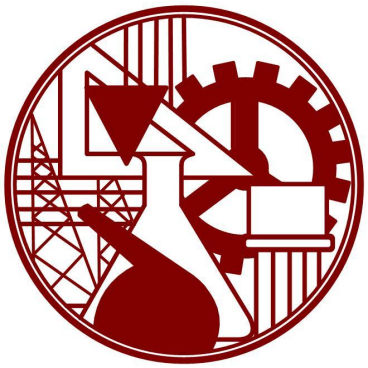


INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Configuração e Interrogação de Sistemas de Informação Federados

Hugo Miguel Ferreira da Cruz
Licenciado

Dissertação de natureza científica realizada para obtenção do grau de Mestre em Engenharia Informática e de Computadores

(Documento Provisório)

Orientador: Paulo Manuel Trigo Cândido da Silva

Setembro de 2010

Resumo

Apesar da existência de produtos comerciais e investigação na área, a implementação de sistemas de informação com diversos componentes distribuídos e heterogéneos, conhecidos como sistemas de informação federados, é ainda um desafio. Desde logo, existe o desafio de modelar estes sistemas, já que modelos de dados como o modelo relacional não incluem informação sobre a distribuição e resolução de heterogeneidade. Outro problema é a interacção com estes sistemas de informação, nomeadamente o problema de realizar interrogações sobre os diversos componentes dos sistemas sem ser necessário conhecer os detalhes dos mesmos. Com esta dissertação, pretende-se apresentar uma solução para estes problemas, através da utilização da *Ontology Web Language* (OWL) para modelar uma descrição unificada dos seus diversos componentes. Sobre este modelo, é criado um componente de *software* que executa interrogações sobre o sistema federado, resolvendo os problemas de distribuição e heterogeneidade existentes.

Palavras-chave: Sistemas de Informação, Sistemas Federados, Integração, Ontology Web Language, Lógica de Descrição

CONTEÚDO

CAPÍTULO 1

INTRODUÇÃO

A maioria dos produtos, e mesmo muitas soluções *ad-hoc*, utilizam os seus próprios repositórios de dados, tipicamente bases de dados. No entanto, as organizações utilizam outros repositórios de dados como XML (*eXtended Markup Language*), folhas de cálculo ou mesmo informação disponibilizada através de *web services*. Mesmo quando todas as fontes de dados envolvidas utilizam o modelo relacional, existem problemas provocados pela utilização de produtos de diferentes fabricantes, como a falta de comunicação ou a utilização de formatos de dados incompatíveis. Sistemas de informação com estes requisitos são conhecidos como sistemas de informação federados [1] e são constituídos por uma série de fontes de dados com as seguintes características:

- Distribuição: as fontes de dados podem-se encontrar distribuídas fisicamente;
- Heterogeneidade: as fontes de dados podem ser heterogéneas não só na tecnologia utilizada mas também nos conceitos e na própria modelação dos dados;
- Autonomia: cada uma das fontes de dados foi criada independentemente do sistema federado e continua a funcionar independentemente deste.

No entanto, outros problemas surgem ao construir este tipo de sistemas. Um desses problemas é o facto de não existir um modelo formal que permita descrever os dados envolvidos. É necessário ter informação não só sobre a estrutura dos dados, mas também sobre a distribuição dos mesmos e sobre o contributo de cada fonte de dados para o sistema global. Outro problema é evitar que a solução construída seja mais uma solução *ad-hoc*.

Nesta dissertação, colocam-se duas hipóteses para dar resposta a estes problemas:

- Hipótese 1: Os formalismos que suportam descrição semântica e têm capacidade para realizar inferência permitem responder aos requisitos de desenho de um novo modelo.
- Hipótese 2: A adopção de um standard (SQL) de interrogação do novo modelo permite ter uma visão uniforme (homogénea) sobre modelos heterogéneos.

No capítulo 2 começa-se por apresentar os conceitos, problemas e soluções existentes dos sistemas de informação federados. No capítulo 3 é descrito um modelo de dados que serve para validar a hipótese 1, seguindo-se, no capítulo 4, a descrição de uma solução de interrogação do sistema descrito por esse modelo, como validação da hipótese 2. O capítulo 5 apresenta a implementação de uma aplicação gráfica que auxilia na configuração e teste de um sistema de informação federado baseado nas hipóteses apresentadas. Segue-se a apresentação, no capítulo 6, de um exemplo complecto que ajude a avaliar as soluções apresentadas. O relatório termina com as conclusões do trabalho de dissertação.

CAPÍTULO 2

SISTEMAS DE INFORMAÇÃO FEDERADOS

A implementação de sistemas de informação federados levanta uma série de problemas que têm sido alvo de investigação desde o início dos anos 80 [2]. Desde então, acompanhando as evoluções nas várias áreas relacionadas (bases de dados [3], integração de sistemas [4] [5], comunicações), o foco da investigação tem variado muito ao longo dos anos. Nos anos mais recentes, a arquitectura proposta em [1] e apresentada na figura 2.1 tem sido sido adoptada como referência por vários autores e projectos de investigação [6] [7] [8].

A investigação mais recente tem-se focado em outros aspectos deste tipo de sistemas, nomeadamente:

- Definição de um modelo de dados global [9] [10] [11];
- Criação automática de modelos de dados globais a partir dos modelos de dados das fontes de dados participantes na federação [7];
- Processamento de interrogações [6] [5];
- Detecção e processamento de informação semântica como, por exemplo, informação repetida entre fontes de dados [4];
- Controlo de transações [3].

Existem alguns produtos comerciais e projectos open-source para a implementação de sistemas federados. No entanto, alguns utilizam o termo "sistema federado" com outros significados: por exemplo, a definição utilizada pela no produto Microsoft SqlServer *SqlServer* é a de um conjunto de servidores com partições horizontais de dados [12]). Outros produtos disponibilizam funcionalidades especialmente viradas para as bases de dados e para a *Web* (vista como uma colecção de páginas *html*), ignorando outros tipos de fontes de dados.

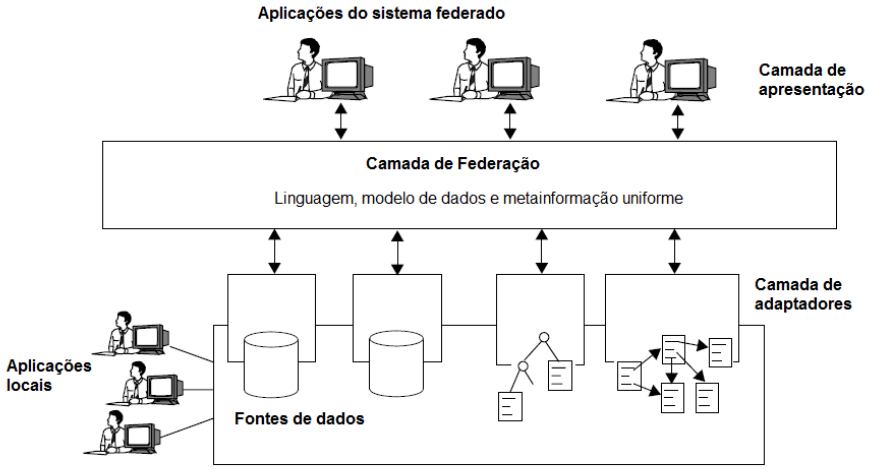


Figura 2.1: Arquitectura de referência (adaptado de [1])

Segue-se a análise de alguns produtos que se apresentam como referências na área e que apresentam soluções mais complectas.

2.1 IBM DB2 [13]

Talvez a implementação mais completa dos conceitos seja a disponibilizada pela IBM através do seu sistema de gestão de bases de dados DB2.

