requerem um cuidado especial. Em SQL, *null* = *null* não resulta em verdadeiro, pois todas comparações que utilizam o operando *null* resultam em *null*. Por isto HQL fornece operando *is [not] null* para avaliar se uma propriedade é nula.

Além de expressões que tratam propriedades quem possuem um único valor, a cláusula WHERE também pode tratar propriedades que resultam em coleções, com os devidos operadores. *is not empty* e *member of* são dois exemplos de operadores utilizados com coleções e, respectivamente, avaliam se uma coleção não esta vazia e se um dado objeto faz parte de uma coleção.

HQL também possuí uma poderosa característica, a chamada de funções, que permite a chamada de funções SQL na cláusula *WHERE*. Por exemplo, as funções do padrão *ANSI SQL UPPER* e *LOWER*, podem ser utilizadas para uma busca *case-insensitive*, como na figura 2.10.

```
1  from Usuario u where lower(u.email) = 'joao@gmail.com'
```

Figura 2.10: Utilização da função *lower*

Agora que a cláusula WHERE foi apresentada, deve-se dizer que há também a possibilidade de se realizar junções implicitamente. Junções implícitas são realizadas simplesmente pelo uso de expressões com caminhos.

A figura 2.11 apresenta duas consultas equivalentes, porém uma utiliza a cláusula *join*, enquanto a outra realiza o *join* implicitamente.

A projeção em HQL é feita através da cláusula *SELECT*. Ela permite especificar exatamente quais objetos ou propriedades apareceram no resultado da consulta. A cláusula *SELECT* também permite o uso da palavra chave *DISTINCT* para eliminar duplicatas no resultado. Além disto, assim como a cláusula *WHERE*, é possível invocar funções SQL através da cláusula *SELECT*.

```
1  from Oferta o
2  where o.item.tipo = 'Computador'
3
4  from Oferta o join Item i on i.item_id = b.item_id
5  where i.tipo = 'Computador'
```

Figura 2.11: *Join* implícito

As funções de agregação *COUNT*, *AVG*, *MIN*, *MAX* e *SUM* também são expressões válidas em HQL e são utilizadas do mesmo modo que em SQL. Elas geralmente aparecem junto a agrupamento.

A cláusula *GROUP BY* permite a construção de resultados agregados. Qualquer propriedade de uma classe retornada ou componentes podem ser usados para realizar a agregação. Em uma consulta onde a agregação é usada, a cláusula *WHERE* é aplicada sobre os valores não agregado. Para fazer a restrição de valores agregados, a cláusula *HAVING* é utilizada.

Para fazer a ordenação do resultado, assim como SQL, HQL utiliza a cláusula *ORDER BY*. Esta cláusula utiliza uma propriedade de uma classe para fazer a ordenação, e pode conter as palavras chaves *asc* e *desc* para indicar se a ordenação deve ser crescente ou decrescente.

2.2.4 Criteria

A API Java Persitence (BAUER; KING, 2005) apresenta uma forma de consulta tipificada, isto é, com segurança sobre tipos, prevenindo assim que erros de tipo sejam cometidos. Esta consulta é realizada através da a API Criteria.

A API Criteria é uma forma programática e segura em relação tipos de expressar consultas. Ela possuí segurança sobre tipos em termos de utilizar interfaces e classes para representar varias partes da estrutura de uma consulta, como a consulta em si, a cláusula *SELECT*, a cláusula *WHERE*, e assim por diante. Ela também pode oferecer segurança sobre tipo em termos de referenciar atributos.

Uma consulta Criteria é essencialmente um grafo de objetos, onde cada parte do grafo representa, conforme se navega adentro deste grafo, uma parte mais atômica da consulta.

A consulta da figura 2.12 representa uma das consulta mais simples feitas em SQL:

```
1  select p from Pais p
```

Figura 2.12: Consulta simples em SQL

A consulta equivalente utilizando a API Criteria é apresentada na figura 2.13.

A interface *CriteriaBuilder* desempenha o papel de fabrica para a consulta e seus elementos. Ela pode ser obtida tanto pelo método *getCriteriaBuilder* da classe *EntityManagerFactory* como pelo método *getCriteriaBuilder* da classe *EntityManager*.

```
1  CriteriaBuilder cb = em.getCriteriaBuilder();
2  CriteriaQuery<Pais> q = cb.createQuery(Pais.class);
3  Root<Pais> p = q.from(Pais.class);
4  q.select(p);
```

Figura 2.13: Consulta simples em Criteria

Na figura 2.13, uma instancia *CriteriaQuery* é criada para representar a consulta que será construída. Então uma instancia *Root* é criada para definir a variável que será utilizada na cláusula *FROM*. Por fim, a variável *p*, é usada na cláusula *SELECT* como a expressão de resultado da consulta.

As consultas feitas pela API Criteria, assim como em SQL, são composta por cláusulas.

A cláusula *SELECT* pode ser montada de várias formas. A mais simples delas é quando se utiliza uma única expressão. A figura 2.14 ilustra este caso. O método *select* recebe um argumento do tipo *Selection* e o estabelece como conteúdo para a cláusula *SELECT* (sobrescrevendo qualquer conteúdo que tenha sido estabelecido anteriormente). Qualquer expressão válida para API Criteria pode ser usada como argumento, pois elas são representadas por uma sub interface de *Selection*, a interface *Expression*. O método *distinct* pode ser usado para eliminar duplicatas no resultado, como apresentado no exemplo.

```
1  CriteriaQuery<Pais> q = cb.createQuery(Pais.class);
2  Root<Pais> c = q.from(Pais.class);
3  q.select(c.get("moeda")).distinct(true);
```

Figura 2.14: Utilização do método *select*

A interface *Selection* também é interface mãe da interface *CompoundSelection*, que é utilizada para fazer seleções múltiplas. A interface *CriteriaBuilder* fornece três métodos de fabrica para construir instâncias *CompoundSelection*.

O primeiro deles é o método *array*. Ele constrói uma instância *CompoundSelection* que representa resultados como vetores.

A segunda alternativa é o método *tuple*. Este método constrói uma instância *CompoundSelection* que representa os resultado através da interface *Tuple* ao invés de vetores. Está interface fornece vários métodos para acessar os dados resultantes.

O terceiro método para realizar seleções múltiplas é o método *construct*. Neste método, a instância *CompoundSelection* representa o resultado através de uma classe definida pelo usuário.

As instâncias *CompoundSelection* podem ser criadas pelo um método de fábrica do *CriteriaBuilder* e serem passados para o método *select*. Um outro modo de criar instâncias *CompoundSelection* é através da interface *CriteriaQuery*. Ela fornece o método *multiselect* que recebe um número variável de argumentos representando a seleção múltipla e constrói uma instância *CompoundSelection*.

O modo como o resultado será apresentado (vetor, *Tuple* ou classe do usuário) é indicado durante a instanciação da *CriteriaQuery*.

A cláusula *FROM* declara variáveis de identificação de consulta que fazem iteração sobre objetos no banco de dados. Existem dois tipos de variáveis de identificação de consulta, as variáveis *Range* e as variáveis *Join*, e cada uma delas é representada por uma sub interfaces da interface *From*.

As variáveis *Range* iteram sobre todos os objetos do banco de dados da hierarquia de uma especifica classe. Ela é representada pela interface *Root*. O método *from* da classe *CriteriaQuery* é utilizado para produzir instâncias *Root* e diferentemente de outros métodos desta classe, sua invocação não sobrescreve uma invocação anterior do método, ou seja, toda vez que o método é invocado uma nova variável é incluída na consulta.

As variáveis *Join* representam uma iteração mais limitada sobre coleções de objetos. Elas são representadas pela interface *Join* e suas sub interfaces. A interface *From* fornece varias formas do método *join*, sendo possível realizar *left joins*, *right joins* e *inner joins*. Cada vez que o método *join* é invocado, uma nova variável *Join* é adicionada a consulta e, uma vez que a interface *From* é mãe tanto de *Root* quanto de *Join*, métodos *join* podem ser invocados tanto em instâncias *Root* quanto sobre instâncias *Join* construídas previamente.

Estas duas interfaces, *Root* e *Join*, também são sub interfaces da interface *Path*. A interface *Path* é responsável por representar expressões de caminho e fornece o método *fetch* para criar *fetch joins*.

Para configurar a cláusula *WHERE*, a interface *CriteriaQuery* fornece dois métodos *where*. O primeiro deles recebe uma argumento do tipo *Expression<Boolean>* e o utiliza como o conteúdo da cláusula *WHERE*. O segundo método *where* recebe um número variável de argumento do tipo *Predicate* e usa uma conjunção *AND* como conteúdo da cláusula *WHERE*. Para se usar a conjunção *OR* é necessário construir uma expressão *OR* explicitamente. Ambos métodos sobrescrevem qualquer conteúdo estabelecido anteriormente.

A interface *CriteriaQuery* também possui métodos para construir as cláusulas *GROUP BY* e *HAVING*. O método *groupBy* recebe um número variável de argumentos especificando uma ou mais expressões de agrupamento.

A construção da cláusula *HAVING* é bem similar a construção da cláusula *WHERE*. Assim como a cláusula *WHERE*, duas formas do método *having* são fornecidas. Uma forma recebe um argumento do tipo *Expression<Boolean>* e a outra recebe um número variável de argumentos do tipo *Predicate*. Esta segunda forma usa a conjunção *AND*, sendo necessário construir uma expressão *OR* explicitamente para se usar a conjunção *OR*.

A interface *CriteriaBuilder* fornece os seguintes métodos para se construir expressões de agregação:

- *count*, *countDistinct* - retornam uma expressão do tipo *long* representando o número de elementos.
- *sum* , *sumAsLong*, *sumAsDouble* - retorna uma expressão representando a soma dos valores.
- *avg* - retorna uma expressão do tipo *double* representando a média dos valores.
- *min*, *least* - retorna uma expressão menor dos valores comparados.
- *max*, *greatest* - retorna uma expressão representando o maior dos valores comparados.

Tanto o método *groupBy* quanto o *having* sobrescreve qualquer valor estabelecido anteriormente.

Por fim, a interface *CriteriaQuery* fornece o método *orderBy* para construir a cláusula *ORDER BY*. Diferente dos outros métodos para construir as cláusulas de consulta, o método *orderBy* um número variável de instâncias *Order* como argumento ao invés de instâncias *Expression*. A interface *Order* é simplesmente uma camada adicional à interface *Expression*, que adiciona uma direção ordenada, podendo ser ascendente (*ASC*) ou descendente (*DESC*). Os métodos *asc* e *desc* da interface CriteriaBuilder recebem uma expressão e retornam uma instância *Order* ascendente ou descendente.

2.3 Framework Object-Inject

Object-Inject (CARVALHO et al., 2013) é um *framework* de persistência e indexação de objetos escrito na linguagem de programação Java. Ele utiliza índices primários para realizar persistência e índices segundários para realiza indexação.

Índices primários são índices sobre conjuntos de atributos que contém as chaves primárias. Eles determinam a localização de registros em arquivos de dados, contendo os atributos do objeto respectivo. Já índices secundários são índices sobre conjuntos de atributos que não contém chaves primárias. Eles conservam apenas referências à entradas de índices primários (RAMAKRISHNAN; GEHRKE, 2003; GARCIA-MOLINA; ULLMAN; WIDOM, 2008).

As chaves primárias são definidas como identificadores UUID (Universally Unique Identifier; identificador universalmente único), que são valores de 128 bits (ZAHN; DINEEN; LEACH, 1990; LEACH; MEALLING; SALZ, 2005; ISO, 1996; ITU, 2005). Para garantir a unicidade destes identificadores é utilizado o algoritmo de geração de números aleatórios proposto por ITU (2004). Neste algoritmo, o esgotamento de possibilidades, considerando uma taxa de geração de UUIDs de 100 trilhões por nanosegundo, ocorreria em aproximadamente 3,1 trilhões de anos.

Índices primários utilizam estas chaves primárias para armazenamento e recuperação de objetos em meio persistente. São disponibilizadas implementações de índices primários baseados nas estruturas de dados hash extensível e lista duplamente encadeada.

Índices secundários são definidos por domínios de indexação e conjuntos de atributos-chave. Estes atributos e o UUID são replicados em chaves que são empregadas em uma estrutura de indexação apropriada para o domínio. São disponibilizadas estruturas de indexação baseadas em árvores, com armazenamento de chaves de roteamento em nós internos e das chaves em nós folha. Desta forma, a partir de um valor de atributo-chave, pode-se navegar entre chaves de roteamento para alcançar uma chave com um UUID. Este identificador é então utilizado para a recuperação do objeto em um índice primário. As implementações de estruturas de indexação disponíveis são Árvore B+ (domínio ordenável) (COMER, 1979), Árvore M (domínio métrico) (CIACCIA; PATELLA; ZEZULA, 1997), Árvore R (GUTTMAN, 1984) (domínio espacial).

A seguir, serão detalhados os módulos utilizados para realizar a persistência e indexação do *framework*.

2.3.1 Módulos

O *framework* é organizado em 4 módulos: Meta, Armazenamento, Blocos e Dispositivos (figura 2.15).

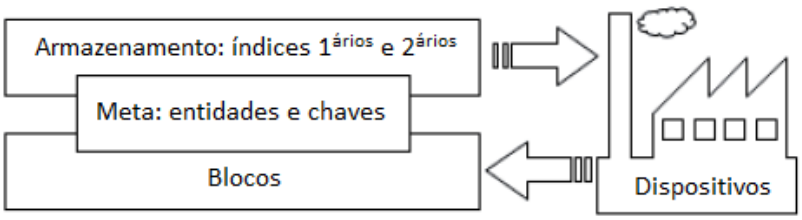

Figura 2.15: Ilustração dos pacotes do *framework Object-Inject*

O módulo Meta define interfaces de entidades (objetos persistentes), chaves (objetos indexáveis contendo atributos-chave de um objeto persistente), domínios de indexação e classes ajudantes de (des)serialização.

O módulo Armazenamento define estruturas de dados para índices primários e secundários, responsáveis por gerenciar entidades e chaves sobre unidades de armazenamento do módulo Blocos.

O módulo Blocos define unidades de armazenamento para estruturas do módulo Armazenamento, possibilitando a divisão dos dados armazenados. Estas unidades são independentes de meio de armazenamento e são disponibilizadas pelo módulo Dispositivos.

O módulo Dispositivos define gerenciadores de recursos computacionais para unidades de armazenamento. Esta é uma camada de abstração sobre meios de armazenamento, responsável por gerenciar alocação, manipulação e liberação de recursos, dispondo unidades de armazenamento para estruturas do módulo Armazenamento.

Módulo Meta

O módulo Meta define o vínculo entre aplicação e *framework*, através de entidades e chaves (associadas a domínios de indexação) (Figura 2.16).

