UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

INTERPRETADOR DE CONSULTAS PARA OBJETOS JAVA

DOUGLAS MATHEUS DE SOUZA

BLUMENAU
2011

2011/2-05

DOUGLAS MATHEUS DE SOUZA

INTERPRETADOR DE CONSULTAS PARA OBJETOS JAVA

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Marcel Hugo , Mestre - Orientador

BLUMENAU
2011

2011/2-05

INTERPRETADOR DE CONSULTAS PARA OBJETOS JAVA

Por

DOUGLAS MATHEUS DE SOUZA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	____ Prof. Marcel Hugo, Mestre – Orientador, FURB
Membro:	____ Prof. Joyce Martins, Mestre – FURB
Membro:	____ Prof. Fernanda Gums – FURB

Blumenau, 14 de dezembro de 2011

Dedico este trabalho aos que me apoiaram durante o seu desenvolvimento, em especial minha família e amigos.

AGRADECIMENTOS

À minha família, que mesmo longe, sempre esteve presente.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Marcel Hugo, por ter acreditado que seria possível o desenvolvimento deste trabalho.

A mente que se abre a uma nova idéia jamais volta ao seu tamanho original.

Albert Einstein

RESUMO

Este trabalho apresenta o desenvolvimento de uma biblioteca de consulta para objetos que implementam a interface *Collection* e vetores da linguagem Java. As consultas são feitas através de uma linguagem específica em conjunto com o interpretador que recebe como entrada uma *String* contendo uma sequência de comandos. Se a entrada é processada com sucesso, uma lista contendo as informações requisitadas é retornada. Caso contrário, uma exceção é lançada informando o programador sobre o erro ocorrido.

Palavras-chave: Consultas. Filtros. Coleções Java.

ABSTRACT

This work presents the development of a library for querying Java objects that implement the Collection interface and arrays of Java language. Queries are made through a specific language beside an interpreter which receives a String containing a sequence of commands. If the received String is successfully processed, the interpreter returns a list with the requested information. Otherwise, the interpreter throws an exception informing the developer about the error occurred.

Key-words: Queries. Filters. Java Collections.

LISTA DE ILUSTRAÇÕES

Quadro 1 - Código em C# com utilização da extensão LINQ to Objects 17
Quadro 2 - Exemplo de consulta usando a linguagem SQL 18
Quadro 3 – Exemplo de uma lista de *tokens* 24
Quadro 4 - Exemplo de construção BNF 25
Quadro 5 - Comparação entre o uso da API *Collections* e a biblioteca Lambdaj 27
Quadro 6 - Exemplo de código com utilização de funções da biblioteca Coollection 28
Quadro 7 – Descrição das classes da primeira etapa do analisador 35
Quadro 8 – Representação do algoritmo de busca linear para restrições simples 42
Quadro 9 – Algoritmo de junção de laços aninhados 42
Quadro 10 – Pseudocódigo do algoritmo de junção baseada em *hash* 43
Quadro 11 – Algoritmo do operador de agrupamento 44
Quadro 12 – Algoritmo de projeção 45
Quadro 13 – Detalhamento das classes de exceção do analisador semântico 46
Quadro 14 – Exemplo de código Java para utilização do interpretador 47
Quadro 15 – Exemplos de uso das cláusulas da linguagem 48
Quadro 16 – Exemplos de *queries* para recuperar dados das coleções 49
Quadro 17 – Comparação entre as bibliotecas de consultas do Java 51
Quadro 18 – Recursos disponíveis em cada uma das ferramentas analisadas 52
Quadro 19 – Gramática da linguagem de consulta o interpretador 58

Figura 1 – Exemplo de seleção 19
Figura 2 - Exemplo de funcionamento do operador projeção 20
Figura 3 - Exemplo de produto cartesiano entre duas relações 21
Figura 4 – Exemplo de junção natural 22
Figura 5 - Exemplo de operação de diferença entre duas tabelas de um banco de dados........ 22
Figura 6 – Árvore de operadores da álgebra relacional 23
Figura 7 – Estrutura básica de um compilador 24
Figura 8 – Árvore sintática de uma expressão matemática 26
Figura 9 – Diagrama de classes dos analisadores léxico e sintático 31
Figura 10 – Diagrama de classes do analisador semântico 33
Figura 11 – Diagrama de atividades do interpretador 35

Figura 12 – Diagrama da montagem da árvore de consulta 36
Figura 13 – Inserção do operador de projeção 37
Figura 14 – Inserção do operador de ordenação 37
Figura 15 – Inserção do operador de agrupamento 37
Figura 16 – Inserção das restrições 38
Figura 17 – Descida das restrições simples 38
Figura 18 – Inserção das relações 39
Figura 19 – Ordenação das restrições simples 39
Figura 20 – Ordenação das junções 40
Figura 21 – Diagrama de classes do estudo de caso 48
Figura 22 – *Query* para buscar a quantidade de álbuns por artista 50
Figura 23 – Diagrama de classes do *parser* simplificado 59
Figura 24 – Diagrama de classes do *parser* gerado pelo JavaCC 60
Figura 25 - Diagrama de classes do *parser* gerado pelo JavaCC 61
Figura 26 – Classes do pacote `br.com.joqi.semantico.consulta` 62
Figura 27 - Classes do pacote `br.com.joqi.semantico.consulta.agrupamento` 63
Figura 28 - Classes do pacote `br.com.joqi.semantico.consulta.ordenacao` .. 63
Figura 29 – Diagrama de classes do plano de execução 64
Figura 30 - Classes do pacote `br.com.joqi.semantico.consulta.projecao` 65
Figura 31 - Classes do pacote `br.com.joqi.semantico.consulta.restricao` .. 66
Figura 32- Classes do pacote `br.com.joqi.semantico.exception` 67

LISTA DE SIGLAS

ANSI - *American National Standards Institute*

API - *Application Program Interface*

BNF - *Backus-Naur Form*

DDL - *Data Definition Language*

DML - *Data Manipulation Language*

EBNF - *Extended Backus-Naur Form*

HQL - *Hibernate Query Language*

IBM - *International Business Machine*

JavaCC - Java *Compiler Compiler*

LINQ - *Language INtegrated Query*

RF – Requisito Funcional

RNF – Requisito Não-Funcional

SEQUEL - *Structured English QUEry Language*

SGBD - Sistema Gerenciador de Banco de Dados

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

XML - *eXtensible Markup Language*

LISTA DE SÍMBOLOS

= - igual

π - pi

σ - sigma

> - maior

< - menor

× - produto cartesiano

- sustenido

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS DO TRABALHO 15
1.2 ESTRUTURA DO TRABALHO 15
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 LINQ 16
2.2 LINGUAGEM SQL 17
2.3 ÁLGEBRA RELACIONAL 18
2.3.1 Seleção 19
2.3.2 Projeção 19
2.3.3 Produto cartesiano 20
2.3.4 Junção 21
2.3.5 Diferença 22
2.3.6 Combinação dos operadores 22
2.4 COMPILADORES 23
2.4.1 Análise léxica 24
2.4.2 Análise sintática 25
2.4.3 Análise semântica 26
2.5 TRABALHOS CORRELATOS 26
2.5.1 Lambdaj 27
2.5.2 Coollection 27
3 DESENVOLVIMENTO DA FERRAMENTA 29
3.1 REQUISITOS DA FERRAMENTA 29
3.2 ESPECIFICAÇÃO DO INTERPRETADOR 29
3.2.1 Ferramentas para geração de compiladores 30
3.2.2 Analisadores léxico e sintático 30
3.2.3 Analisador semântico 32
3.2.4 Diagrama de atividades 35
3.3 IMPLEMENTAÇÃO 36
3.3.1 Geração da árvore de consulta 36
3.3.2 Execução da consulta 40
3.3.2.1 Renomeação 41

3.3.2.2 Busca linear 41
3.3.2.3 Junção de laços aninhados 42
3.3.2.4 Junção baseada em *hash* 43
3.3.2.5 Agrupamento por *hash* 44
3.3.2.6 Ordenação 44
3.3.2.7 Projeção 45
3.3.3 Tratamento de erros 45
3.3.4 Operacionalidade da implementação 47
3.3.4.1 Estudo de caso 48
3.4 RESULTADOS E DISCUSSÃO 50
4 CONCLUSÕES 53
4.1 EXTENSÕES 54
REFERÊNCIAS BIBLIOGRÁFICAS 55
APÊNDICE A – Gramática da linguagem de consulta do interpretador 57
APÊNDICE B – Diagrama de classes do parser gerado pelo JavaCC 59
APÊNDICE C – Diagramas de classes do analisador semântico 62

1 INTRODUÇÃO

Uma característica presente em quase todas as aplicações é a consulta de dados. Uma consulta pode ser efetuada de variadas formas e em diferentes tipos de fontes. Um tipo de fonte, usada em grande parte dos casos, é um Sistema Gerenciador de Banco de Dados (SGBD) ou um banco de dados. Em outras situações, os dados podem estar na memória, em estruturas conhecidas como vetores, listas, filas, pilhas, entre outras.

Quando armazenados em um SGBD, os dados podem ser facilmente recuperados através de linguagens especializadas de consulta como a *Structured Query Language* (SQL), que é utilizada nos bancos de dados relacionais. Com o SQL o desenvolvedor tem a possibilidade de criar bases de dados, adicionar e manter dados em bases já existentes e selecionar informações (TAYLOR, 2010, p. 7), além de realizar operações lógicas e aritméticas sobre um conjunto de dados com uma quantidade de código pequena.

Para objetos em memória, tomando como exemplo a maioria das linguagens de programação, é necessário que sejam feitos laços que encapsulem estruturas condicionais, o que pode tornar o código de difícil entendimento dependendo da quantidade de condições. No entanto, um recurso conhecido como *Language INtegrated Query* (LINQ), oferecido pela Microsoft e que vem sendo bem recebido pela comunidade de desenvolvedores da plataforma .NET, pode facilitar a realização de consultas em memória.

A LINQ nada mais é do que uma tecnologia que fornece mecanismos de apoio para consulta de dados de diferentes tipos incluindo vetores e coleções em memória, bases de dados, arquivos *eXtensible Markup Language* (XML), entre outros (FREEMAN; RATTZ JR., 2010, p. 7). Atualmente integrado as linguagens C# e Visual Basic, possui diferentes extensões, entre elas: LINQ to *Entities*, LINQ to SQL, LINQ to XML. Além das três extensões citadas, está disponível a LINQ to *Objects* que permite ao desenvolvedor efetuar consultas em memória com uma quantidade de código relativamente pequena. Como resultado tem-se um código mais legível, sem utilização de *loops* e condicionais.

A criação de uma linguagem de consulta para objetos Java similar ao SQL pode vir facilitar o desenvolvimento, principalmente para os programadores já familiarizados em efetuar consultas em bancos de dados. A necessidade de codificar laços e estruturas condicionais será menor, haja vista que a própria linguagem terá a responsabilidade de criar tais estruturas internamente. Assim sendo, a codificação tornar-se-á mais legível e concisa.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um interpretador de consultas para objetos Java que permita ao programador realizar consultas em memória através de uma linguagem pré-definida, baseando-se na extensão LINQ to *Objects*.

Os objetivos específicos do trabalho podem ser descritos da seguinte forma:

a) especificar uma linguagem de consulta semelhante ao SQL;
b) disponibilizar os operadores de projeção, seleção e junção da álgebra relacional, além dos operadores de agrupamento e ordenação;
c) disponibilizar funções de agregação existentes na linguagem SQL (`count`, `sum` `max`, `min`, `avg`).

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado em quatro capítulos, sendo eles: introdução, fundamentação teórica, desenvolvimento da ferramenta e considerações finais.

O capítulo 2 apresenta fundamentos teóricos sobre LINQ, linguagem SQL, álgebra relacional, compiladores e trabalhos correlatos a este.

Os requisitos deste trabalho, bem como detalhes do desenvolvimento e diagramas representados pela *Unified Modeling Language* (UML) são abordados no capítulo 3. Ainda no capítulo 3, um estudo de caso é apresentado seguido de uma breve discussão sobre os resultados obtidos.

Finalmente, no capítulo 4, são apresentadas as conclusões finais segundo os resultados obtidos bem como sugestões de melhorias e implementações futuras.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados aspectos teóricos fundamentais para o entendimento deste trabalho. Os aspectos abordados são LINQ, linguagem SQL, álgebra relacional, compiladores e trabalhos correlatos.

2.1 LINQ

A integração entre algumas fontes de dados e linguagens de programação .NET nem sempre é fácil de ser feita pois, ao contrário das linguagens .NET, os dados podem não ser orientados a objetos. Tentativas de criação de fontes de dados orientadas a objetos que facilitassem esse tipo de integração existiram, porém, mesmo com o passar dos anos, o modelo relacional continua sendo o mais utilizado (MARGUERIE; EICHERT; WOOLEY, 2009, p. 6).

Motivada por este problema, a Microsoft, desenvolveu um recurso conhecido como LINQ para as linguagens C# e Visual Basic.NET. Inicialmente, a intenção da Microsoft era solucionar os problemas relacionados ao mapeamento de objetos com os bancos de dados relacionais. No entanto, o LINQ evoluiu e, segundo Marguerie, Eichert e Wooley (2009, p. 6), hoje pode ser definido como "um conjunto de ferramentas de consulta, de uso geral e integrada à linguagem.". Com este conjunto de ferramentas, o desenvolvedor vê-se livre para consultar dados em memória (LINQ to *Objects*), em bancos de dados (LINQ to SQL), em arquivos XML (LINQ to XML) e em outros tipos de fontes.

Anteriormente à criação do LINQ, o desenvolvedor era obrigado a utilizar linguagens diferentes dentro de uma mesma aplicação. Um exemplo disto é que, se o programador quisesse recuperar informações de um banco de dados, seria necessário o uso da linguagem SQL dentro da aplicação codificada em C# ou Visual Basic. Sendo assim, pode-se citar que um dos principais aspectos do LINQ é que sua sintaxe e seus conceitos são os mesmos para qualquer fonte de dados, ou seja, a necessidade da utilização de diferentes linguagens dentro do mesmo projeto é menor. Além disso, uma vez que o desenvolvedor aprenda a escrever uma consulta com a extensão LINQ to *Objects*, ele saberá codificar nas outras extensões.

Outro aspecto interessante do LINQ é que, por ser um recurso nativo da linguagem, a

verificação de erros no código é feita durante a compilação do programa (MARGUERIE; EICHERT; WOOLEY, 2009, p. 7).

No Quadro 1, pode-se observar um exemplo de consulta em uma coleção, que filtre as pessoas com idade entre 12 e 20 anos, usando a extensão LINQ to *Objects*.

```
//Cria uma lista de pessoas
var pessoas = new List<Pessoa>(){
    new Pessoa {Idade=12,Nome="Carlos"},
    new Pessoa {Idade=18,Nome="Rita"},
    new Pessoa {Idade=16,Nome="Juliana"},
    new Pessoa {Idade=25,Nome="Paulo"},
    new Pessoa {Idade=28,Nome="José"},
    new Pessoa {Idade=18,Nome="Andréa"}
};

//Consulta uma coleção de pessoas adolescentes com LINQ
var pes = from p in pessoas where p.Idade > 12 && p.Idade < 20 select p;
```

Quadro 1 - Código em C# com utilização da extensão LINQ to Objects

2.2 LINGUAGEM SQL

SQL, diferentemente de outras linguagens de programação como Java, C#, Visual Basic e outras mais, não tem seu foco voltado ao desenvolvimento de aplicações. Ao invés disso, a linguagem SQL foi criada para "facilitar o acesso de informações (por meio de consultas, atualizações e manipulações de dados) armazenadas em bancos de dados do tipo relacional." (MANZANO, 2002, p. 16).

Em sua primeira versão, lançada em 1974 pela *International Business Machine* (IBM), a linguagem foi chamada de SEQUEL (uma abreviatura de *Structured English QUEry Language*) e utilizada em um protótipo de banco de dados relacional da IBM. Mais tarde, uma segunda versão denominada SEQUEL/2 é lançada e, finalmente, no final de 1976, é apresentada a linguagem SQL, sendo posteriormente implementada nos sistemas gerenciadores de bancos de dados comerciais DB2 e SQL/DS (MANZANO, 2002, p. 17).

Como a IBM já era um nome muito forte no meio computacional, a linguagem SQL conseguiu ser aceita por outros fabricantes, tornando-se padrão para acesso de bancos de dados relacionais. Porém, em conjunto com a boa aceitação começaram a surgir os primeiros problemas, tal como a falta de padronização, pois cada empresa incorporava novos comandos à linguagem original. A partir daí, o *American National Standards Institute* (ANSI) passou a interferir diretamente nos padrões do SQL, estabelecendo normas e critérios. Sendo assim, a

linguagem passa a ser chamada de ANSI/SQL (MANZANO, 2002, p. 18).

Segundo os padrões ANSI, a linguagem SQL é composta por dois grupos distintos de instruções. O primeiro deles é o *Data Definition Language* (DDL) e suas instruções são usadas para manipular a estrutura do banco de dados, ou seja, criar, alterar ou excluir tabelas e índices. As principais instruções do grupo DDL são: `create table`, `drop table`, `alter table`, `create index` e `drop index`. Já no segundo grupo, chamado de *Data Manipulation Language* (DML), as instruções servem para manipular os dados propriamente ditos, selecionando, incluindo, alterando e excluindo-os do banco de dados, respectivamente, com as instruções `select`, `insert`, `update` e `delete`. Além das instruções dos grupos DDL e DML, ainda existem cláusulas que permitem a aplicação de condições a uma consulta, sendo elas: `from`, `where`, `group by`, `having` e `order by` (MANZANO, 2002, p. 22).