Uma instância do SGBD BD2 assume o papel de servidor central e assume as funções de camada federada para o sistema. Esta camada proporciona transparência de localização, esquema e linguagem e contempla um conjunto de fontes de dados relacionais (Db2, SqlServer, Oracle) e não relacionais (Excel, WebServices) através de um conjunto de wrappers distribuídos com o produto. Estes wrappers são componentes de software que são invocados pelo servidor para fazer a comunicação com cada uma das fonte de dados. É da responsabilidade destes a tradução da entrada (em linguagem SQL) para as operações a realizar efectivamente, seja a tradução para outra linguagem ou para um conjunto de operações a realizar (por exemplo, um conjunto de chamadas a funções). As fontes de dados têm, no entanto, de ser representadas sobre a forma tabular no servidor federado. Note-se também que todas as fontes de dados não relacionais apenas suportam operações de leitura. A autonomia das fontes é mantida excepto quando o contexto de uma transacção requer que essa transacção corra na fonte de dados.

O facto de todo o controlo correr num servidor central torna necessária a existência de uma máquina com grande capacidade para não comprometer a disponibilidade do sistema federado. Não existem também muitas possibilidades de definição de alguns tipos de meta-dados, como por exemplo para definir relações e restrições entre participantes. Para além disso, as partições/replicações de dados só são suportados indirectamente através de vistas, que estão limitadas a operações de leitura.

O modelo de dados global é definido tal como o de uma base de dados comum no sistema DB2, através da linguagem DDL do SGBD. É então utilizado o modelo relacional com as habituais extensões de tipos (que permitem campos com ficheiros binários, xml, etc). Também não existe processamento semântico da informação e o processamento transaccional está limitado pelas capacidades dos adaptadores.

2.2 Sybase Adaptive Server Enterprise [14]

Um sistema de informação federado baseado neste no *Sybase Adaptative Server* assenta numa base de dados local que contém proxy tables. Estas proxy tables contêm a meta-informação importada das fontes originais dos dados e permitem aceder às fontes de dados através de interrogações em linguagem SQL. Quando uma tabela da vista global é constituída por mais de uma tabela, a solução passa pela criação de vistas, com a limitação de essas vistas apenas permitirem operações de leitura.

O suporte a fontes de dados para além de bases de dados é realizada através de ligações a outros produtos do mesmo fabricante. Todas estas fontes de dados são mapeadas para o formato tabular para manter a consistência com os restantes esquemas parciais. Por exemplo, é possível criar uma proxy table baseada num sistemas de ficheiros, sendo criada uma tabela com colunas como o nome do ficheiro, a sua localização, meta-informação (datas de criação, acesso, tamanho, etc) e conteúdo (pesquisável como texto).

2.3 UniSql

O UniSql é um SGBD orientado a objectos que permite a criação de um sistema de informação federado através de objectos proxy. A linguagem de interrogação utilizada é a mesma que é utilizada para as interrogações realizadas sobre os objectos: SQL/X. A comunicação com outras fontes de dados é feita através de drivers já incluídos na distribuição. Por ser um SGBD orientado a objectos, permite modelações distintas das possíveis nas soluções baseadas no modelo relacional puro. É possível, por exemplo, a execução de algum processamento semântico dos dados.

2.4 GaianDB [15]

O GaianDB [15] é outro projecto da IBM, desta vez disponibilizado como uma biblioteca open-source escrita em Java, e que apresenta soluções para sistemas muito dinâmicos. No entanto, o seu modelo de dados não é público e tem poucas opções de especificação de distribuição de dados: só é possível definir partições horizontais de dados. Também são reutilizadas as soluções já existentes para bases de dados, como o modelo relacional e a criação de vistas sobre as fontes de dados externas. A heterogeneidade é resolvida através da utilização de objectos especiais como *proxies* para outras fontes de dados.

2.5 A implementação proposta

Mantendo a arquitectura apresentada na figura 2.1, pretende-se manter o modelo de aplicações que utilizam bases de dados como fonte de dados. Para isso, são mantidos os conceitos do modelo relacional, com o SQL como linguagem de interrogação, permitindo o acesso através de interfaces como o *jdbc* ou mesmo de técnicas de *object-relational mapping*. O modelo de dados é, no entanto, descrito numa taxonomia *OWL*, pretendendo-se tirar partido tirar partido de alguma da expressividade adicional disponibilizada e permitir trabalho futuro na definição de semântica adicional dos modelos.

A camada de federação é implementada numa componente de código *Java* que permite realizar interrogações SQL sobre o modelo de dados definido. O acesso à fontes de dados é realizado através de adaptadores com uma interface aberta e que têm como requisitos receber interrogações *sql* e retornar objectos *RowSet*, podendo ser facilmente implementados novos adaptadores.

CAPÍTULO 3

O MODELO DE REPRESENTAÇÃO DO SISTEMA

3.1 Objectivos e Possíveis Soluções

A modelação de um sistema de informação federado partilha algumas das características da modelação de um sistema de informação comum mas apresenta também outros desafios. Nomeadamente, é essencial modelar:

- O modelo de dados público (global) do sistema federado;
- A estrutura das fontes de dados envolvidas (os modelos de dados parciais), tendo em conta a heterogeneidade das mesmas;
- A contribuição de cada fonte de dados para o sistema federado.

Outros tipos de informação podem ser incluídos na construção de um sistema federado (qualidade, semântica) mas não são essenciais para a disponibilidade do sistema, pelo que apenas a informação essencial foi considerada no modelo criado.

Para a criação deste modelo foram consideradas algumas alternativas utilizadas em implementações destes sistemas e apresentadas em artigos da área:

- Modelo relacional: utilizado por várias implementações, principalmente por sistemas de gestão de bases de dados que também suportam sistemas federados. A escolha deste modelo permite a reutilização das ferramentas já existentes para utilização com bases de dados, mas não permite muita flexibilidade na definição de qual a participação das fontes de dados no sistema federado.

- Modelo de objectos: os modelos de dados das fontes e da visão global dos dados são definidos através de classes. A heterogeneidade e o contributo das fontes para o sistema são resolvidos através das hierarquias das classes e de código executável. Este modelo é muito flexível, mas tem como desvantagens misturar a estrutura dos dados com código e a dificuldade de o modelo poder ser configurado por uma pessoa especialista na modelação dos dados mas que não conheça as particularidades da linguagem em que as classes. estão implementadas.
- Um novo modelo de dados para o problema: independentemente da implementação do modelo (XML, OWL ou outro), o objectivo é ter um modelo de dados separado da implementação. Neste caso é importante a existência de uma aplicação de configuração do sistema que utilize conceitos familiares aos especialistas do desenho dos sistemas de informação, para que não seja necessário conhecer a linguagem de implementação do modelo.

Optou-se pela criação de um modelo de dados sobre a linguagem OWL, que até ser guardado como um ficheiro XML, mas permite tirar partido da maior expressividade da OWL. Esta linguagem tem três sub-linguagens: *OWL Full*, *OWL-DL* e *OWL Lite*, por ordem decrescente de capacidades descritivas. No entanto, os motores de inferência que suportam a sub-linguagem *OWL Full* não garantem que seja possível chegar a resultados em tempo útil, pelo que se optou por utilizar manter o modelo dentro dos limites da *OWL-DL*. Esta sub-linguagem tem a sua base na lógica de descrição, mais concretamente na linguagem SHOIN^{D}:

- S - abreviatura da linguagem ALC, com papeis (propriedades) transitivos:
  - AL - *Attributive Language* - negação de conceitos atómicos, intersecções, uniões, restrições, qualificação existencial limitada;
  - C - negação de conceitos;
- H - hierarquia de papeis (propriedades);
- O - nominais, restrições sobre valores;
- I - propriedades inversas;
- N - restrições de cardinalidade;
- D - utilização de tipos XML.

Para a modelação do sistema são criadas duas taxonomias: uma para representar as fontes de dados (os modelos parciais) e outra para representar o modelo de dados do sistema federado (o modelo global). A figura 3.1 apresenta uma visão geral do mapeamento entre estas taxonomias.