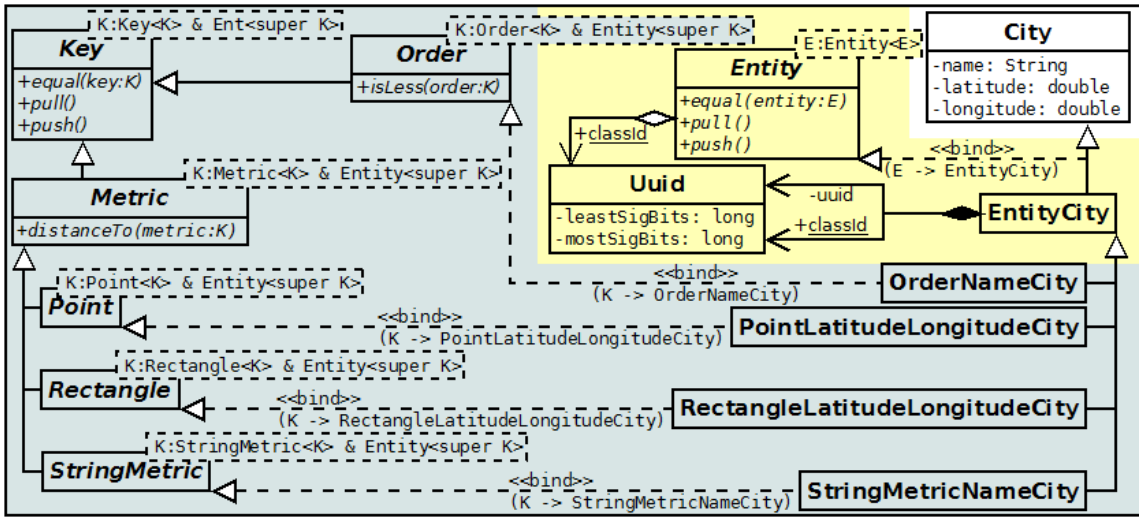

Figura 2.16: *Framework Object-Inject*: pacote Meta

Meta adiciona significado ao *framework* em classes da aplicação, com resolução e segurança de tipos (type safety) garantidas em tempo de compilação. As metaclasses do *framework* são as interfaces Entity, Key e interfaces relacionadas a domínios de indexação Order, Metric, Point, Rectangle, e StringMetric.

Uma entidade é a representação de um objeto persistente, definida na interface Entity. Uma chave é a representação de um objeto persistente e indexável através de atributos-chave, definida na interface Key. Em uma aplicação, classes devem implementar a interface Entity para se tornarem persistentes e a interface Key para se tornarem indexáveis.

A interface Entity especifica métodos de identificação, comparação de igualdade e (des)serialização de entidades, além de um atributo no escopo da classe para identificação e comparação de igualdade entre classes de entidades. A interface Key especifica métodos de comparação de igualdade e de (des)serialização de chaves. Ela é especializada em domínios de indexação, com especificações de métodos particulares à cada domínio, através das interfaces Order (domínio ordenável), StringMetric (domínio métrico textual) e Metric (domínio métrico), especializada em Point e Rectangle (domínio métrico espacial).

A interface Order especifica um método de comparação de ordem entre chaves. Este método é empregado na ordenação de chaves, satisfazendo propriedades de relação de ordem total (transitividade, anti-simetria, totalidade) (ZEZULA et al., 2005). A interface StringMetric especifica métodos acessadores para a string (texto) de indexação. A interface Metric especifica um método de comparação de distância entre chaves. Este método é empregado em chaves com relação de (dis)similaridade, porém sem relação de ordem, satisfazendo propriedades de função de distância (não-negatividade, simetria, reflexividade) (ZEZULA et al., 2005). As interfaces Point e Rectangle especificam métodos de acesso à coordenadas dimensionais. Além disso, a interface Rectangle especifica métodos de acesso à coordenadas de origem e extensão em dimensões.

Com adoção de funcionalidades especificadas em interfaces, são garantidas transparência e retrocompatibilidade para classes de uma aplicação. É possível eximir classes existentes da aplicação de modificações, através de classes empacotadoras dedicadas ao vínculo com o *framework*. Uma classe empacotadora de persistência ou indexação pode especializar uma classe da aplicação e implementar a interface Entity ou Key. Isto torna a classe empacotadora compatível, em tipos, atributos e métodos, com a classe da aplicação e interface. Este arranjo é transparente à classe da aplicação através de conversão de tipo entre classe da aplicação e classe empacotadora, realizada automaticamente pelo *framework*. Outro benefício é que as classes da aplicação, por vezes indisponíveis em formato fonte, não precisam se modificadas, tornando o *framework* retrocompatível com classes da aplicação.

As interfaces Entity e Key adotam o padrão de projeto CRTP (Curiously Recurring Template Pattern; padrão com template curiosamente recursivo) a fim de garantir restrições de comparação entre entidades ou chaves. Neste padrão de projeto, um parâmetro de tipo de uma classe recebe como argumento a própria classe e, logo, si próprio. Estas interfaces requerem um parâmetro de tipo, empregado em métodos de comparação. Desta forma, a assinatura dos métodos de comparação restringe comparações de igualdade à mesma classe de entidades ou chaves. Por definição, a comparação de igualdade entre classes distintas tem resultado falso em qualquer caso.

Ilustrando o módulo Meta, a figura 2.16 apresenta o cenário de uma aplicação. City é uma classe da aplicação, representando uma cidade. EntityCity é uma classe empacotadora de persistência. São definidas as classes empacotadoras de indexação OrderNameCity e OrderMayorCity (chaves para indexação em domínio ordenável, através dos atributos name e mayor, respectivamente), PointLatitudeLongitudeCity e RectangleLatitudeLongitudeCity (chaves para indexação em domínio métrico espacial, através dos atributos latitude e longitude, simultaneamente), e StringMetricNameCity e StringMetricMayorCity (chaves para indexação em domínio métrico textual, através dos atributos name e mayor, respectivamente).

Módulo Armazenamento

O módulo Armazenamento define a especificação de estruturas de dados no *framework* (Figura 2.17).

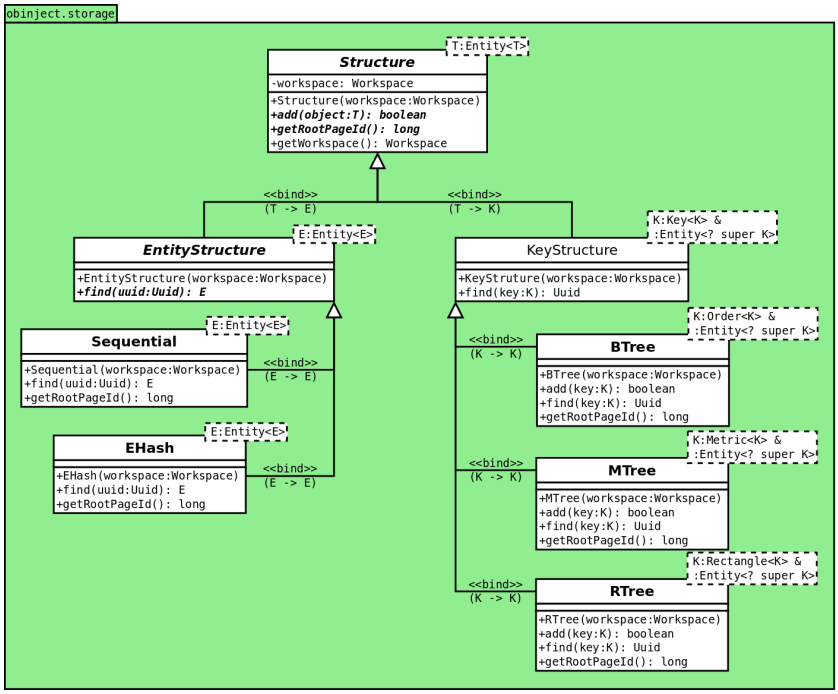

Figura 2.17: *Framework Object-Inject*: pacote Armazenamento

Estruturas para índices primários, responsáveis por gerenciar entidades, são especificadas na classe EntityStructure. Estruturas para índices secundários, responsáveis por gerenciar chaves, são especificadas na classe KeyStructure. Ambas classes são especializações da classe abstrata Structure. Esta classe define a dependência em relação à interface Workspace. Estruturas de dados organizam os dados armazenados (entidades e chaves) em unidades de armazenamento, gerenciadas por um Workspace (espaço de trabalho). Esta organização permite compartilhamento de um espaço de entidades e chaves entre diferentes estruturas de dados.

Com o *framework*, são disponibilizadas as implementações de estruturas de dados Sequential (lista circular duplamente encadeada), EHash (hash extensível) - especializações da classe EntityStructure - e BTree (árvore B+), MTree (árvore M) e RTree (árvore R) - especializações da classe KeyStructure.

Módulo Blocos

O módulo Blocos define unidades de armazenamento de estruturas de dados (figura 2.18).

Unidade de armazenamento é uma unidade funcional em uma estrutura de dados, consistindo de uma sequência de bytes em formato específico para armazenamento de um ou mais objetos. Normalmente, a sequência de bytes é associada a blocos de meios de armazenamento, tendo tamanho definido em função do tamanho de um bloco.

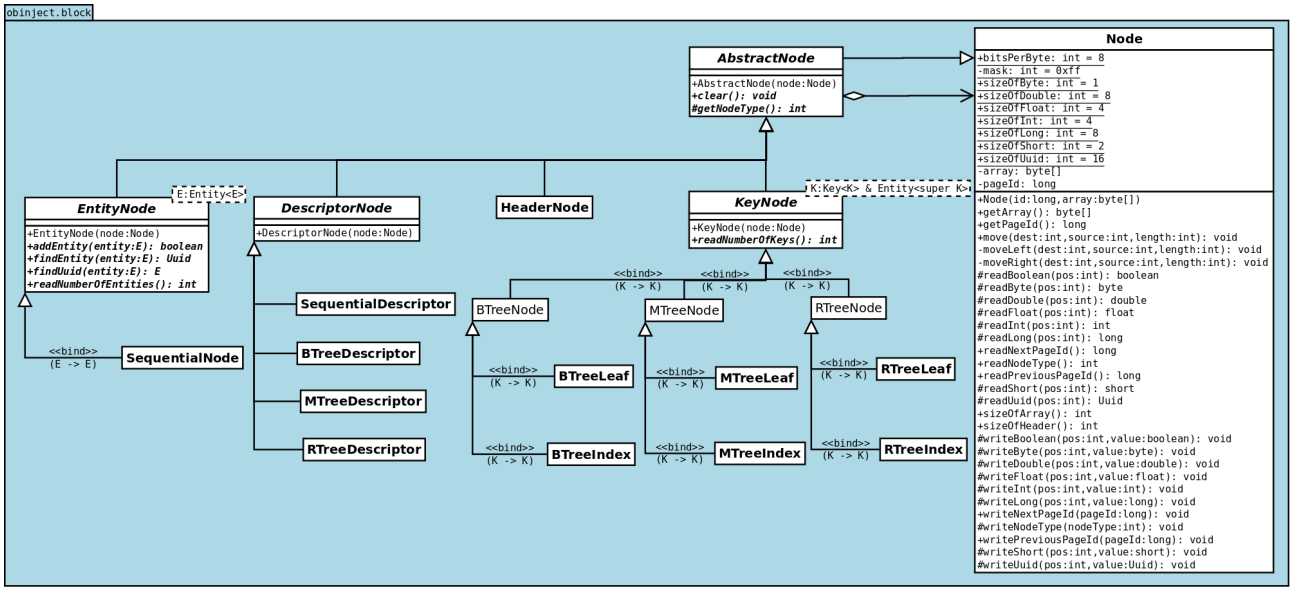

Figura 2.18: *Framework Object-Inject*: pacote Blocos

A classe Node (nó) e especializações especificam formato de dados, métodos e classes funcionais de unidades de armazenamento. A classe Node especifica métodos de (des)serialização de atributos. Dentre as especializações da classe Node estão a classe AbstractNode, contendo métodos de identificação de estruturas de dados, e as classes funcionais DescriptorNode, para descrição de propriedades de estruturas de dados (por exemplo, nó raiz, nós livres, estatísticas de acesso), EntityNode e KeyNode, para armazenamento de entidades e chaves, respectivamente. Estas classes são especializadas, ainda, segundo particularidades de estruturas de dados.

O formato dos dados em nó é descrito em 5 seções: cabeçalho (header), propriedades (features), índice de entradas (entries), objetos (entities, keys) e espaço disponível (free space). O formato da seção cabeçalho é idêntico entre todos os nós; o formato das seções propriedades, índice de entradas e objetos é definido por estruturas de dados.

A seção cabeçalho contém atributos para identificação de estruturas de dados, prevenindo manipulação de nós por estruturas de dados incorretas, e referências de nós adjacentes, permitindo navegação em estruturas de dados. A seção propriedades contém atributos de descrição de um nó (por exemplo, número de objetos, nível ou profundidade na estruturas de dados). A seção índice de entradas contém atributos para localização de objetos (por exemplo, deslocamento, ou offset, nós filhos, identificadores de objetos representativos). A seção objetos contém um vetor (array) de entidades ou chaves em formato sequencial (serial). A seção espaço disponível contém bytes disponíveis para extensão de outras seções.

Suportando objetos de tamanho variável, seções são posicionadas da seguinte forma em unidades de armazenamento: seções de tamanho fixo na extremidade inicial (bytes mais significativos), seções

de tamanho variável na extremidade final (bytes menos significativos), e espaço disponível entre extremidades. Os objetos são armazenados em espaço disponível no sentido da extremidade final para a extremidade inicial, mantendo espaço disponível entre extremidades. Em caso de insuficiência de espaço disponível, ocorre adição ou divisão (split) de nós, segundo políticas de estruturas de dados.

Algumas responsabilidades de Nós são delegadas a Workspace, como interações com meios de armazenamento e definição do comprimento da sequência de bytes manipulada (tamanho do nó) - arbitrário e constante, idêntico entre os nós em um Workspace.

Módulo Dispositivos

O módulo Dispositivos (figura 2.19) define gerenciadores de dispositivos para estruturas de dados.

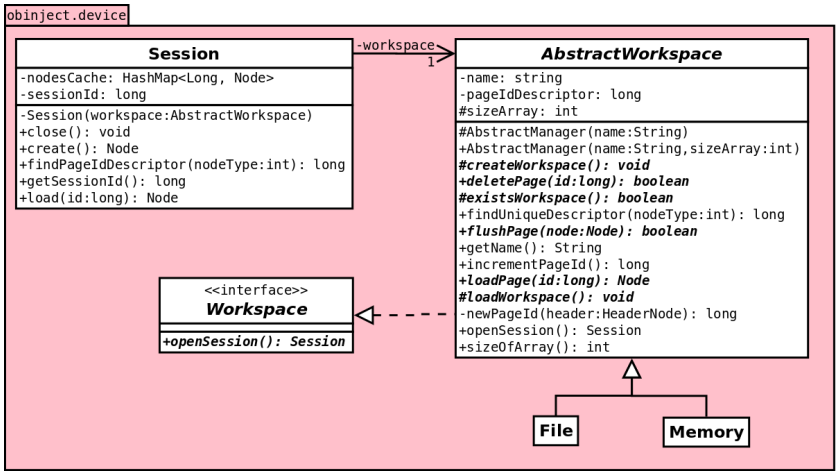

Figura 2.19: *Framework Object-Inject*: pacote Dispositivos

Um gerenciador de dispositivo é responsável por gerenciar alocação, manipulação e liberação de unidades de armazenamento em meios de armazenamento para estruturas de dados, através de uma interface transparente, independente de detalhes de dispositivos.

Um Workspace (espaço de trabalho) consiste em um conjunto de unidades de armazenamento disponível para acesso e compartilhamento por estruturas de dados através de operações de leitura e escrita.

Provendo gerenciadores de dispositivos com interface workspace, são definidas as classes File (arquivo) e Memory (memória), especializações de AbstractWorkspace, e Session, e a interface Workspace.

A classe abstrata AbstractWorkspace especifica construtores de inicialização e carregamento de workspace e métodos de leitura, escrita e remoção de unidades de armazenamento. As classes File e Memory especializam a classe AbstractWorkspace como gerenciadores de dispositivos sobre arquivos (meio de armazenamento persistente) e vetores em memória (meio de armazenamento volátil),

respectivamente.

A classe Session (sessão) representa uma sessão de trabalho temporária sobre um workspace, mantendo unidades de armazenamento em cache. Session é, também, uma abstração sobre dispositivos, especificando somente métodos de criação e leitura de unidades de armazenando e finalização de sessão, momento em que ocorre escrita de unidades de armazenamento em cache para um workspace.

A interface Workspace é o vínculo entre estruturas de dados e gerenciadores de dispositivos, definido na classe Structure.

Uma vez que a interface Workspace especifica um método de obtenção de sessão e é implementada pela classe AbstractWorkspace, é possível realizar operações através de sessões ou gerenciadores de dispositivo, disponibilizando diferentes níveis de abstracação para aplicações e desenvolvedores.

2.3.2 Geração de classes empacotadoras

Em Oliveira (2012) foi introduzida a automatização da implementação de interfaces de persistência e indexação através da metaprogramação, combinando metaclasses do *framework* e metadados da aplicação.

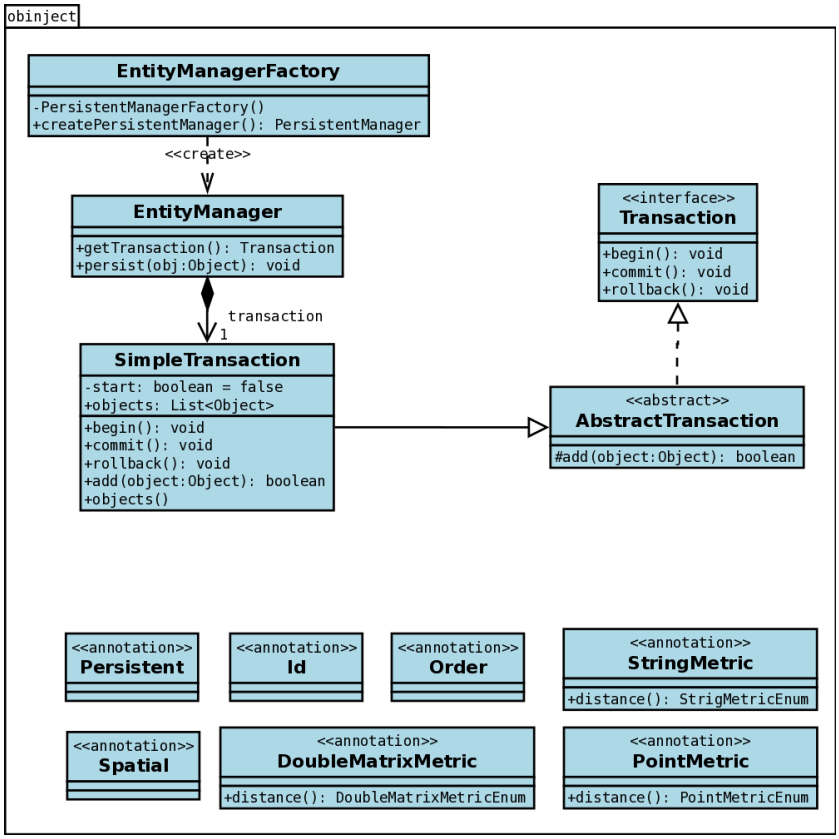

Figura 2.20: Diagrama de classes: pacote *obinject*

Assim como metaclasses, metadados adicionam significado para o *framework* em classes da aplicação, em termos de análise sintática, entretanto, metadados são menos restritos e mais significativos

que metaclasses, devido à existência de construções de linguagem de programação especializadas à representação de metadados. Na linguagem de programação Java, metadados são representados através de anotações e tipos-anotação.

Utilizando-se disso, para alimentar analisadores e geradores de código, foi feita a automatização da implementação de classes da aplicação. Para isto, foram criadas as anotações de persistência e indexação descritas a seguir.

A anotação de persistência é *@Persistent*. Esta é uma anotação marcadora (*marker*), sem elementos, aplicável a tipos (classes). Um gerador de classes gera uma classe empacotadora com a implementação da interface *Entity*, baseado em uma classe anotada com esta anotação, e um índice primário para entidades desta classe de entidades.

As anotações de indexação são descritas abaixo. Estas anotações são aplicáveis a atributos (campos). Um gerador de classes gera uma classe empacotadora com as implementações da interface *Key* e sub-interface correspondente à anotação e um índice secundário para chaves, baseado em um atributo anotado com uma anotação de indexação. Algumas anotações são marcadoras e outras contém elementos para configuração de indexação. As anotações de indexação são:

- *@Id*: atributo é chave primária da classe, com relação de ordem, indexado em árvore *B+*.
- *@Order*: atributo com relação de ordem, indexado em árvore *B+*.
- *@Spatial*: atributo com relação espacial, indexado em árvore *R*.
- *@StringMetric* (*distance=levenshtein*, *damerauLevenshtein*, *xuDamerau*): atributo com relação de distância entre cadeias de caracteres, indexado em árvore *M*; funções de distância podem ser *levenshtein* e variantes *damerauLevenshtein* (transposição) e *xuDamerau* (proteínas).
- *@DoubleMatrixMetric* (*distance=veryLeastDistance*): atributo com relação de distância matricial, indexado em árvore *M*; função de distância pode ser *veryLeastDistance*.
- *@PointMetric* (*distance=euclidean*, *manhattan*, *earthSpherical*): atributo com relação de distância, indexado em árvore *M*; função de distância pode ser *euclidean* (euclidiana), *manhattan*, e esférica terrestre.

Anotações de persistência e indexação tem política de retenção de tempo de execução, permitindo acesso por reflexão para validadores e geradores de classe e outras ferramentas.

Implementações provenientes de geração de código podem ser armazenadas em classes empacotadoras, conhecidas somente pelo *framework*, evitando intrusão de código em classes originais,

tornando o processo de geração de código transparente para a aplicação. A adoção de metadados e classes empacotadoras conduz à uma abordagem conveniente e pouco intrusiva para a aplicação. No contexto de *frameworks*, usabilidade e intrusão são critérios relevantes para adoção e desenvolvimento de aplicações (HOU; RUPAKHETI; HOOVER, 2008; CORRITORE; WIEDENBECK, 2001).

Para garantir que classes da aplicação satisfaçam requisitos de persistência e indexação do *framework* (por exemplo, construtor sem argumentos, métodos de acesso a atributos, etc), também foram desenvolvidos validadores de classes, invocados por geradores de classes e responsáveis por lançar erros de validação, abortando o processo de geração de classes.

Além da implementação das interfaces de persistência e indexação, através de geradores de classes, foram inseridos atributos de escopo de classe representando índices primários e secundários da entidade, especificados através de anotações de indexação. Desta forma, todas as instâncias de uma entidade tem acesso aos seus índices para persistência e indexação. Assim, a responsabilidade de gerenciamento de índices primários e secundários é transferido para as entidades, tonando-as autogerenciáveis (*self-managed entity*).

Capítulo 3
ObInject Query Language

Neste capítulo serão apresentados as modificações feitas no *framework* Object-Inject e a linguagem de consulta desenvolvida para este *framework*, objetivo principal deste trabalho.

3.1 Extensão do framework Object-Inject

Esta seção apresenta as extensões do *framework Object-Inject* no módulo de metaprogramação (OLIVEIRA, 2012), onde estão definidos os metadados para classes da aplicação usando anotações. Estas anotações possibilitam que a geração das classes de empacotamento pelo módulo de geração de código seja automatizada através das técnicas de instrumentação de objetos e compilação de novas classes em tempo de execução.

Uma classe da aplicação deve ser persistente e possuir uma chave primária para ser manipulável pelo *framework* e, opcionalmente, pode conter índices.

Para se tornar persistente, uma classe da aplicação deve receber a anotação de escopo de classe *@Persistent*. Esta anotação indica ao módulo de geração de código que uma classe de empacotamento, responsável pela persistência do objeto, deve ser criada. Foi adicionado nesta anotação o elemento *blockSize*. Este elemento define o tamanho dos blocos da estrutura de dados responsável pelo armazenamento do objeto.

Toda classe da aplicação deve conter um ou mais atributos que definem a chave primária. Estes atributos devem receber a anotação *@Id*, como descrito por Oliveira (2012). Esta anotação indica ao módulo de geração de código que uma classe de empacotamento responsável pela chave primária do objeto deve ser criada.

Para ilustrar o uso das anotações, foram criadas as classes da aplicação *Terreno* e *Mapa* (figura 3.1), sendo que a classe *Terreno* possuí uma associação unidirecional com a classe *Mapa* com valor de multiplicidade 1.

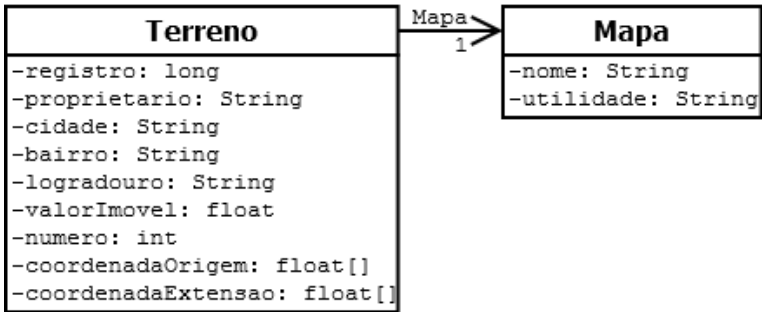

Figura 3.1: Diagrama de classes: Relacionamento entre as classes *Terreno* e *Mapa*

A adição de índices em classe da aplicação podia ser feita pela adição das anotações @*Order*, @*Spatial*, @*StringMetric* e @*PointMetric* antes de atributos que determinam um dos quatro domínios de índice. Cada uma destas anotações define a indexação do atributo pelas árvores *B+*, *R*, *M* e *M*, respectivamente. No entanto, esta nomeação apresenta o problema em que cada classe da aplicação pode ter um único índice para cada domínio de índice. Dessa maneira, cada classe da aplicação podia ter definido 4 índices, um para cada domínio de índice.

A figura 3.2 ilustra o problema descrito. A classe da aplicação *Terreno* é anotada com @*Persistent* (linha 12 e 13) para se tornar persistente. Já o atributo registro (linha 16) recebe a anotação @*Id* (linha 15) para se tornar a chave primária. O atributo proprietário (linha 18) recebe a anotação @*StringMetric* (linha 17) se tornando indexável por uma árvore *M* usando a função de distância de Levenshtein. A anotação @*Order* é utilizada 3 vezes (linhas 19, 21 e 23) sem gerar erro, porém, apenas o primeiro @*Order* é reconhecido pelo *framework*. Desta forma, apenas o atributo cidade (linha 120) se torna indexável por uma árvore *B+*. O atributo coordenadaOrigem (linha 29) recebe a anotações @*PointMetric* (linha 27) e @*Spatial* (linha 28) se tornando indexável por uma árvore *M* usando a função de distância euclidiana e por uma árvore *R*. Observe que apenas um índice foi criado para cada anotação e apenas um atributo foi utilizado para tal índice, mesmo quando a mesma anotação é utilizada em mais de um atributo.

Afim de permitir a criação de mais de um índice para cada domínio de índice, foram criadas 20 variações de cada anotação, sendo nomeadas de *first* (primeira) a *twentieth* (vigésima). Cada variação fica responsável por um índice diferente para o mesmo domínio, possibilitando a criação de múltiplos índices em um mesmo domínio de índice. Adicionalmente, alguns elementos das anotações foram modificados e algumas anotações foram renomeadas para ficarem mais coerente com suas funções. A seguir são dadas as anotações na forma atual:

- @*OrderFirst* à @*OrderTwentieth*: atributos com relação de ordem (implementação da classe *Order* figura2.16), indexados em árvores *B+*;

- @*GeoPointFirst* à @*GeoPointTwentieth*: atributos que estabelecem coordenadas esféricas (im-

```
1   package org.obinject.sample.terreno.problema;
2
3   import org.obinject.annotation.StringMetricEnum;
4   import org.obinject.annotation.Id;
5   import org.obinject.annotation.Order;
6   import org.obinject.annotation.Persistent;
7   import org.obinject.annotation.PointMetricEnum;
8   import org.obinject.annotation.PointMetric;
9   import org.obinject.annotation.Spatial;
10  import org.obinject.annotation.StringMetric;
11
12  @Persistent
13  public class Terreno {
14      @Id
15      private long registro;
16      @StringMetric(distance = StringMetricEnum.LEVENSHTEIN)
17      private String proprietario;
18      @Order
19      private String cidade;
20      @Order
21      private String bairro;
22      @Order
23      private String logradouro;
24      private int numero;
25      @PointMetric(distance = PointMetricEnum.EUCLIDEAN)
26      @Spatial
27      private float latitudeMinima;
28      @PointMetric(distance = PointMetricEnum.EUCLIDEAN)
29      @Spatial
30      private float longitudeMinima;
31      private float latitudeMaxima;
32      private float longitudeMaxima;
33
34      // gets e sets ...
35
36  }
```

Figura 3.2: Exemplo problema: Classe *Terreno*

plementação da classe *Point* figura 2.16), indexados em árvores M utilizando a geometria esférica. Esta anotação possui o elemento *radius* (valor padrão = 6378) que define o raio da esfera e a anotação *angle* (valor padrão = grau) que define se o ângulo será em graus ou radianos;

- *@EditionFirst* à *@EditionTwentieth* (substituiu *@StringMetric*): atributos que estabelecem uma cadeia de caracteres (implementação da classe *Metric* figura 2.16), indexados em árvore *M*. Esta anotação possui o elemento *distancia* onde o usuário pode escolher uma das funções de distâncias definidas: LEVENSHTEIN (valor padrão do elemento) ou uma de suas variantes *DAMERAU* (problema da transposição) e *PROTEIN* que utiliza *DAMERAU* com pesos para substituição de aminoácidos;

- *@PointFirst* à *@PointTwentieth*: atributos com relação de distância (implementação da classe

Point figura 2.16), indexados em árvores *M*; possui o elemento *distancia* onde o usuário pode escolher uma das funções de distâncias definidas: *EUCLIDEAN* (valor padrão) para distâncias euclidianas e *MANHATTAN* para distâncias de Manhattan;

- *@RectangleFirst* à *@RectangleTwentieth* (substituiu *@Spatial*): atributos com relação de distância euclidiana (implementação da classe *Rectangle* figura 2.16), indexados em árvores *R*;

- *@RectangleExtensionFirst* à *@RectangleExtensionTwentieth*: funciona em conjunto com as anotações *@RectangleFirst* à *@RectangleTwentieth* para indicar os atributos que estabelecem extensão do retângulo.

Além disto, a implementação não permitia o uso de múltiplos atributos para a criação de um índice, pois o *framework* reconhecia apenas a primeira anotação de cada tipo. Para resolver isto, a geração de classes foi remodelada e foi criado um novo pacote chamado *generator* (figura 3.3). Este novo pacote é encarregado de encapsular todos os metadados (nomes de atributos, nome de métodos, tipos, etc) necessários para geração de classes de empacotamento das classes da aplicação. A partir destes metadados é gerado um código fonte de cada classe de empacotamento no formato de uma string, em tempo de execução, que será então compilada também em tempo de execução.

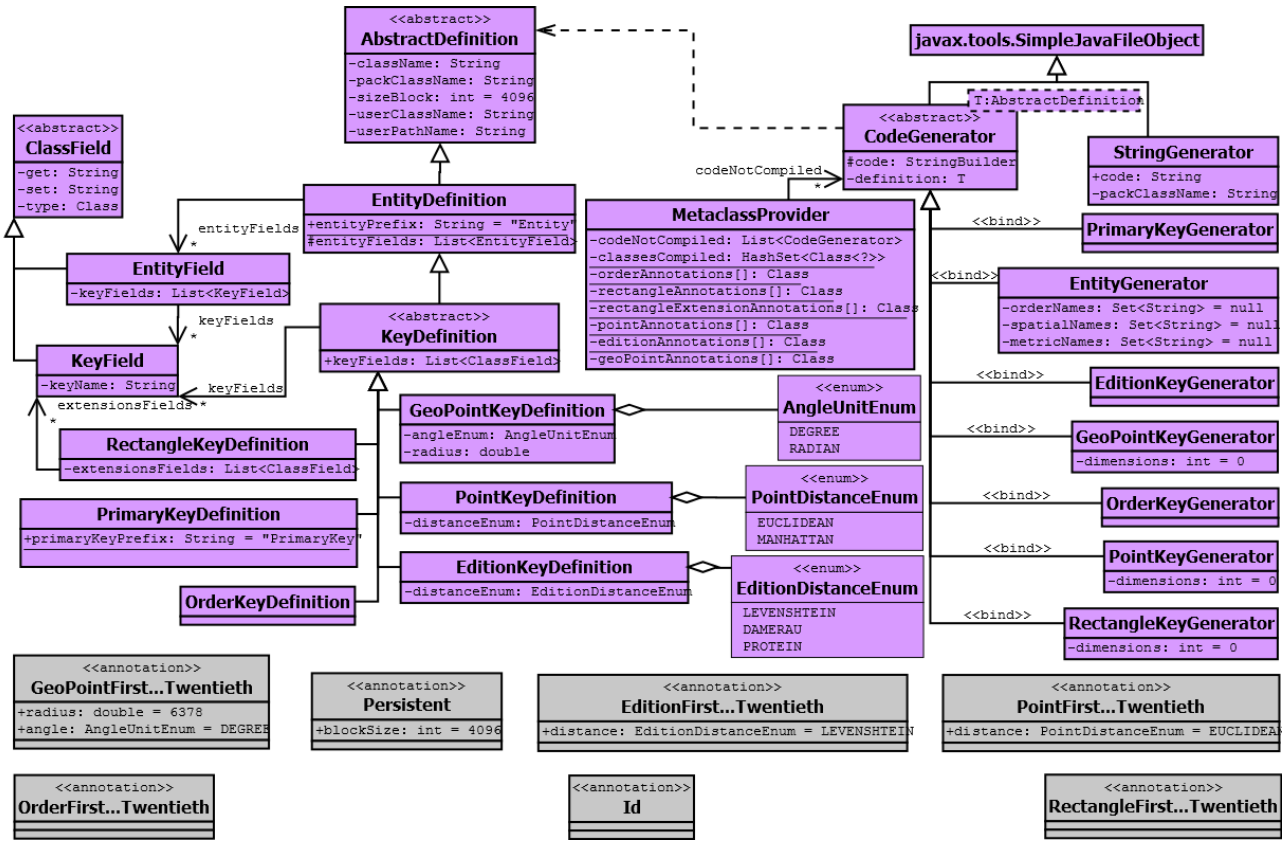

Figura 3.3: Diagrama de Classes: pacote *generator*

A classe *CodeGenerator* delega a responsabilidade de gerar as strings que serão compiladas para suas classes descendentes:

- *EntityGenerator*: responsável pela classe de empacotamento *Entity*;
- *PrimaryKeyGenerator*: responsável pela classe de empacotamento *PrimaryKey*;
- *OrderKeyGenerator*: responsável pela classe de empacotamento *OrderKey*;
- *GeoPointKeyGenerator*: responsável pela classe de empacotamento *GeoPointKey*;
- *EditionKeyGenerator*: responsável pela classe de empacotamento *EditionKey*;
- *PointKeyGenerator*: responsável pela classe de empacotamento *PointKey*;
- *RectangleKeyGenerator*: responsável pela classe de empacotamento *RectangleKey*.

Todas as classes descendentes da classe *CodeGenerator* implementam o método *getCharContent* que retorna a *string* a ser compilada. Para gerar esta *string* este método chama um conjunto de outros métodos, cada um responsável pela geração de uma parte do código fonte.

A classe *CodeGenerator* possui um atributo genérico que descende de *AbstractDefinition* e que contém as definições que serão utilizadas na geração do código fonte. As classes descendentes de *AbstractDefinition* são:

- *EntityDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *Entity*.
- *KeyDefinition*: possui as definições comuns as classes que geram classes de empacotamento para índices.
- *PrimaryKeyDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *PrimaryKey*.
- *OrderKeyDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *OrderKey*.
- *GeoPointKeyDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *GeoPointKey*.
- *EditionKeyDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *EditionKey*.

- *PointKeyDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *PointKey*.

- *RectangleKeyDefinition*: possui as definições utilizadas para a geração da classe de empacotamento *RectangleKey*.

Cada atributo da classe da aplicação esta associado a um tipo, uma função *get* e uma *set*. Para encapsular todos estes metadados como uma única entidade, foram criadas as classes *ClassField*, que contém os metadados comuns a atributos normais e atributos indexáveis (tipo e nome das funções set e get), e suas generalizações *EntityField*, que contém uma lista de atributos chaves, e *KeyField*, que contém o nome do atributo.

Com estas modificações, a classe *Terreno* (figura 3.2) foi reescrita para ficar de acordo com as extensões realizadas. A nova classe *Terreno* é apresentada na figura 3.4.

Da mesma forma que anteriormente, a classe da aplicação *Terreno* é anotada com @*Persistent* (linha 16) para se tornar persistente, implementando a classe *EntityTerreno* (figuras A.1, A.2, A.3 e A.4). O atributo registro (linha 19) recebe a anotação @*Id* (linha 18) para se tornar a chave primária, implementando a classe *PrimaryKeyTerreno* (figura A.5). O atributo proprietário (linha 22) recebe as anotações @*OrderSecond* e @*EditionFirst* (linhas 20 e 21) se tornando indexável por uma árvore B+ e por uma árvore *M* usando a função de distância de Levenshtein e implementando as classes *OrderSecondTerreno* (figura A.7) e *EditionFirstTerreno* (figura A.8), respectivamente. Os atributos cidade (linha 24), bairro (linha 26) e logradouro (linha 28) recebem a anotação @*OrderFirst* (linhas 21, 23 e 25) formando um conjunto indexável por uma árvore *B+* e implementando a classe *OrderFirstTerreno* (figura A.6). O atributo coordenadaOrigem (linha 34) recebe as anotações @*GeoPointFirst*, @*PointFirst* e @*RectangleFirst* (linhas 31, 32, 33) se tornando indexável por uma árvore M utilizando a geometria esférica, por uma árvore M usando a função de distância Euclidiana e por uma árvore R, além de implementar as classes *GeoPointFirstTerreno* (figura A.9), *PointFirstTerreno* (figura A.11) e *RectangleFirstTerreno* (figura A.12), respectivamente. O atributo coordenadaExtensao (linha 37) recebe a anotação @RectangleFirstExtension (linhas 35) para estabelecer a extensão do retângulo formado em conjunto com o atributo coordenadaOrigem (linha 34). Além disto, ele também recebe a anotação @GeoPointSecond se tornando indexável por uma árvore M utilizando a geometria esférica e implementando a classe *GeoPointSecondTerreno* (figura A.10).

```java
package org.obinject.sample.terreno;

import org.obinject.annotation.EditionDistanceEnum;
import org.obinject.annotation.EditionFirst;
import org.obinject.annotation.GeoPointFirst;
import org.obinject.annotation.GeoPointSecond;
import org.obinject.annotation.Id;
import org.obinject.annotation.OrderFirst;
import org.obinject.annotation.OrderSecond;
import org.obinject.annotation.Persistent;
import org.obinject.annotation.PointDistanceEnum;
import org.obinject.annotation.PointFirst;
import org.obinject.annotation.RectangleFirst;
import org.obinject.annotation.RectangleFirstExtension;

@Persistent
public class Terreno {
    @Id
    private long registro;
    @OrderSecond
    @EditionFirst(distance = EditionDistanceEnum.LEVENSHTEIN)
    private String proprietario;
    @OrderFirst
    private String cidade;
    @OrderFirst
    private String bairro;
    @OrderFirst
    private String logradouro;
    private int numero;
    @GeoPointFirst
    @PointFirst(distance = PointDistanceEnum.EUCLIDEAN)
    @RectangleFirst
    private float latitudeMinima;
    @GeoPointFirst
    @PointFirst(distance = PointDistanceEnum.EUCLIDEAN)
    @RectangleFirst
    private float longitudeMinima;
    @RectangleFirstExtension
    @GeoPointSecond
    private float latitudeMaxima;
    @RectangleFirstExtension
    @GeoPointSecond
    private float longitudeMaxima;

    // gets e sets ...

}
```

Figura 3.4: Exemplo Problema: Classe *Terreno* após implementação das extensões

3.2 ObInject Query Language

Esta seção apresenta a criação do módulo de consultas para o *framework* Object-Inject, que é a principal contribuição desta monografia. Este módulo foi criado para possibilitar a obtenção de dados da

base de dados de forma estruturada e intuitiva. Para tal, foi criado uma "linguagem de consulta" baseada apenas em métodos. Através destes métodos a consulta é definida e então executada.

O pacote *queries* (figura 3.5) define as classes responsáveis por realizar consultas para o *framework* Object-Inject. A consulta é definida pelo usuário através dos métodos *select*, *from*, *where*, *groupBy*, *having* e *orderBy* presentes na classe *Query*. Optou-se para que estes métodos possuíssem nomes semelhantes as cláusulas utilizadas em SQL para facilitar a compreensão e melhorar a usabilidade do modelo. Além disto, o uso de métodos torna a linguagem uma linguagem tipificada, isto é, com segurança sobre tipos.

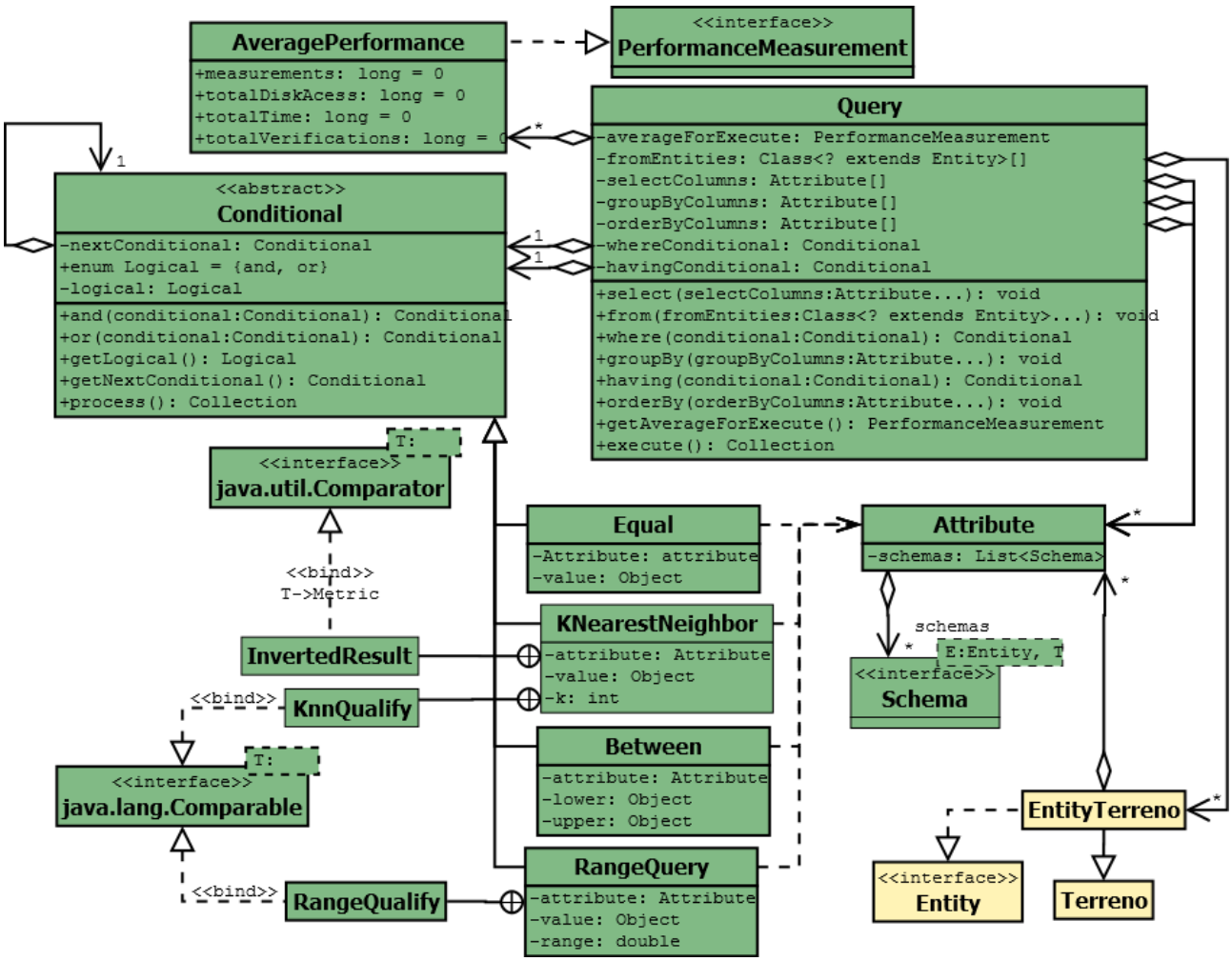

Figura 3.5: Diagrama de classes: pacote *queries*

Para encapsular os atributos da classe do usuário que serão utilizados na consulta, foi criada a classe *Attribute* (figura 3.5), de forma que cada atributo fique responsável por saber quais estruturas podem ser indexadas. Para isto, cada objeto Attribute recebe uma lista de esquemas, onde cada esquema indica uma estrutura em que o atributo foi indexado. Estes atributos foram agregados a classe de empacotamento *Entity*, os tornando acessíveis por tal classe.

O método *select* é utilizado para definir o retorno da consulta. Este retorno é definido através dos

atributos presentes nas classes de empacotamento *Entity*. A figura 3.6, mostra uma consulta utilizando o método *select* para definir os atributos *bairro* e *valorImovel* como retorno da consulta. Desta forma, a consulta retorna uma coleção de vetores de objetos, cada vetor contendo um objeto do tipo *String* (para *bairro*) e um objeto do tipo *Integer* (para *valorImovel*).

```
Query q0 = new Query();
q0.select(EntityTerreno.$bairro, EntityTerreno.$valorImovel);
q0.from(EntityTerreno.class);
Collection<Object[]> l0 = q0.execute();
for (Object[] obj : l0) {
    System.out.println(obj[0] + " " + obj[1]);
}
```

Figura 3.6: Exemplo de consulta: método *select*.

O método from, define de quais entidades os dados serão recuperados. Para isto utilizam-se as próprias classes de empacotamento *Entity*. Na figura 3.7, a consulta *q1* utiliza o método *from* para definir que os dados serão recuperados das entidades *Terreno* e *Mapa*. Neste caso, sem a cláusula *select*, a consulta retorna uma coleção de vetores de objetos, cada vetor contendo um objeto do tipo *Terreno* e um objeto do tipo *Bairro*.

```
Query q1 = new Query();
q1.from(EntityTerreno.class, EntityMapa.class);
q1.where(new Equal(EntityTerreno.$mapa, EntityMapa.class));
Collection<EntityTerreno> l1 = q1.execute();
for (Terreno terreno : l1) {
    System.out.println(terreno.getRegistro());
}
```

Figura 3.7: Exemplo de consulta: método *from*

Para restringir a consulta é utilizado o método *where*. Este método recebe como parâmetro um condição definida pela classe *Conditional* (figura 3.5). Esta classe é responsável por implementar os algoritmos que buscam as informações nas estruturas de dados para responder a consulta. A classe *Conditional* é reflexiva, possibilitando a concatenação de condições através dos métodos *and* e *or*. Estes métodos são utilizadas para estabelecer a lógica booleana entre as condições. A implementação dos algoritmos é feita através do método *process*, que direciona a execução da condição para o algoritmo menos custoso. A seguir são exemplificadas as condições já implementadas.

A condição *Equal* realiza a comparação de igualdade entre dois elementos. Estes elementos podem ser atributos da classe da aplicação ou valores. A implementação desta condição pode usar um atributo indexado em árvore *B+* ou usar uma busca sequencial sobre toda a base. Na figura 3.8, o método *where* recebe duas condições *Equal* ligadas pelo operando *and*. A primeira delas especifica

que apenas terrenos localizados no bairro "Centro" devem ser retornadas e a segunda que apenas terrenos localizados na cidade "Itajubá - MG". Com isto a consulta retorna todos os terrenos que estão localizados no bairro "Centro" da cidade de "Itajubá - MG".

```
Query q2 = new Query();
q2.from(EntityTerreno.class);
q2.where(new Equal(EntityTerreno.$bairro, "Centro")).
        and(new Equal(EntityTerreno.$cidade, "Itajubá - MG"));
Collection<EntityTerreno> l2 = q2.execute();
for (Terreno terreno : l2) {
    System.out.println(terreno.getLogradouro() + " "
            + terreno.getNumero());
}
```

Figura 3.8: Exemplo de consulta: método *where*, condição *Equal*

A condição *Between* define o conjunto de elementos que possuem um dado atributo com valor dentro de uma certa faixa de valores. A implementação desta condição pode usar um atributo indexado em árvore *B+* ou usar uma busca sequencial sobre toda a base. Na figura 3.9, método *where* recebe uma condição *Equal* e uma *Between* ligadas pelo operando *and*. A condição *Equal* restringe a consulta à terrenos localizados na cidade "Itajubá - MG" e a condição *Between* restringe a consulta à terrenos com valor do Imóvel entre 10000 e 20000. Com isto a consulta retorna todos os terrenos que estão localizados na cidade "Itajubá - MG" e que tenham seu valor de Imóvel entre 10000 e 20000.

```
Query q3 = new Query();
q3.from(EntityTerreno.class);
q3.where(new Equal(EntityTerreno.$cidade, "Itajubá - MG")).
        and(new Between(EntityTerreno.$valorImovel, 10000, 20000));
Collection<EntityTerreno> l3 = q3.execute();
for (Terreno terreno : l3) {
    System.out.println(terreno.getLogradouro() + " "
            + terreno.getNumero());
}
```

Figura 3.9: Exemplo de consulta: método *where*, condição *Between*

A condição *KNearestNeighbor* define o conjunto dos K elementos mais próximos a entidade em relação a certo atributo. A implementação desta condição pode usar um atributo indexado em árvore *M* ou usar uma busca sequencial sobre toda a base. Na figura 3.9, o método *where* recebe uma condição *KNearestNeighbor* para restringir a consulta. Ela restringe a consulta aos 5 terrenos mais próximos do ponto (1,1) em relação a coordenada de Origem da entidade Terreno.

A condição *RangeQuery* define o conjunto de elementos que estão dentro de um raio de distância de um valor definido. A implementação desta condição pode usar um atributo indexado em árvore *M* ou usar uma busca sequencial sobre toda a base. Na figura 3.10, o método *where* recebe uma condição

```
float ponto[] = {1f,1f};

Query q4 = new Query();
q4.from(EntityTerreno.class);
q4.where(new KNearestNeighbor(EntityTerreno.$coordenadaOrigem,ponto,5));
Collection<EntityTerreno> l4 = q4.execute();
for (Terreno terreno : l4) {
    System.out.println(terreno.getLogradouro() + " "
            + terreno.getNumero());
}
```

Figura 3.10: Exemplo de consulta: método *where*, condição *KNearestNeighbor*

RangeQuery para restringir a consulta. Ela restringe a consulta aos terrenos dentro de um raio de 1,5 do ponto (2,2) em relação a coordenada de Origem da entidade Terreno.

```
float ponto2[] = {2f,2f};

Query q5 = new Query();
q4.from(EntityTerreno.class);
q4.where(new RangeQuery(EntityTerreno.$coordenadaOrigem, ponto2, 1.5));
Collection<EntityTerreno> l5 = q5.execute();
for (Terreno terreno : l5) {
    System.out.println(terreno.getLogradouro() + " "
            + terreno.getNumero());
}
```

Figura 3.11: Exemplo de consulta: método *where*, condição *RangeQuery*

Continuando com os métodos para definir a consulta, o método *groupBy* é utilizado para realizar o agrupamento do resultado. Este método recebe atributos da classe de empacotamento *Entity* e realiza o agrupamento de acordo com estes atributos. Na figura 3.12, o método orderBy indica que a consulta deve ser agrupado pelo atributo Proprietário. Desta forma, a consulta retorna todos os Terrenos da base localizados na cidade de "Itajubá - MG" agrupados pelo mesmo proprietário.

```
Query q6 = new Query();
q6.from(EntityTerreno.class);
q6.where(new Equal(EntityTerreno.$cidade, "Itajubá - MG"));
q6.groupBy(EntityTerreno.$proprietario);
Collection<EntityTerreno> l6 = q6.execute();
for (Terreno terreno : l6) {
    System.out.println(terreno.getProprietario());
}
```

Figura 3.12: Exemplo de consulta: método *groupBy*

Após fazer o agrupamento do resultado, é possível restringir a consulta em relação a esse agrupamento através do método *having*. Assim como o método *where*, o método *having* recebe como parâmetro um condição definida pela classe *Conditional* (figura 3.5). Portanto, também é possível

concatenar condições através dos métodos *and* e *or* e a implementação dos algoritmos para este tipo de restrição também é feita através do método *process*.

Por último temos método *orderBy*, que define a ordenação do resultado. Este método recebe atributos presentes nas classes de empacotamento *Entity* e realiza a ordenação de acordo com estes atributos. Na figura 3.13, método *orderBy* indica que a consulta deve ser ordenada pelo atributo valorImovel. Desta forma, assim como a consulta da figura 3.12, a consulta retorna todos os Terrenos da base localizados na cidade de "Itajubá - MG" agrupados pelo mesmo proprietário, porém ordenada pelo proprietário em ordem alfabética.

```
Query q7 = new Query();
q7.from(EntityTerreno.class);
q7.where(new Equal(EntityTerreno.$cidade, "Itajubá - MG"));
q7.groupBy(EntityTerreno.$proprietario);
q7.orderBy(EntityTerreno.$valorImovel);
Collection<EntityTerreno> l7 = q7.execute();
for (Terreno terreno : l7) {
    System.out.println(terreno.getLogradouro() + " "
            + terreno.getNumero());
}
```

Figura 3.13: Exemplo de consulta: método orderBy

Após definir a consulta, utiliza-se método *execute* para processar todas as cláusulas (definidas pelos métodos descritos anteriormente) e retornar uma coleção com o resultado da consulta.

3.3 Considerações Finais

Apesar da estrutura da consulta ter sido concluída, a implementação de todos os métodos das classes *Query* e *Conditional* (figura 3.5) ainda não foi finalizada. Na classe *Query*, o método *select* sempre retorna objetos, não retornando apenas atributos. As funções de agregação que fazem parte dos métodos *groupBy* e *having* e o método *orderBy* ainda não foram totalmente finalizados. Na classe *Conditional*, os métodos *and* e *or* ainda não realizam a concatenação de várias condições.

Capítulo 4
Experimentos

Este Capítulo descreve os experimentos realizados para testar a persistência de objetos e consultas utilizando a linguagem ObInject Query Language (seção 3.2) através do *framework* Object-Inject. Visando avaliar o desempenho deste *framework* foram realizados os mesmos experimentos com o *framework* Hibernate associado ao banco de dado MySQL, com posterior comparação entre as duas soluções.

Para realizar os experimentos, foi desenvolvido um modelo de dados simples que simula uma eleição. O diagrama UML deste modelo é apresentando na figura 4.1.

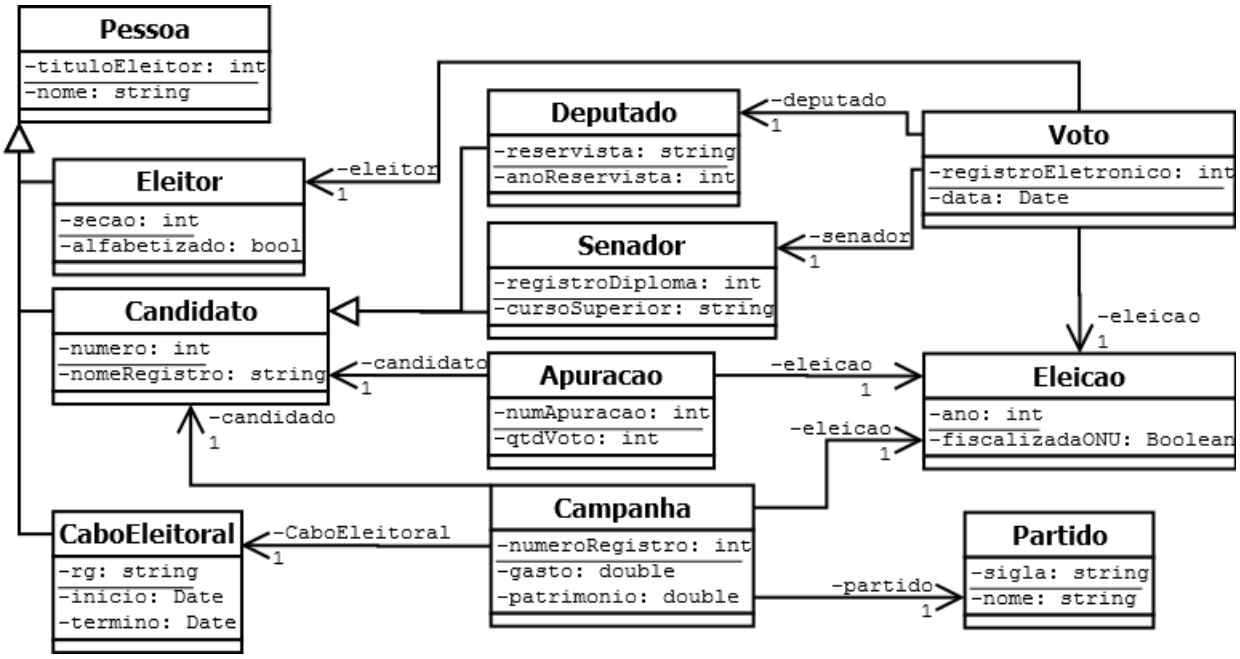

Figura 4.1: Diagrama UML: eleicoes

A partir deste modelo, foi gerado o mesmo conjunto de testes para cada *framework*. Além disto, foram gerados dados para serem persistidos em ambos *frameworks* e posteriormente consultados. Deste forma, foram realizados experimentos para medir o desempenho em relação a persistência dos dados e experimentos para medir o desempenho em relação a consulta de alguns dados.

Além das anotações necessárias para tornar as classes persistentes, alguns atributos também receberam as anotações *@OrderFist* para o Object-Inject e *@Index* para o Hibernate. Os atributos que receberam estas anotação foram: *totalVoto* na classe *Apuração*, *gasto* na classe *Campanha* e *nome* na classe *Pessoa*.

Os experimentos foram realizados num computador com processador Intel® Core™ i7-930, *clock* de 2.8 GHz, respectivamente 32 KB, 256 KB e 8 MB de *cache* L1, L2 e L3, com 24 GB de memória RAM (triplo canal) e sistema operacional Windows 8.1 e 500 GB de HD SATA 7200 RPM. O *framework* Object-Inject foi implementado em Java, usando Oracle Java SE 7u45 (JDK).

4.1 Persistência de dados

Afim de realizar os experimentos relacionados a persistência dos dados, foram gerados 5 conjuntos de dados no formato txt. Cada conjunto de dados representa os atributos e os relacionamentos dos objetos a serem persistidos. Estes atributos foram criados de forma aleatória (inclusive o tamanho das Strings), com exceção de alguns atributos que foram tratados como números sequenciais. Os relacionamentos também foram gerados de forma aleatória.

A tabela 4.1 mostra o número de objetos criados para cada conjunto de dados:

Tabela 4.1: Quantidade de Objetos para Inserção

Experimento	1	2	3	4	5
Eleição	1	1	1	1	1
Partido	10	20	30	40	50
Voto	100.000	200.000	300.000	400.000	500.000
Eleitor	100.000	200.000	300.000	400.000	500.000
Senador	6	25	80	135	222
Deputado	125	390	983	1.495	2.818
Cabo Eleitoral	131	611	2.113	4.137	9077
Apuração	131	415	1.063	1.630	3.040
Campanha	131	415	1.063	1.630	3.040
Total	200.535	401.873	605.333	809.068	1.018.248

Para cada um dos 5 conjuntos foi gerado uma quantidade de objetos diferente. Alguns objetos tiveram um quantidade pré-definida para cada conjunto, enquanto outros tiveram uma quantidade aleatória. Entre os objetos de quantidade pré-definida estão Eleição, Partido, Voto e Eleitor. Os objetos de quantidade variável são Senador, Deputado, Cabo Eleitoral, Apuração e Campanha. A quantidade de Deputados e Senadores variaram de acordo com o número de partidos. Podem haver de 0 à 2, 4, 6, 8 ou 10 senadores por partido e de 0 à 20, 40, 60, 80 ou 100 deputados por partido,

respectivamente para cada experimento. A quantidade de Cabos Eleitorais varia de acordo com o número de Candidatos, sendo um mínimo de 1 e um máximo de 1, 2, 3, 4 ou 5 Cabos Eleitorais por candidato, respectivamente para cada experimento. O número de Apurações e Campanhas é o mesmo número de candidatos.

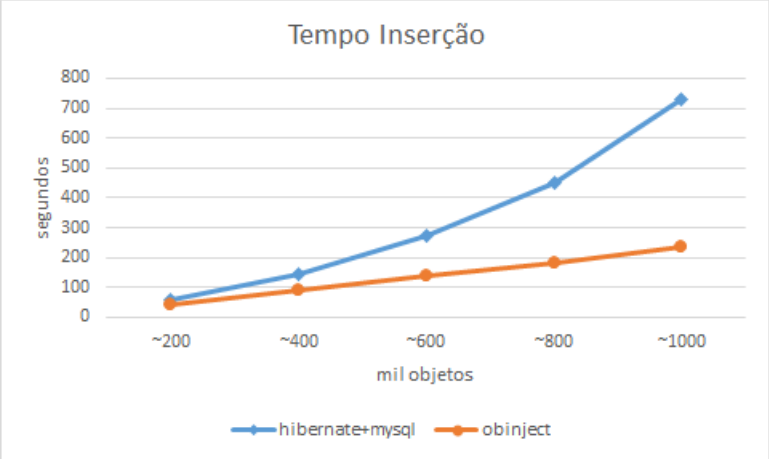

Figura 4.2: Tempo para inserção

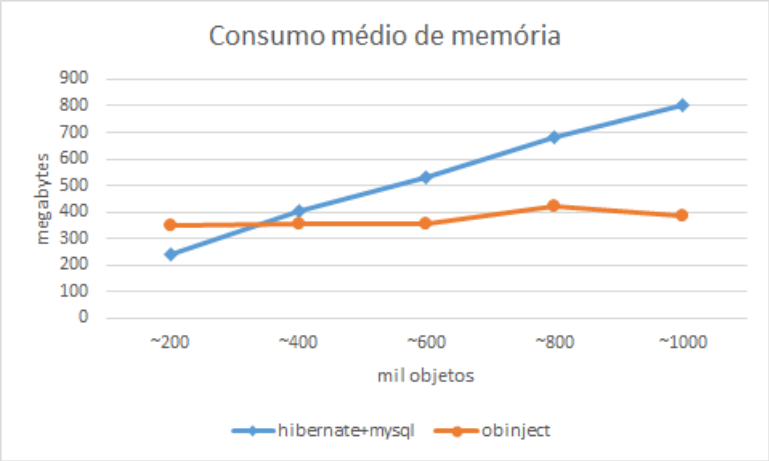

Figura 4.3: Consumo médio de memória.

Com este conjunto de dados, foi criado um aplicativo para ambos *frameworks* afim de fazer a geração e persistência dos objetos. O tempo de execução deste aplicativo foi medido e os resultados obtidos podem ser observados na figura 4.2. Como pode ser observado, ambos *frameworks* possuem um tempo bem próximo para o primeiro conjunto de dados, porém, conforme o número de objetos aumenta, o tempo para a inserção no Hibernate se torna consideravelmente maior que para o Object-Inject.

Também foi medido a quantidade de memória utilizada durante a execução do aplicativo através da interface de desenvolvimento (IDE) Netbeans 7.4. Esta IDE permite a instrumentação da aplicação através de um Perfil, onde o consumo de memoria da aplicação é medido e armazenado a cada

segundo. Através desta medição, foi calculada o consumo médio de memória, apresentada na figura 4.3. Pode-se observar que o Object-Inject apresenta um consumo médio de memória maior apenas para o primeiro conjunto de dados.

Após os objetos serem persistidos, foi medido o espaço total em disco necessário para armazenar os objetos. Para isto, foram considerados os arquivos gerados por cada *framework*.

O Object-Inject gerou os seguintes arquivos: EntityApuracao.dbo, EntityCaboEleitoral.dbo, EntityCampanha.dbo, EntityCandidato.dbo, EntityDeputado.dbo, EntityEleicao.dbo, EntityEleitor.dbo, EntityPartido.dbo, EntitySenador.dbo, EntityVoto.dbo, OrderFirstApuracao.btree, OrderFirstCampanha.btree, OrderFirstPartido.btree, PrimaryKeyApuracao.pk, PrimaryKeyCaboEleitoral.pk, PrimaryKeyCampanha.pk, PrimaryKeyCandidato.pk, PrimaryKeyDeputado.pk, PrimaryKeyEleicao.pk, PrimaryKeyEleitor.pk, PrimaryKeyPartido.pk, PrimaryKeySenador.pk PrimaryKeyVoto.pk. Sendo que o formato dbo é utilizado para armazenar os objetos na forma sequencial, o formato btree é utilizado para armazenar índices com relação de ordem e o formato pk é utilizado para armazenar os índices de chave primaria.

Já o Hibernate transforma os objetos e suas relações em tabelas. Elas e seus relacionamentos podem ser vistas na figura 4.4. Estas tabelas são representadas em disco pelo MySql através do formato frm, sendo então gerados os arquivos apuracao.frm, campanha.frm, campanha_pessoa.frm, eleicao.frm, partido.frm, pessoa.frm e voto.frm.

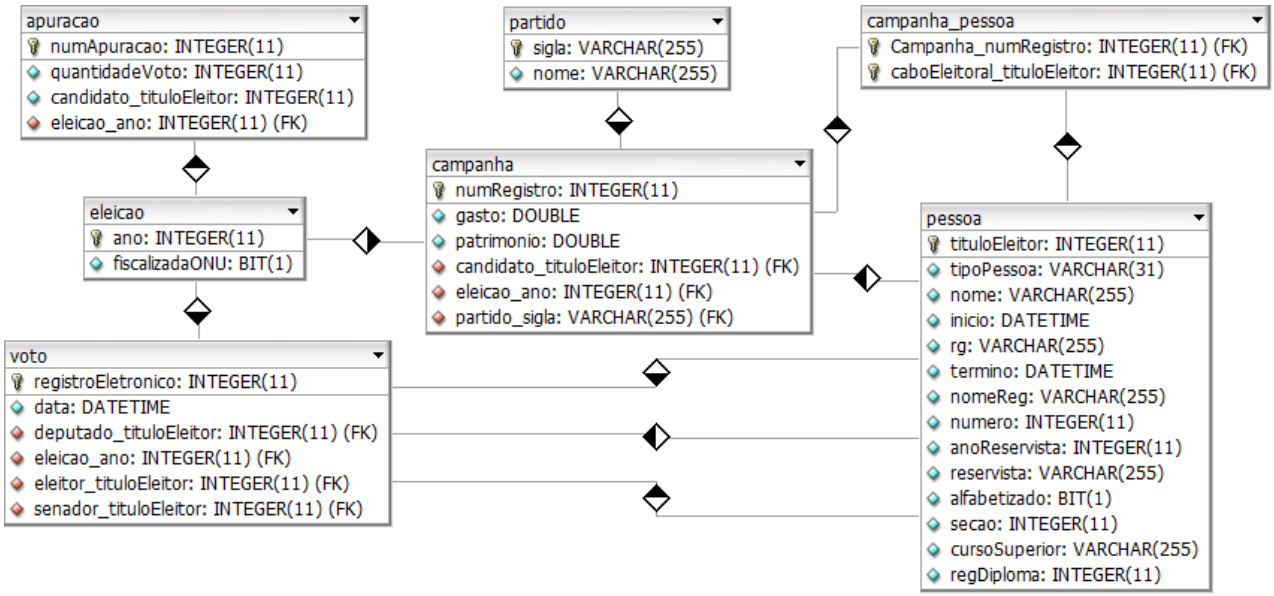

Figura 4.4: Diagrama de Tabelas: Eleição

A figura 4.5 mostra o espaço total em disco utilizado por ambos *frameworks* para a persistência dos objetos. Como pode ser observado, o comportamento de ambos os *frameworks* é similar, mas o

Object-Inject apresenta um consumo de disco menor para todos conjuntos de dados.

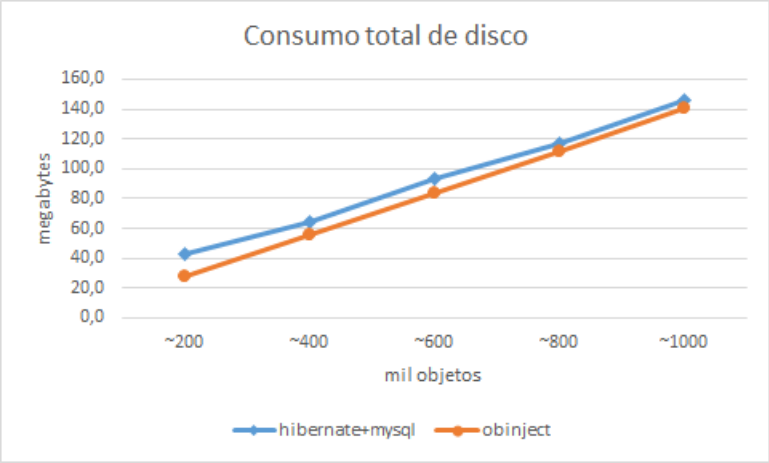

Figura 4.5: Consumo total de disco

4.2 Consultas

Assim como para a inserção, foram gerados 5 conjuntos de dados para realizar os teste relacionados à consulta. Cada conjunto é um subconjunto dos dados utilizados para realizar a inserção que foram foram gerados anteriormente. Para isto, foi sorteado um número para cada objeto, com valor entre 0 e 1. Entretanto, algum objetos receberam numeros iguais. A tabela 4.2 mostra a quantidade de objetos que serão consultados para cada conjunto de dados.

Tabela 4.2: Quantidade de Objetos para Consulta

Experimento	1	2	3	4	5
Partido	2	3	4	3	4
Voto	10.116	20.009	29.928	39.934	50.001
Eleitor	10.116	20.009	29.928	39.934	50.001
Senador	1	1	4	13	20
Deputado	13	45	89	146	289
Cabo Eleitoral	14	67	193	428	932
Apuração	11	33	115	190	327
Campanha	14	46	93	159	309
Total	20.287	40.213	60.354	80.807	101.883

Os Votos receberam o mesmo número de seus Eleitores relacionados e cabos receberam o mesmo número de seus Candidatos relacionados. Então, objetos com número menor que 0,1 foram escolhidos para compor o conjunto, ou seja, aproximadamente 10% dos objetos foram sorteados.

Após a geração destes 5 conjuntos de dados, foram criadas várias consultas para testar cada *framework*. Estas consultas buscam os objetos através de seus atributos chave primaria. Para isto, o Object-Inject utilizou o método *where* junto ao operador *Equal*, como pode ser visto na figura B.1.

De modo semelhantes, para medir o desempenho no *framework* Hibernate foram realizadas consultas equivalentes através da linguagem SQL, como pode ser visto na figura B.2.

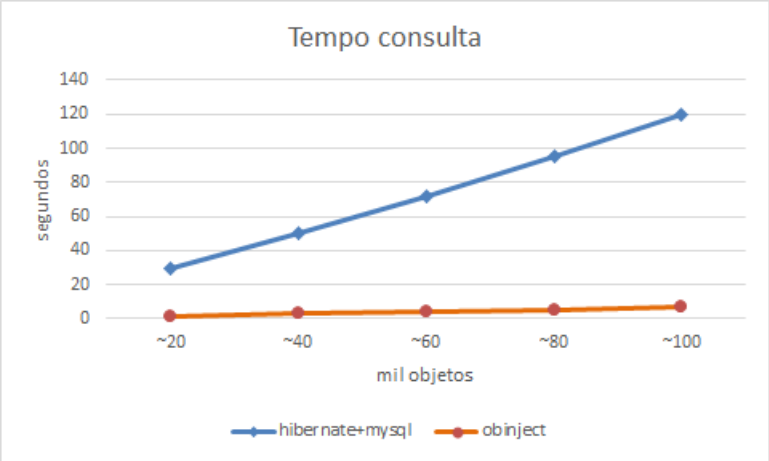

Figura 4.6: Tempo para consulta

O tempo de execução para realizar estas consultas foi medido e os resultados obtidos podem ser vistos na figura 4.6. Como pode ser observado, o Object-Inject apresenta um tempo muito inferior ao Hibernate.

Também foi medido a quantidade de memória utilizada durante a execução do aplicativo através do próprio Netbeans. A figura 4.7 mostra a quantidade média de memória utilizada. Pode-se observar que o Object-Inject apresenta um consumo de memória muuito inferior ao Hibernate.

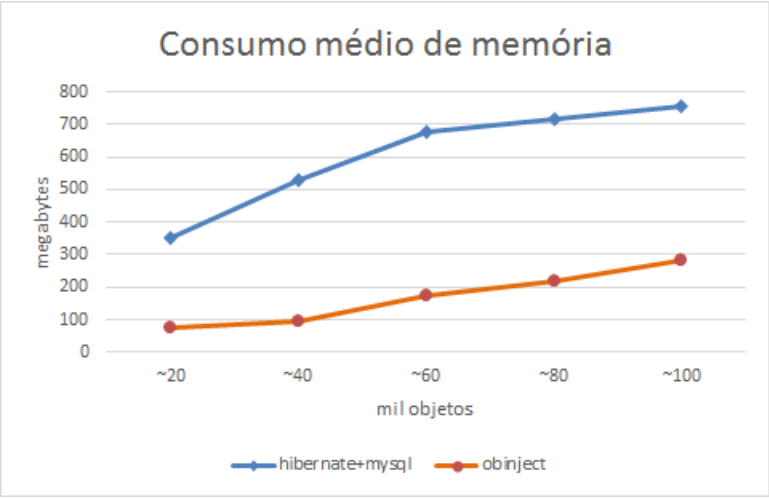

Figura 4.7: Consumo médio de memória

4.3 Considerações Finais

Em todos os testes feitos o Object-Inject demonstrou um desempenho melhor que o Hibernate. Porém, temos que observar que através do MySQL, o Hibernate oferece controle de transação, o que contribui para um pior desempenho do *framework*.

Capítulo 5
Conclusões

Ao longo deste trabalho, foi estudado o modo como se implementa consultas através de operadores da álgebra relacional, bem como várias linguagens de consulta. Estas linguagens permitem realizar extração de dados da base de forma estruturada e simples, sem a necessidade de se preocupar de que forma ou por quais estruturas os dados foram persistidos. Estas consultas são realizadas simplesmente descrevendo quais tipos de dados devem ser procurados e quais suas características.

A partir disto, foi desenvolvida a linguagem ObInject Query Language. Esta linguagem, permite a criação e definição de consultas para o *framework* Object-Inject através de métodos. Desta forma, o *framework* passa a apresentar ao usuário um meio de buscar dados na base de forma simples e estruturada, sem a necessidade de manipular as estruturas utilizadas para a persistência. Além disto, a linguagem possibilita a criação de consultas mais complexas, tornando o *framework* mais completo.

Adicionalmente, foram feitas extensões no módulo de metaprogramação e criado o pacote *generation*. Com estas extensões é possível a criação de mais de um índice para cada domínio de índice. Também se tornou possível o uso de múltiplos atributos para a criação de um índice.

Através dos experimentos, pode-se concluir que o *framework* Object-Inject apresenta um ótimo desempenho quando comparado ao Hibernate associado ao MySql, apesar de o Object-Inject não possuir controle de transação. Em relação ao tempo necessário para se fazer a persistência e a consulta, o Object-Inject demonstrou ser mais eficiente para ambos os casos. Em relação ao consumo médio de memória, o Object-Inject apresentou um consumo maior apenas para o primeiro conjunto dados durante a persistência e um consumo sempre menor durante a consulta. Em relação ao consumo de disco para armazenar os dados, o Object-Inject também apresentou um consumo menor.

5.1 Trabalhos Futuros

Como trabalho futuro, fica sugerido a expansão da linguagem ObInject Query Language. É necessário o ajuste do retorno da consulta, para permitir que apenas atributos sejam retornados. Também é preciso aplicar os operadores união e intersecção quando os operadores lógicos *and* e *or* são utilizados, respectivamente. Além disso, é necessário adicionar suporte à operadores binários, ordenação descendente para o método *orderBy* e a funções de agregação (*sum*, *count*, *max*, etc).

Referências Bibliográficas

BAUER, C.; KING, G. *Hibernate in Action*. Manning Publications Company, 2005. (In Action Series). ISBN 9781932394153. Disponível em: <http://books.google.com.br/books?id=WCmSQgAACAAJ>.

BERLER, M. et al. *The object data standard: ODMG 3.0*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. ISBN 1-55860-647-5.

BLASGEN, M. W.; ESWARAN, K. P. Storage and access in relational data bases. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 16, n. 4, p. 363–377, dez. 1977. ISSN 0018-8670. Disponível em: <http://dx.doi.org/10.1147/sj.164.0363>.

CARVALHO, L. O. et al. Obinject: a noodmg indexing and persistence framework. In: *Proceedings of the 2013 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2013. (SAC '13).

CATTELL, R. G. G. *The Object Database Standard: ODMG-93*. [S.l.]: Morgan Kaufmann, 1993. ISBN 1-55860-302-6.

CHAMBERLIN, D. D. et al. A history and evaluation of system r. *Commun. ACM*, ACM, New York, NY, USA, v. 24, n. 10, p. 632–646, out. 1981. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/358769.358784>.

CHAMBERLIN, D. D.; BOYCE, R. F. Sequel: A structured english query language. In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. New York, NY, USA: ACM, 1974. (SIGFIDET '74), p. 249–264. Disponível em: <http://doi.acm.org/10.1145/800296.811515>.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: *Proc. 23rd VLDB Conf.* Greece: [s.n.], 1997. p. 426–435.

CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM*, ACM, New York, NY, USA, v. 13, n. 6, p. 377–387, jun. 1970. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/362384.362685>.

COMER, D. The ubiquitous B-tree. *ACM Comput. Surv.*, v. 11, n. 2, p. 121–137, 1979.

CORRITORE, C. L.; WIEDENBECK, S. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, v. 54, n. 1, p. 1 – 23, 2001. ISSN 1071-5819. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1071581900904233>.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. *Database Systems: The Complete Book*. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 9780131873254.

GUPTA, A.; HARINARAYAN, V.; QUASS, D. Aggregate-query processing in data warehousing environments. In: *Proceedings of the 21th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. (VLDB '95), p. 358–369. ISBN 1-55860-379-4. Disponível em: <http://dl.acm.org/citation.cfm?id=645921.673150>.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Rec.*, v. 14, n. 2, p. 47–57, 1984.

HOU, D.; RUPAKHETI, C. R.; HOOVER, H. J. Documenting and evaluating scattered concerns for framework usability: A case study. In: *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008. (APSEC '08), p. 213–220. ISBN 978-0-7695-3446-6. Disponível em: <http://dx.doi.org/10.1109/APSEC.2008.39>.

ISO. Norm, *Information technology - Open Systems - Interconnection Remote Procedure Call (RPC)*. [S.l.]: International Organization for Standardization, 1996.

ITU. Norm, *Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components*. [S.l.]: International Telecommunication Union, 2004.

ITU. Norm, *"Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components*. [S.l.]: International Organization for Standardization, 2005.

KITSUREGAWA, M.; TANAKA, H.; MOTO-OKA, T. Application of hash to data base machine and its architecture. *New Generation Computing*, Springer-Verlag, v. 1, n. 1, p. 63–74, 1983. ISSN 0288-3635. Disponível em: <http://dx.doi.org/10.1007/BF03037022>.

LEACH, P. J.; MEALLING, M.; SALZ, R. *A Universally Unique IDentifier (UUID) URN Namespace*. 2005. Internet RFC 4122.

OLIVEIRA, E. S. Maurï¿½cio Faria de. *Metaprogramaï¿½ï¿½o e Metadados de Persistï¿½ncia e Indexaï¿½ï¿½o para o Framework Object-Injection*. Dissertação (Mestrado) — Universidade Federal de Itajubï¿½, 2012.

RAMAKRISHNAN, R.; GEHRKE, J. *Database management systems (3. ed.)*. [S.l.]: McGraw-Hill, 2003. I-XXXII, 1-1065 p. ISBN 978-0-07-115110-8.

ZAHN, L.; DINEEN, T.; LEACH, P. *Network computing architecture*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990. ISBN 0-13-611674-4.

ZEZULA, P. et al. *Similarity Search: The Metric Space Approach*. [S.l.]: Springer, 2005.

Apêndice

A

Classes de empacotamento para classe Terreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.device.File;
4   import org.obinject.meta.*;
5   import org.obinject.queries.Attribute;
6   import org.obinject.queries.Schema;
7   import org.obinject.storage.*;
8
9   public class EntityTerreno extends Terreno implements Entity<EntityTerreno> {
10
11      private Uuid uuid = Uuid.generator();
12      public static final Uuid classId = Uuid.fromString("80114B3D-345B-3B17-9B4D-4745C72A40D0");
13      public static final Sequential<EntityTerreno> entityStructure = new Sequential<EntityTerreno>(
14          new File("build/classes/org/obinject/sample/terreno/EntityTerreno.dbo", 4096)) { };
15      public static final BTree<PrimaryKeyTerreno> primaryKeyTerrenoStructure = new BTree<PrimaryKeyTerreno>(
16          new File("build/classes/org/obinject/sample/terreno/PrimaryKeyTerreno.pk", 4096)) { };
17      public static final BTree<OrderFirstTerreno> orderFirstTerrenoStructure = new BTree<OrderFirstTerreno>(
18          new File("build/classes/org/obinject/sample/terreno/OrderFirstTerreno.btree", 4096)) { };
19      public static final BTree<OrderSecondTerreno> orderSecondTerrenoStructure = new BTree<OrderSecondTerreno>(
20          new File("build/classes/org/obinject/sample/terreno/OrderSecondTerreno.btree", 4096)) { };
21      public static final RTree<RectangleFirstTerreno> rectangleFirstTerrenoStructure = new RTree<RectangleFirstTerreno>(
22          new File("build/classes/org/obinject/sample/terreno/RectangleFirstTerreno.rtree", 4096)) { };
23      public static final MTree<EditionFirstTerreno> editionFirstTerrenoStructure = new  MTree<EditionFirstTerreno>(
24          new File("build/classes/org/obinject/sample/terreno/EditionFirstTerreno.mtree", 4096)) { };
25      public static final MTree<GeoPointFirstTerreno> geoPointFirstTerrenoStructure = new MTree<GeoPointFirstTerreno>(
26          new File("build/classes/org/obinject/sample/terreno/GeoPointFirstTerreno.mtree", 4096)) { };
27      public static final MTree<GeoPointSecondTerreno> geoPointSecondTerrenoStructure = new MTree<GeoPointSecondTerreno>(
28          new File("build/classes/org/obinject/sample/terreno/GeoPointSecondTerreno.mtree", 4096)) { };
29      public static final MTree<PointFirstTerreno> pointFirstTerrenoStructure = new MTree<PointFirstTerreno>(
30          new File("build/classes/org/obinject/sample/terreno/PointFirstTerreno.mtree", 4096)) { };
31      public static final Attribute $registro = new Attribute();
32      public static final Attribute $proprietario = new Attribute();
33      public static final Attribute $cidade = new Attribute();
34      public static final Attribute $bairro = new Attribute();
35      public static final Attribute $logradouro = new Attribute();
36      public static final Attribute $numero = new Attribute();
37      public static final Attribute $latitudeMinima = new Attribute();
38      public static final Attribute $longitudeMinima = new Attribute();
39      public static final Attribute $latitudeMaxima = new Attribute();
40      public static final Attribute $longitudeMaxima = new Attribute();
```

Figura A.1: Classe de empacotamento EntityTerreno (Parte 1)

```
42      public EntityTerreno() {
43      }
44
45      public EntityTerreno(Terreno obj) {
46          this.setRegistro(obj.getRegistro());
47          this.setProprietario(obj.getProprietario());
48          this.setCidade(obj.getCidade());
49          this.setBairro(obj.getBairro());
50          this.setLogradouro(obj.getLogradouro());
51          this.setNumero(obj.getNumero());
52          this.setLatitudeMinima(obj.getLatitudeMinima());
53          this.setLongitudeMinima(obj.getLongitudeMinima());
54          this.setLatitudeMaxima(obj.getLatitudeMaxima());
55          this.setLongitudeMaxima(obj.getLongitudeMaxima());
56      }
57
58      @Override
59      public boolean isEqual(EntityTerreno obj) {
60          return (this.getRegistro() == obj.getRegistro()) && (((this.getProprietario() == null) &&
61                  (obj.getProprietario() == null)) || ((this.getProprietario() != null) &&
62                  (obj.getProprietario() != null) && (this.getProprietario().equals(obj.getProprietario())))) &&
63                  (((this.getCidade() == null) && (obj.getCidade() == null)) || ((this.getCidade() != null) &&
64                  (obj.getCidade() != null) && (this.getCidade().equals(obj.getCidade())))) &&
65                  (((this.getBairro() == null) && (obj.getBairro() == null)) || ((this.getBairro() != null) &&
66                  (obj.getBairro() != null) && (this.getBairro().equals(obj.getBairro())))) &&
67                  (((this.getLogradouro() == null) && (obj.getLogradouro() == null)) ||
68                  ((this.getLogradouro() != null) && (obj.getLogradouro() != null) &&
69                  (this.getLogradouro().equals(obj.getLogradouro())))) && (this.getNumero() == obj.getNumero()) &&
70                  (this.getLatitudeMinima() == obj.getLatitudeMinima()) &&
71                  (this.getLongitudeMinima() == obj.getLongitudeMinima()) &&
72                  (this.getLatitudeMaxima() == obj.getLatitudeMaxima()) &&
73                  (this.getLongitudeMaxima() == obj.getLongitudeMaxima());
74      }
75
76      @Override
77      public Uuid getUuid() {
78          return this.uuid;
79      }
80
81      @Override
82      public EntityStructure<EntityTerreno> getEntityStructure() {
83          return entityStructure;
84      }
```

Figura A.2: Classe de empacotamento EntityTerreno (Parte 2)

```
86      @Override
87      public boolean inject() {
88          PrimaryKeyTerreno primaryKey = new PrimaryKeyTerreno(this);
89          Uuid uuidInject = EntityTerreno.primaryKeyTerrenoStructure.find(primaryKey);
90          if (uuidInject == null) {
91              EntityTerreno.entityStructure.add(this);
92              EntityTerreno.primaryKeyTerrenoStructure.add(primaryKey);
93              EntityTerreno.orderFirstTerrenoStructure.add(new OrderFirstTerreno(this));
94              EntityTerreno.orderSecondTerrenoStructure.add(new OrderSecondTerreno(this));
95              EntityTerreno.rectangleFirstTerrenoStructure.add(new RectangleFirstTerreno(this));
96              EntityTerreno.editionFirstTerrenoStructure.add(new EditionFirstTerreno(this));
97              EntityTerreno.geoPointFirstTerrenoStructure.add(new GeoPointFirstTerreno(this));
98              EntityTerreno.geoPointSecondTerrenoStructure.add(new GeoPointSecondTerreno(this));
99              EntityTerreno.pointFirstTerrenoStructure.add(new PointFirstTerreno(this));
100             return true;
101         } else {
102             this.uuid = uuidInject;
103             return false;
104         }
105     }
106
107     @Override
108     public boolean pullEntity(byte[] array, int position) {
109         PullStream pull = new PullStream(array, position);
110         Uuid storedClass = pull.pullUuid();
111         if (classId.equals(storedClass) == true) {
112             uuid = pull.pullUuid();
113             this.setRegistro(pull.pullLong());
114             this.setProprietario(pull.pullString());
115             this.setCidade(pull.pullString());
116             this.setBairro(pull.pullString());
117             this.setLogradouro(pull.pullString());
118             this.setNumero(pull.pullInt());
119             this.setLatitudeMinima(pull.pullFloat());
120             this.setLongitudeMinima(pull.pullFloat());
121             this.setLatitudeMaxima(pull.pullFloat());
122             this.setLongitudeMaxima(pull.pullFloat());
123             return true;
124         }
125         return false;
126     }
```

Figura A.3: Classe de empacotamento EntityTerreno (Parte 3)

```
128         @Override
129         public void pushEntity(byte[] array, int position) {
130             PushStream push = new PushStream(array, position);
131             push.pushUuid(classId);
132             push.pushUuid(uuid);
133             push.pushLong(this.getRegistro());
134             push.pushString(this.getProprietario());
135             push.pushString(this.getCidade());
136             push.pushString(this.getBairro());
137             push.pushString(this.getLogradouro());
138             push.pushInt(this.getNumero());
139             push.pushFloat(this.getLatitudeMinima());
140             push.pushFloat(this.getLongitudeMinima());
141             push.pushFloat(this.getLatitudeMaxima());
142             push.pushFloat(this.getLongitudeMaxima());
143         }
144
145         @Override
146         public int sizeOfEntity() {
147             return Stream.sizeOfUuid + Stream.sizeOfUuid + Stream.sizeOfLong + Stream.sizeOfString(this.getProprietario()) +
148                     Stream.sizeOfString(this.getCidade()) + Stream.sizeOfString(this.getBairro()) +
149                     Stream.sizeOfString(this.getLogradouro()) + Stream.sizeOfInt + Stream.sizeOfFloat + Stream.sizeOfFloat +
150                     Stream.sizeOfFloat + Stream.sizeOfFloat;
151         }
152
153         static {
154             $registro.getSchemas().add(new Schema<PrimaryKeyTerreno, Long>() {...13 linhas });
167             $proprietario.getSchemas().add(new Schema<OrderSecondTerreno, java.lang.String>() {...13 linhas });
180             $proprietario.getSchemas().add(new Schema<EditionFirstTerreno, java.lang.String>() {...13 linhas });
193             $cidade.getSchemas().add(new Schema<OrderFirstTerreno, java.lang.String>() {...13 linhas });
206             $bairro.getSchemas().add(new Schema<OrderFirstTerreno, java.lang.String>() {...13 linhas });
219             $logradouro.getSchemas().add(new Schema<OrderFirstTerreno, java.lang.String>() {...13 linhas });
232             $latitudeMinima.getSchemas().add(new Schema<RectangleFirstTerreno, Float>() {...13 linhas });
245             $latitudeMinima.getSchemas().add(new Schema<PointFirstTerreno, Float>() {...13 linhas });
258             $latitudeMinima.getSchemas().add(new Schema<GeoPointFirstTerreno, Float>() {...13 linhas });
271             $longitudeMinima.getSchemas().add(new Schema<RectangleFirstTerreno, Float>() {...13 linhas });
284             $longitudeMinima.getSchemas().add(new Schema<PointFirstTerreno, Float>() {...13 linhas });
297             $longitudeMinima.getSchemas().add(new Schema<GeoPointFirstTerreno, Float>() {...13 linhas });
310             $latitudeMaxima.getSchemas().add(new Schema<RectangleFirstTerreno, Float>() {...13 linhas });
323             $latitudeMaxima.getSchemas().add(new Schema<GeoPointSecondTerreno, Float>() {...13 linhas });
336             $longitudeMaxima.getSchemas().add(new Schema<RectangleFirstTerreno, Float>() {...13 linhas });
349             $longitudeMaxima.getSchemas().add(new Schema<GeoPointSecondTerreno, Float>() {...13 linhas });
362         }
363     }
```

Figura A.4: Classe de empacotamento EntityTerreno (Parte 4)

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class PrimaryKeyTerreno extends EntityTerreno implements Order<PrimaryKeyTerreno>,Comparable<PrimaryKeyTerreno> {
7
8       private long calculatedComparisons;
9
10      public PrimaryKeyTerreno() {...2 linhas }
12
13      public PrimaryKeyTerreno(Terreno obj) {...12 linhas }
25
26      @Override
27      public int compareTo(PrimaryKeyTerreno obj) {...10 linhas }
37
38      @Override
39      public long getCalculatedComparisons() {...3 linhas }
42
43      @Override
44      public KeyStructure<PrimaryKeyTerreno> getKeyStructure() {...3 linhas }
47
48      @Override
49      public boolean pullKey(byte[] array, int position) {...5 linhas }
54
55      @Override
56      public void pushKey(byte[] array, int position) {...4 linhas }
60
61      @Override
62      public int sizeOfKey() {...3 linhas }
65  }
```

Figura A.5: Classe de empacotamento PrimaryKeyTerreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class OrderFirstTerreno extends EntityTerreno implements Order<OrderFirstTerreno>, Comparable<OrderFirstTerreno> {
7
8       private long calculatedComparisons;
9
10      public OrderFirstTerreno() {...2 linhas }
12
13      public OrderFirstTerreno(Terreno obj) {...12 linhas }
25
26      @Override
        public int compareTo(OrderFirstTerreno obj) {...10 linhas }
37
38      @Override
        public long getCalculatedComparisons() {...3 linhas }
42
43      @Override
        public KeyStructure<OrderFirstTerreno> getKeyStructure() {...3 linhas }
47
48      @Override
        public boolean pullKey(byte[] array, int position) {...7 linhas }
56
57      @Override
        public void pushKey(byte[] array, int position) {...6 linhas }
64
65      @Override
        public int sizeOfKey() {...3 linhas }
69  }
```

Figura A.6: Classe de empacotamento OrderFirstTerreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class OrderSecondTerreno extends EntityTerreno implements Order<OrderSecondTerreno>, Comparable<OrderSecondTerreno
7
8       private long calculatedComparisons;
9
10      public OrderSecondTerreno() {...2 linhas }
12
13      public OrderSecondTerreno(Terreno obj) {...12 linhas }
25
26      @Override
        public int compareTo(OrderSecondTerreno obj) {...10 linhas }
37
38      @Override
        public long getCalculatedComparisons() {...3 linhas }
42
43      @Override
        public KeyStructure<OrderSecondTerreno> getKeyStructure() {...3 linhas }
47
48      @Override
        public boolean pullKey(byte[] array, int position) {...5 linhas }
54
55      @Override
        public void pushKey(byte[] array, int position) {...4 linhas }
60
61      @Override
        public int sizeOfKey() {...3 linhas }
65  }
```

Figura A.7: Classe de empacotamento OrderSecondTerreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class EditionFirstTerreno extends EntityTerreno implements Edition<EditionFirstTerreno> {
7
8       private long calculatedDistance;
9       private double preservedDistance;
10
11      public EditionFirstTerreno() {...2 linhas }
13
14      public EditionFirstTerreno(Terreno obj) {...12 linhas }
26
27      @Override
        public long getCalculatedDistance() {...3 linhas }
31
32      @Override
        public double getPreservedDistance() {...3 linhas }
36
37      @Override
        public String getString() {...3 linhas }
41
42      @Override
        public double distanceTo(EditionFirstTerreno obj) {...4 linhas }
47
48      @Override
        public KeyStructure<EditionFirstTerreno> getKeyStructure() {...3 linhas }
52
53      @Override
        public boolean pullKey(byte[] array, int position) {...5 linhas }
59
60      @Override
        public void pushKey(byte[] array, int position) {...4 linhas }
65
66      @Override
        public void setPreservedDistance(double distance) {...3 linhas }
70
71      @Override
        public int sizeOfKey() {...3 linhas }
75  }
```

Figura A.8: Classe de empacotamento EditionFirstTerreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class GeoPointFirstTerreno extends EntityTerreno implements Point<GeoPointFirstTerreno> {
7
8       private long calculatedDistance;
9       private double preservedDistance;
10
11      public GeoPointFirstTerreno() {...2 linhas }
13
14      public GeoPointFirstTerreno(Terreno obj) {...12 linhas }
26
27      @Override
        public double getOrigin(int axis) {...10 linhas }
38
39      @Override
        public void setOrigin(int axis, double value) {...10 linhas }
50
51      @Override
        public long getCalculatedDistance() {...3 linhas }
55
56      @Override
        public double getPreservedDistance() {...3 linhas }
60
61      @Override
        public int numberOfDimensions() {...3 linhas }
65
66      @Override
        public void setPreservedDistance(double distance) {...3 linhas }
70
71      @Override
        public double distanceTo(GeoPointFirstTerreno obj) {...4 linhas }
76
77      @Override
        public KeyStructure<GeoPointFirstTerreno> getKeyStructure() {...3 linhas }
81
82      @Override
        public boolean pullKey(byte[] array, int position) {...8 linhas }
91
92      @Override
        public void pushKey(byte[] array, int position) {...7 linhas }
100
101     @Override
        public int sizeOfKey() {...3 linhas }
105 }
```

Figura A.9: Classe de empacotamento GeoPointFirstTerreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class GeoPointSecondTerreno extends EntityTerreno implements Point<GeoPointSecondTerreno> {
7
8       private long calculatedDistance;
9       private double preservedDistance;
10
11      public GeoPointSecondTerreno() {...2 linhas }
13
14      public GeoPointSecondTerreno(Terreno obj) {...12 linhas }
26
27      @Override
        public double getOrigin(int axis) {...10 linhas }
38
39      @Override
        public void setOrigin(int axis, double value) {...10 linhas }
50
51      @Override
        public long getCalculatedDistance() {...3 linhas }
55
56      @Override
        public double getPreservedDistance() {...3 linhas }
60
61      @Override
        public int numberOfDimensions() {...3 linhas }
65
66      @Override
        public void setPreservedDistance(double distance) {...3 linhas }
70
71      @Override
        public double distanceTo(GeoPointSecondTerreno obj) {...4 linhas }
76
77      @Override
        public KeyStructure<GeoPointSecondTerreno> getKeyStructure() {...3 linhas }
81
82      @Override
        public boolean pullKey(byte[] array, int position) {...8 linhas }
91
92      @Override
        public void pushKey(byte[] array, int position) {...7 linhas }
100
101     @Override
        public int sizeOfKey() {...3 linhas }
105 }
```

Figura A.10: Classe de empacotamento GeoPointSecondTerreno

```
1   package org.obinject.sample.terreno;
2
3   import org.obinject.meta.*;
4   import org.obinject.storage.KeyStructure;
5
6   public class PointFirstTerreno extends EntityTerreno implements Point<PointFirstTerreno> {
7
8       private long calculatedDistance;
9       private double preservedDistance;
10
11      public PointFirstTerreno() {...2 linhas }
13
14      public PointFirstTerreno(Terreno obj) {...12 linhas }
26
27      @Override
        public double getOrigin(int axis) {...10 linhas }
38
39      @Override
        public void setOrigin(int axis, double value) {...10 linhas }
50
51      @Override
        public long getCalculatedDistance() {...3 linhas }
55
56      @Override
        public double getPreservedDistance() {...3 linhas }
60
61      @Override
        public int numberOfDimensions() {...3 linhas }
65
66      @Override
        public void setPreservedDistance(double distance) {...3 linhas }
70
71      @Override
        public double distanceTo(PointFirstTerreno obj) {...4 linhas }
76
77      @Override
        public KeyStructure<PointFirstTerreno> getKeyStructure() {...3 linhas }
81
82      @Override
        public boolean pullKey(byte[] array, int position) {...8 linhas }
91
92      @Override
        public void pushKey(byte[] array, int position) {...7 linhas }
100
101     @Override
        public int sizeOfKey() {...3 linhas }
105 }
```

Figura A.11: Classe de empacotamento PointFirstTerreno

```
1   package org.obinject.sample.terreno;
2   import ...2 linhas
4   public class RectangleFirstTerreno extends EntityTerreno implements Rectangle<RectangleFirstTerreno> {
5
6       private long calculatedDistance;
7       private double preservedDistance;
8
9       public RectangleFirstTerreno() {...2 linhas }
11
12      public RectangleFirstTerreno(Terreno obj) {...12 linhas }
24
25      @Override
        public KeyStructure<RectangleFirstTerreno> getKeyStructure() {...3 linhas }
29
30      @Override
        public boolean pullKey(byte[] array, int position) {...9 linhas }
40
41      @Override
        public void pushKey(byte[] array, int position) {...8 linhas }
50
51      @Override
        public int sizeOfKey() {...3 linhas }
55
56      @Override
        public double getOrigin(int axis) {...10 linhas }
67
68      @Override
        public void setOrigin(int axis, double value) {...10 linhas }
79
80      @Override
        public int numberOfDimensions() {...3 linhas }
84
85      @Override
        public double getExtension(int axis) {...10 linhas }
96
97      @Override
        public void setExtension(int axis, double value) {...10 linhas }
108
109     @Override
        public long getCalculatedDistance() {...3 linhas }
113
114     @Override
        public double getPreservedDistance() {...3 linhas }
118
119     @Override
        public void setPreservedDistance(double distance) {...3 linhas }
123
124     @Override
        public double distanceTo(RectangleFirstTerreno obj) {...4 linhas }
129 }
```

Figura A.12: Classe de empacotamento RectangleFirstTerreno

Apêndice

B

Consultas geradas para o Object-Inject e Hibernate

```
Query q1 = new Query();
q1.from(EntitySenador.class);
q1.where(new Equal(EntitySenador.$tituloEleitor, senador.getTituloEleitor()));
Collection<Senador> resSenador = q1.execute();

Query q2 = new Query();
q2.from(EntityDeputado.class);
q2.where(new Equal(EntityDeputado.$tituloEleitor, deputado.getTituloEleitor()));
Collection<Deputado> resDeputado = q2.execute();

Query q3 = new Query();
q3.from(EntityCaboEleitoral.class);
q3.where(new Equal(EntityCaboEleitoral.$tituloEleitor, caboEleitoral.getTituloEleitor()));
Collection<CaboEleitoral> resCaboEleitoral = q3.execute();

Query q4 = new Query();
q4.from(EntityEleitor.class);
q4.where(new Equal(EntityEleitor.$tituloEleitor, eleitor.getTituloEleitor()));
Collection<Eleitor> resEleitor = q4.execute();

Query q5 = new Query();
q5.from(EntityPartido.class);
q5.where(new Equal(EntityPartido.$sigla, partido.getSigla()));
Collection<Partido> resPartido = q5.execute();

Query q6 = new Query();
q6.from(EntityCampanha.class);
q6.where(new Equal(EntityCampanha.$numeroRegistro, campanha.getNumeroRegistro()));
Collection<Campanha> resCampanha = q6.execute();

Query q7 = new Query();
q7.from(EntityVoto.class);
q7.where(new Equal(EntityVoto.$registroEletronico, voto.getRegistroEletronico()));
Collection<Voto> resVoto = q7.execute();

Query q8 = new Query();
q8.from(EntityApuracao.class);
q8.where(new Equal(EntityApuracao.$numeroApuracao, apuracao.getNumeroApuracao()));
Collection<Apuracao> resApuracao = q8.execute();
```

Figura B.1: Consultas realizadas no Object-Inject

```
EntityManager em1 = emf.createEntityManager();
Collection<Senador> resSenador = em1.createQuery(
        "from Senador o where o.tituloEleitor = "
        + senador.getTituloEleitor()).getResultList();

EntityManager em2 = emf.createEntityManager();
Collection<Deputado> resDeputado = em2.createQuery(
        "from Deputado o where o.tituloEleitor = "
        + deputado.getTituloEleitor()).getResultList();

EntityManager em3 = emf.createEntityManager();
Collection<CaboEleitoral> resCaboEleitoral = em3.createQuery(
        "from CaboEleitoral o where o.tituloEleitor = "
        + caboEleitoral.getTituloEleitor()).getResultList();

EntityManager em4 = emf.createEntityManager();
Collection<Eleitor> resEleitor = em4.createQuery(
        "from Eleitor o where o.tituloEleitor = "
        + eleitor.getTituloEleitor()).getResultList();

EntityManager em5 = emf.createEntityManager();
Collection<Partido> resPartido = em4.createQuery(
        "from Partido o where o.sigla = '"
        + partido.getSigla() + "'").getResultList();

EntityManager em6 = emf.createEntityManager();
Collection<Campanha> resCampanha = em6.createQuery(
        "from Campanha o where o.numeroRegistro = "
        + campanha.getNumeroRegistro()).getResultList();

EntityManager em7 = emf.createEntityManager();
Collection<Voto> resVoto = em7.createQuery(
        "from Voto o where o.registroEletronico = "
        + voto.getRegistroEletronico()).getResultList();

EntityManager em8 = emf.createEntityManager();
Collection<Apuracao> resApuracao = em8.createQuery(
        "from Apuracao o where o.numeroApuracao = "
        + apuracao.getNumeroApuracao()).getResultList();
```

Figura B.2: Consultas realizadas no Hibernate