No Quadro 2, é exemplificado o código de uma consulta em SQL onde são selecionados clientes com idade entre 12 e 20 anos na tabela `cliente` de uma determinada base de dados.

```
select codigo,
       nome
from   cliente
where  idade > 12 and idade < 20;
```

Quadro 2 - Exemplo de consulta usando a linguagem SQL

2.3 ÁLGEBRA RELACIONAL

A álgebra relacional é uma linguagem teórica com operadores que trabalham sobre uma ou mais relações a fim de definir uma relação resultante, porém, sem modificar a relação original (SUMATHI; ESAKKIRAJAN, 2007, p. 72). Uma propriedade fundamental é que todo operador tem como entrada uma ou duas relações e retorna uma relação resultante (RAMAKRISHNAN; GEHRKE, 1999, p. 92). Desta maneira, expressões podem ser aninhadas, servindo uma de entrada para outra.

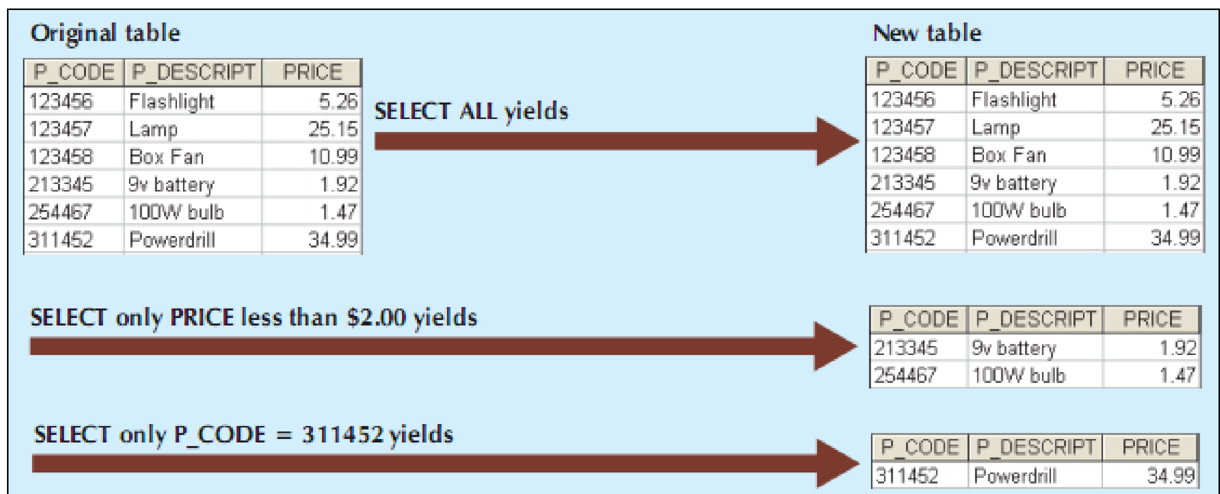
Os operadores da álgebra podem ser classificados como unários, que são aqueles que executam operações em uma só relação, como por exemplo a seleção e a projeção, e binários, sendo estes os que operam sobre duas relações, como os operadores de produto cartesiano, junção, união, intersecção e também o de diferença.

Nas subseções a seguir são apresentados alguns dos operadores mais utilizados.

2.3.1 Seleção

O operador de seleção, também chamado de restrição, tem como função selecionar linhas de uma relação de acordo com uma condição. A representação deste operador geralmente é feita através da letra grega sigma (σ) seguida da condição. A relação sobre qual a seleção irá operar é identificada entre parênteses, após a condição. Por exemplo, para selecionar de uma relação denominada clientes somente as linhas cuja coluna idade seja maior que 18, deve-se usar expressão $\sigma_{\texttt{idade>18}}(\texttt{clientes})$.

Na Figura 1 é possível ver como o operador de seleção age sobre uma determinada relação. Ao lado esquerdo, é mostrada a relação com todas as suas linhas. Ao lado direito, são exibidas três relações obtidas através do uso da seleção.



Original table / New table (SELECT ALL yields):

P_CODE	P_DESCRIPT	PRICE
123456	Flashlight	5.26
123457	Lamp	25.15
123458	Box Fan	10.99
213345	9v battery	1.92
254467	100W bulb	1.47
311452	Powerdrill	34.99

SELECT only PRICE less than $2.00 yields:

P_CODE	P_DESCRIPT	PRICE
213345	9v battery	1.92
254467	100W bulb	1.47

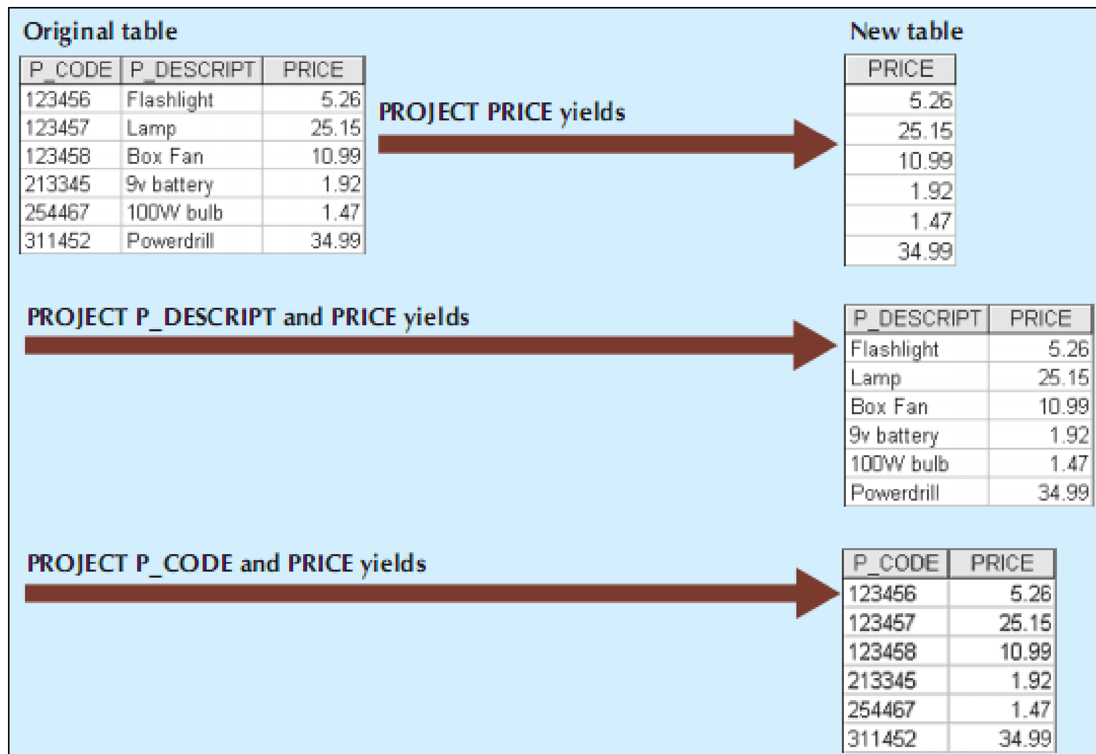
SELECT only P_CODE = 311452 yields:

P_CODE	P_DESCRIPT	PRICE
311452	Powerdrill	34.99

Figura 1 – Exemplo de seleção

2.3.2 Projeção

Representado pela letra grega pi (π), o operador de projeção é usado para extrair colunas de uma relação. Desta maneira, pode-se concluir que a relação resultante da expressão $\pi_{\texttt{nome,idade}}(\texttt{clientes})$ terá somente duas colunas: nome e idade. Na Figura 2, pode-se observar o resultado de uma projeção aplicada a uma determinada relação, onde ao lado esquerdo é mostrada a relação com todas as suas colunas e ao lado direito as projeções resultantes.



P_CODE	P_DESCRIPT	PRICE
123456	Flashlight	5.26
123457	Lamp	25.15
123458	Box Fan	10.99
213345	9v battery	1.92
254467	100W bulb	1.47
311452	Powerdrill	34.99

PRICE
5.26
25.15
10.99
1.92
1.47
34.99

P_DESCRIPT	PRICE
Flashlight	5.26
Lamp	25.15
Box Fan	10.99
9v battery	1.92
100W bulb	1.47
Powerdrill	34.99

P_CODE	PRICE
123456	5.26
123457	25.15
123458	10.99
213345	1.92
254467	1.47
311452	34.99

Fonte: Coronel, Morris e Rob (2010, p. 69).

Figura 2 - Exemplo de funcionamento do operador projeção

2.3.3 Produto cartesiano

O operador de produto cartesiano, que é representado pelo símbolo ×, define uma relação que é uma concatenação de todas as linhas de uma relação R com todas as linhas de uma relação S (SUMATHI; ESAKKIRAJAN, 2007, p. 82). O resultado do produto cartesiano possui todos os atributos de R e S. Neste caso, o número de linhas da relação resultante será a quantidade de linhas na relação R multiplicado pela quantidade de linhas na relação S.

Como exemplo, na Figura 3 é possível ver o resultado de um produto cartesiano efetuado entre as relações A, que possui as colunas A1 e A2, e B, que tem as colunas B1, B2 e B3. Nota-se que a relação resultante da expressão A × B possui todos as linhas e colunas de A e B combinadas.

A_1	A_2
a	b
c	d
e	f

B_1	B_2	B_3
g	h	i
j	k	l

$S.A_1$	$S.A_2$	$T.B_1$	$T.B_2$	$T.B_3$
a	b	g	h	i
a	b	j	k	l
c	d	g	h	i
c	d	j	k	l
e	f	g	h	i
e	f	j	k	l

Fonte: Roman (2002, p. 74).

Figura 3 - Exemplo de produto cartesiano entre duas relações

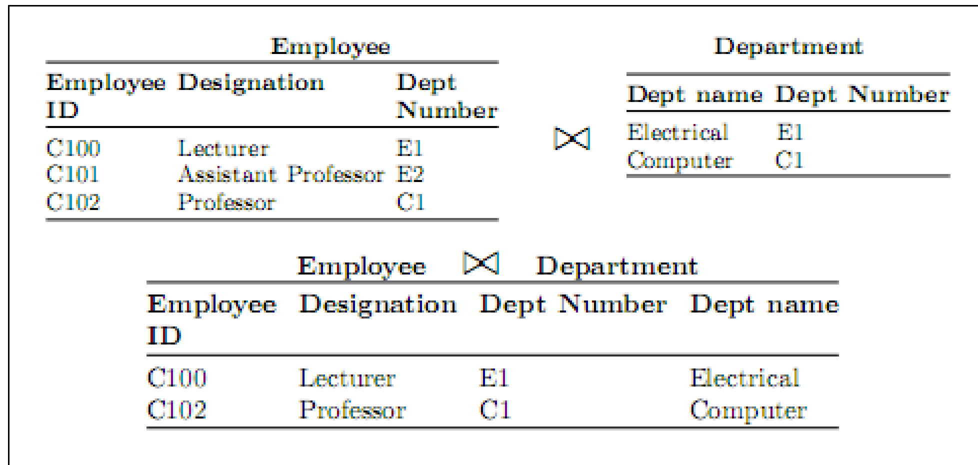
2.3.4 Junção

A junção é uma das operações mais utilizadas na álgebra relacional e serve para combinar informações de duas ou mais relações. Embora uma junção possa ser definida como um produto cartesiano seguido de seleções e projeções, estas acabam sendo usadas com maior frequência, uma vez que os produtos cartesianos geram, em grande parte dos casos, resultados maiores que as junções (RAMAKRISHNAN; GEHRKE, 1999, p. 97).

Existem diferentes formas de junção, sendo a junção natural a mais simples. A junção natural, representada pelo símbolo $\bowtie$, baseia-se na igualdade dos atributos comuns entre duas relações `R` e `S` para realizar a seleção e a projeção. Na maioria dos sistemas, a junção natural requer que os atributos que serão comparados tenham o mesmo nome em cada uma das relações envolvidas na operação. Porém, mesmo que os atributos tenham denominações diferentes é possível efetuar a junção natural, desde que sejam do mesmo tipo (SUMATHI; ESAKKIRAJAN, 2007, p. 83).

Outro tipo de junção é a *theta*. Neste tipo de junção, existe a possibilidade do uso de outros operadores relacionais além do operador de igualdade, como >, <, >= e <=. A junção *theta*, diferentemente da natural, não envolve a projeção de atributos.

Na Figura 4, é possível observar o resultado obtido através da junção natural de duas relações, `employee` e `department`. No exemplo, a junção natural faz a combinação das relações com base na coluna `Dept Number`.



Employee

Employee ID	Designation	Dept Number
C100	Lecturer	E1
C101	Assistant Professor	E2
C102	Professor	C1

⋈

Department

Dept name	Dept Number
Electrical	E1
Computer	C1

Employee ⋈ Department

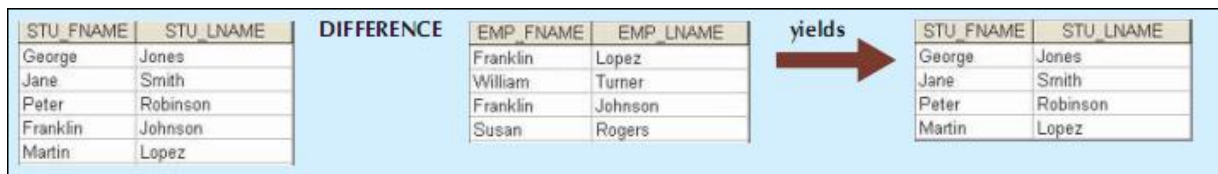
Employee ID	Designation	Dept Number	Dept name
C100	Lecturer	E1	Electrical
C102	Professor	C1	Computer

Fonte: Sumathi e Esakkirajan (2007, p. 84).

Figura 4 – Exemplo de junção natural

2.3.5 Diferença

A operação de diferença consiste em definir uma relação das linhas que estão em uma relação `R`, mas não estão em `S` (SUMATHI; ESAKKIRAJAN, 2007, p. 79). A representação deste operador é feita através da expressão `R-S`, onde `R` e `S` são duas relações de entrada. A Figura 5 exemplifica a operação de diferença entre duas tabelas de um banco de dados.



STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Franklin	Johnson
Martin	Lopez

DIFFERENCE

EMP_FNAME	EMP_LNAME
Franklin	Lopez
William	Turner
Franklin	Johnson
Susan	Rogers

yields

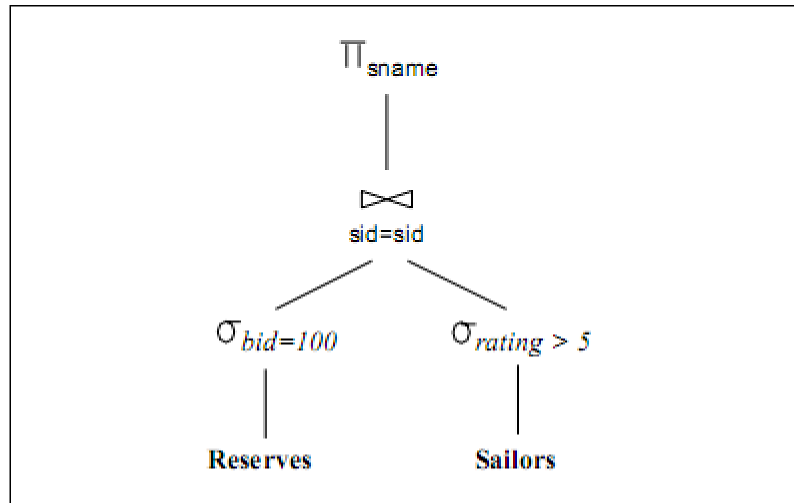
STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Martin	Lopez

Fonte: Coronel, Morris e Rob (2010, p. 70).

Figura 5 - Exemplo de operação de diferença entre duas tabelas de um banco de dados

2.3.6 Combinação dos operadores

É possível combinar os diferentes operadores da álgebra relacional em uma expressão aplicando um operador ao resultado de um ou mais operadores. Tal combinação pode ser representada através de uma árvore de expressões, onde as folhas são os nomes das relações e cada nó interno é identificado por um operador que faz sentido quando aplicado à(s) relação(ões) representada(s) por seu filho ou filhos (GARCIA-MOLINA; ULLMAN; WIDOM, 2001, p. 269). Na Figura 6 é exibido um exemplo de uma árvore de expressões.



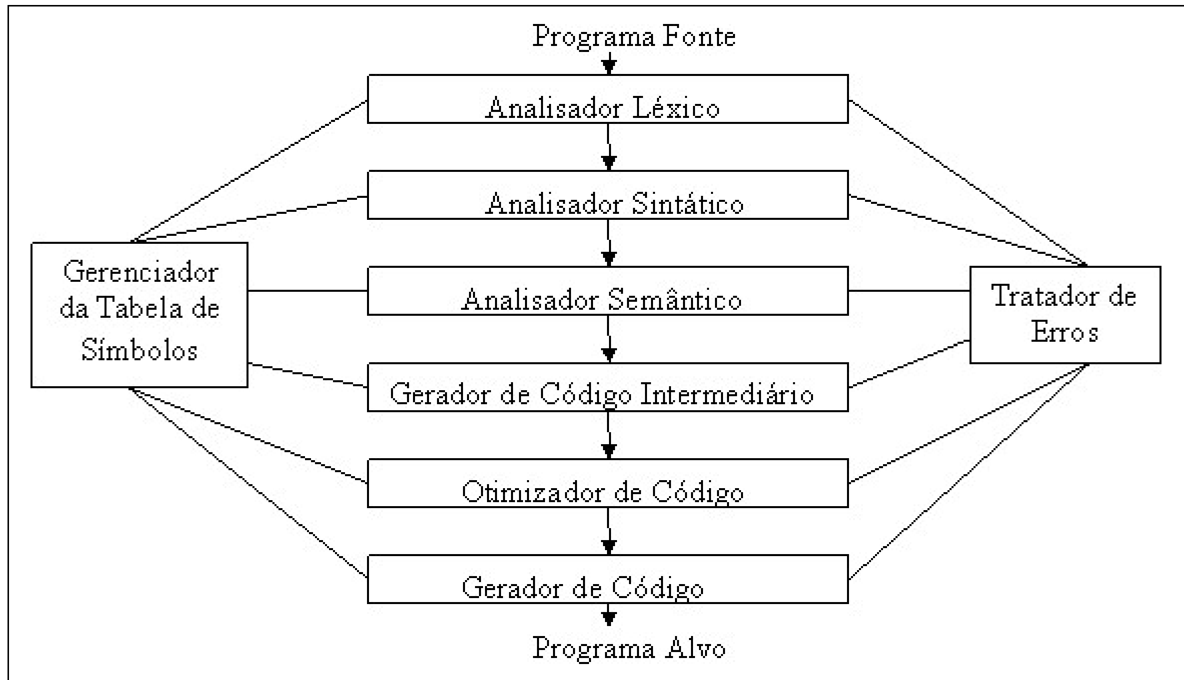
Fonte: Ramakrishnan e Gehrke (1999, p. 396).