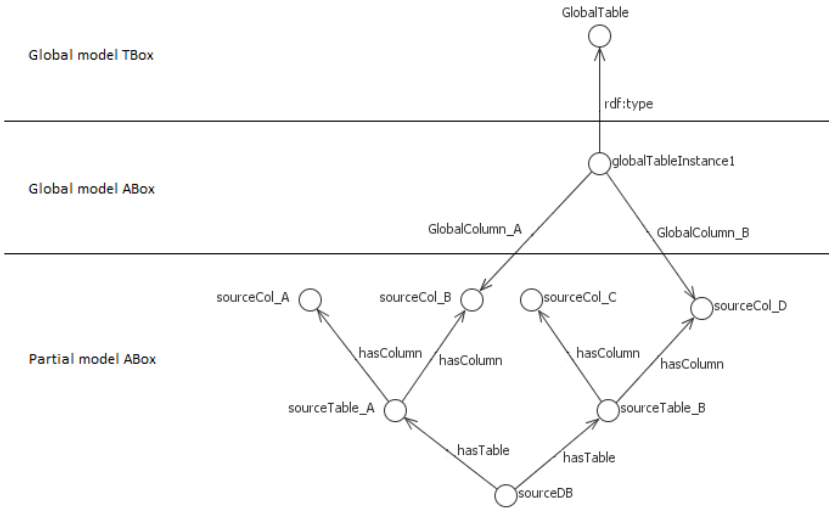


Figura 3.1: Visão geral do mapeamento entre as taxonomias

3.2 O Modelo das Fontes de Dados

Os modelos parciais são descritos sobre uma versão alterada da taxonomia *relational.owl*[16]. Esta taxonomia permite descrever a estrutura de fontes de dados relacionais, pelo que as fontes de dados não relacionais têm de ser mapeadas para este modelo. As alterações efectuadas têm três objectivos:

1. Remover a informação não relevante para a realização das operações de leitura;
2. Introduzir informação para o acesso às fontes de dados, incluindo a sua localização, o adaptador a utilizar nesse acesso e as credenciais necessárias.

Para o acesso às fontes de dados, é necessária mais informação. Para cada fonte de dados, é necessário saber a sua localização (um Unique Resource Identifier), qual o adaptador a usar para aceder a essa fonte e, possivelmente, as credenciais necessárias (username e password ou apenas um dos dois). Como o acesso às colunas pode não poder ser feito através pelo seu nome, é registada informação que permita o seu acesso. O formato desta informação não é limitado pelo modelo, mas tem de ser compreendido pelo adaptador da fonte de dados respectiva. A TBox para esta informação está na listagem 1.

A TBOX desta taxonomia é fixa (listagem 1), sendo toda a informação inserida como parte da ABox.

```
1 dbs:Database ⊑ rdf:Bag
2 dbs:Table ⊑ rdf:Seq
3 dbs:Column ⊑ rdf:Resource
4
```

```
dbs:hasTable (dbs:Database, dbs:Table)
dbs:hasColumn (dbs:Table, dbs:Column)

dbs:provider (dbs:Database, xsd:string)
dbs:uri (dbs:Database, xsd:string)
dbs:username (dbs:Database, xsd:string)
dbs:password (dbs:Database, xsd:string)
dbs:columnAccess (dbs:Column, xsd:string)
```

Listagem 1: TBox relational.owl modificada

3.3 O Modelo de Dados Global

O modelo de dados global é representado pela TBox de uma outra taxonomia. Um modelo de dados relacional pode ser mapeado para esta TBOX seguindo apenas duas regras:

1. Para cada tabela, é criada uma subclasse de "fm:FederatedEntity".
2. Para cada coluna dessa tabela, é criada uma propriedade com a classe correspondente à tabela como domínio e com o conceito "dbs:Column" como contradomínio.

A TBox resultante é semelhante à TBox das taxonomias utilizada para conter dados na ABox, como é descrito em [17]. Neste caso, a ABox vai ser utilizada de forma diferente, nomeadamente para fazer o mapeamento com as fontes de dados. O exemplo da listagem 2 contém uma TBox que representa uma tabela "Product" com as colunas "prodId" e "prodDesc".

```
fm:FederatedEntity ⊑ rdf:Bag
fm:Product ⊑ fm:FederatedEntity

fm:prodId (fm:Product, dbs:Column)
fm:prodDesc (fm:Product, dbs:Column)
```

Listagem 2: Exemplo de uma TBox de um esquema global

3.4 Mapeamento entre os Modelos

O mapeamento entre os modelos permite modelar:

- Partições horizontais;
- Partições verticais;
- Réplicas;
- Colunas constantes;

- Colunas calculadas.

A ABox da taxonomia do modelo global é o ponto de conexão entre o modelo global e os diversos modelos parciais.

3.5 Partições Horizontais

No cenário mais simples, o mapeamento é feito criando uma instância da classe referente a uma tabela global e mapeando directamente as colunas de ambos os modelos, como apresentado na listagem 3.

```
1 fm:Product (productInstance1)
2 fm:prodId (productInstance1, dbSource1.product.id)
3 fm:prodDesc (productInstance1, dbSource1.product.description)
```

Listagem 3: Exemplo de ABox de mapeamento de modelos

Neste exemplo, cada instância de "fm:Product" representa uma partição horizontal da tabela correspondente. Se existirem duas instâncias, os dados correspondentes à tabela serão a união dos dados obtidos através de cada instância.

3.6 Partições Verticais

Para representar uma partição vertical, o primeiro passo é realizar o mapeamento das colunas envolvendo várias tabelas dos modelos parciais. As colunas podem ter origem em tabelas diferentes da mesma fonte de dados ou mesmo de fontes de dados diferentes. Cada tabela envolvida será vista como uma partição vertical dos dados.

O passo extra neste caso é o estabelecimento da relação existente entre as partições verticais. Esta relação é semelhante a uma chave estrangeira e a informação registada é:

1. A tabela esquerda da relação;
2. A tabela direita da relação;
3. Pares de colunas que irão constituir a condição de junção.

A TBox relativa a esta informação é apresentada na listagem 4.

```
1 fm:FederatedRelation ⊑ rdf:Bag
2 fm:ColumnRelation ⊑ rdf:Bag
3
4 fm:implicitJoin (fm:FederatedEntity, fm:FederatedRelation)
5 fm:tableLeft (fm:FederatedRelation, dbs:Table)
6 fm:tableRight (fm:FederatedRelation, dbs:Table)
7
8 fm:relatedColumns (fm:FederatedRelation, fm:ColumnRelation)
```

```
9  fm:fromColumn (fm:ColumnRelation, dbs:Column)
10 fm:toColumn (fm:ColumnRelation, dbs:Column)
```

Listagem 4: TBox para registar relações entre tabelas

3.7 Réplicas

Para além das partições, é possível definir réplicas. As réplicas são definidas ao nível das instâncias das tabelas do modelo global. Isto é, pode-se indicar que duas partições horizontais são réplicas. Esta definição é suportada pela propriedade "fm:replic", cuja definição é apresentada na listagem 5.

```
1 fm:replic (fm:FederatedEntity, fm:FederatedEntity)
2
3 #replic is symmetric and transitive
4 fm:replic$^{-}_{+}$
```

Listagem 5: TBox para o modelamento de réplicas

3.8 Constantes

Quando as fontes de dados são heterogéneas, é comum existir informação em falta ou em formatos incompatíveis. Para ultrapassar estas barreiras é possível inserir valores constantes ou colunas calculadas através da execução de funções.

Para o primeiro caso, foi criada uma fonte de dados especial para conter apenas a informação das constantes. Esta fonte de dados não tem existência física e representa uma tabela virtual apenas com uma linha, que contém o valor da constante. Por essa razão, esta fonte de dados tem de ser tratada de forma diferente pelas aplicações implementada sobre o modelo. Por outro lado, esta solução permite alguma liberdade sem quebrar os princípios já existentes no resto do modelo. A descrição desta fonte de dados está na listagem 6.

```
1 dbs:Database (srcConstants)
2 dbs:Table (srcConstants.constantsTable)
3 dbs:Column (srcConstants.constantsTable.int#3)
4 dbs:Column (srcConstants.constantsTable.string#NA)
```

Listagem 6: Exemplo de uma ABox para a fonte de dados de constantes

Ao inserir colunas de constantes com colunas de outras fontes de dados, estamos a formar partições verticais, e é necessário estabelecer a relação entre as tabelas. Neste caso, como a tabela com as constantes pode ser vista como uma tabela apenas com uma linha, queremos juntar esse valor a todos as linhas da outra partição vertical. Isto é conseguido com o estabelecimento da relação entre as tabelas das partições verticais sem indicar colunas para estabelecer a

condição de junção, equivalente, na prática, a um *cross join*. Um exemplo do estabelecimento de uma relação deste tipo é apresentado na listagem 7.

```
1  fm:Product (prodInstance1)
2  fm:prodId (prodInstance1, dbSource1.product.id)
3  fm:prodDesc (prodInstance1, srcConstants.constantsTable.string#NA)
4
5  fm:FederatedRelation (fm:relationWithConstants)
6  fm:tableLeft (fm:relationWithConstants, dbSource1.product)
7  fm:tableRight (fm:relationWithConstants,
       srcConstants.constantsTable)
8  fm:relatedColumns (⊥)
9
10 fm:implicitJoin(prodInstance1, fm:relationWithConstants)
```

Listagem 7: Exemplo de uma ABox com utilização de constantes

3.9 Funções

Para a utilização de colunas calculadas, é necessário modelar a execução de funções que executem as transformações pretendidas. Modelar o comportamento destas funções na taxonomia daria origem a um modelo complexo e potencialmente incompleto. A solução adoptada passa por ter na taxonomia a indicação de qual a função a executar e quais as colunas que são utilizadas como parâmetros, como é apresentado na listagem 8. O comportamento das funções é da responsabilidade das aplicações construídas sobre o modelo.

```
1 fm:FunctionOperation ⊑ ⊤
2 fm:Function ⊑ ⊤
3
4 fm:operation (fm:Function, fm:FunctionOperation)
5 fm:arguments (fm:Function, dbs:Column)
```

Listagem 8: TBox para modelação de execução de funções

As funções disponíveis são as instâncias da classe fm:FunctionOperation. Instâncias de fm:Function são execuções concretas de funções. Os parâmetros são mapeados através da propriedade fm:arguments. Tal como no mapeamento directo de colunas, as colunas usadas podem ser provenientes de tabelas diferentes, desde que a relação entre as tabelas esteja definida.

O contradomínio das propriedades correspondentes às colunas do modelo global será a união de dbs:Column (mapeamento directo de colunas) e fm:Function (mapeamento através da execução de funções), como apresentado na listagem 9.

```
1 fm:prodId (fm:Product, dbs:Column ⊔ fm:Function)
```

Listagem 9: Exemplo da TBox de uma coluna com suporte a funções

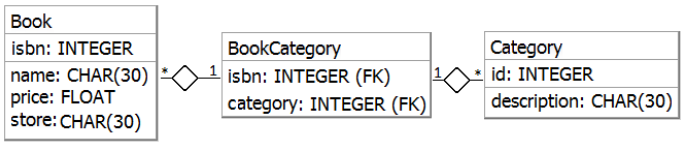


Figura 3.2: Diagrama do modelo de dados utilizado pelo *webste*

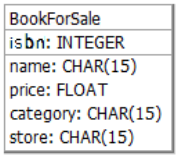


Figura 3.3: O novo modelo de dados global para ser utilizado nas pesquisas de livros

3.10 Exemplo

O exemplo apresentado de seguida é um exemplo minimal mas funcional, que tem como objectivo mostrar como cada elemento do modelo é materializado. Não se pretende que este exemplo apresente um caso real. Neste exemplo, um *website* realiza vendas de uma cadeia de livrarias e usa uma base de dados relacional para guardar a informação dos livros disponíveis. O diagrama da figura 3.2 mostra o modelo de dados já em utilização.

Pretende-se que este *website* passe também a vender livros cuja informação é originada de uma rede social. Esta rede tem a informação dos livros disponíveis registada numa folha de cálculo com apenas três colunas <isbn, name, price>.

O novo modelo de dados global, para ser utilizado nas pesquisas de livros, é apresentado na figura 3.3.

3.10.1 Modelos parciais

Criar a ABOX que modela a fonte de dados relacional do *website* é bastante simples, já que os conceitos do modelo são mantidos. A figura 3.4 apresenta a ABOX resultante.

Para a folha de cálculo, o primeiro passo a realizar é mapear os seus conceitos para o modelo relacional. A figura 3.5 apresenta a ABOX que representa a folha de cálculo, em que os conceitos foram mapeados segundo as seguintes regras:

- um ficheiro de folha de cálculo é uma base de dados no modelo relacional;
- uma folha dentro de um ficheiro é uma tabela no modelo relacional

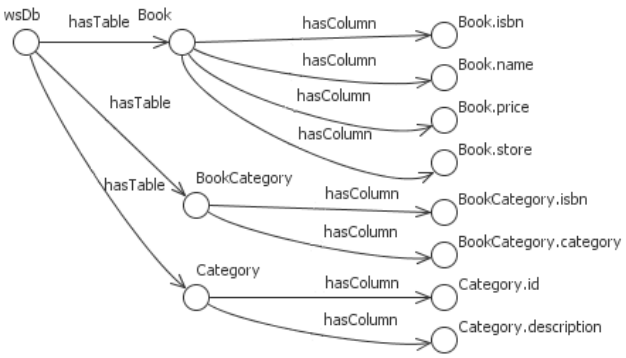


Figura 3.4: ABox do esquema parcial que descreve a base de dados do *website*

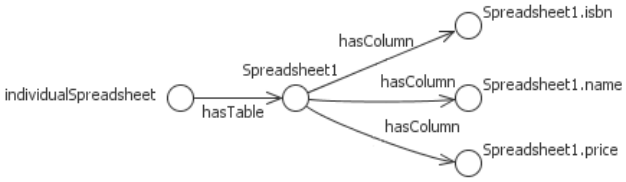


Figura 3.5: ABox of partial schema description for spreadsheets

- uma coluna continua a ser uma coluna no modelo relacional.

3.10.2 Modelo global

Para a descrição do modelo global, é criada uma TBOX. Para cada tabela, é criada uma subclasse de *fm:FederatedEntity*. Para cada coluna de uma tabela, é criada uma propriedade que tem a classe correspondente à tabela como domínio. O contradomínio destas propriedades é sempre a união entre *dbs:Column* e *fm:Function*. A TBOX do cenário de exemplo é apresentada na listagem 10.

```
1 fm:FederatedRelation ⊑ rdf:Bag
2 fm:BookForSale ⊑ fm:FederatedRelation
3
4 fm:isbn (fm:BookForSale, dbs:Column ⊔ fm:Function)
5 fm:name (fm:BookForSale, dbs:Column ⊔ fm:Function)
6 fm:price (fm:BookForSale, dbs:Column ⊔ fm:Function)
7 fm:category (fm:BookForSale, dbs:Column ⊔ fm:Function)
8 fm:store (fm:BookForSale, dbs:Column ⊔ fm:Function)
```

Listagem 10: TBOX do modelo global para o exemplo

3.10.3 Mapeamento entre os modelos

Através da tabela do modelo global, pretendemos que seja possível aceder a ambas as fontes parciais de dados. Para o efeito, criam-se duas instâncias da classe *fm:BookForSale*, modelando duas partições horizontais de dados.

A primeira dessas instâncias vai mapear o modelo parcial da base de dados do *website*. Neste modelo parcial, a informação está dispersa em diversas tabelas, isto é, existe uma partição vertical dos dados. É então necessário não só realizar o mapeamento das colunas, mas também indicar as relações que permitem estabeler o particionamento vertical. A listagem 11 apresenta a ABOX resultante deste mapeamento.

```
#Mapping the columns
fm:BookForSale (bookInstance1)
fm:isbn (bookInstance1, Book.isbn)
fm:name (bookInstance1, Book.name)
fm:price (bookInstance1, Book.price)
fm:category (bookInstance1, Category.description)
fm:store (bookInstance1, Book.store)

#Create the relation between Book and BookCategory
fm:FederatedRelation (relBookBookCategory)
fm:tableLeft (relBookBookCategory, Book)
fm:tableRight (relBookBookCategory, BookCategory)

#Set the joining condition for the relation
fm:ColumnsRelation (colRelBookBookCategory)
fm:fromColumn (colRelBookBookCategory, Book.isbn)
fm:toColumn (colRelBookBookCategory, BookCategory.isbn)
fm:relatedColumns (relBookBookCategory, colRelBookBookCategory)

#Add the relation to the instance
fm:implicitJoin (bookInstance1, relBookBookCategory)

#Relation between BookCategory and Category is ommited but follows the same rule
```

Listagem 11: ABox of schema mapping for website database

A fonte de dados correspondente à folha de cálculo apresenta outros desafios. Primeiro, o preço dos livros deve inclur uma margem de lucro para o *website*, pelo que vai ser usada uma função para acrescentar 10% ao valor dos livros, multiplicando o valor original por 1.1. A indicação da categoria dos livros está em falta, pelo que é estabelecida uma partição vertical para cruzar com a informação existente no *website*. Finalmente, a indicação da loja tem de ser substituida por uma constante que indique que os livros são originados da rede social. A ABOX resultante deste processo de mapeamento é apresentada na listagem 12.

```
#Create and directly map columns
fm:BookForSale(bookInstance2)
fm:isbn (bookInstance2, Spreadsheet1.isbn)
fm:name (bookInstance2, Spreadsheet1.name)
fm:category (bookInstance2, Category.description)
```

```
6
7  #Set function for price calculation
8  fm:Function (addMarginProfit)
9  fm:operation (addMarginProfit, fm:Multiply)
10 fm:arguments (addMarginProfit, Spreadsheet1.price)
11 fm:arguments (addMarginProfit, Constants.#1.1)
12 fm:price (bookInstance2, addMarginProfit)
13
14 #Create the relation between tables to obtain category information
15 fm:FederatedRelation (relSSBookCategory)
16 fm:tableLeft (relSSBookCategory, Book)
17 fm:tableRight (relSSBookCategory, BookCategory)
18
19 #Set the joining condition for the relation
20 fm:ColumnsRelation (colRelBookBookCategory)
21 fm:fromColumn (colRelBookBookCategory, Book.isbn)
22 fm:toColumn (colRelBookBookCategory, BookCategory.isbn)
23 fm:relatedColumns (relSSBookCategory, colRelBookBookCategory)
24
25 #Add the relations to the table instance reusing relBookCategoryCategory relation
26 fm:implicitJoin (bookInstance2, relSSBookCategory)
27 fm:implicitJoin (bookInstance2, relBookCategoryCategory)
28
29 #Set constant for the store column
30 fm:store (bookInstance2, Constants.#individualSeller)
```

Listagem 12: ABox do mapeamento da folha de cálculo

CAPÍTULO 4

EXECUÇÃO DE INTERROGAÇÕES

A construção da interface aplicacional (API) tem como objectivo permitir executar interrogações sobre o sistema federado de forma o mais transparente possível, como se de uma base de dados se tratasse. Para o efeito, a interface para interrogações utiliza a linguagem SQL e os resultados são retornados sobre a forma de objectos *RowSet*. Esta API foi desenvolvida em *Java* e a operação fundamental disponibilizada é:

```
1 RowSet execute(String queryString);
```

A arquitectura sobre a qual a execução de interrogações está implementada é a apresentada no capítulo 2. Sobre esta arquitectura, o executor de interrogações executa-se na camada de federação, em colaboração com a camada de adaptadores para o acesso aos dados.

É essencial que a implementação não interfira com a autonomia das fontes de dados, o que tem algumas consequências no planeamento e execução das interrogações. Nomeadamente, não é possível manter estruturas auxiliares como estatísticas e índices actualizados, pelo que muitas das técnicas de optimização utilizadas em cenários de bases de dados não podem ser aproveitadas para o cenário dos sistemas de informação federados.

O executor de interrogações implementado está dividido em três fases essenciais:

1. **Análise da interrogação SQL**: partindo de uma *string*, são construídos objectos que representam a estrutura da interrogação.
2. **Planeamento das interrogações**: é necessário planear a execução da interrogação tendo em conta a modelação do sistema federado.

3. **Execução da interrogação**: utilizando o planeamento criado no ponto anterior, são executadas as sub-interrogações locais (através de adaptadores) e todo o pós-processamento necessário.

4.1 Interpretação da interrogação SQL

Uma interrogação SQL chega à API sobre a forma de uma *String*, sendo necessário realizar a análise da mesma. É necessário que a interpretação das interrogações suporte as cláusulas essenciais do *SQL standard*[18] (*select, from, where, group by, order, having*) e que seja suficientemente flexivel para permitir a escrita de interrogações sem demasiadas preocupações de rigidez sintática. Foram consideradas duas hipóteses para abordar este problema:

1. Implementar um novo *parser* SQL;
2. Utilizar um *parser* disponibilizado como *open-source* e alterar/acrescentar o necessário para os requisitos.

Para a implementação de um novo *parser* SQL, considerou-se a utilização de uma ferramenta de construção de *parsers* como a antlr [19]. Existem, inclusive, algumas implementações da linguagem SQL através desta ferramenta. No entanto, devido à complexidade da linguagem SQL, nenhuma das implementações disponiblizadas era complecta, e a criação de uma nova implementação obrigaria a um esforço demasiado grande sem suficiente retorno de valor acrescentado para os objectivos traçados.

Optou-se então pela utilização de um *parser open-source* e foram analisadas algumas alternativas:

- SQLJEP [20]: *parser* gerado com recurso ao gerador JavaCC [21] mas que se baseia numa versão incomplecta do *standard* SQL, centrando-se mais nas particularidades dos dialectos Oracle e MaxDB.
- Apache Derby [22]: é uma implementação bastante complecta mas foi criada para funcionar em modo "ligado", pelo que a utilização num sistema federado (sobre um modelo de dados virtual) obrigaria a recorrer a "truques" que por vezes provocam comportamentos pouco previsíveis.
- *Eclipse Data Tools*[1] [24]: é uma implementação bastante complecta, apesar de gerar um modelo de objectos complexo, pouco documentado e por vezes pouco organizado.

Optou-se por utilizar o parser do pacote *Eclipse Data Tools* e acrescentar código numa segunda fase da interpretação da *query*, de forma a:

1. Simplificar o modelo de objectos;

[1] Eclipse Public License 1.0 [23]

2. Complementar os objectos com informação do modelo de dados do sistema federado;
3. Realizar logo algumas validações à interrogação, como a utilização de colunas não existentes ou a não indicação das tabelas correctas na cláusula *where*.

O interpretador implementado dá origem a um novo conjunto de objectos que suporta as cláusulas:

- **select**: colunas, funções e constantes. Também é válido o modificador distinct.
- **from**: inner, outter, left e right joins. Inner queries não são suportadas.
- **where**: condições booleanas com colunas, funções e constantes. Inner queries não são suportadas.
- **order by**.
- **group by**.
- **having**: colunas.

O diagrama de classes utilizado é apresentado na figura 4.1, com as propriedades e os métodos ocultados por uma questão de simplificação do diagrama.

O código apresentado na listagem 13 mostra como utilizar o pacote *Eclipse Data Tools* para interpretar uma *string* e gerar uma árvore de objectos que representam a interrogação.

```
1  public void parseAndFill(String query) throws SQLParserException,
       SQLParserInternalException
2  {
3     //Get SQLQueryParserManager object
4     //Parameters: product, version
5     //Both parameters as null for sql standard
6     SQLQueryParserManager parserManager =
          SQLQueryParserManagerProvider.
7           getInstance().getParserManager(null, null);
8
9     //Parse
10    SQLQueryParseResult parseResult =
          parserManager.parseQuery(query);
11
12    //Get the sql statement
13    QueryStatement resultObject = parseResult.getQueryStatement();
14
15    //Process the sql objects
16    runThroughQueryTree(resultObject);
17 }
```

Listagem 13: Utilização do pacote Eclipse Data Tools

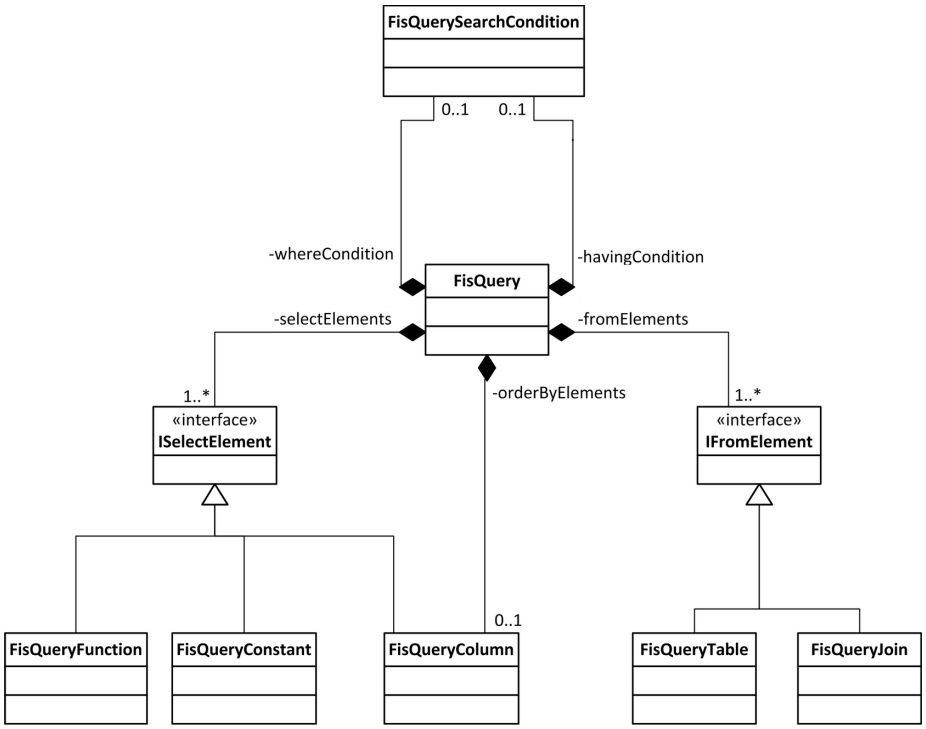


Figura 4.1: Diagrama de classes para as *queries* interpretadas

4.2 Planeamento das interrogações

O planeamento das interrogações tem como objectivo definir como cada interrogação tem de ser efectivamente realizada. Isto implica cruzar as informações da interrogação a executar com os dados do modelo federado. Nomeadamente, é necessário realizar o planeamento quais as interrogações que devem ser executadas em cada fonte, assim como o planeamento de como juntar os resultados parciais e realizar as operações globais sobre os dados.

Este planeamento é composto por quatro passos essenciais:

1. Identificar quais as operações que têm de ser realizadas como pós-processamento de forma centralizada, p. e., junções globais, alguns filtros e agregações.
2. Expandir a interrogação para conter informação de possíveis partições, funções definidas no modelo global do sistema, etc.
3. Criar sub-interrogações que podem ser executadas nas fontes de dados locais.

4. Planear o pós-processamento necessário para juntar os resultados de cada uma das sub-interrogações locais.

4.2.1 Acesso ao modelo do sistema federado

No processo de planeamento de uma interrogação, é necessário aceder ao modelo que descreve o sistema federado para alterar, complementar ou dividir a interrogação. O modelo está descrito em linguagem OWL que tem de ser acedido em código Java, no entanto o acesso a estes dados não é feito de forma directa, já que é necessário ter em conta a semântica que é proporcionada pela linguagem OWL, nomeadamente lógica de descrição (Description Logic - DL [17]). Para o efeito, é necessário utilizar um motor de inferência, que conhece e processa a semântica da linguagem OWL. O motor de inferência *Pellet*[2] [26] permite aceder aos dados através de diversas interfaces como a interface OWLAPI[27] e a interface *Jena*[28]. Ambas permitem aceder a taxonomias escritas em OWL através de modelos de objectos, mas a interface *Jena* permite também a utilização da linguagem de interrogação SPARQL-DL [29].

A linguagem SPARQL-DL tem como origem a linguagem SPARQL, utilizada para interrogar grafos RDF. A SPARQL-DL é em tudo semelhante à SPARQL, mas realiza as pesquisas com auxílio de um motor de inferência. O resultado prático é a realização de inferência DL, que tem influência nos resultados das pesquisas.

A listagem que se segue apresenta como exemplo uma parcela de uma taxonomia em DL. Neste exemplo, diz-se que "fm:instancia1" é uma réplica de "fm:instancia2". Note-se que "fm:replic" é uma propriedade simétrica e transitiva.

```
1 replic(instancia1, instancia2)
```

A seguinte interrogação pode ser executada em SPARQL ou SPARQL-DL, sendo válida nas duas linguagens. O objectivo é obter as réplicas de "instancia2". Os elementos iniciados pelo caracter "?" são as variáveis da interrogação.

```
1 SELECT ?replic
2 WHERE { instancia2 replic ?replic }
```

O resultado da execução da interrogação em SPARQL é vazio, enquanto em SPARQL-DL o resultado é "fm:instancia1". A diferença dos resultados está no facto de as interrogações SPARQL apenas terem em conta o que existe directamente no modelo, enquanto as interrogações SPARQL-DL realizam inferências sobre o modelo, obtendo informações que não têm de ser directamente inseridas no modelo.

[2]GNU Affero General Public License, Version 3 [25]

O código para executar uma interrogação SPARQL-DL através da interface *Jena* é apresentado na listagem 14.

```
1  public ResultSet getReplicsOfInstance(String instance) {
2      //Create the query to be executed
3      //The query is in a separate file
4      final String queryFile = "file:" + getSparqlDirectory() +
           "FindReplicsOfInstance.sparql";
5      Query q = QueryFactory.read( queryFile );
6
7      //Create the execution environment over an existing model
8      //The environment will use SparqlDL
9      QueryExecution qe = SparqlDLExecutionFactory.create(
           q,getRdfsModel());
10
11     //Set the variables of the query
12     Resource parameterInstance =
           getRdfsModel().getResource(instance);
13
14     QuerySolutionMap qs;
15     qs = new QuerySolutionMap();
16     qs.add("instance", parameterInstance);
17     qe.setInitialBinding(qs);
18
19     //Execute and get results
20     ResultSet rs = qe.execSelect();
21     return rs;
22 }
```

Listagem 14: Executar uma interrogação SPARQL-DL através da interface *Jena*

4.2.2 Resultado do planeamento

Partindo da análise da interrogação interpretada, é aplicado um conjunto de regras dando origem a um objecto do tipo *FisQueryMasterPlan*. Este objecto contém um ou mais planos de execução de sub-interrogações, representadas por objectos de tipos que implementam a interface *FisQueryPlan*. Existem três classes que implementa esta interface:

- *FisQueryTablePlan*: Plano de interrogação referente a uma tabela do esquema global.
- *FisQueryInstancePlan*: Plano de interrogação referente a uma instância de uma tabela do esquema global.
- *FisQuerySourcePlan*: Plano de interrogação executável numa fonte de dados.

O diagrama de classes desta componente é apresentado na figura 4.2.

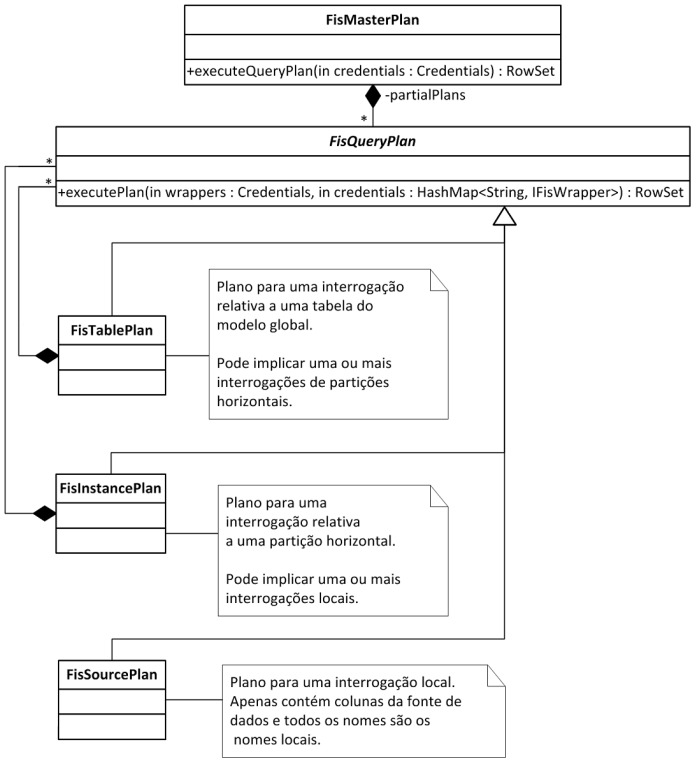


Figura 4.2: Diagrama de classes para os planos de execução das *queries*

4.2.3 Regras aplicadas no planeamento

Regra 1. Por cada interrogação sobre o modelo federado, é criado um objecto do tipo *FisQueryMasterPlan*.

FisQueryMasterPlan

Regra 2. Por cada tabela global envolvida na interrogação, é criado um objecto do tipo *FisQueryTablePlan*. Cada um desses objectos contém uma interrogação parcial contendo apenas as colunas existente na tabela respectiva.

Regra 3. As colunas necessárias para a realização das operações globais de junção, filtragem e agrupamento são adicionadas à clausula de selecção do objecto *FisQueryTablePlan* correspondente à tabela que contém cada coluna.

FisQueryTablePlan

Regra 4. Por cada intancia existente para a tabela global (anexo A.1), é criado um objecto do tipo *FisQueryInstancePlan*.

FisQueryInstancePlan

Regra 5. Se mais de uma tabela parcial estiver envolvida na instância (anexo A.2), a junção das tabelas é adicionada à interrogação inicial (anexo A.3).

Regra 6. Cada coluna do modelo global é substituida pela coluna do modelo parcial correspondente (anexo A.2).

Regra 7. Se uma coluna do modelo global está mapeada para uma função, as colunas utilizadas como argumento são adicionadas à cláusula de selecção da interrogação (anexo A.4) e a função fica registada para ser realizada de forma centralizada.

Regra 8. Para cada fonte de dados da instância, é criado um objecto do tipo *FisQuerySourcePlan* que contém a sub-interrogação que apenas contém as colunas e tabelas que lhe correspondem.

4.3 Execução da interrogação

A execução das interrogações utiliza os objectos resultantes do planeamento (figura 4.2) como base para o seu funcionamento. Existem duas classes essenciais de operações que são realizadas pelo executor de interrogações:

1. Entrega das interrogações parciais aos adaptadores das fontes de dados;
2. Realização centralizada das operações que não podem ser realizadas nas fontes de dados.

A figura 4.3 apresenta o diagrama de sequência das operações realizadas pelo executor de interrogações. De seguida são descritas algumas destas operações pela sua ordem de execução.

4.3.1 Detectar réplicas

As réplicas estão definidas ao nível das instâncias de uma tabela do modelo global (anexo A.5). Se o acesso a uma dessas instâncias falhar, é possível tentar aceder a uma das suas réplicas. Basta que uma das réplicas seja acedida com sucesso para a interrogação poder ser realizada com sucesso.

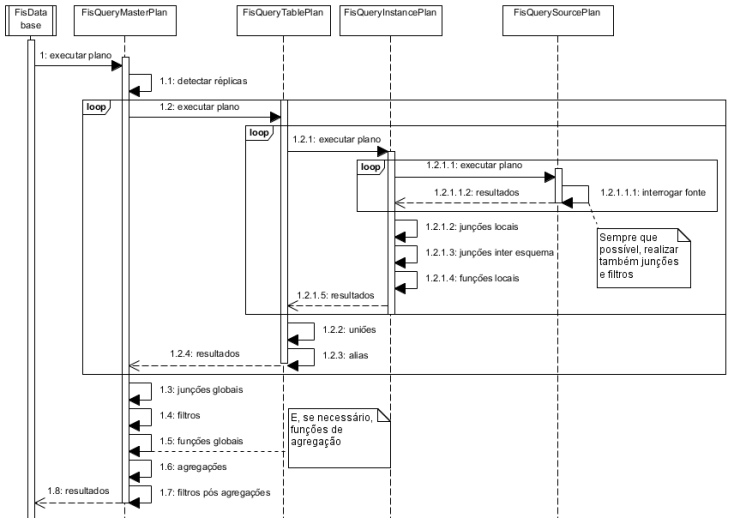


Figura 4.3: Diagrama de sequência do executor de interrogações

Por outro lado, se o acesso a uma instância for realizado com sucesso, todas os acessos às suas réplicas são dados como termidados, de forma a não aceder desnecessáriamente a informação duplicada.

4.3.2 Interrogar fonte

As interrogações parciais são realizadas sempre através dos adaptadores (wrappers). Estes componentes, que implementam a interface presente na listagem 15, são responsáveis por realizar o acesso aos dados resolvendo o problema da heterogeneidade física dos dados: os dados retornados através do *RowSet* estão sempre no formato relacional.

```
1 public interface IFisWrapper {
2     public RowSet executePlan(FisQuery query, Credentials
          credentials) throws FISWrapperException;
3 }
```

Listagem 15: Interface dos adaptadores de fontes de dados

Adaptadores implementados

Dois adaptadores foram implementados para servirem de exemplo e de validação aos restantes componentes. Nomeadamente, foram criados adaptadores que permitem o acesso a fontes de dados relacionais através da interface JDBC e a ficheiros criados pela aplicação *Microsoft Excel*.

O acesso a fontes de dados *jdbc* é realizado de forma directa: os objectos do tipo *FisQuery* geram *strings SQL standard* que podem ser executadas através de uma ligação *jdbc* e o resultado pode ser retornado directamente.

O acesso a fontes de dados *Excel* tem de ser realizado através de uma API que permita o acesso a ficheiros deste tipo. Para o efeito foi utilizada a biblioteca *POI*[3] [31]. O acesso através desta biblioteca é feito ao nível de folhas, colunas e células, como é possível ver na listagem 16.

```
1  Sheet sheet = null;
2  Table table = null;
3  Column column = null;
4
5  //for each sheet...
6  for(int i = 0; i < wb.getNumberOfSheets(); ++i)
7  {
8    sheet = wb.getSheetAt(i);
9
10   for (Cell cell : row) {
11     CellReference cellRef = new CellReference(row.getRowNum(),
           cell.getColumnIndex());
12
13     //do something with cell
14
15   }
16 }
```

Listagem 16: Aceder a células num ficheiro *xls*

4.3.3 Junções locais, inter-esquema e globais

Sempre que possível, tenta-se que as junções sejam realizadas nas fontes de dados, de forma a minimizar o tráfego, a distribuir o processamento e a tirar partido das optimizações que possam existir, como por exemplo a existência de índices. No entanto, existem várias situações em que isso não é possível, pois os dados a juntar provêm de fontes de dados distintas. Nesses casos, é necessário realizar as junções em memória.

O pacote *javax.sql.rowset* contém a interface *JoinRowSet* para dar suporte a objectos *RowSet* que permitam realizar junções em modo desligado das fontes. Infelizmente, a única implementação disponibilizada neste pacote - *JoinRowSetImpl* - apenas suporta a variante *inner join*, pelo que não pode ser utilizada. Foi por isso necessário realizar a implementação das junções de raíz.

Na escolha do algoritmo a implementar, foram consideradas as duas variantes tipicamente utilizadas pelos SGBDs para realizar este tipo de operações:

- *Nested-loop join*: dois ciclos aninhados percorrem todas as colunas das tabelas a juntar, sendo necessário percorrer a tabela interior para cada

[3] Apache License, Version 2.0 [30]

linha da tabela exterior.

- *Merge-scan join*: as tabelas a juntar estão ordenadas pela chave de junção, sendo cada tabela percorrida apenas uma vez.

A utilização do *merge-scan loop* tem o problema de ser necessária a ordenação dos dados. Não sendo possível a utilziação de índices, as operações de ordenação em memória têm um custo muito elevado, sendo mais viável optar pela implementação de um algoritmo do tipo *nested-loop join*, apesar do seu custo também elevado.

O algoritmo implementado é composto pelos seguintes passos:

1. Escolher como tabela interna a tabela mais pequena, de forma a optimizar a execução;
2. Para uma linha da tabela externa, a tabela interna é percorrida para encontrar uma linha que cumpra o predicado de junção;
3. Quando termina a tabela interna, é escolhida a linha seguinte da tabela externa;
4. Repete desde o ponto 2, até terminar a tabela externa.

A implementação deste algoritmo tem algumas diferenças consoante o tipo de junção a efectuar:

- *Inner join*: a tabela resultante contém todas os pares de linhas que cumpram o predicado;
- *Left outter join*: se uma linha da tabela à esquerda não tem nenhuma linha correspondente na tabela à direita, a linha é inserida apenas com os dados da tabela à esqueda e com as restantes colunas com o valor nulo;
- *Right outter join*: identico ao *left outter join*, mas com o papel das tabelas invertido;
- *Full outter join*: mistura das variantes *left* e *right outter join*;
- *Cross join*: junção de todas as linhas da tabela à esquerda com todas as linahs da tabela exterior. Não existe predicado a verificar.

4.3.4 Funções

As funções a executar podem ser de uma de duas categorias:

- Funções de linha: aplicadas sobre cada linha da tabela;
- Funções de agregado: aplicadas sobre um conjunto de linhas agrupadas pela cláusula de agregação *group by*.

Uma função de linha implementa a interface apresentada na listagem 17 e realiza, de uma só vez, as suas operações sobre a sua lista de parâmetros.

```
1 public interface IFISRowFunction extends IFISFunction {
2     public Object execute(List parameters) throws
          FISExecutionException;
3 }
```

Listagem 17: Interface para funções de linha

Um função de agregação implementa a interface apresentada na listagem 18 e realiza, de forma fazeada, as suas operações sobre os dados de cada linha. Para obter o valor final da sua execução, é necessário chamar o método *getFinalResult*.

```
1 public interface IFISAgregationFunction extends IFISFunction {
2     public void initialize();
3     public void accumulate(Object parameter) throws
          FISExecutionException;
4     public Object getFinalResult();
5
6 }
```

Listagem 18: Interface para funções de agregação

4.3.5 Uniões

A união objectos *RowSet* é realizada copiando linha entre objectos. Isto implica que todas as linha são percorridas de forma a serem copiadas. Devido à forma como os objectos do pacote *javax.sql.rowset* estão implementados, não é possível garantir de que forma os dados estão a ser guardados, pelo que se torna muito difícil fazer optimizações a esta operação.

4.3.6 Alias

Sendo utilizados *alias* numa interrogação, há momentos em que é necessário alterar o nome das colunas, que passam a ser conhecidas pelo seu *alias*. A alteração do nome de uma coluna num objecto *rowset* implica a alteração da sua *metadata*, alteração essa que não pode ser realizada sobre um objecto pre-enchido. Por essa razão, tenta-se realizar esta alteração de nomes juntamente com outras operações que obriguem a utilizar novos objectos *rowset*, como por exemplo a realização da união entre objectos.

4.3.7 Filtros

O pacote *javax.sql.rowset* contém a interface *FilteredRowSet* para dar suporte a objectos *RowSet* que possam ser filtrados em modo desligado das fontes. A implementação de referência incluída no pacote é a classe *FilteredRowSetImpl* e deixa a decisão de quais as linahs a filtrar para um objecto que implemente a interface *Predicate*, apresentada na listagem 19.

```
1  public interface Predicate {
2        //utilizado para validar o predicado sobre um objecto
             preenchido
3      public boolean evaluate(RowSet rs);
4
5        //utilizados ao inserir novas linhas
6      public boolean evaluate(Object value, int column) throws
           SQLException;
7      public boolean evaluate(Object value, String columnName)
           throws SQLException;
8  }
```

Listagem 19: Interface javax.sql.rowset.Predicate

Na fase de interpretação da interrogação, são criados objectos do tipo *FisQuerySearchCondition* para conter as informações dos predicados. Para validar estas condições, foi implementada a classe *FisQueryPredicate* (listagem 20).

```
1  public class FisQueryPredicate implements Predicate, Cloneable {
2      protected FisQuerySearchCondition _conditionToEvaluate;
3
4      public FisQueryPredicate(FisQuerySearchCondition
           conditionToEvaluate) { ... }
5
6      public boolean evaluate(RowSet rs) { ... }
7
8        //não é utilizado no contexto de inserções
9      public boolean evaluate(Object value, int column) throws
           SQLException { ... }
10
11       //não é utilizado no contexto de inserções
12     public boolean evaluate(Object value, String columnName)
           throws SQLException { ... }
13
14     @Override
15     public Object clone() throws CloneNotSupportedException { ... }
16
17 }
```

Listagem 20: Classe FisQueryPredicate

4.3.8 Agregações

A primeira acção a realizar para a realização de uma agregação é a ordenação dos dados. Depois de os dados estarem ordenados pelas colunas de agregação, agregar é apenas remover as colunas repetidas e, se necessário, executar as funções de agregação.

A ordenação dos dados é, no entanto, uma operação problemática. Mais uma vez, a inexistência de objectos auxiliares como índices dos dados, limita-nos a escolha de algoritmos. Os objectos *RowSet* têm também um custo de acesso sequencial muito inferior ao do acesso *random* (que corresponde a uma pesquisa

numa tabela), pelo que a escolha do algoritmo teve também esse factor em conta. O algoritmo utilizado foi o *mergesort* [32], que tem um custo *O(n log(n))*. No entanto, se a quantidade de dados for muito elevada, qualquer algoritmo de ordenação em memória tem problemas de eficiência e de utilização de memória.

CAPÍTULO 5

APLICAