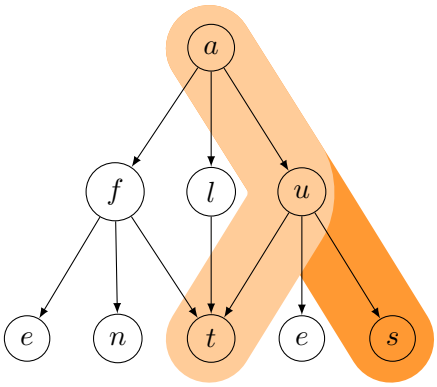
Afluentes

Um Framework para Programação Paralela

Saulo Medeiros de Araujo

13 de fevereiro de 2011



1 Introdução

O Afluentes é um framework para programação paralela livremente inspirado na técnica de redução de grafos utilizada pelas linguagens funcionais. Ele permite que uma computação seja descrita como um conjunto de subcomputações que, respeitando-se suas interdependências, serão executadas em paralelo.

2 Exemplo

Para explicar o funcionamento do Afluentes escreveremos um programa que calcula as raízes reais de uma equação do segundo grau. Dada uma equação $ax^2 + bx + c = 0$ onde $a \neq 0$ a fórmula de Bhaskara diz que se $\Delta = b^2 - 4ac$:

- é menor do que zero, então a equação não possui raízes;
- é maior ou igual a zero, então a equação possui raízes reais, não necessariamente distintas, dadas por $\frac{-b-\sqrt{\Delta}}{2a}$ e $\frac{-b+\sqrt{\Delta}}{2a}$.

2.1 Calculando Δ

A figura 1 ilustra a árvore sintática abstrata do cálculo do Δ. Nela, cada ocorrência de um operador representa uma subcomputação, o que resulta nas subcomputações e interdependências exibidas na figura 2.

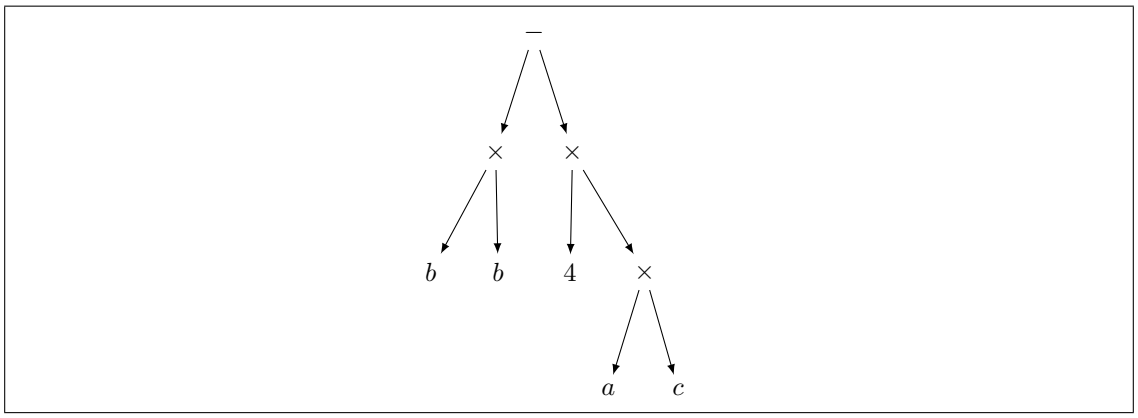


Figura 1: árvore sintática abstrata do cálculo do Δ

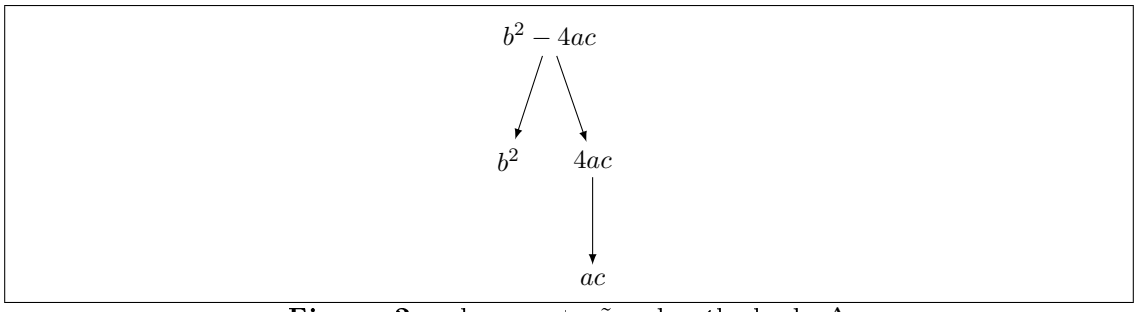


Figura 2: subcomputações do cálculo do Δ

Para cada subcomputação precisamos definir uma classe que encapsule seus parâmetros e seu resultado. No cálculo do Δ, entretanto, todas as subcomputações são operações binárias entre números reais, sendo suficiente, portanto, uma única classe, que chamaremos de `OpRes` cujo código é exibido na figura 3.

```
class OpRes {
  double x, y, result;

  OpRes() {}

  OpRes(double x) { this.x = x; }

  OpRes(double x, double y) { this.x = x; this.y = y; }
}
```

Figura 3: classe OpRes

Também é necessário, pada cada subcomputação, prover uma implementação da interface `VerticesProcessor` que será responsável por efetuar a subcomputação a partir dos parâmetros encapsulados na classe `OpRes` e nela armazenar o resultado obtido. A figura 4 exibe o código do processador de subtração utilizado no cálculo do Δ. O código do processador de multiplicação é similar.

```
class SubProcessor implements VerticesProcessor<OpRes> {
  @Override
  public void processVertices(List<? extends Vertex<OpRes>> vertices) {
    OpRes content;

    for (Vertex<OpRes> vertex: vertices) {
      content = vertex.getContent();
      content.result = content.x - content.y;
    }
  }
}
```

Figura 4: processador de subtração

A figura 5 exibe o código que descreve o cálculo do Δ como um conjunto de subcomputações e em seguida utiliza esta descrição para executá-las, respeitando suas interdependências, em paralelo. Eis uma descrição de seu funcionamento:

- a linha 11 cria uma instância do redutor de grafos;
- as linhas 13 e 14 cadastram os processadores de subtração e multiplicação no redutor obtendo, como resultado, instâncias da interface `VertexBuilder`;
- a linha 16 cria a subcomputação que irá calcular b^2;
- a linha 19 cria a subcomputação que irá calcular ac;
- a linha 22 cria a subcomputação que irá calcular $4ac$;
- a linha 23 cadastra a dependência entre as subcomputações $4ac$ e ac e uma instância da classe `SetX` como ouvinte da remoção desta dependência;
- a linha 26 cria a subcomputação $b^2 - 4ac$;
- a linha 27 cadastra a dependência entre as subcomputações $b^2 - 4ac$ e b^2 e uma instância da classe `SetX` como ouvinte da remoção desta dependência;
- a linha 28 cadastra a dependência entre as subcomputações $b^2 - 4ac$ e $4ac$ e uma instância da classe `SetY` como ouvinte da remoção desta dependência;
- a linha 31 utiliza o redutor de grafos para executar as subcomputações.

```
      double delta(double a, double b, double c)
          throws InterruptedException, VerticesProcessingException {
        GraphReducer reducer;
        VertexBuilder<OpRes> subBuilder;
        VertexBuilder<OpRes> mulBuilder;
        Vertex<OpRes> delta;
        Vertex<OpRes> b2;
        Vertex<OpRes> ac;
        Vertex<OpRes> _4ac;

/*11*/  reducer = new GraphReducerImpl();

/*13*/  subBuilder = reducer.addProcessor(new SubProcessor());
/*14*/  mulBuilder = reducer.addProcessor(new MulProcessor());

/*16*/  mulBuilder.createVertex(new OpRes(b, b));
        b2 = mulBuilder.getVertex();

/*19*/  mulBuilder.createVertex(new OpRes(a, c));
        ac = mulBuilder.getVertex();

/*22*/  mulBuilder.createVertex(new OpRes(4));
/*23*/  mulBuilder.appendEdge(ac, SetY.INSTANCE);
        _4ac = mulBuilder.getVertex();

/*26*/  subBuilder.createVertex(new OpRes());
/*27*/  subBuilder.appendEdge(b2, SetX.INSTANCE);
/*28*/  subBuilder.appendEdge(_4ac, SetY.INSTANCE);
        delta = subBuilder.getVertex();

/*31*/  reducer.reduceGraph(delta);

        return delta.getContent().result;
      }
```

Figura 5: cálculo do Δ

Quando o método `reduceGraph` é invocado, o grafo formado pelas subcomputações é percorrido e aquelas que não possuem dependências têm sua execução agendada. Quando uma subcomputação é executada ela é removida do grafo, os ouvintes de sua remoção são notificados e as subcomputações que passaram a não possuir dependências têm sua execução agendada. Esse processo continua até que a subcomputação passada como parâmetro para o método `reduceGraph` tenha sido executada.

Cada dependência entre subcomputações pode estar associada a uma instância da interface `EdgeRemovalListener` que desempenha as seguintes funções:

1. propaga o resultado da subcomputação que foi executada para a subcomputação dependente;

2. cria novas subcomputações.

As classes `SetX` e `SetY`, cujos códigos são exibidos na figura 6, desempenham a primeira função. A segunda função será detalhada na próxima seção.

```
class SetX implements EdgeRemovalListener<OpRes, OpRes> {
  static final SetX INSTANCE = new SetX();

  @Override
  public void edgeRemoved(OpRes dependent, OpRes dependable,
                          EdgeAppender<OpRes> appender) {
    dependent.x = dependable.result;
  }
}

class SetY implements EdgeRemovalListener<OpRes, OpRes> {
  static final SetY INSTANCE = new SetY();

  @Override
  public void edgeRemoved(OpRes dependent, OpRes dependable,
                          EdgeAppender<OpRes> appender) {
    dependent.y = dependable.result;
  }
}
```

Figura 6: classes `SetX` e `SetY`

2.2 Calculando as Raízes

Se Δ é maior ou igual a zero suas raízes podem ser calculadas pelas subcomputações ilustradas na figura 7. Note que, neste caso, existem subcomputações, tais como $\sqrt{\Delta}$, das quais dependem mais de uma subcomputação.

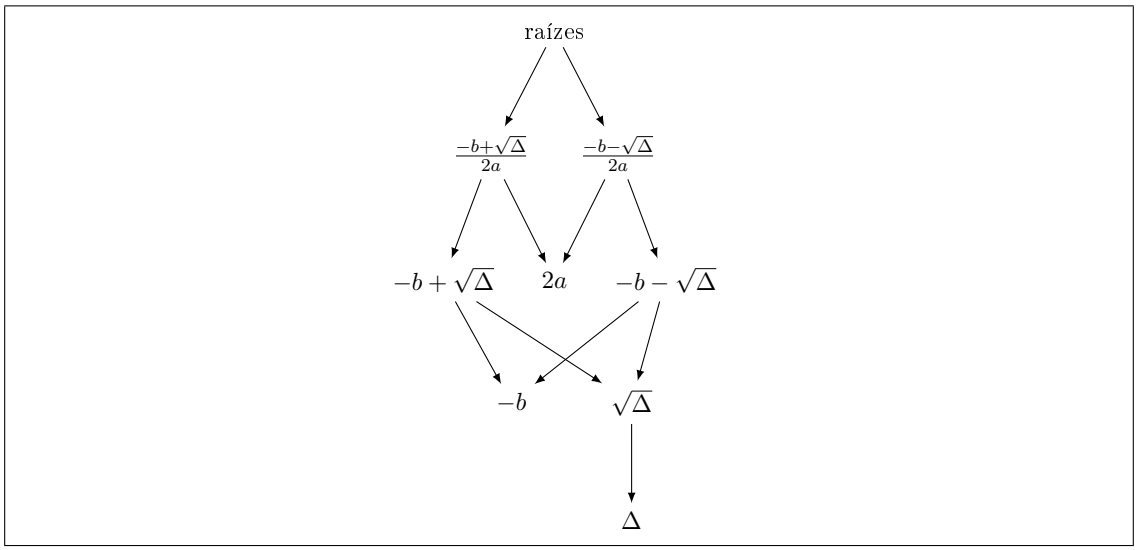


Figura 7: subcomputações do cálculo das raízes quando $\Delta \geq 0$

A figura 8 exibe o código que descreve o cálculo das raízes como um conjunto de subcomputações e em seguida utiliza esta descrição para executá-las, respeitando suas interdependências, em paralelo. Eis uma descrição de seu funcionamento:

- a linha 10 cadastra o processador da subcomputação que irá armazenar as raízes. Como esta subcomputação não tem outra função, seu processador, cujo código é exibido na figura 9, nada faz;
- a linha 12 cria a subcomputação que irá calcular Δ;
- a linha 22 cria a subcomputação que irá armazenar as raízes;
- a linha 23 cadastra a dependência entre a subcomputação que irá armazenar as raízes e a que calcula Δ e uma instância da classe `DeltaListener` como ouvinte da remoção desta dependência;
- a linha 26 utiliza o redutor de grafos para executar as subcomputações.

```
      Collection<Double> roots(double a, double b, double c)
          throws InterruptedException, VerticesProcessingException {
        /* ... */
        VertexBuilder<Collection<Double>> rootsBuilder;
        Vertex<OpRes> delta;
        DeltaListener listener;
        Vertex<Collection<Double>> roots;

        /* ... */
/*10*/  rootsBuilder = reducer.addProcessor(new RootsProcessor());

/*12*/  delta = createDelta(a, b, c, subBuilder, mulBuilder);

        listener = new DeltaListener();
        listener.a = a;
        listener.b = b;
        listener.c = c;
        listener.addBuilder = addBuilder;
        listener.subBuilder = subBuilder;
        /* ... */

/*22*/  rootsBuilder.createVertex(new ArrayList<Double>(2));
/*23*/  rootsBuilder.appendEdge(delta, listener);
        roots = rootsBuilder.getVertex();

/*26*/  reducer.reduceGraph(roots);

        return roots.getContent();
      }
```

Figura 8: cálculo das raízes

```
class RootsProcessor implements VerticesProcessor<Collection<Double>> {
  @Override
  public void processVertices(List<? extends Vertex<Collection<Double>>> vertices)
    throws Throwable {
  }
}
```

Figura 9: processador das raízes

```
class DeltaListener implements EdgeRemovalListener<Collection<Double>, OpRes> {
      /* ... */

      @Override
      public void edgeRemoved(Collection<Double> roots, OpRes delta,
                              EdgeAppender<Collection<Double>> appender) {
         Vertex<OpRes> _b;
         /* ... */

         if (delta.result >= 0) {
          subBuilder.createVertex(new OpRes(0, b));
          _b = subBuilder.getVertex();

          sqrtBuilder.createVertex(new OpRes(delta.result));
          _sqrt = sqrtBuilder.getVertex();

          subBuilder.createVertex(new OpRes());
          subBuilder.appendEdge(_b, SetX.INSTANCE);
          subBuilder.appendEdge(_sqrt, SetY.INSTANCE);
          _b_m_sqrt = subBuilder.getVertex();

          addBuilder.createVertex(new OpRes());
          addBuilder.appendEdge(_b, SetX.INSTANCE);
          addBuilder.appendEdge(_sqrt, SetY.INSTANCE);
          _b_p_sqrt = addBuilder.getVertex();

          mulBuilder.createVertex(new OpRes(2, a));
          _2a = mulBuilder.getVertex();

          divBuilder.createVertex(new OpRes());
          divBuilder.appendEdge(_b_m_sqrt, SetX.INSTANCE);
          divBuilder.appendEdge(_2a, SetY.INSTANCE);
          _b_m_sqrt_2a = divBuilder.getVertex();

/*35*/    appender.appendEdge(_b_m_sqrt_2a, RootsListener.INSTANCE);

          /* ... */
        }
      }
    }
```

Figura 10: classe `DeltaListener`

Quando a subcomputação que calcula Δ é executada, o ouvinte criado na linha 12, cujo código é exibido na figura 10, é notificado. Se $\Delta \geq 0$, as subcomputações que calculam as raízes a partir do Δ são criadas e, na linha 35, cadastradas como dependências da subcomputação que armazena as raízes. O código do ouvinte da remoção destas dependências é exibido na figura 11.

```
class RootsListener
    implements EdgeRemovalListener<Collection<Double>, OpRes> {
  static final RootsListener INSTANCE = new RootsListener();

  private RootsListener() {}

  @Override
  public void edgeRemoved(Collection<Double> roots,
                          OpRes root,
                          EdgeAppender<Collection<Double>> appender) {
    roots.add(root.result);
  }
}
```

Figura 11: classe `RootsListener`

3 Execução Especulativa

Na seção anterior as subcomputações $-b$ e $2a$ são criadas apenas se $\Delta \geq 0$. Podemos aumentar o desempenho do cálculo das raízes estruturando suas subcomputações como na figura 12.

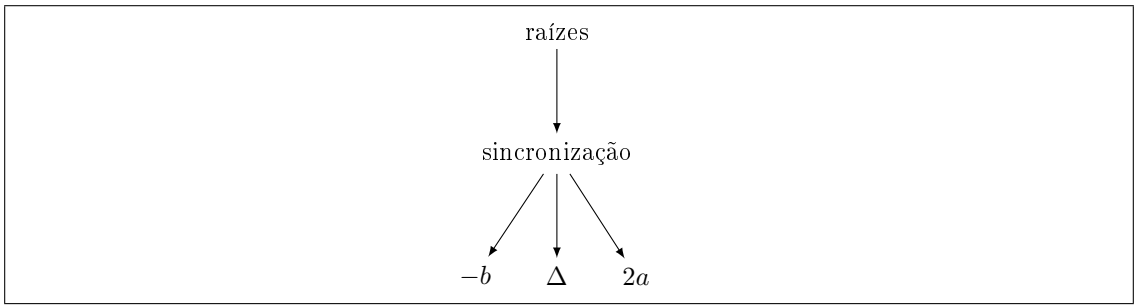


Figura 12: subcomputações do cálculo das raízes especulativo

A subcomputação sincronização tem como função armazenar o resultado de $-b$, Δ e $2a$. Quando sua execução for notificada, o ouvinte verificará se $\Delta \geq 0$. Em caso positivo, ele criará as subcomputações necessárias ao cálculo das raízes, ilustradas na figura 13, a partir dos resultados de $-b$, Δ e $2a$.

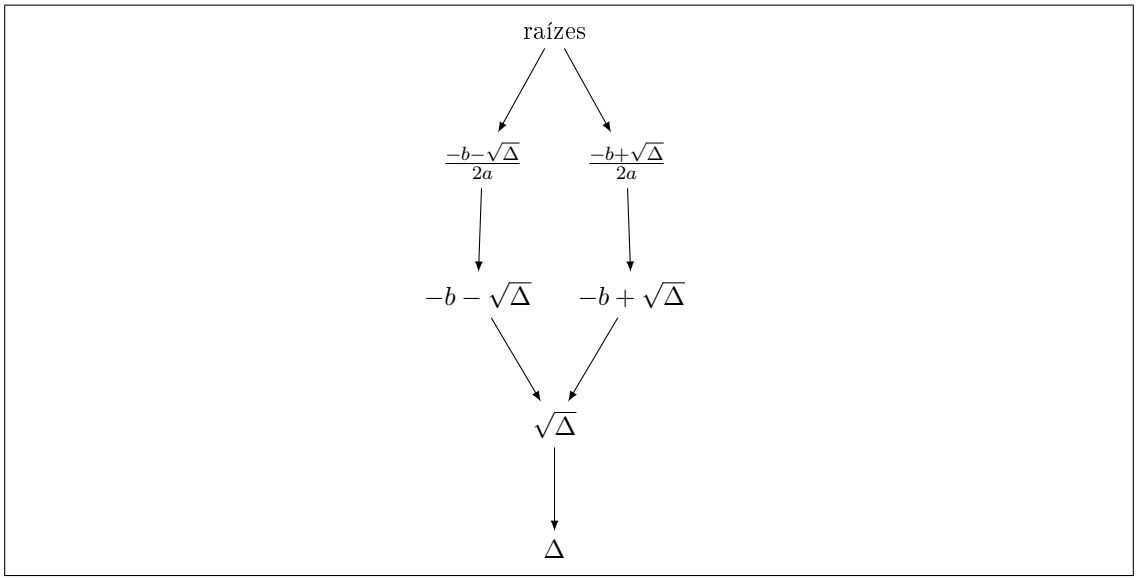


Figura 13: subcomputações do cálculo das raízes especulativo a partir de $-b$, Δ e $2a$

4 Bancos de Dados

O pacote `net.sf.afluentes.jdbc` contém classes que facilitam a execução paralela de comandos e consultas SQL — Structured Query Language. Para explicar seu uso, escreveremos um programa que importa compras efetuadas por clientes de uma loja hipotética, de arquivos XML — eXtensible Markup Language — para um banco de dados.

4.1 Modelo Entidade-Relacionamento

O modelo do banco de dados para o qual as compras serão importadas é exibido na figura 14. Eis uma descrição de suas entidades e relacionamentos:

- A entidade CUSTOMER representa os clientes da loja. Além de seu nome e endereço, cada cliente possui um número do seguro social, armazenado no atributo Ssn, que, tal como o CPF — Cadastro de Pessoas Físicas — no Brasil, o identifica unicamente.
- A entidade PRODUCT representa os produtos comercializados pela loja. Cada produto possui um número serial, que o identifica unicamente, e um nome.
- A entidade ORDER representa as compras feitas pelos clientes e deseja-se armazenar a data nas quais elas ocorreram.
- o relacionamento LINE_ITEM representa a ocorrência de um produto como um item de uma compra.

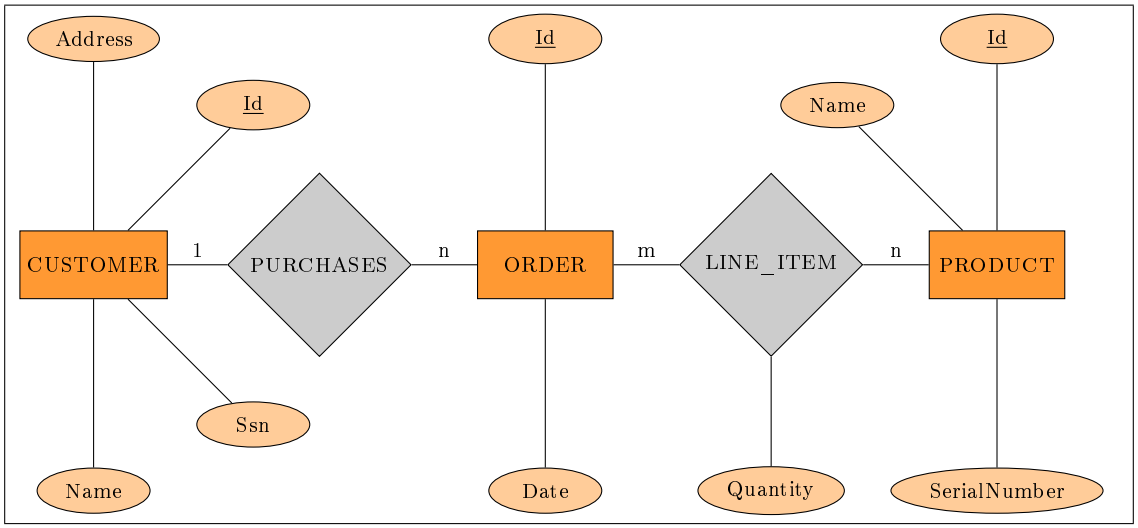


Figura 14: modelo entidade-relacionamento do banco de dados para o qual as compras serão importadas

4.2 Esquema SQL

A figura 15 exibe os comandos que criam o esquema SQL do banco de dados.

```
create table CUSTOMERS (ID integer not null auto_increment primary key,
  SSN integer not null unique,
  NAME varchar(255) not null,
  ADDRESS varchar(255) not null);

create table PRODUCTS (ID integer not null auto_increment primary key,
  SERIAL_NUMBER varchar(255) not null unique,
  NAME varchar(255) not null);

create table ORDERS (ID integer not null auto_increment primary key,
  DATE datetime, CUSTOMER_ID integer not null,
  foreign key (CUSTOMER_ID) references CUSTOMER(ID));

create table LINE_ITEMS (ORDER_ID integer not null, PRODUCT_ID integer not null,
  QUANTITY integer not null,
  primary key (ORDER_ID, PRODUCT_ID),
  foreign key (ORDER_ID) references ORDERS(ID),
  foreign key (PRODUCT_ID) references PRODUCT(ID));
```

Figura 15: esquema SQL do banco de dados para o qual as compras serão importadas

4.3 DTD dos Arquivos XML

A figura 16 mostra o DTD — Document Type Definition — dos arquivos XML que armazenam compras que serão importadas para o banco de dados. O atributo `customer` da tag `order` armazena o número do seguro social do cliente que efetuou a compra. Já

o atributo `product` armazena o número serial do produto adquirido. A figura 17 exibe um exemplo de arquivo XML que satisfaz este DTD.

```
<!DOCTYPE orders [
  <!ELEMENT orders (order+)>

  <!ELEMENT order (lineItem+)>
    <!ATTLIST order customer CDATA #REQUIRED>
    <!ATTLIST order date CDATA #REQUIRED>

  <!ELEMENT lineItem EMPTY>
    <!ATTLIST lineItem product CDATA #REQUIRED>
    <!ATTLIST lineItem quantity CDATA #REQUIRED>
]>
```

Figura 16: DTD dos arquivos XML que armazenam compras

```
<orders>
  <order customer="111111111" date="01/01/2011 13:00:00">
    <lineItem product="A-1" quantity="1"/>
  </order>
  <order customer="222222222" date="01/02/2011 14:00:00">
    <lineItem product="A-1" quantity="1"/>
    <lineItem product="B-2" quantity="2"/>
  </order>
  <order customer="333333333" date="01/03/2011 15:00:00">
    <lineItem product="A-1" quantity="1"/>
    <lineItem product="B-2" quantity="2"/>
    <lineItem product="C-3" quantity="3"/>
  </order>
  <order customer="444444444" date="01/04/2011 16:00:00">
    <lineItem product="A-1" quantity="1"/>
    <lineItem product="B-2" quantity="2"/>
  </order>
</orders>
```

Figura 17: exemplo de arquivo XML que armazena compras

```
interface OrdersHandler {
  public void startOrders();
  public void endOrders();

  public void startOrder(String customerName, Date date);
  public void endOrder();

  public void startLineItem(String productSerialNumber, int quantity);
  public void endLineItem();
}
```

Figura 18: interface `OrdersHandler`

4.4 Lendo os Arquivos XML

A interface `OrdersHandler`, cujo código é exibido na figura 18, define os eventos que ocorrem, no estilo SAX (Simple API for XML), durante a leitura dos arquivos XML que armazenam as compras.

A classe `OrdersHandlerAdapter`, cujo código é exibido na figura 19, converte os eventos SAX que ocorrem durante a leitura de arquivos XML que armazenam compras nos eventos definidos na interface `OrdersHandler`.

```
class OrdersHandlerAdapter extends DefaultHandler {
  private static final String ORDERS = "orders";
  /* ... */
  private DateFormat formatter = new SimpleDateFormat("MM/dd/yyyy hh:mm:ss");
  private OrdersHandler handler;

  OrdersHandlerAdapter(OrdersHandler handler) { this.handler = handler; }

  @Override public void startElement(String uri, String localName, String qName,
                                     Attributes attributes) throws SAXException {
    if (ORDERS.equals(qName)) {
      startOrders(attributes);
    } else if (ORDER.equals(qName)) {
      startOrder(attributes);
    /* ... *;
  }

  private void startOrders(Attributes attributes) { handler.startOrders(); }

  private void startOrder(Attributes attributes) throws SAXException {
    String customerName;
    Date date;

    customerName = attributes.getValue(ORDER_CUSTOMER);
    date = parseDate(attributes.getValue(ORDER_DATE));
    handler.startOrder(customerName, date);
  }

  private Date parseDate(String date) throws SAXException { /* ... */ }

  private void startLineItem(Attributes attributes) {
    String productSerialNumber;
    int quantity;

    productSerialNumber = attributes.getValue(LINE_ITEM_PRODUCT);
    quantity = Integer.parseInt(attributes.getValue(LINE_ITEM_QUANTITY));
    handler.startLineItem(productSerialNumber, quantity);
  }

  /* ... */
}
```

Figura 19: classe `OrdersHandlerAdapter`

A classe `OrdersParser`, cujo código é exibido na figura 20, utiliza um parser SAX para ler um arquivo XML que armazena compras e, à medida que ocorrem os eventos definidos na interface `OrdersHandler`, notifica a implementação passada como parâmetro.

```
class OrdersParser {
  void parse(OrdersHandler handler, InputSource input)
      throws ParserConfigurationException, SAXException, IOException {
    OrdersHandlerAdapter adapter;
    SAXParserFactory factory;
    SAXParser parser;

    adapter = new OrdersHandlerAdapter(handler);
    factory = SAXParserFactory.newInstance();
    factory.setValidating(true);
    parser = factory.newSAXParser();
    parser.parse(input, adapter);
  }
}
```

Figura 20: classe `OrdersParser`

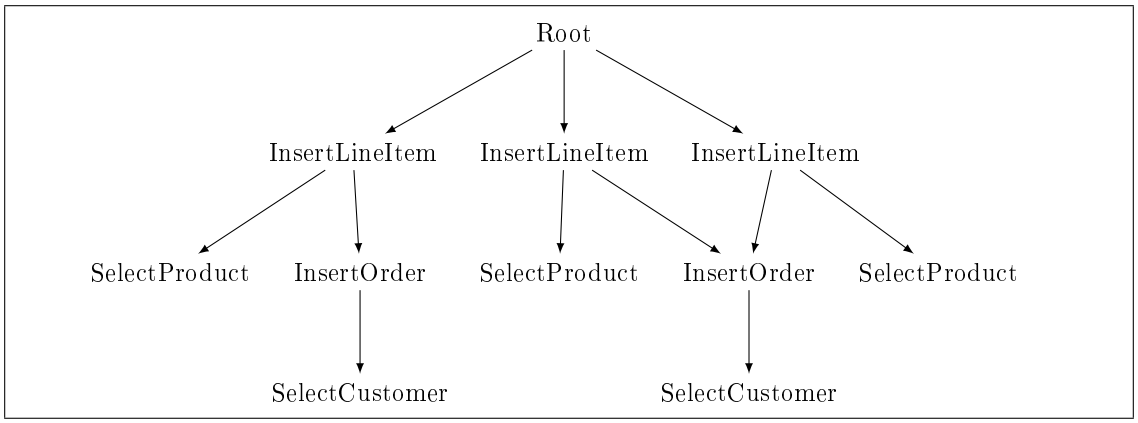


Figura 21: subcomputações da inclusão das duas primeiras compras

4.5 Importando os Arquivos XML

A inclusão no banco de dados das compras armazenadas em arquivos XML pode ser decomposta nas seguintes subcomputações:

- SelectCustomer — obtenção de um cliente a partir de seu número do seguro social;
- InsertOrder — inclusão de uma compra;
- SelectProduct — obtenção do ID de um protudo a partir de seu número serial;
- InsertLineItem — inclusão de um item de uma compra;

A figura 21 exemplifica como estas subcomputações podem ser utilizadas para incluir no banco de dados as duas primeiras compras contidas no arquivo exibido na figura 17.

A subcomputação InsertOrder depende da subcomputação SelectCustomer já que, devido às restrições de integridade do banco de dados, só é possível incluir uma compra após obtermos o ID do cliente que a efetuou a partir de seu número do seguro social. Pelo mesmo motivo, a subcomputação InsertLineItem depende das subcomputações InsertOrder e SelectProduct.

As figuras 22 e 23 exibem, respectivamente, o código das classes `Customer` e `SelectCustomerProcessor`. A primeira é responsável por encapsular os parâmetros e o resultado da subcomputação SelectCustomer, enquanto a última é responsável por efetuá-la.

```
class Customer {
  Integer id;
  Integer ssn;
  String name;
  String address;

  Customer(Integer ssn) {
    this.ssn = ssn;
  }
}
```

Figura 22: classes `Customer`

A classe `SelectProcessor<P, T>` facilita a implementação de processadores que efetuam consultas. O parâmetro `P` indica a classe do parâmetro da consulta (caso ela possua apenas um parâmetro) ou a classe que encapsula os parâmetros da consulta (caso ela possua mais de um parâmetro). O parâmetro `T` indica a classe do conteúdo dos vértices que serão processados.

O construtor da classe `SelectProcessor` recebe quatro parâmetros:

1. `dataSource` — um data source JDBC — Java Database Connectivity — que fornecerá a conexão com a qual a consulta será efetuada;
2. `queryPrefix` — o prefixo da consulta;
3. `parametersSeparator` — o separador dos parâmetros da consulta;
4. `querySuffix` — o sufixo da consulta.

Para criar uma subclasse concreta da classe `SelectProcessor<P, T>` é necessário implementar seus quatro métodos abstratos:

1. `P getParameters(T)` — extrai o(s) parâmetro(s) da consulta do conteúdo do vértice;
2. `void appendParameters(StringBuilder, T);` — concatena os parâmetros da consulta;

3. `P getParameters(ResultSet)` — extrai o(s) parâmetro(s) da consulta de seu resultado;

4. `void setResult(T, ResultSet)` — armazena o resultado da consulta.

```
class SelectCustomerProcessor extends SelectProcessor<Integer, Customer> {
  private static final String PREFIX = "select ID, SSN, NAME, ADDRESS" +
    " from CUSTOMERS where SSN in (";
  private static final String SUFFIX = ")";
  private static final String SEPARATOR = ", ";

  SelectCustomerProcessor(DataSource dataSource) {
    super(dataSource, PREFIX, SEPARATOR, SUFFIX);
  }

  @Override
  protected Integer getParameters(Customer customer) {
    return customer.ssn;
  }

  @Override
  protected void appendParameters(StringBuilder builder, Integer ssn) {
    JdbcUtils.append(builder, ssn);
  }

  @Override
  protected Integer getParameters(ResultSet result) throws SQLException {
    return result.getInt("SSN");
  }

  @Override
  protected void setResult(Customer customer,
      ResultSet result) throws SQLException {
    customer.id = result.getInt("ID");
    customer.ssn = result.getInt("SSN");
    customer.name = result.getString("NAME");
    customer.address = result.getString("ADDRESS");
  }
}
```

Figura 23: classes `SelectCustomerProcessor`

As figuras 24 e 25 exibem, respectivamente, o código das classes `Order` e `InsertOrderProcessor`. A primeira é responsável por encapsular os parâmetros e o resultado da subcomputação InsertOrder enquanto a última é responsável por efetuá-la.

A classe `GeneratedKeysProcessor<T>` facilita a implementação de processadores que efetuam comandos SQL de inclusão. O parâmetro `T` indica a classe do conteúdo dos vértices que serão processados.

O construtor da classe `GeneratedKeysProcessor` recebe dois parâmetros:

1. `dataSource` — um data source JDBC que fornecerá a conexão com a qual a inclusão será efetuada;

2. `command` — o comando de inclusão que será efetuado.

Para criar uma subclasse concreta da classe `GeneratedKeysProcessor<T>` é necessário implementar seus dois métodos abstratos:

1. `setParameters(PreparedStatement, T)` — define os parâmetros do comando de inclusão;

2. `setGeneratedKey(T, ResultSet);` — armazena a chave primária gerada pelo banco de dados.

```
class Order {
  Integer id;
  Date date;
  Integer customerId;

  Order(Date date) {
    this.date = date;
  }
}
```

Figura 24: classes `Order`

```
class InsertOrderProcessor extends GeneratedKeysProcessor<Order> {
  private static final String COMMAND = "insert into ORDERS (DATE, CUSTOMER_ID)" +
    " values (?, ?)";

  InsertOrderProcessor(DataSource dataSource) {
    super(dataSource, COMMAND);
  }

  @Override
  protected void setParameters(PreparedStatement statement,
                               Order order) throws SQLException {
    statement.setTimestamp(1, new Timestamp(order.date.getTime()));
    statement.setInt(2, order.customerId);
  }

  @Override
  protected void setGeneratedKey(Order order, ResultSet result) throws SQLException {
    order.id = result.getInt(1);
  }
}
```

Figura 25: classes `InsertOrderProcessor`

A classe `GraphBuilder`, cujo código é exibido na figura 26, cria as subcomputações que irão incluir no banco de dados as compras armazenadas em um arquivo XML à medida que ele é lido.

```
class GraphBuilder implements OrdersHandler {
  VertexBuilder<Object> rootBuilder;
  /* ... */

  private Vertex<Object> root;
  /* ... */

  Vertex<Object> getRoot() { return root; }

  @Override public void startOrders() { rootBuilder.createVertex(null); }

  @Override public void endOrders() { root = rootBuilder.getVertex(); }

  @Override public void startOrder(int customerSsn, Date date) {
    Customer customer;
    Order order;

    customer = new Customer(customerSsn);
    selectCustomerBuilder.createVertex(customer);
    selectCustomer = selectCustomerBuilder.getVertex();

    order = new Order(date);
    insertOrderBuilder.createVertex(order);
    insertOrderBuilder.appendEdge(selectCustomer, CustomerSelected.INSTANCE);
    insertOrder = insertOrderBuilder.getVertex();
  }

  @Override public void startLineItem(String productSerialNumber, int quantity) {
    Product product;
    LineItem lineItem;

    product = new Product(productSerialNumber);
    selectProductBuilder.createVertex(product);
    selectProduct = selectProductBuilder.getVertex();

    lineItem = new LineItem(quantity);
    insertLineItemBuilder.createVertex(lineItem);
    insertLineItemBuilder.appendEdge(insertOrder, OrderInserted.INSTANCE);
    insertLineItemBuilder.appendEdge(selectProduct, ProductSelected.INSTANCE);
    insertLineItem = insertLineItemBuilder.getVertex();

    rootBuilder.appendEdge(insertLineItem);
  }

  /* ... */
}
```

Figura 26: classe `GraphBuilder`

A figura 27 exibe o código dos ouvintes de remoção de dependência utilizados pela classe `GraphBuilder`. Eles têm como função propagar o ID que foi consultado ou gerado por uma subcomputação para as subcomputações dependentes.

```
class CustomerSelected implements EdgeRemovalListener<Order, Customer> {
  static final CustomerSelected INSTANCE = new CustomerSelected();

  private CustomerSelected() {}

  @Override
  public void edgeRemoved(Order dependent,
                          Customer dependable,
                          EdgeAppender<Order> appender) {
    dependent.customerId = dependable.id;
  }
}

class OrderInserted implements EdgeRemovalListener<LineItem, Order> {
  static final OrderInserted INSTANCE = new OrderInserted();

  private OrderInserted() {}

  @Override
  public void edgeRemoved(LineItem dependent,
                          Order dependable,
                          EdgeAppender<LineItem> appender) {
    dependent.orderId = dependable.id;
  }
}

class ProductSelected implements EdgeRemovalListener<LineItem, Product> {
  static final ProductSelected INSTANCE = new ProductSelected();

  private ProductSelected() {}

  @Override
  public void edgeRemoved(LineItem dependent,
                          Product dependable,
                          EdgeAppender<LineItem> appender) {
    dependent.productId = dependable.id;
  }
}
```

Figura 27: classes `CustomerSelected`, `OrderInserted` e `ProductSelected`

```
class OrdersImporter {
  void importOrders(DataSource dataSource, InputSource inputSource)
      throws SQLException, /* ... */ {
    Connection connection;
    SingletonDataSource updateDataSource;
    ExecutorServiceFactory factory;
    GraphReducer reducer;
    GraphBuilder builder;
    OrdersParser parser;
    Vertex<Object> root;

    connection = null;
    try {
      connection = dataSource.getConnection();
      connection.setAutoCommit(false);
      updateDataSource = new SingletonDataSource(connection);

      factory = new FixedThreadPoolFactory(3);

      reducer = new GraphReducerImpl(factory);

      builder = new GraphBuilder();
      builder.rootBuilder = reducer.addProcessor(
        new SingleVertexProcessor<Object>());
      builder.selectCustomerBuilder = reducer.addProcessor(
        new SelectCustomerProcessor(dataSource), 50);
      builder.selectProductBuilder = reducer.addProcessor(
        new SelectProductProcessor(dataSource), 50);
      builder.insertOrderBuilder = reducer.addProcessor(
        new InsertOrderProcessor(updateDataSource), 50);
      builder.insertLineItemBuilder = reducer.addProcessor(
        new InsertLineItemProcessor(updateDataSource), 50);

      parser = new OrdersParser();
      parser.parse(builder, inputSource);
      root = builder.getRoot();

      reducer.reduceGraph(root);
    } catch (SQLException exception) {
      JdbcUtils.rollback(connection);
      throw exception;
    /* ...*/
    } finally {
      JdbcUtils.commit(connection);
      JdbcUtils.close(connection);
    }
  }
```

Figura 28: classe `OrdersImporter`