Figura 6 – Árvore de operadores da álgebra relacional

2.4 COMPILADORES

Um compilador pode ser definido como "um programa que lê um programa escrito numa linguagem – a linguagem fonte – e o traduz num programa equivalente numa outra linguagem – a linguagem alvo" (AHO; SETHI; ULLMAN, 1995, p. 1).

Em geral, os compiladores são programas muito complexos e dividem-se em várias fases de interpretação. No entanto, possuem certa padronização, já que são compostos por algumas fases de análise pré-definidas que, apesar de serem construídas separadamente, acabam trabalhando em conjunto, uma vez que cada fase processa o resultado obtido na fase anterior. A estrutura mais básica de um compilador pode ser vista na Figura 7.



Fonte: Martinotto (2009).

Figura 7 – Estrutura básica de um compilador

Nas subseções a seguir são apresentadas três tipos de análises, respectivamente conhecidas como léxica, sintática e semântica.

2.4.1 Análise léxica

O analisador léxico é a primeira fase de um compilador. Sua tarefa é ler os caracteres de entrada a fim de encontrar os *tokens* que serão utilizados pelo *parser* para que seja feita a análise sintática (AHO; SETHI; ULLMAN, 1995, p. 38). Os *tokens* são unidades léxicas que são representadas por cada símbolo especificado na gramática da linguagem contido dentro do texto fonte. Deste modo, o analisador léxico lê o texto fonte caractere por caractere identificando os *tokens* e colocando-os em uma estrutura chamada de lista de *tokens*. Uma vez terminada a leitura do texto, o analisador léxico retorna a lista de *tokens*, que é usada como entrada na próxima fase, chamada de análise sintática. No Quadro 3, é apresentado um exemplo de uma lista de *tokens* gerada a partir do texto fonte `soma := a + b`.

identificador	soma
símbolo de atribuição	:=
identificador	a
símbolo de adição	+
identificador	b

Quadro 3 – Exemplo de uma lista de *tokens*

No nível léxico poucos erros podem ser identificados, uma vez que este analisador

possui uma visão muito local do texto fonte (AHO; SETHI; ULLMAN, 1995, p. 40). Em alguns casos, o analisador léxico reconhece uma entrada como um *token* válido, porém, ao ser analisado sintaticamente, este *token* pode não estar de acordo com as regras gramaticais da linguagem. O *token* `fi` quando encontrada na sequência `fi(1 == 1){return;}` em um código Java, por exemplo, pode ser um identificador ou um comando `if` escrito incorretamente. Neste caso, somente o analisador sintático é capaz de identificar se existe ou não algum erro.

2.4.2 Análise sintática

Na fase de análise sintática, o objetivo é verificar se a estrutura gramatical do texto fonte está correta segundo as regras definidas na gramática da linguagem (PRICE; TOSCANI, 2001, p. 8). As regras gramaticais de uma linguagem geralmente são descritas através da notação *Backus-Naur Form* (BNF) ou extensões da mesma, como a *Extended Backus-Naur Form* (EBNF). No Quadro 4 é mostrado um exemplo de construção com a notação BNF.

```
<programa> ::= program id <declaracao> begin <listaComandoAux> end id ;
<declaracao> ::= î | <declaracao1> ;
<declaracao1> ::= <declaracaoAux> <declaracao>;
<declaracaoAux> ::= <tipo> ":" <listaIdentificadores>;
```

Quadro 4 - Exemplo de construção BNF

O analisador sintático identifica estruturas como expressões e comandos, percorrendo toda a lista de *tokens* gerada na fase de análise léxica. Além disso, assim como o analisador léxico, o sintático também identifica erros.

Para armazenar as estruturas sintáticas identificadas, o analisador constrói uma árvore de derivação (ou árvore sintática), que é gerada a partir das regras gramaticais especificadas na linguagem. A construção da árvore de derivação pode ser feita de várias formas, no entanto, dois métodos são mais utilizados, sendo eles chamados de *top-down* e *bottom-up*. No método *top-down*, a árvore é construída da raiz para as folhas, enquanto que no *bottom-up* a construção inicia-se nas folhas e termina na raiz (AHO; SETHI; ULLMAN, 1995, p. 72).

Na Figura 8, é possível ver um exemplo de uma árvore sintática gerada para a expressão matemática `soma := a + b`.

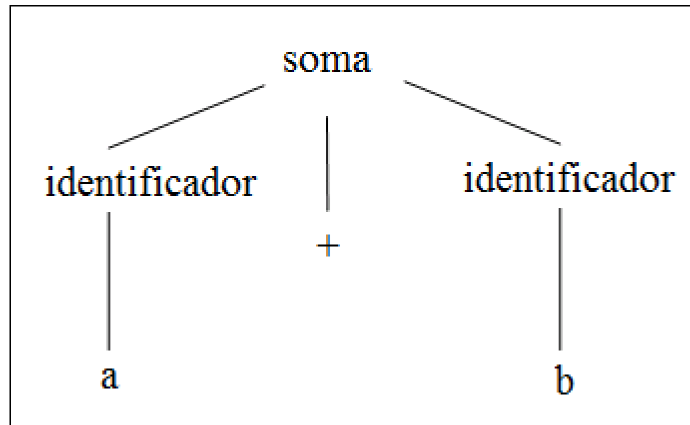


Figura 8 – Árvore sintática de uma expressão matemática

2.4.3 Análise semântica

De acordo com Martinotto (2009), “semântica de uma linguagem é a interpretação que se pode atribuir ao conjunto de todas as suas sentenças.”. Desta forma, pode-se dizer que a principal função do analisador semântico é interpretar o texto fonte, baseando-se nas informações recebidas dos outros analisadores. Outro papel importante do analisador semântico é a verificação de tipos, checando se cada operador recebe os operandos que são permitidos pelas definições da linguagem (AHO; SETHI; ULLMAN, 1995, p. 72). Como exemplo, o compilador deve acusar erro caso um valor literal tente ser atribuído a uma variável numérica.

2.5 TRABALHOS CORRELATOS

A seguir são apresentados algumas bibliotecas para a linguagem Java que facilitam a realização de consultas em memória. Dentre as ferramentas analisadas, as selecionadas foram: Lambdaj (FUSCO, 2009) e Coollection (ANDRADE, 2010).

2.5.1 Lambdaj

Lambdaj é uma biblioteca para a linguagem Java que tem como objetivo eliminar a necessidade da utilização de *loops* ao interagir com coleções (FUSCO, 2009) e consequentemente reduzir a quantidade de código dentro de um programa. Usando a ferramenta, o desenvolvedor tem a possibilidade de:

a) filtrar itens dada uma condição;
b) selecionar determinados atributos dos itens;
c) atribuir valores as propriedades de cada item;
d) ordenar os itens através de seus atributos;
e) agrupar itens;
f) somar valores de um determinado atributo numérico dos itens;
g) concatenar atributos dos itens.

No Quadro 5, observa-se a ordenação de uma lista de pessoas usando a *Application Program Interface* (API) *Collections* do Java e posteriormente a biblioteca Lambdaj. Nota-se que a pequena quantidade de código utilizada na ordenação com a Lambdaj, torna a sintaxe mais fácil de ser entendida.

```
//Ordenação de uma coleção usando a API Collections
Collections.sort(listaPessoas, new Comparator<Pessoa>() {
      @Override
      public int compare(Pessoa p1, Pessoa p2) {
            return p1.getNome().compareToIgnoreCase(p2.getNome());
      }
});

//Ordenação de uma coleção usando a biblioteca Lambdaj
Object argumento = Lambda.on(Pessoa.class).getNome();
listaPessoas = Lambda.sort(listaPessoas, argumento);
```

Quadro 5 - Comparação entre o uso da API *Collections* e a biblioteca Lambdaj

2.5.2 Coollection

Iterar sobre coleções nem sempre é considerado a melhor maneira de filtrar, mapear ou ordenar dados (ANDRADE, 2010). A biblioteca Coollection, assim como a Lambdaj, tem como propósito oferecer uma série de métodos que auxiliam o programador a executar consultas e realizar ordenações e agrupamentos. Utilizando as funções da Coollection para filtrar uma coleção, como apresentado no Quadro 6, pode-se observar que o código torna-se

mais claro se comparado à forma tradicional de filtragem usando *loops*.

```
//Filtra em uma lista de pessoas somente as que possuem nome "Carlos"
List<Pessoa> p = new ArrayList<Pessoa>();
for(Pessoa pes : listaPessoas){
      if(pes.getNome().equals("Carlos")){
            p.add(pes);
      }
}

//Filtra em uma lista de pessoas somente as que possuem nome "Carlos"
//usando a biblioteca Coollection
List<Pessoa> p = from(pessoas).where("getNome", eq("Carlos")).all();
```

Quadro 6 - Exemplo de código com utilização de funções da biblioteca Coollection

3 DESENVOLVIMENTO DA FERRAMENTA

Nas seções a seguir são mostrados detalhes e técnicas relacionadas ao desenvolvimento da ferramenta.

Primeiramente, são descritos os requisitos funcionais e não funcionais da ferramenta.

Em seguida, é apresentada a especificação do interpretador, ou seja, dos analisadores léxico, sintático e semântico.

Após a especificação dos analisadores, são mostrados os algoritmos de recuperação dos dados.

Por fim, é feito um estudo de caso e uma discussão sobre os resultados obtidos com a ferramenta desenvolvida.

3.1 REQUISITOS DA FERRAMENTA

O interpretador de consultas para objetos Java deve:

a) permitir a realização de consultas em objetos que implementem a interface `Collection` ou vetores (Requisito Funcional – RF);
b) permitir ao desenvolvedor informar uma sequência de comandos de consulta (RF);
c) interpretar a sequência de comandos informada pelo usuário (RF);
d) exibir mensagens de erro caso comandos inválidos sejam encontrados (RF);
e) retornar uma coleção com o resultado da consulta efetuada (RF);
f) ter uma linguagem de consulta similar ao SQL (Requisito Não Funcional - RNF);
g) ser desenvolvido em Java (RNF).

3.2 ESPECIFICAÇÃO DO INTERPRETADOR

Nesta seção são detalhadas as especificações dos analisadores léxico e sintático, bem como das classes que executam a consulta propriamente dita.

3.2.1 Ferramentas para geração de compiladores

> Atualmente existem várias ferramentas que automatizam a construção de um compilador, ou pelo menos parte da construção. Os analisadores léxicos e sintáticos geralmente são gerados com ferramentas chamadas geradores de *parser* ou *compiler-compiler*. Estas ferramentas normalmente têm como entrada as expressões regulares que definem os *tokens* da linguagem para gerar o analisador léxico, e a BNF da linguagem para gerar o analisador sintático. Estas entradas permitem a geração das partes de um compilador de modo muito preciso e eficiente, e possui a vantagem de facilitar a manutenção e o entendimento da sintaxe da linguagem. Existem ainda geradores de analisadores semânticos, porém não são tão eficientes. (GESSER, 2007, p. 32).

Neste trabalho foi utilizado um dos geradores de *parser* mais conhecidos, o Java Compiler Compiler (JavaCC). O JavaCC é uma ferramenta que gera código puramente Java baseando-se em uma gramática especificada no formato EBNF.

De acordo com Tavares (2000), o JavaCC aproveita todas as características da linguagem Java para prover facilidades na construção de analisadores sintáticos, facilitando a geração e adaptação dos códigos que avaliam os nós da árvore sintática. Também é possível, através do uso de recursos do Java, gerenciar exceções no analisador sintático.

Juntamente com o pacote JavaCC pode-se encontrar as ferramentas *jjdoc* e *jjtree*. O *jjdoc* é responsável por gerar uma documentação em formato HTML da especificação da linguagem. O *jjtree* é um utilitário para criação e manipulação de árvores sintáticas.

3.2.2 Analisadores léxico e sintático

Por ser uma ferramenta desenvolvida para a linguagem Java, a especificação dos analisadores léxico e sintático do interpretador foi feita através do JavaCC, visando maior facilidade na definição dos *tokens* e na criação das regras sintáticas, uma vez que com o JavaCC é possível definir tais regras utilizando-se de código Java.

A gramática foi especificada para que a linguagem seja semelhante ao SQL, com algumas pequenas diferenças e pode ser visualizada no Apêndice A. Sendo assim, as cláusulas reconhecidas pelo interpretador são:

a) `select`: cláusula de declaração das projeções, ou seja, os atributos de cada objeto que devem fazer parte do resultado final da consulta;
b) `from`: cláusula de declaração das coleções ou vetores de consulta;
c) `where`: cláusula de declaração das restrições;

d) `group by`: cláusula de declaração dos atributos de agrupamento;

e) `order by`: cláusula de declaração dos atributos de ordenação. Os atributos de ordenação podem ser sucedidos pelos comandos `asc` ou `desc`, que definem, respectivamente, se a ordenação será de feita maneira crescente ou decrescente;

f) `range`: cláusula que define um intervalo `N-M` para a recuperação dos registros, sendo `N` o início e `M` o fim do intervalo.

Uma das diferenças da linguagem do interpretador com relação ao SQL é a não obrigatoriedade da cláusula `select`, como na *Hibernate Query Language* (HQL) do *framework* de acesso a banco de dados para Java, o Hibernate. Portanto, a consulta mais básica que pode ser reconhecida pelo interpretador possui somente a cláusula `from`.

Além de verificar se a consulta escrita pelo usuário está de acordo com as regras gramaticais, os analisadores têm como função gerar a entrada necessária para o analisador semântico funcionar corretamente, criando instâncias de cada um dos operadores algébricos disponibilizados pela linguagem.

As classes geradas pelo JavaCC são representadas no diagrama de classes mostrado na Figura 9. No diagrama é possível observar somente os nomes das classes e suas associações, uma vez que os nomes dos métodos e atributos de cada classe são considerados triviais neste caso, mas podem ser consultados no Apêndice B.

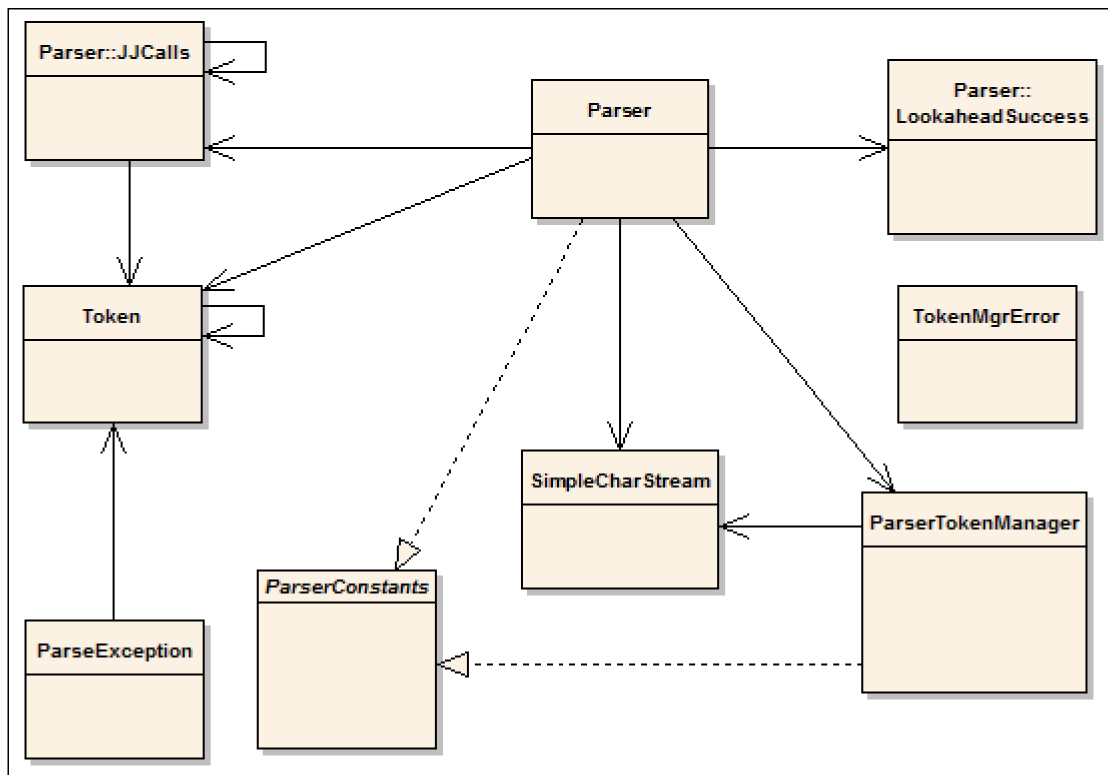


Figura 9 – Diagrama de classes dos analisadores léxico e sintático

As classes que merecem maior destaque no diagrama apresentado na Figura 9 são:

a) `ParserTokenManager`: classe responsável pelo reconhecimento dos *tokens*;
b) `Parser`: classe de implementação do analisador sintático. Tem a função de analisar a sequência de *tokens* e verificar se estão de acordo com as regras sintáticas especificadas na linguagem;
c) `TokenMgrError`: classe de tratamento de erros léxicos encontrados no programa fonte;
d) `ParseException`: classe responsável por emitir as mensagens de erro na análise sintática.

3.2.3 Analisador semântico

O analisador semântico é o analisador que tem o papel de executar a consulta, retornando uma lista como resultado final. As principais funções do analisador semântico são:

a) verificar a existência das relações da cláusula `from`;
b) verificar se os atributos informados nos operandos de cada umas das cláusulas existem nas relações;
c) verificar compatibilidade de tipos de dados das restrições;
d) montar a árvore de consulta;
e) executar a consulta.

A primeira etapa do analisador semântico é a montagem da árvore de consulta, cuja responsabilidade fica a cargo da classe `PlanoExecucao`. Os nós da árvore de consulta são representados pelos operadores da álgebra relacional que ficam armazenados nas listas de operadores produzidas pelo analisador sintático. Juntamente com o processo de montagem da árvore de consulta também é feita a verificação da existência das relações informadas na cláusula `from`. Devido ao fato de que as relações podem armazenar instâncias de diferentes classes, a existência dos nomes dos atributos nos operandos e a compatibilidade dos tipos de dados são verificadas somente durante a execução da consulta, que é a segunda etapa da análise semântica. Na Figura 10 é exibido o diagrama de classes do analisador semântico, o qual pode ser conhecido em detalhes no Apêndice C.

«static,enum... ItemOrdenacao:: TipoOrdenacao

ItemOrdenacao

Comparator ResultListComparator

HashMap ResultObject

ArrayList ResultList

Ordenacao

ProjecaoCampo

Agrupamento

«interface» IPossuiRestricoes

ProjecaoString

Projecao

ArrayList ListaProjecoes

Query

RestricaoConjunto

ProjecaoBooleana

Relacao

Comparable Restricao

ProjecaoDate

ProjecaoFuncaoAgregacao

PlanoExecucao

RestricaoSimples

Sum

Max

QueryImpl

ArvoreConsulta

NoArvore

FuncaoAgregacao

Avg

Min

Count

Figura 10 – Diagrama de classes do analisador semântico

No Quadro 7 são detalhadas as classes mostradas no diagrama da Figura 10.

Classe	Descrição
Agrupamento	Classe de representação do operador de agrupamento da álgebra relacional. Contém uma lista dos campos informados na cláusula group by.
ArvoreConsulta	Classe que representa a árvore de consulta. Armazena todos os operadores da álgebra relacional que serão utilizados no momento da execução da consulta.
Avg	Classe que representa a função de agregação avg.
Count	Classe que representa a função de agregação count.
FuncaoAgregacao	Classe usada para representar uma função de agregação.
IPossuiRestricoes	Interface que define quais classes armazenam uma lista restrições.
ItemOrdenacao	Classe que representa cada campo informado na cláusula order by. Esta classe guarda o nome do campo e também o tipo de ordenação (ascendente ou descendente).
ListaProjecoes	Lista que armazena todas as projeções declaradas na cláusula select. Esta é uma subclasse de ArrayList.
Max	Classe que representa a função de agregação max.
Min	Classe que representa a função de agregação min.
NoArvore	Classe que representa um nó da árvore de consulta. O operador algébrico relacionado ao nó é armazenado no atributo operacao.
Ordenacao	Classe que representa o operador de ordenação. Armazena uma lista de itens de ordenação, representados pela classe ItemOrdenacao.
PlanoExecucao	Classe responsável por montar e otimizar a árvore de consulta.
Projecao	Representação do operador de projeção.
ProjecaoBooleana	Classe de projeção para valores os lógicos true ou false
ProjecaoCampo	Projeção que faz referência a um campo de um objeto.
ProjecaoDate	Projeção do tipo date, para que seja possível projetar datas.
ProjecaoFuncaoAgregacao	Projeção que representa uma função de agregação na cláusula select.
ProjecaoString	Projeção do tipo texto, para que seja possível projetar textos.
Query	Classe que armazena os operadores referentes as cláusulas (select, from, where, group by e order by) da consulta. Possui uma instância de PlanoExecucao, que será utilizada para montar a árvore de consulta. Também armazena uma referência de QueryImpl, classe que implementa os algoritmos de busca. Recebe como parâmetro o objeto que armazena as coleções a serem consultadas.
QueryImpl	Classe de implementação dos algoritmos de busca dos dados. Efetua a consulta propriamente dita com base na árvore construída pelo PlanoExecucao.
Relacao	Classe que representa uma relação.
Restricao	Classe que representa uma restrição. Armazena os operadores lógicos and/or e not.
RestricaoConjunto	Classe que representa um conjunto de restrições escrita entre parênteses.
RestricaoSimples	Classe que representa uma restrição simples, fora de parênteses.
ResultList	Lista que armazena os resultados intermediários da consulta.
ResultSet	HashSet que armazena os resultados intermediários da consulta.

`ResultListComparator`	Classe que implementa a interface `Comparator` e é responsável por ordenar do resultado da consulta.
`ResultObject`	Classe que representa cada objeto da `ResultList`. Também representa cada objeto retornado no resultado final da consulta.
`Sum`	Classe que representa a função de agregação sum.
`TipoOrdenacao`	Classe que indica se uma ordenação é feita de modo crescente ou decrescente.

Quadro 7 – Descrição das classes da primeira etapa do analisador

3.2.4 Diagrama de atividades

Na Figura 11 é apresentado o diagrama de atividades do interpretador, com a sequência de análises do compilador até a execução da consulta, retornando uma lista.

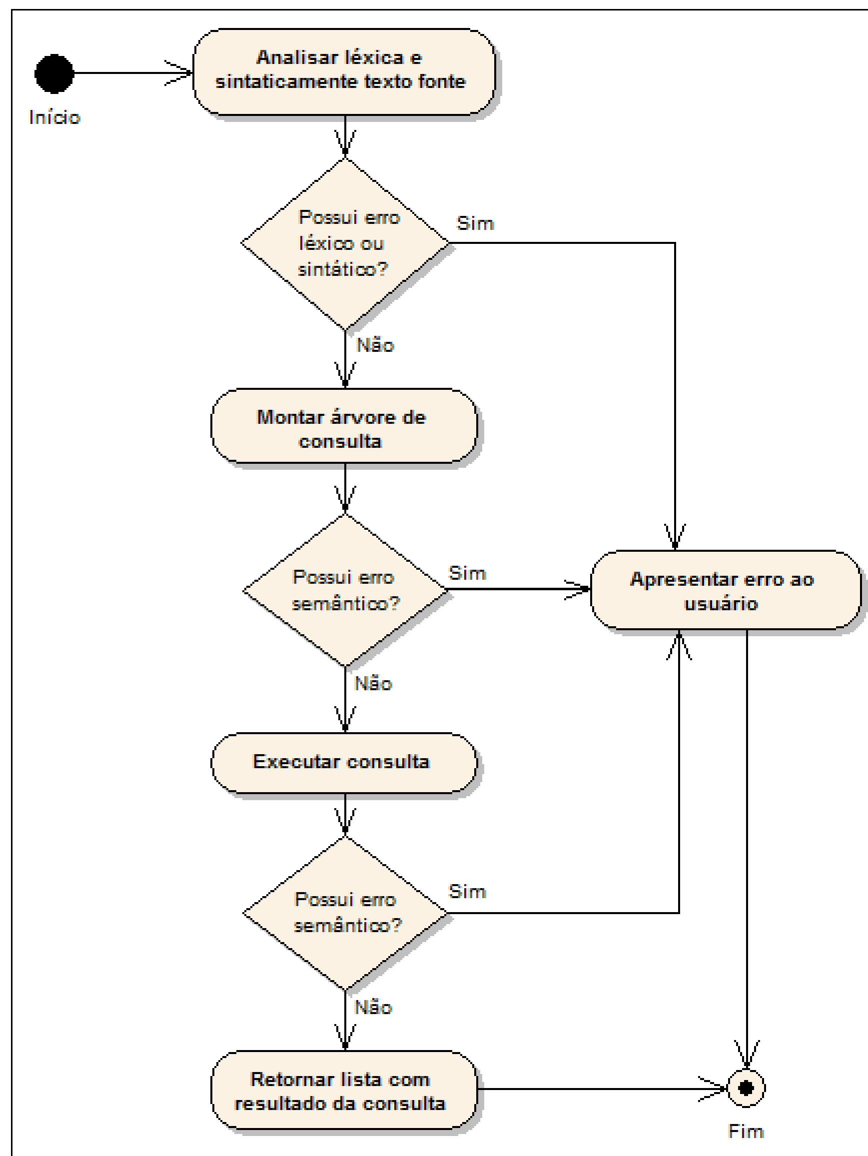


Figura 11 – Diagrama de atividades do interpretador

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentados detalhes da implementação da análise semântica. Além geração da árvore e da execução da consulta, o analisador semântico também tem a responsabilidade de tratar os erros semânticos no código a ser interpretado.

3.3.1 Geração da árvore de consulta

Para representar uma consulta internamente, o interpretador utiliza uma estrutura de árvore. A montagem da árvore de consulta é iniciada juntamente com a análise sintática, quando as projeções, as ordenações e os agrupamentos, caso existam, são inseridos a partir da raiz. Em seguida, utilizando as listas de operadores produzidas pelo analisador sintático, o plano de execução monta o resto da árvore, inserindo e organizando as restrições. A organização das restrições na árvore é muito importante para que a execução da consulta tenha uma boa performance e consequentemente um tempo de resposta reduzido.

As etapas de montagem de árvore podem ser observadas no diagrama mostrado na Figura 12.

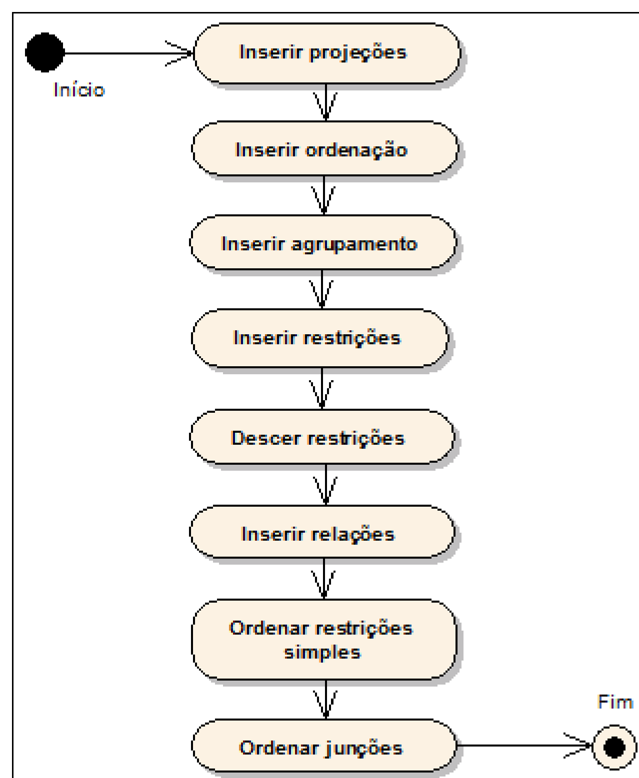


Figura 12 – Diagrama da montagem da árvore de consulta

O primeiro passo da montagem da árvore é a inserção das projeções. Como mostrado na Figura 13, os operadores de projeção são inseridos no início da árvore, pois são os últimos a serem executados.

Figura 13 – Inserção do operador de projeção

Após as projeções, o operador utilizado para ordenar o resultado final da consulta é inserido na árvore, como mostra a Figura 14. O operador de ordenação armazena o nome do campo e o tipo de ordenação a ser realizada, que pode ser ascendente ou descendente.

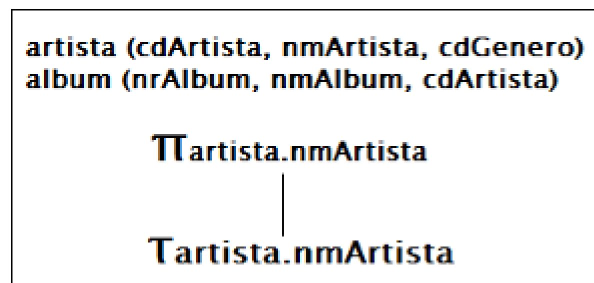


Figura 14 – Inserção do operador de ordenação

A próxima etapa na construção da árvore de consulta é inserção do agrupamento, que guarda uma lista de campos que serão utilizados para agrupar a consulta e também armazena as referências das funções de agregação. Na Figura 15 pode-se ver a inserção do operador de agrupamento.

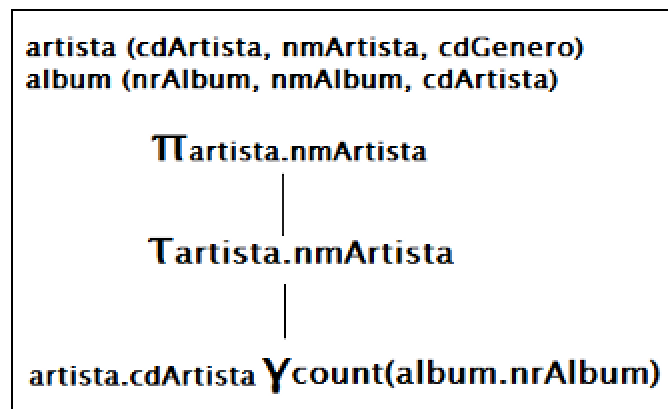


Figura 15 – Inserção do operador de agrupamento

Depois de inserir as projeções, a ordenação e o agrupamento, o plano de execução inicia o processo de inserção e organização das restrições. Tendo como raiz o último operador inserido na árvore, as restrições conjuntivas são inseridas uma abaixo da outra, na ordem em que foram escritas pelo usuário. Já as restrições disjuntivas, que usam o operador lógico `or`, são inseridas na própria raiz, para que posteriormente seja feita a união entre elas. Para o caso

de um conjunto de restrições escritas entre parênteses, é criada uma subárvore onde as restrições são inseridas. A Figura 16 mostra as restrições inseridas na parte mais baixa da árvore.

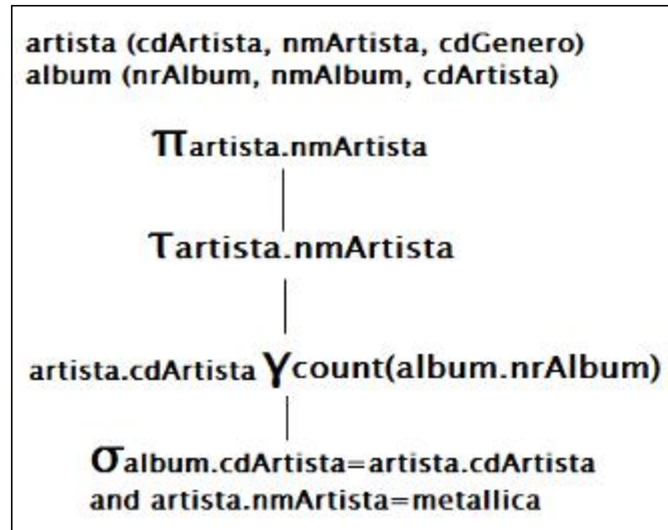


Figura 16 – Inserção das restrições

Para diminuir o tamanho das relações de entrada para as operações de junção, as restrições simples (que não efetuam junção) são colocadas na parte mais baixa da árvore (folhas) para que sejam executadas primeiro. Como mostra a Figura 17, a restrição simples `artista.nmArtista=metallica` é colocada mais abaixo da junção `album.cdArtista=artista.cdArtista`.

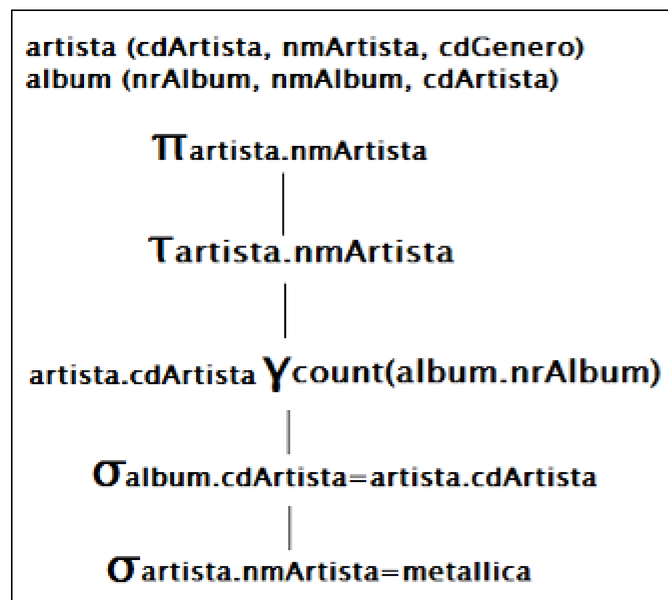


Figura 17 – Descida das restrições simples

Prosseguindo com a construção da árvore, as relações são inseridas em cada nó folha, transformando estes nós em produtos cartesianos entre todas relações escritas na cláusula `from`. As relações são instâncias de coleções recuperadas através da API *Reflection*. A Figura

18 mostra o produto cartesiano `album × artista` inserido ao final da árvore.

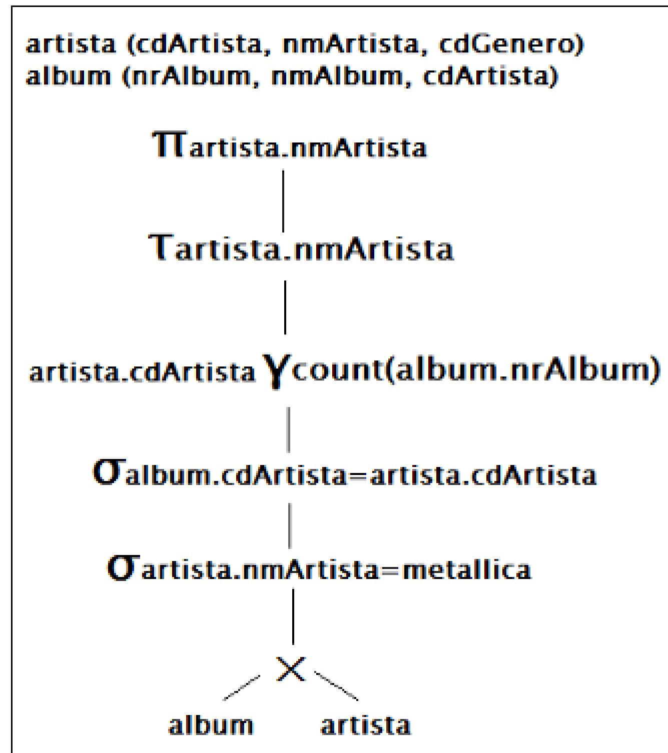


Figura 18 – Inserção das relações

Visando reduzir o tamanho das entradas para os produtos cartesianos criados durante a inserção das relações, as restrições simples devem ser organizadas de maneira que se relacionem com suas relações, como visto na Figura 19, onde a restrição `artista.nmArtista=metallica` é diretamente relacionada a relação `artista`.

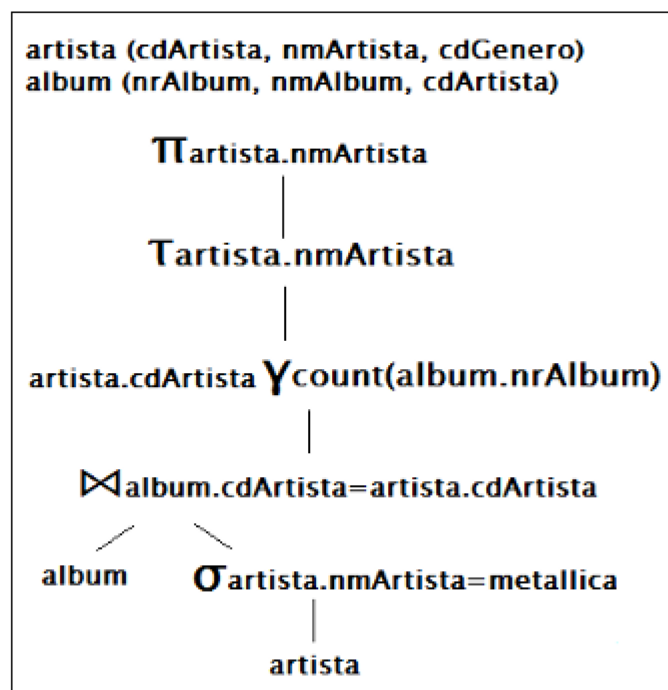


Figura 19 – Ordenação das restrições simples

O último passo a ser realizado pelo plano de execução é ordenar as junções. O arranjo

das junções é feito de maneira que seus dois nós filhos sejam subárvores com caminho para as relações de entrada necessárias. Desta forma, é possível eliminar os produtos cartesianos existentes na árvore de consulta, como mostra a Figura 20, onde a operação `album × artista` foi substituída pela junção `album.cdArtista=artista.cdArtista`.

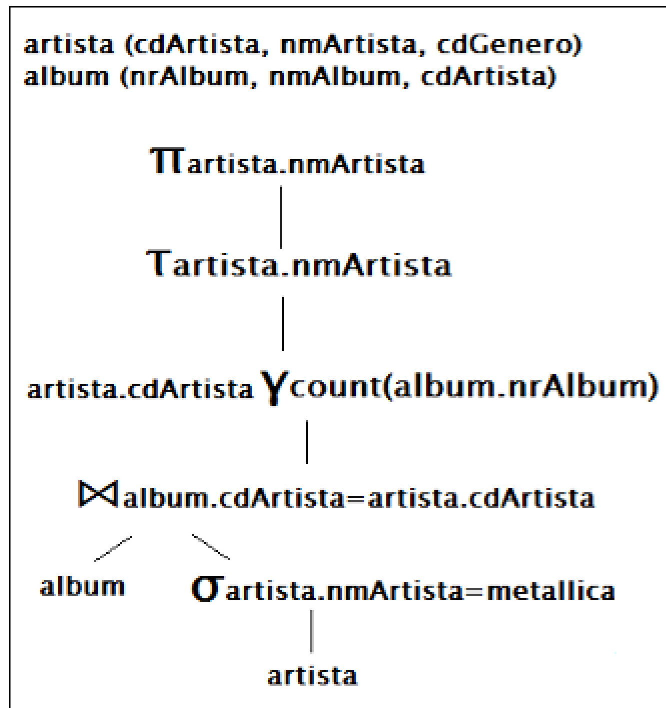


Figura 20 – Ordenação das junções

Uma vez terminada a construção da árvore por parte do plano de execução, a classe `Query` encarrega-se de passar a instância da árvore para sua instância de `QueryImpl`, que implementa os algoritmos referentes a cada um dos operadores. A instância de `QueryImpl`, então, percorre os nós da árvore recursivamente, executando cada operador até chegar à raiz, onde o resultado final é devolvido ao usuário.

3.3.2 Execução da consulta

A implementação da execução da consulta, ou seja, dos algoritmos referentes a cada operador, encontra-se na classe `QueryImpl`. A consulta é feita percorrendo-se cada um dos nós da árvore de consulta gerada pelo `PlanoExecucao` e resolvendo suas respectivas operações algébricas. O início deste processo é dado na raiz da árvore e é recursivo, ou seja, a entrada de cada nó visitado é calculada com base nas operações de seus nós filhos ou subárvores. Utilizando este método, conhecido como canalização (do inglês *pipelining*), não há a necessidade de armazenar as relações resultantes de cada nó em memória, uma vez que

estas são imediatamente usadas pelos operadores seguintes.

O valor de um operando referente a um campo é recuperado com a utilização da API *Reflection*. Para que seja possível recuperar valores deste tipo, os operandos devem ser identificados pelo nome de um atributo que possui um *getter* conforme os padrões utilizados no Java ou um método que não possua parâmetros de entrada e retorne algum valor. Caso não seja possível encontrar o campo identificado no operando, uma exceção é lançada, informando ao usuário de que o campo é inexistente.

Nas subseções a seguir são apresentadas implementações dos algoritmos para resolução de cada um dos operadores algébricos utilizados neste trabalho.

3.3.2.1 Renomeação

O operador de renomeação é utilizado para apelidar relações e projeções de uma consulta. Utilizando o operador de renomeação, é possível substituir o nome original dos operandos da consulta por um apelido. Neste trabalho, a operação de renomeação se faz necessária quando uma relação é declarada mais de uma vez na cláusula `from`.

Para renomear uma operação, o analisador semântico armazena em um `HashMap` todos os apelidos declaradas na cláusula `select`. A medida que os apelidos vão sendo declarados durante a consulta, as operações referentes a estes apelidos são recuperadas do `HashMap` e associadas ao operando declarado.

3.3.2.2 Busca linear

O algoritmo mais simples implementado neste trabalho é o algoritmo de busca linear. Para recuperar os dados, o algoritmo de busca linear executa uma única passagem sobre a coleção, testando os objetos visitados e adicionando-os ao resultado final caso satisfaçam a condição da restrição.

A busca linear é usada principalmente para resolver restrições simples, cujos operandos estão relacionados a uma ou nenhuma relação. Desta maneira, sendo `R` uma relação declarada na cláusula `from`, as expressões `R.atributo = 1`, `R.atributo is null`, `R.atributo1 = R.atributo2` e até mesmo a expressão `1 = 1` são consideradas restrições

simples. Além das restrições simples, algumas junções também podem ser resolvidas com o algoritmo de busca linear, desde que as tuplas referentes à junção já tenham sido calculadas previamente por algum dos algoritmos de junção implementados neste trabalho.

Uma representação do algoritmo de busca linear pode ser vista no Quadro 8.

```
R → {}
para cada objeto X na coleção R1 faça
   se X obedece condição então
      R = R ∪ {X}
fim para
```

Quadro 8 – Representação do algoritmo de busca linear para restrições simples

3.3.2.3 Junção de laços aninhados

O algoritmo de laços aninhados para as operações de junção pode ser utilizado para as restrições que usam qualquer um dos operadores relacionais, porém, devido ao alto custo de execução, foi usado somente para resolver restrições cujo operador relacional não seja de igualdade (=). Segundo Garcia-Molina, Ullman e Widom (2001, p. 306), um exemplo em que o uso de laços aninhados se faz essencial é quando existe apenas um valor dos atributos da junção na primeira relação de entrada. Caso na primeira relação de entrada já existam os dois valores dos atributos de junção, o algoritmo de laços aninhados pode ser substituído pela busca linear para que uma melhor performance seja alcançada.

O papel do algoritmo de laços aninhados, sendo `R` e `S` duas relações de entrada da junção, é agir de maneira que, para cada objeto de `R` é feita uma busca linear em `S` com o objetivo de encontrar os objetos que satisfaçam a condição de junção e adicioná-los no resultado final.

O algoritmo de laços aninhados é representado através de pseudocódigo no Quadro 9.

```
R → {}
para cada objeto X na coleção R1 faça
   para cada objeto Y na coleção R2 faça
      se X e Y satisfazem condição de junção então
         J → junção de X com Y
         R = R ∪ {J}
      fim se
   fim para
fim para
```

Quadro 9 – Algoritmo de junção de laços aninhados

3.3.2.4 Junção baseada em *hash*

O fato de ter que realizar uma busca linear em `S` para cada objeto em `R`, torna o processamento de uma junção de laços aninhados bastante custoso e, devido a este fato, a implementação de uma solução alternativa para realizar junções foi necessária.

O algoritmo de junção baseado em *hash* é aplicado para as junções que utilizam o operador relacional de igualdade ou um equivalente (`not <>`). O objetivo principal deste algoritmo é encontrar, nas duas relações de entrada, os objetos que possuem o mesmo *hash* para o valor dos atributos da junção. Para isso, a relação de entrada do primeiro operando é percorrida, gerando um *hash* para o valor do atributo da junção de cada um dos objetos da coleção. Ao término da iteração sobre a primeira coleção tem-se uma tabela *hash*, representada por um `HashMap` do Java, onde as chaves são os valores dos atributos de junção e o valor associado a cada chave é uma lista de objetos que possuem tal *hash*. Sendo assim, a próxima etapa do algoritmo é iterar sobre a segunda coleção e fazer a junção dos objetos que possuem um *hash* equivalente na tabela gerada na etapa anterior. Como é necessário efetuar a varredura somente uma vez em cada coleção, o custo da junção *hash* é muito menor se comparado com o custo da junção de laços aninhados. Em certos casos, assim como no algoritmo de laços aninhados, também há a possibilidade de substituir uma junção *hash* pela busca linear.

No Quadro 10 é apresentado o pseudocódigo do algoritmo de junção *hash*.

```
R → {}
T → tabela hash

para cada objeto X na coleção R1 faça
   h → hash do atributo de junção em X
   inserir h em T
fim para

para cada objeto Y na coleção  R2 faça
   h → hash do atributo de junção em Y
   O → objeto em T cujo hash é h
   se O não é nulo
      J → junção de O com Y
      R = R ∪ {J}
   fim se
fim para
```

Quadro 10 – Pseudocódigo do algoritmo de junção baseada em *hash*

3.3.2.5 Agrupamento por *hash*

O objetivo do operador de agrupamento é eliminar registros duplicados e também realizar as operações matemáticas referentes às funções de agregação `avg`, `count`, `max`, `min` e `sum`.

Os objetos de uma coleção podem ser agrupados com a utilização de uma tabela *hash*, tendo seu processamento similar ao da junção *hash*. O processo de agrupamento é realizado visitando-se cada elemento da coleção e gerando um *hash* com base nos valores dos atributos de agrupamento, usando para isso a estrutura `HashMap` do Java. Cada um dos objetos da lista é inserido no `HashMap` se e somente se já não existir nenhum outro objeto com o mesmo *hash* dentro do mapa de *hash*. O algoritmo de agrupamento também é responsável pelo cálculo das funções de agregação `count`, `sum`, `max`, `min` e `avg`, que, caso tenham sido declaradas na consulta, têm suas respectivas operações matemáticas efetuadas à medida que objetos de mesmo *hash* são encontrados na varredura da coleção. O resultado final é obtido através do método `values`, que retorna a coleção de objetos inseridos no `HashMap`.

O algoritmo de execução do operador de agrupamento é apresentado no Quadro 11.

```
R → {}
T → tabela hash

para cada objeto X na coleção R1 faça
   h → hash do agrupamento
   se não existe h em T então
      inserir h em T
      para cada função de agregação F faça
         inserir cópia de F como atributo de X
      fim para
   fim se
   para cada função de agregação F faça
      v → valor do atributo agregado na função
      Fo → cópia de F no objeto com hash de agrupamento h
      atualizar resultado final de Fo com v
   fim para
fim para
```

Quadro 11 – Algoritmo do operador de agrupamento

3.3.2.6 Ordenação

A ordenação dos dados recuperados na consulta é realizada após o agrupamento, visando a economia de tempo na execução, uma vez que o agrupamento diminui ainda mais o

tamanho do resultado da consulta. A classe `ResultListComparator`, que efetua a ordenação da consulta, é uma implementação da interface `Comparator` do Java e é usada em conjunto com o método `sort` da classe `Collections`. Sendo assim, o algoritmo de ordenação propriamente dito não foi explicitamente implementado, uma vez que o Java disponibiliza tal recurso.

3.3.2.7 Projeção

Na árvore de consulta, o operador de projeção é representado por uma lista de campos ou constantes que serão projetadas e é o último operador a ser executado. Para a resolução desta operação, o algoritmo implementado efetua uma passagem em todo resultado da consulta, mantendo em cada objeto somente os valores declarados na lista de projeções. Caso a cláusula `select` não tenha sido declarada na consulta, os valores projetados serão os próprios objetos recuperados das coleções.

O algoritmo responsável por projetar os campos do resultado final de consulta é mostrado no Quadro 13.

```
R → {}
L → lista de projeções
para cada objeto X na coleção R1 faça
   P → {}
   para cada atributo A de projeção em L faça
      P = P ∪{A}
   fim para
   R = R ∪{P}
fim para
```

Quadro 12 – Algoritmo de projeção

3.3.3 Tratamento de erros

O tratamento de erros do interpretador é feito tanto em nível léxico e sintático, quanto em nível semântico. Seguindo a sequência das fases de análise, os erros léxicos e sintáticos são os primeiros a serem detectados. Como estes erros estão diretamente associados à gramática da linguagem, a detecção fica a cargo das classes geradas pelo JavaCC. Para que as mensagens de erro léxico e sintático sejam apresentadas de forma mais clara ao usuário, as classes `TokenMgrError` e `ParseException` sofreram algumas alterações, já que o JavaCC

gera suas próprias mensagens.

Os erros semânticos são tratados em duas etapas. A primeira das etapas ocorre durante a construção da árvore de consulta e os erros que podem ser detectados são:

a) a não existência de uma coleção declarada na cláusula `from` no objeto a ser consultado;
b) a não implementação da interface `Collection` ou do tipo `Object[]` por alguma uma das relações informadas na cláusula `from`;
c) a redeclaração de uma relação na cláusula `from` com o mesmo apelido;
d) a falta do nome da relação nos operandos caso façam referência ao nome de um atributo e haja mais de uma relação de entrada na cláusula `from`;
e) a não declaração da cláusula `group by` caso alguma função de agregação tenha sido escrita na consulta.

A segunda etapa do tratamento dos erros semânticos acontece durante a execução da consulta, onde o interpretador deve verificar se os nomes dos atributos informados nos operandos existem em cada objeto das relações de entrada e também se seus valores implementam a interface `Comparable`. A execução desta etapa não é possível durante a montagem da árvore, uma vez que as coleções podem armazenar instâncias de diferentes tipos e, neste caso, os objetos de uma relação nem sempre terão os mesmos atributos.

Detalhes das classes de lançamento de exceção do analisador semântico são exibidos no Quadro 13.

Classe	**Descrição**
CampoInexistenteException	Exceção para o caso de um atributo não estar declarado em uma classe. Classe de uso interno da ferramenta, a exceção não é mostrada ao usuário.
ClausulaFromException	Classe de exceção para erros semânticos da cláusula `from`.
ClausulaGroupByException	Classe de exceção para erros semânticos da cláusula `group by`.
ClausulaOrderByException	Classe de exceção para erros semânticos da cláusula `order by`.
ClausulaSelectException	Classe de exceção para erros semânticos da cláusula `select`.
ClausulaWhereException	Classe de exceção para erros semânticos da cláusula `where`.
RelacaoInexistenteException	Lança um erro caso uma relação declarada na cláusula `from` não exista.
TiposIncompativeisException	Lança uma exceção caso seja feita uma comparação entre dois valores de tipos de dados diferentes. Classe de uso interno da ferramenta, a exceção não é mostrada ao usuário.
ValorInvalidoException	Lança uma exceção caso o valor recuperado de um atributo não implemente a interface `Comparable`. Classe de uso interno da ferramenta, a exceção não é mostrada ao usuário.

Quadro 13 – Detalhamento das classes de exceção do analisador semântico

3.3.4 Operacionalidade da implementação

A execução da consulta através do código Java é simples, e pode ser dividida em dois passos, que são:

a) instanciar um objeto da classe `Query`, existente no pacote `br.com.joqi.semantico.consulta`. O construtor de `Query` requer a passagem de um parâmetro, que deve ser a instância da classe onde encontram-se as coleções a serem consultadas. Para que seja possível realizar consultas em uma coleção, a mesma deve ser declarada como um atributo na classe em que está contida;
b) invocar o método `getResultCollection` da classe `Query`, que recebe como parâmetro a *String* de consulta. O método irá retornar uma `Collection` contendo instâncias da classe `ResultObject`.

Para recuperar os valores requisitados na consulta, a classe `ResultObject` disponibiliza os métodos `get`, `getString`, `getInt`, `getShort`, `getFloat`, `getDouble`, `getBoolean` e `getDate`, que retornam, respectivamente, instâncias do tipo `Object`, `String`, `int`, `short`, `float`, `double`, `boolean` e `java.util.Date` e recebem como parâmetro o nome do campo cujo valor deve ser recuperado.

No Quadro 14 é exibido um exemplo de código Java para utilização do interpretador, onde são selecionados os artistas cujo nome possua a sequência de caracteres “ch”. As coleções disponíveis para consulta encontram-se na classe `BancoConsulta`.

```
/*Cria a instancia da classe que possui as colecoes de consulta*/
BancoConsulta bancoConsulta = new BancoConsulta();

/*Cria a String de consulta*/
StringBuilder sb = new StringBuilder();
sb.append(" select    cdArtista, nmArtista, cdGenero ");
sb.append(" from      artistas ");
sb.append(" where     nmArtista like '%ch%' ");
sb.append(" order by  nmArtista desc ");

/*Cria a instancia de Query e executa o metodo getResultCollection*/
Query query = new Query(bancoConsulta);
Collection<ResultObject> col = query.getResultCollection(sb.toString());

/*Exibe os resultados da consulta no console*/
for(ResultObject objeto : col){
      System.out.print(objeto.getInt("cdArtista"));
      System.out.print("\t" + objeto.getInt("cdGenero"));
      System.out.print("\t" + objeto.getString("nmArtista"));
      System.out.println();
}
```

Quadro 14 – Exemplo de código Java para utilização do interpretador

Exemplos de código mostrando cada uma das cláusulas da linguagem são apresentados no Quadro 15.

Cláusula	Definição
select	SELECT atributo1 as a1, 'constante de texto' texto, date('01/01/2011') data, atributo2, 1+2*3 as expressao, count(atributo) contador
from	FROM colecao1, colecao2 c2, colecao3 as c3
where	WHERE not atributo1 = 1 AND atributo2 <> 'texto' OR atributo3 >= date('01/01/2011') AND (atributo4 is null or atributo4 is true) AND atributo5 between 1 and 100 Caso exista mais de uma relação declarada na cláusula from: WHERE not c1.atributo1 = 1 AND c2.atributo2 <> 'texto' OR c3.atributo3 >= date('01/01/2011') AND (c4.atributo4 is null or c4.atributo4 is true) AND c5.atributo5 between 1 and 100
group by	GROUP BY atributo1, atributo2, atributo3
order by	ORDER BY atributo1, atributo2, atributo3

Quadro 15 – Exemplos de uso das cláusulas da linguagem

Na subseção a seguir é apresentado um estudo de caso para demonstrar o uso da ferramenta.

3.3.4.1 Estudo de caso

Os testes foram efetuados sobre três coleções que simulam uma base de dados de álbuns musicais e armazenam instâncias das classes `Album`, `Artista` e `Genero`, que são apresentadas no diagrama da Figura 21.

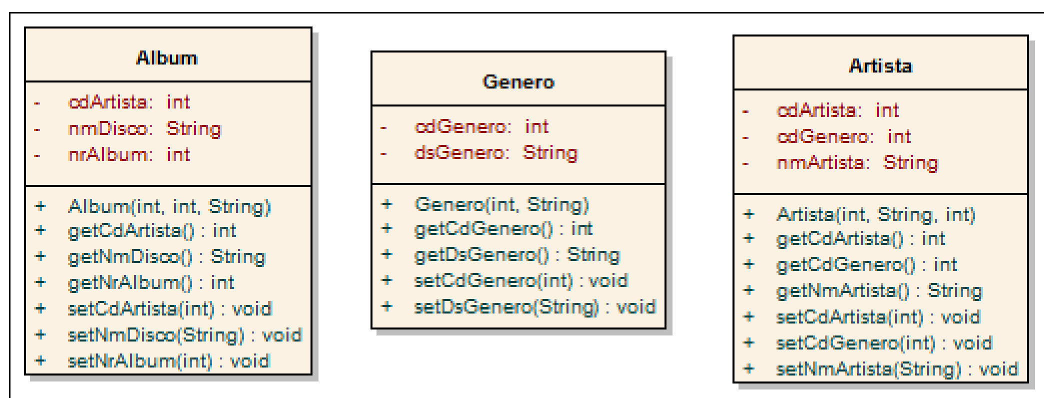


Figura 21 – Diagrama de classes do estudo de caso

Como a ferramenta desenvolvida neste trabalho não possui interface gráfica, um editor de consultas foi criado para demonstrar seu uso. O *layout* básico do editor possui uma área de texto (`JTextArea`) para a escrita do código de consulta (Figura 22). Ao lado da área de edição do texto fonte, é mostrada uma lista de coleções disponíveis para consulta, que encontram-se na instância classe `BancoConsulta` armazena pelo editor. Botões para abertura de arquivos,

limpeza da área de texto e execução da consulta são exibidos na parte de cima da tela. O resultado da consulta é exibido em uma `JTable`, que carrega os objetos do tipo `ResultObject`, abaixo do campo onde o código fonte da *query* é editado. O número de registros recuperados e o tempo de execução da consulta são exibidos logo abaixo da tabela resultante.

Ao clicar-se no botão “Executar”, o editor cria uma instância de `Query` que recebe como parâmetros a referência para a classe `BancoConsulta` e a *String* de consulta informada na área de texto do editor. O método `getResultCollection` de `Query` é executado e o resultado da consulta é exibido na tela. A consulta é executada dentro de um bloco `try-catch` e caso algum erro seja encontrado, ele é apresentado através de uma mensagem utilizando a classe `JOptionPane`.

Utilizando o editor de consultas, diferentes *queries*, que podem ser vistas no Quadro 16, foram escritas para recuperar os dados das coleções.

Descrição	**Query**

Recuperar artistas de reggae, ordenado pelo nome do artista	`select art.nmArtista,`
	`       gen.dsGenero`
	`from   artistas art,`
	`       generos gen`
	`where  art.cdGenero = gen.cdGenero`
	`order by art.nmArtista`

Quantidade de álbuns por gênero	`select gen.dsGenero,`
	`       count(alb.nrAlbum)`
	`from   albuns alb,`
	`       generos gen,`
	`       artistas art`
	`where  alb.cdArtista = art.cdArtista and`
	`       art.cdGenero = gen.cdGenero`
	`group by gen.cdGenero`

Álbuns dos artistas cujo nome contém “ozzy”	`select alb.nmAlbum`
	`from   albuns alb,`
	`       artistas art`
	`where  alb.cdArtista = art.cdArtista and`
	`       art.nmArtista like '%ozzy%'`

Quadro 16 – Exemplos de *queries* para recuperar dados das coleções

Na Figura 22 é exibido o resultado de uma *query* que busca a quantidade de álbuns de cada artista, mostrando também o gênero musical ao qual cada um pertence.

Editor de consultas

Abrir arquivo... Limpar Executar

albuns (291)
artistas (287)
generos (2)

```
select  art.nmArtista,
        gen.dsGenero,
        count(alb.nrAlbum)
from    albuns alb,
        artistas art,
        generos gen
where   alb.cdArtista = art.cdArtista and
        art.cdGenero = gen.cdGenero
group by art.cdArtista
order by art.nmArtista
```

nmArtista	dsGenero	count(alb.nrAlbum)
Alpha Blondy	Reggae	20
Black Sabbath	Metal	17
Bob Marley	Reggae	14
Death	Metal	10
Edson Gomes	Reggae	9
Groundation	Reggae	9
Iron Maiden	Metal	15
Jimmy Cliff	Reggae	25
Judas Priest	Metal	16
Megadeth	Metal	13
Metallica	Metal	9
Motörhead	Metal	21
Ozzy Osbourne	Metal	11
Pantera	Metal	13
Peter Tosh	Reggae	8
Slayer	Metal	11

Registros: 19 Tempo: 156.0 ms

Figura 22 – *Query* para buscar a quantidade de álbuns por artista

3.4 RESULTADOS E DISCUSSÃO

No Java, filtrar uma coleção com várias condições pode ser um processo custoso, uma vez que, para cada condição, é necessário que um `if` seja escrito. Se os dados estiverem separados em coleções diferentes, laços `for` aninhados devem ser codificados. Com isso, uma grande quantidade de código deve ser escrita pelo programador, o que pode tornar o código fonte complicado e pouco legível.

O estudo de caso apresentado mostra como é possível recuperar objetos em uma coleção sem utilizar laços `for` para iterar e comandos `if` para filtrar os dados, através de uma ferramenta que disponibiliza uma linguagem específica para consulta em instâncias de objetos do tipo `Collection` e também em vetores.

Além da ferramenta desenvolvida neste trabalho, existem outras que possibilitam a filtragem de objetos que implementam as interfaces `Collection` e `Iterable` do Java, como o

Lambdaj (FUSCO, 2009) e a biblioteca Coollection (ANDRADE, 2010). Já na linguagem C#, o LINQ permite realizar este tipo de busca em objetos IEnumerable<T>.

Para demonstrar o uso do LINQ e de cada uma das ferramentas de consulta para Java apresentadas neste trabalho, o Quadro 17 apresenta *queries* escritas usando cada uma das bibliotecas. As *queries* são equivalentes e tem como objetivo buscar os artistas cujo nome seja Metallica.

	Recuperar o artista cujo nome seja Metallica	**Tempo médio (milissegundos)**
Interpretador de consultas para objetos Java	from artistas where nmArtista = ‘metallica’	31
Coollection	from(artistas).where("getNmArtista", eq("Metallica")).all();	24

LambdaJ	filter(new Predicate<Artista>() {
	@Override
	public boolean apply(Artista a) {
	return a.getNmArtista().equalsIgnoreCase("Metallica");
	}
	}, artistas)
		26

LINQ	from art in artistas where art.nmArtista == "metallica" select art	2

Quadro 17 – Comparação entre as bibliotecas de consultas do Java

Analisando o Quadro 17, pode-se observar que a principal diferença entre o interpretador e as ferramentas apresentadas, é a maneira de escrever a consulta. Ao contrário do Lambdaj e da Coollection, que utilizam métodos estáticos aninhados, uma consulta no interpretador é escrita diretamente em uma única *String*. Também é possível notar que, se comparado com a biblioteca LambdaJ, a quantidade de código a ser passada para o interpretador é muito menor. Sendo assim, consultas que possuem uma grande combinação de restrições, devem ficar mais legíveis no interpretador e no LINQ.

O interpretador possui uma particularidade em relação às ferramentas Java LambdaJ e Coollection, que é a possibilidade de realizar consultas sobre mais de uma coleção ao mesmo tempo, ou seja, efetuar junções e produtos cartesianos. Levando em conta as versões testadas de cada ferramenta apresentada no Quadro 17, somente o LINQ disponibiliza este recurso.

Com relação ao tempo médio de execução da consulta, ambas as ferramentas tiveram um desempenho bastante parecido, a não ser o LINQ, cujo tempo de execução encontra-se muito abaixo das outras. As consultas feitas através do interpretador tendem a ser mais lentas, pois, além de executar consulta propriamente dita, ainda devem ser feitas as análises léxica, sintática e semântica e também a construção da árvore de consulta.

No Quadro 18 é feita uma comparação dos principais recursos de cada uma das ferramentas analisadas. A filtragem de coleções e a ordenação são duas operações comuns a todas as ferramentas apresentadas.

	Filtragem por atributos	**Junção/ Produto cartesiano**	**Agrupamento/ Agregação**	**Possibilita subconsultas**	**Ordenação**
Interpretador	X	X	X		X
Coollection	X			X	X
LambdaJ	X		X	X	X
LINQ	X	X	X	X	X

Quadro 18 – Recursos disponíveis em cada uma das ferramentas analisadas

4 CONCLUSÕES

A utilização de uma biblioteca para consulta em coleções ou vetores do Java pode agilizar o desenvolvimento de aplicações que buscam dados em memória. Ferramentas como LambdaJ e Coollection, oferecem uma opção para programadores que não desejam trabalhar com repetidos comandos `for` e `if`. Porém, estas ferramentas não disponibilizam uma linguagem de consulta, e sim uma série de classes e métodos que auxiliam na recuperação de objetos em coleções.

Com o uso do interpretador de consultas para objetos Java, códigos que hoje são escritos utilizando os comandos `for` e `if`, podem ser substituídos por *Strings* de consulta. Além de ficar mais claro e conciso, o código também fica mais dinâmico, uma vez que a *String* de consulta pode ser montada em partes.

Levando em consideração que as operações básicas da álgebra relacional foram implementadas, pode-se dizer que o objetivo deste trabalho foi alcançado. O interpretador mostrou-se capaz de executar consultas que combinam diferentes operadores algébricos como projeção, restrição, junção, produto cartesiano, agrupamento, agregação e ordenação. Os analisadores léxico e sintático, gerados pelo JavaCC, foram capazes de detectar erros no código de consulta, assim como certas inconsistências puderam ser detectadas pelo analisador semântico.

É interessante observar que, operações mais complexas, especialmente junções e restrições, devem ser implementadas com bastante cautela, para que a ferramenta possa ter um bom desempenho no que diz respeito ao tempo de execução. Desta maneira, pode-se dizer que uma das grandes dificuldades encontradas neste trabalho foi lidar com restrições disjuntivas (conectadas pelo operador lógico `or`) quando a consulta é feita sobre mais de uma coleção.

Por fim, pode-se concluir que o desenvolvimento de uma ferramenta que auxilie na execução de consultas em memória é uma tarefa possível de ser realizada. A existência de uma linguagem de consulta para o Java pode vir a facilitar o desenvolvimento de aplicações de consulta, assim como o LINQ para a linguagem C#.

4.1 EXTENSÕES

Algumas sugestões de melhorias para o interpretador são:

a) otimizar as projeções, colocando-as junto com as restrições quando possível;
b) dar suporte ao uso de expressões aritméticas complexas, com o uso de parênteses e nomes de atributos;
c) implementar a cláusula `having`, para que seja possível filtrar as coleções a partir dos resultados das funções de agregação;
d) permitir aninhar `n` atributos nas cláusulas da consulta (ex: `pessoa.filho.idade`), uma vez que atualmente é possível referenciar somente os atributos da classe cuja consulta está sendo realizada (ex: `pessoa.filho`);
e) disponibilizar outras funções matemáticas além de `count`, `min`, `max`, `sum` e `avg`;
f) disponibilizar funções para tratamento de *Strings*;
g) permitir que sejam feitas subconsultas;
h) permitir a passagem de parâmetros na *String* de consulta;
i) disponibilizar a ferramenta na forma de *plugin* para ambientes de desenvolvimento Java, o que difundiria a utilização do interpretador.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores**: princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: LTC, 1995.

ANDRADE, Wagner. **A cool way to manipulate collections in Java**. Porto Alegre, 2010. Disponível em: <https://github.com/wagnerandrade/coollection>. Acesso em: 31 mar. 2011.

CORONEL, Carlos; MORRIS, Steven; ROB, Peter. **Database systems**: design, implementation and management. 9th ed. Boston: South-Western, 2010.

FUSCO, Mario. **Manipulating collections without loops with Lambdaj**. [S.l.], 2009. Disponível em: <http://www.codeproject.com/KB/java/lambdaj.aspx>. Acesso em: 26 mar. 2011.

FREEMAN, Adam; RATTZ JR., Joseph C. **Pro LINQ**: language integrated query in C# 2010. New York: Apress, 2010.

GARCIA-MOLINA, Hector; ULLMAN, Jeffrey D.; WIDOM, Jennifer. **Implementação de sistemas de banco de dados**. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

GESSER, Júlio V. **Compilador Java 5.0 para gerar código C++ para plataforma Palm Os**. 2007. 84 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MANZANO, José A. N. G. **Estudo dirigido de SQL**. São Paulo: Érica, 2002.

MARGUERIE, Fabrice; EICHERT, Steve; WOOLEY, Jim. **LINQ em ação**. Tradução Angelo Giuseppe Meira Costa. Rio de Janeiro: Ciência Moderna, 2009.

MARTINOTTO, André L. **Uso de Prolog no desenvolvimento de compiladores**: arquiteturas especiais de computadores. [S.l.], 2009. Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp135/trabs/martinotto/trabII/sintatico.htm>. Acesso em: 30 mar. 2011.

PRICE, Ana M. A.; TOSCANI, Simão S. **Implementação de linguagens de programação**: compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Database management systems**. 2th ed. Boston: McGraw-Hill College, 1999.

ROMAN, Steven. **Access database**: design & programming. 3rd ed. [S.l]: O'Reilly, 2002.

SUMATHI, S.; ESAKKIRAJAN, S. **Fundamentals of relational database management systems**. [S.l.]: Springer, 2007.

TAVARES, Orivaldo de L. **Ferramentas para construção de compiladores**. Vitória, 2000. Disponível em: <http://www.inf.ufes.br/~tavares/labcomp2000/javacc.html>. Acesso em: 11 nov. 2011.

TAYLOR, Allen G. **SQL for dummies**. 7th ed. Indianapolis: Wiley Publishing, 2010.

APÊNDICE A – Gramática da linguagem de consulta do interpretador

No Quadro 19 é apresentada a gramática da linguagem de consulta do interpretador no formato EBNF.

```
<DEFAULT> SKIP : {
" "
| "\r"
| "\t"
| "\n"
}

<DEFAULT> TOKEN : {
<SELECT: "select">
| <FROM: "from">
| <WHERE: "where">
| <HAVING: "having">
| <GROUP_BY: "group by">
| <ORDER_BY: "order by">
| <ASC: "asc">
| <DESC: "desc">
| <AND: "and">
| <OR: "or">
| <NOT: "not">
| <LIKE: "like">
| <IS: "is">
| <NULL: "null">
| <TRUE: "true">
| <FALSE: "false">
| <AS: "as">
| <BETWEEN: "between">
| <RANGE: "range">
}

<DEFAULT> TOKEN : {
<COUNT: "count">
| <SUM: "sum">
| <AVG: "avg">
| <MIN: "min">
| <MAX: "max">
}

<DEFAULT> TOKEN : {
<DATE: "date">
}

<DEFAULT> TOKEN : {
<MAIS: "+">
| <MENOS: "-">
| <MULTIPLICA: "*">
| <DIVIDE: "/">
| <IGUAL: "=">
| <MAIOR: ">">
| <MENOR: "<">
| <MAIOR_IGUAL: ">=">
| <MENOR_IGUAL: "<=">
| <DIFERENTE: "<>">
}
```

```
<DEFAULT> TOKEN : {
<PONTO: ".">
| <VIRGULA: ",">
| <PONTO_E_VIRGULA: ";">
| <ASPA: "\'">
| <ABRE_PARENTESE: "(">
| <FECHA_PARENTESE: ")">
}

<DEFAULT> TOKEN : {
<#DIGITOS: (["0"-"9"])+>
| <NUMERO: <DIGITOS> (<PONTO> <DIGITOS>)?>
| <#INICIO_IDENTIFICADOR: ["a"-"z","A"-"Z","_"]>
| <IDENTIFICADOR: <INICIO_IDENTIFICADOR> (<INICIO_IDENTIFICADOR> |
<DIGITOS>)*>
| <TEXTO: <ASPA> (~["\'"])* <ASPA>>
}

expressaoAritmeticaAux  :=    ( <NUMERO> )
expressaoAritmetica     :=    multiplicacaoDivisao ( <MAIS> | <MENOS> )*
multiplicacaoDivisao    :=    expressaoAritmeticaAux ( <MULTIPLICA> |
<DIVIDE> )*
funcaoAgregacao   :=    ( <COUNT> | <SUM> | <MIN> | <MAX> | <AVG> )
<ABRE_PARENTESE> projecaoCampo <FECHA_PARENTESE>
projecaoCampo     :=    ( <IDENTIFICADOR> <PONTO> <IDENTIFICADOR> |
<IDENTIFICADOR> )
projecaoAritmetica      :=    expressaoAritmetica
projecaoString    :=    <TEXTO>
projecaoDate      :=    <DATE> <ABRE_PARENTESE> projecaoString
<FECHA_PARENTESE>
projecaoFuncaoAgregacao :=    funcaoAgregacao
relacao     :=    <IDENTIFICADOR> ( ( <AS> )? <IDENTIFICADOR> )?
operando    :=    ( projecaoCampo | projecaoAritmetica | projecaoString |
projecaoDate )
restricao   :=    operando ( ( ( <IGUAL> | <MAIOR> | <MENOR> |
<MAIOR_IGUAL> | <MENOR_IGUAL> | <DIFERENTE> | <LIKE> | ( <BETWEEN>
operando <AND> ) ) operando ) | ( <IS> ( ( <NULL> ) | ( ( <TRUE> | <FALSE>
) ) ) ) )
projecao    :=    ( projecaoCampo | projecaoAritmetica | projecaoString |
projecaoDate | projecaoFuncaoAgregacao ) ( ( <AS> )? <IDENTIFICADOR> )?
select      :=    <SELECT> projecao ( <VIRGULA> projecao )*
from  :=    <FROM> relacao ( <VIRGULA> relacao )*
where :=    <WHERE> restricoes
restricoes  :=    restricoesAux ( ( <AND> | <OR> ) restricoesAux )*
restricoesAux     :=    ( <NOT> )? ( ( restricao ) | ( <ABRE_PARENTESE>
restricoes <FECHA_PARENTESE> ) )
tipoOrdenacao     :=    ( <ASC> | <DESC> )?
orderBy     :=    <ORDER_BY> operando tipoOrdenacao ( <VIRGULA> operando
tipoOrdenacao )*
groupBy     :=    <GROUP_BY> operando ( <VIRGULA> operando )*
range :=    <RANGE> expressaoAritmetica ( <VIRGULA> expressaoAritmetica )?
executa     :=    ( select )? from ( where )? ( groupBy )? ( orderBy )? (
range )? ( <PONTO_E_VIRGULA> )? <EOF>
```

Quadro 19 – Gramática da linguagem de consulta o interpretador

APÊNDICE B – Diagrama de classes do parser gerado pelo JavaCC

Pelo fato de ser muito grande, o diagrama de classes do *parser* gerado pelo JavaCC é divido em partes. Os relacionamentos entre as classes são exibidos no diagrama simplificado, na Figura 23.

Nas Figura 23 são exibidas as classes `Parser`, `ParserConstants`, `JJCalls` e `LookaheadSuccess`.

A Figura 24 mostra as classes `SimpleCharStream`, `ParserTokenManager`, `Token`, `ParseException` e `TokenMgrError`.

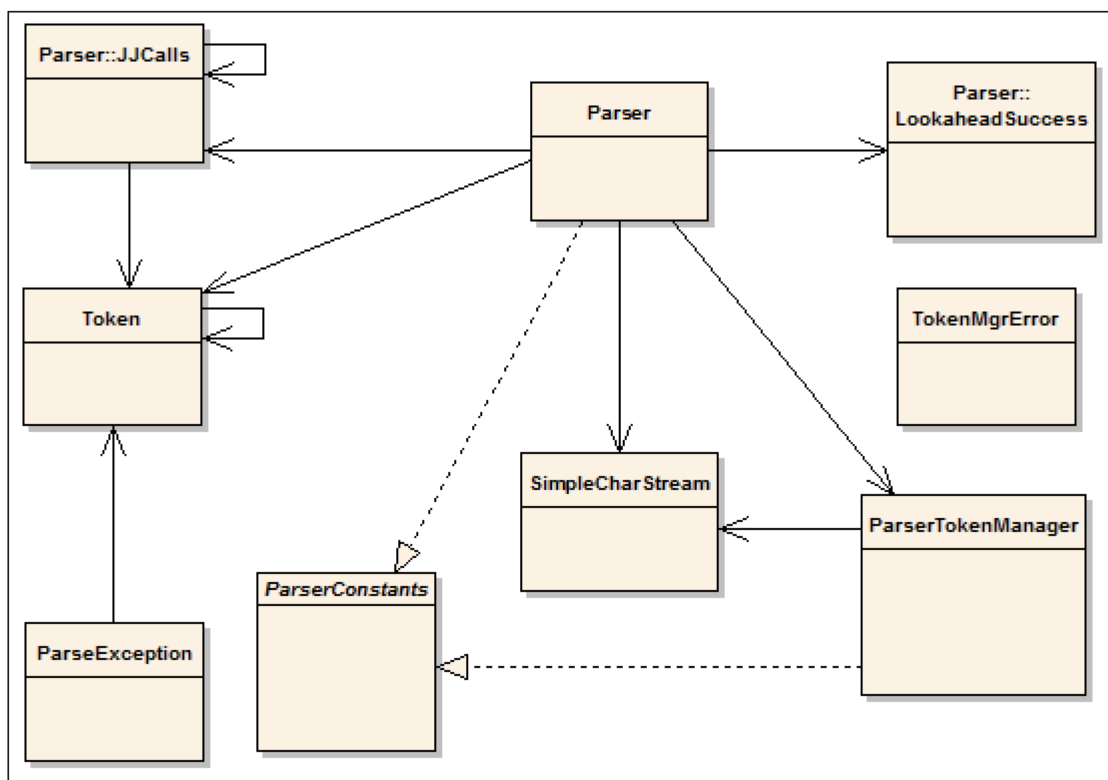


Figura 23 – Diagrama de classes do *parser* simplificado

Figura 24 – Diagrama de classes do *parser* gerado pelo JavaCC

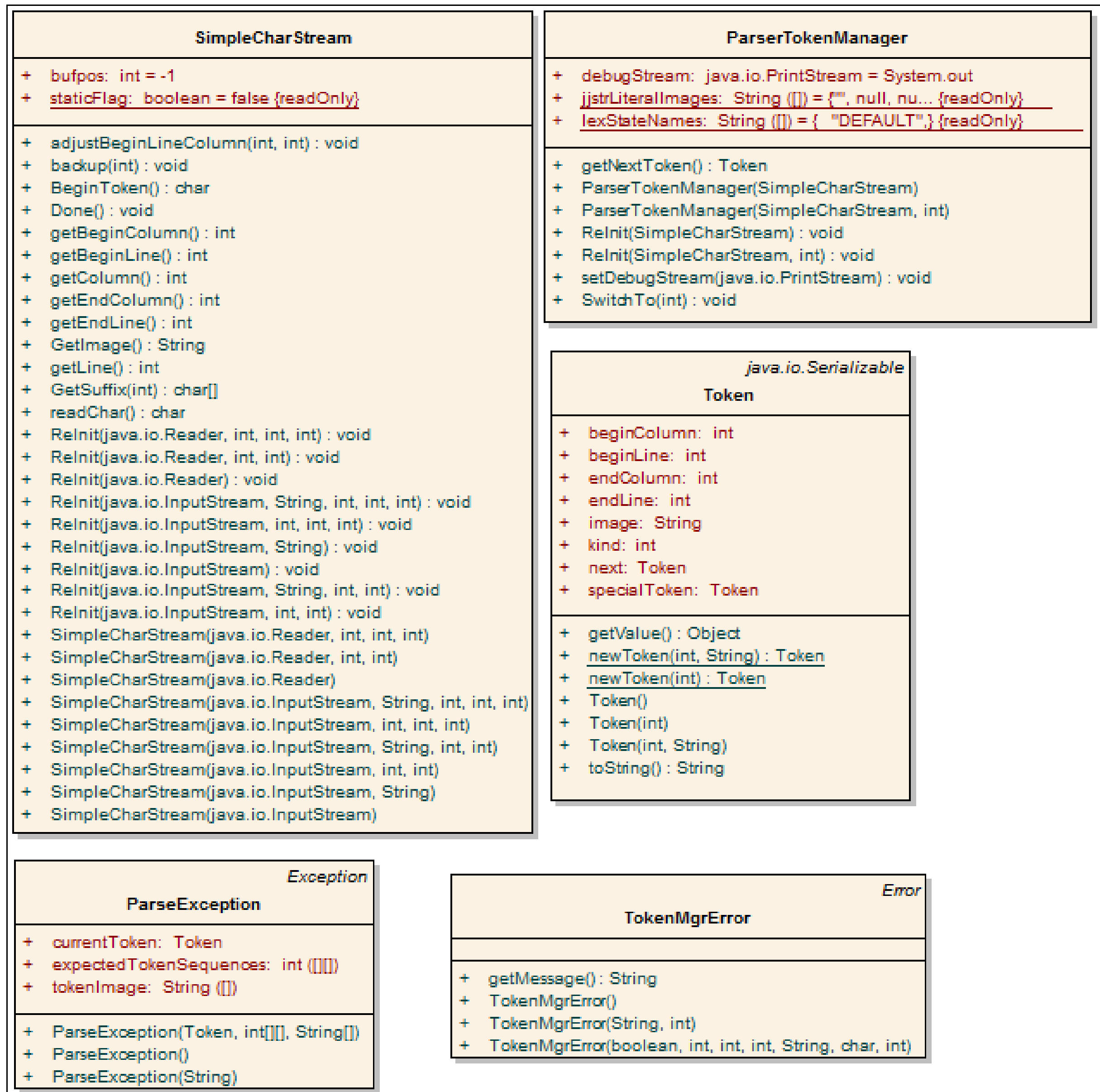


Figura 25 - Diagrama de classes do *parser* gerado pelo JavaCC

APÊNDICE C – Diagramas de classes do analisador semântico

Pelo fato de possuir uma grande quantidade de classes, o diagrama do analisador semântico foi dividido em partes, para seja possível visualizá-lo mais facilmente.

Na Figura 26 é mostrado o diagrama das classes de execução de consulta `Query`, `QueryUtils` e `QueryImpl`.

A Figura 27 apresenta o diagrama de classes dos operadores de agrupamento.

A Figura 28 mostra as classes referentes ao operador de ordenação.

A Figura 29 exibe o diagrama de classes do plano de execução.

A Figura 30 expõe as classes dos diferentes tipos de projeção.

Na Figura 31 são mostrados os diagramas de classes das restrições.

Na Figura 32 são apresentadas as classes de exceção do interpretador.

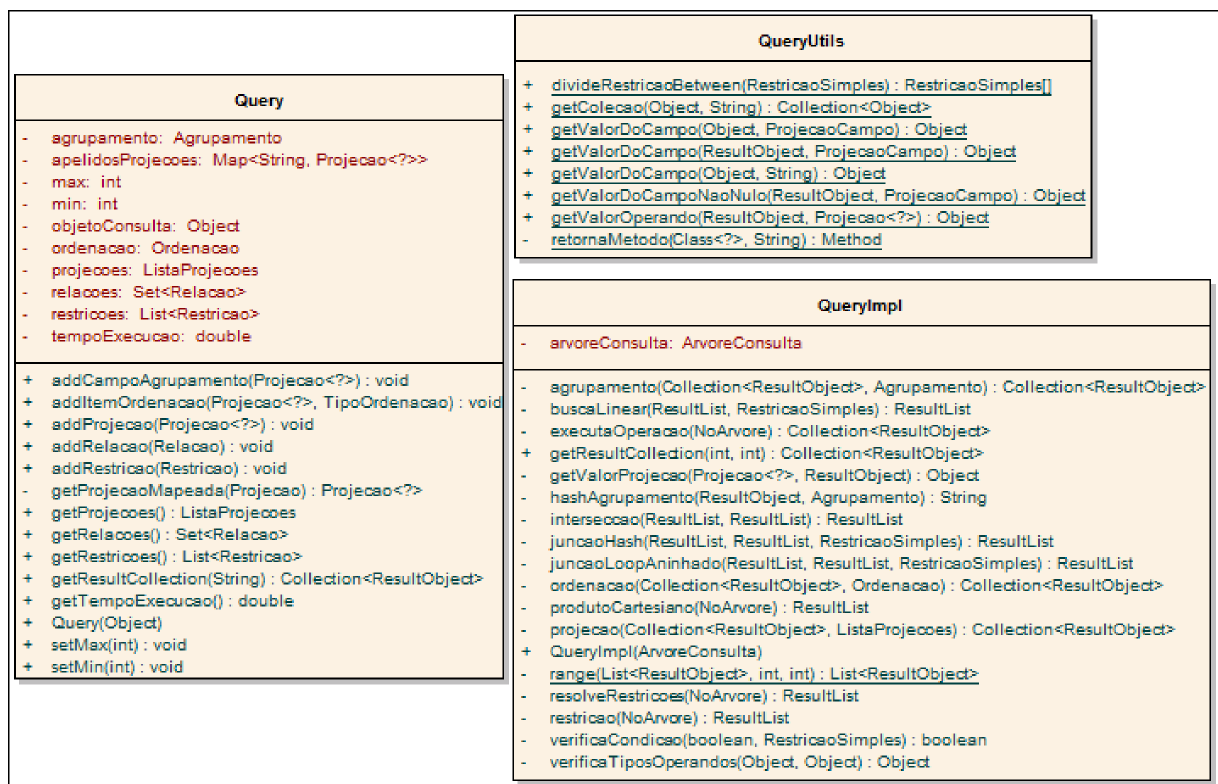


Figura 26 – Classes do pacote `br.com.joqi.semantico.consulta`

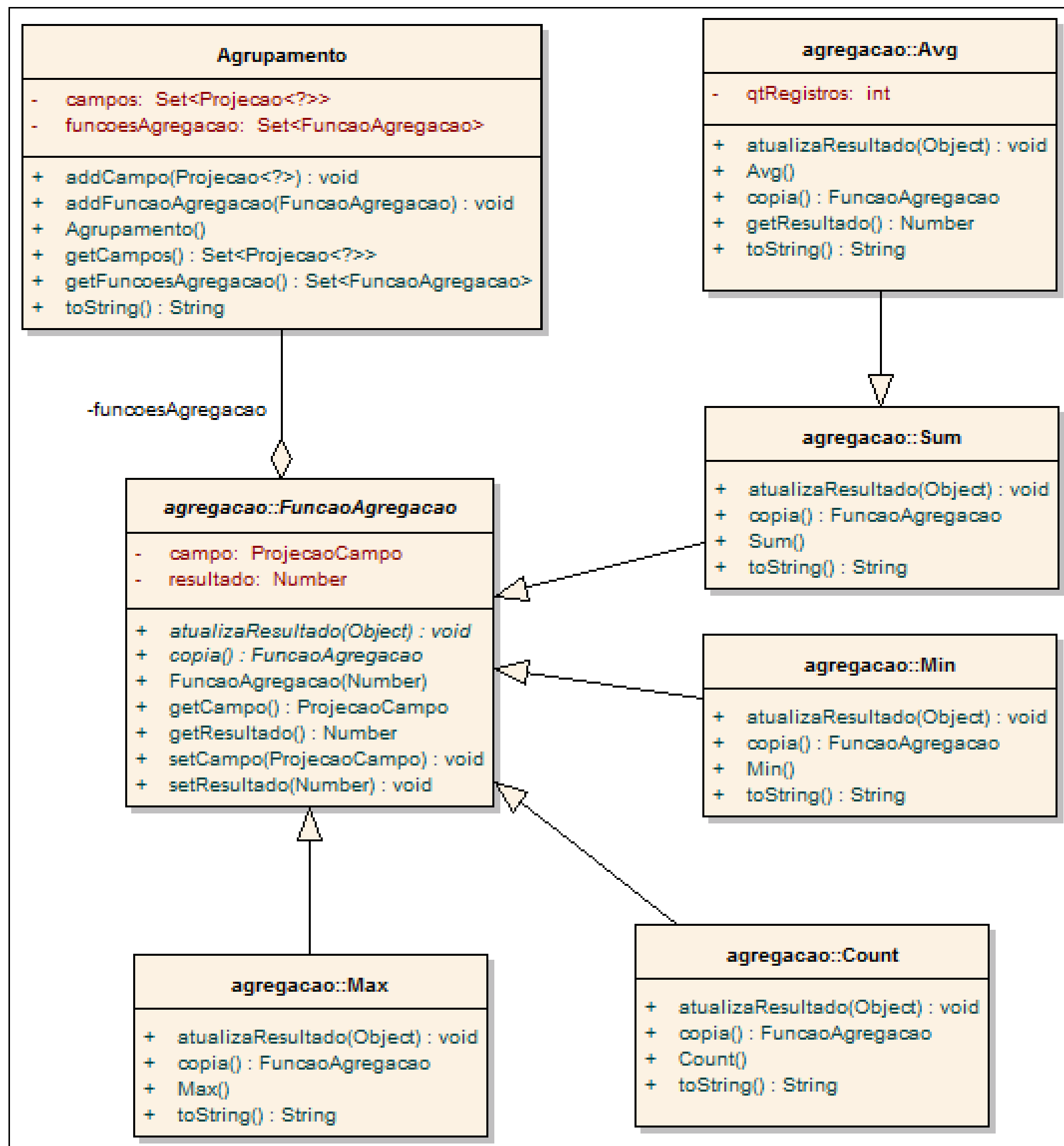


Figura 27 - Classes do pacote `br.com.joqi.semantico.consulta.agrupamento`

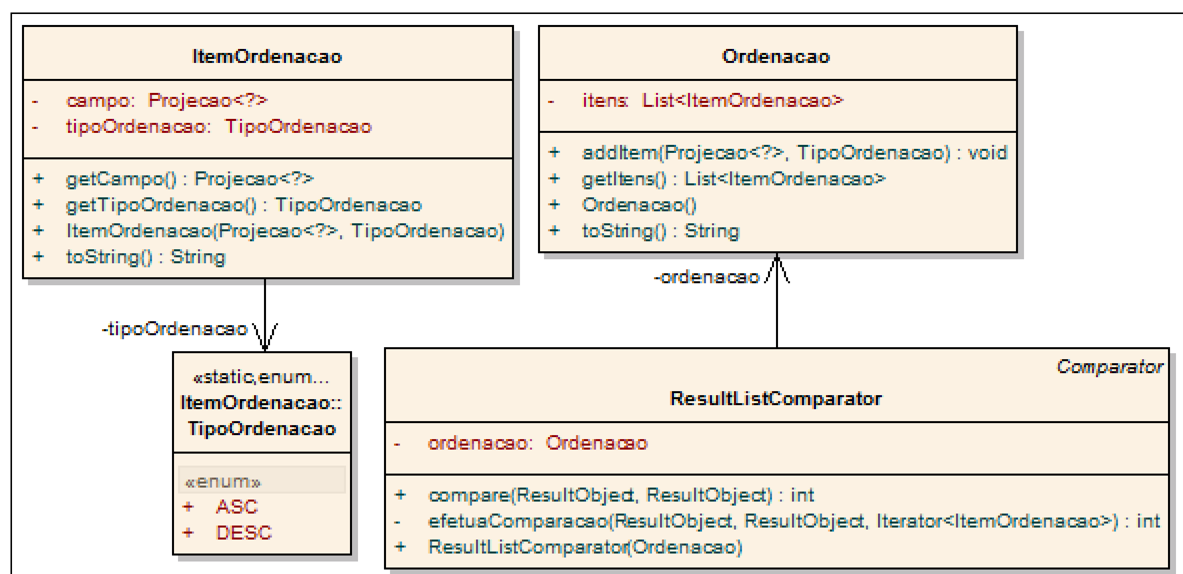


Figura 28 - Classes do pacote `br.com.joqi.semantico.consulta.ordenacao`

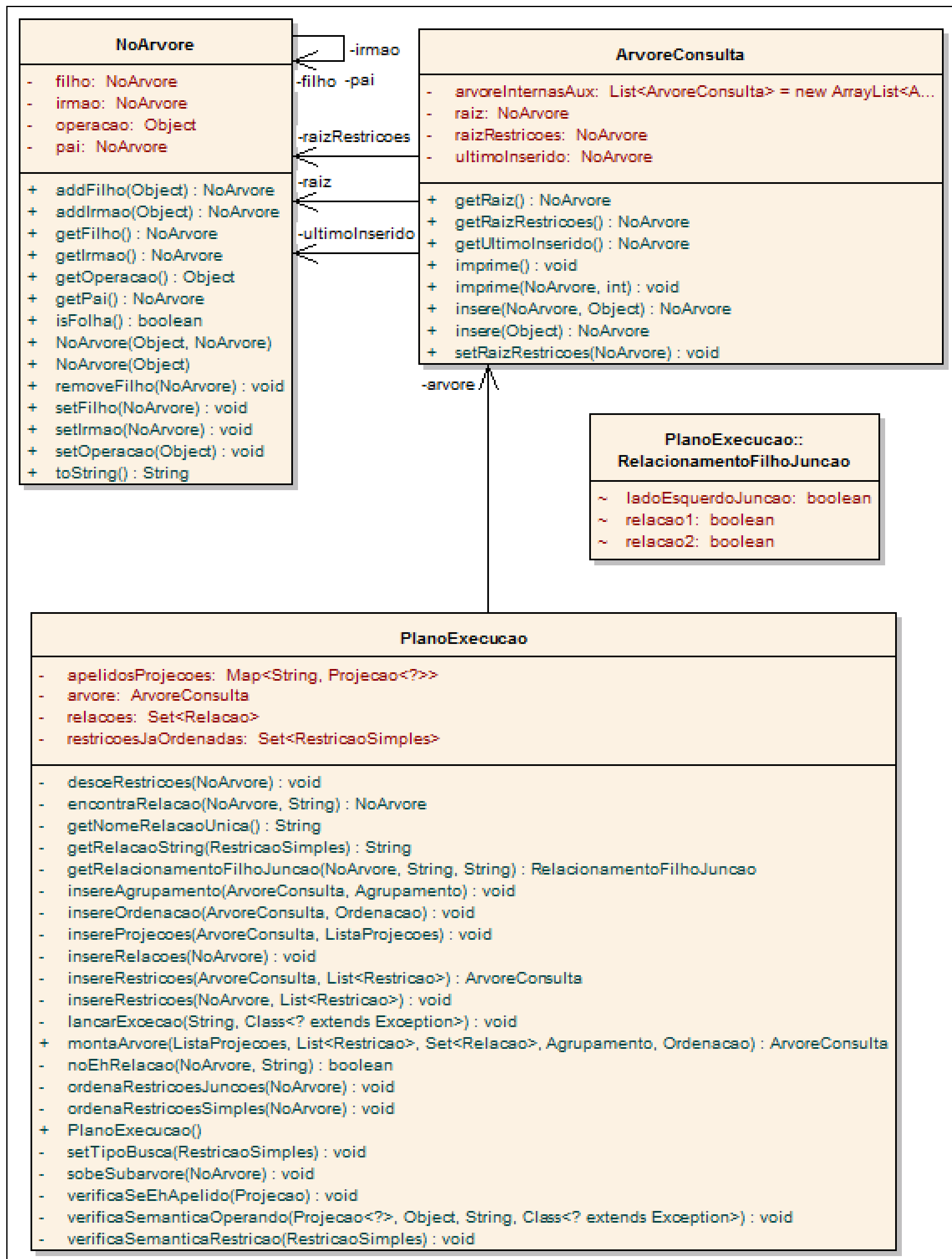


Figura 29 – Diagrama de classes do plano de execução

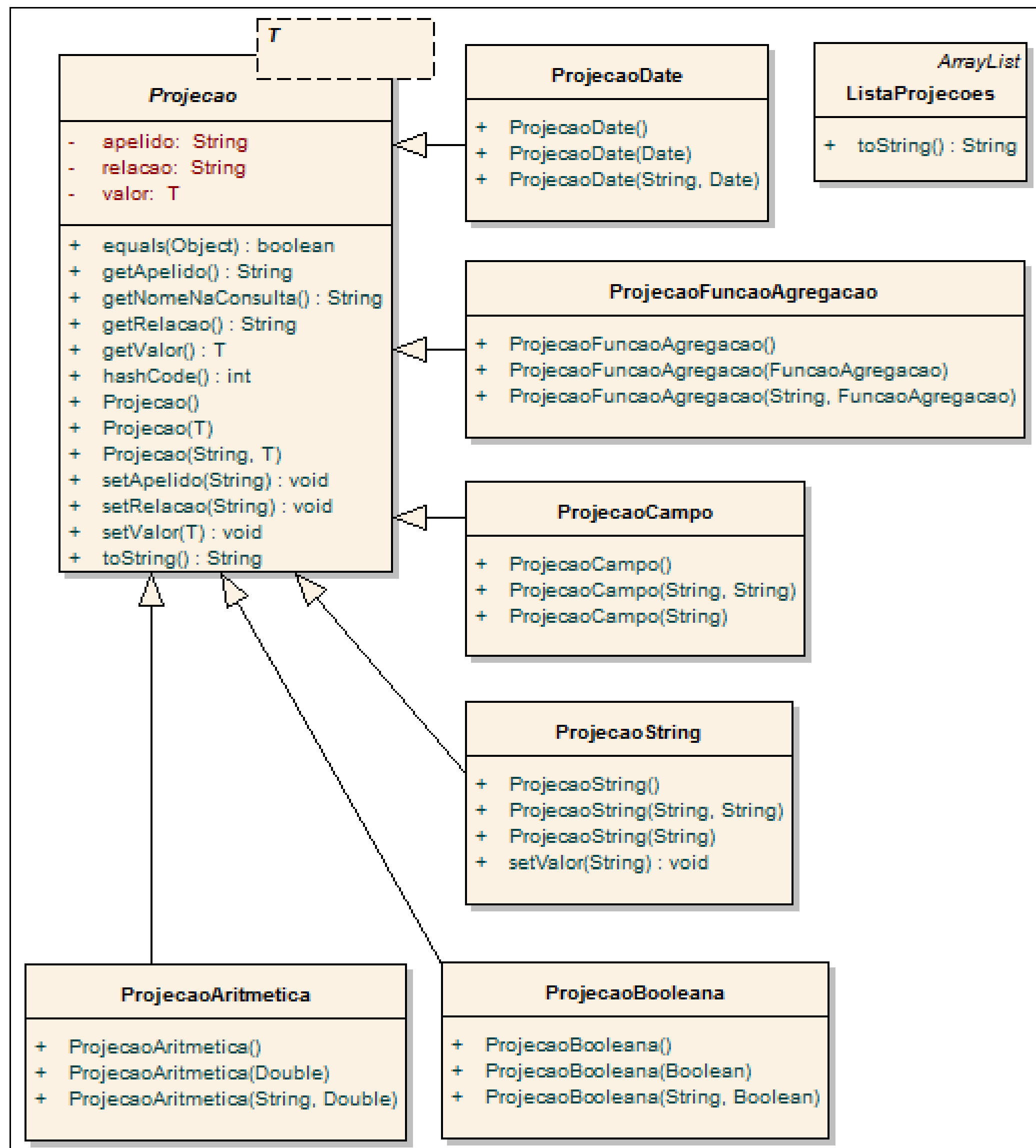


Figura 30 - Classes do pacote `br.com.joqi.semantico.consulta.projecao`

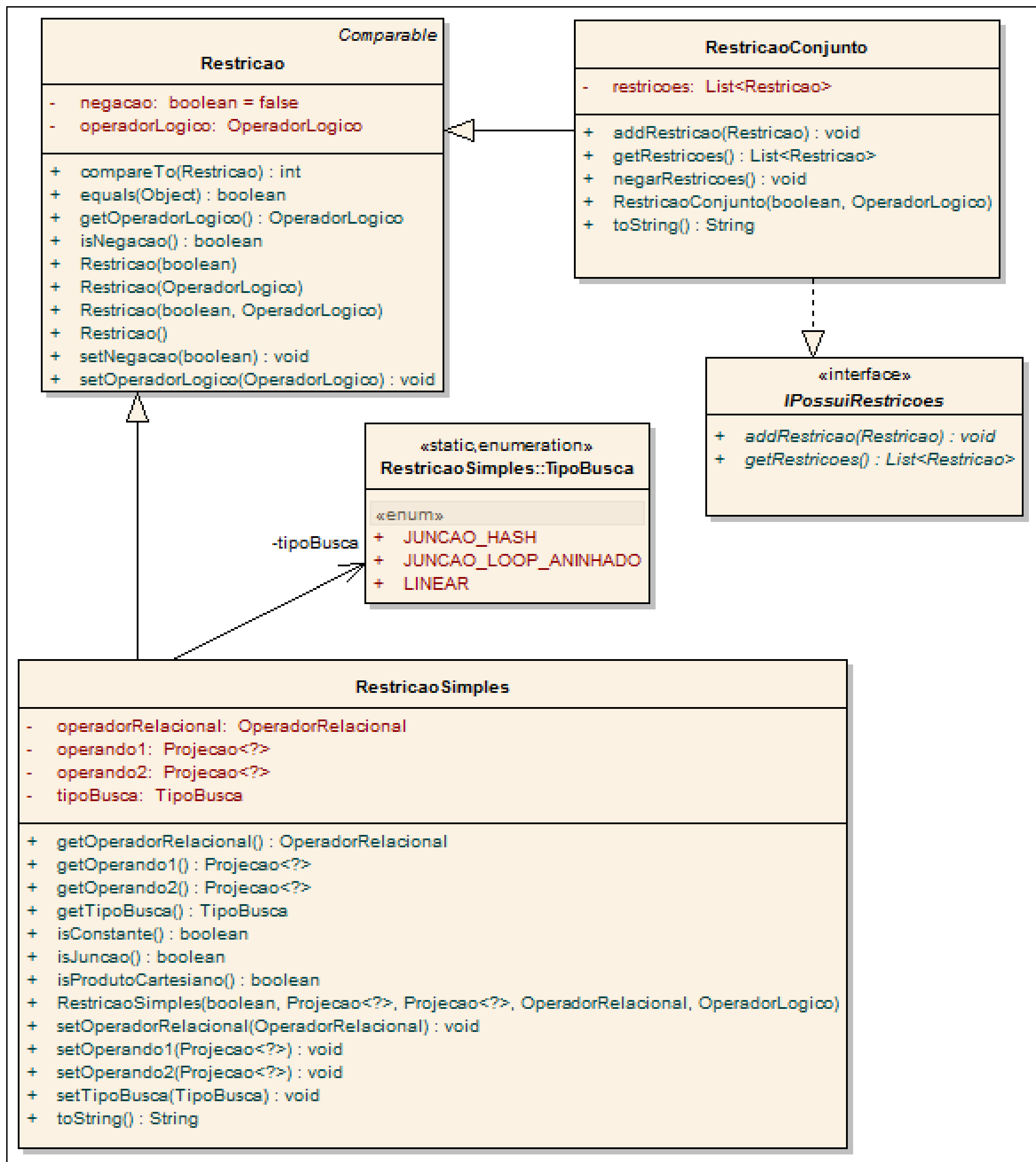


Figura 31 - Classes do pacote `br.com.joqi.semantico.consulta.restricao`

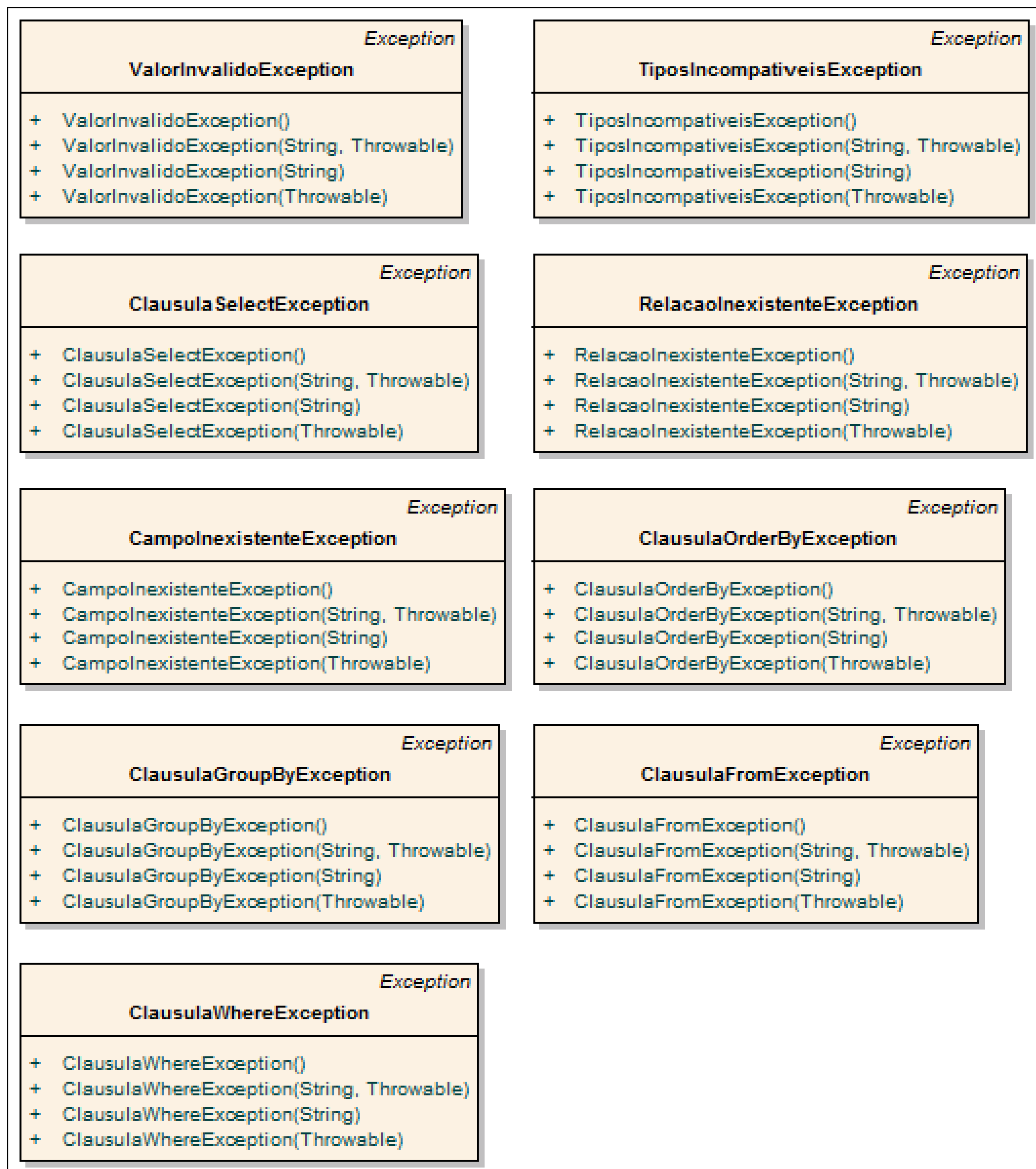


Figura 32- Classes do pacote `br.com.joqi.semantico.exception`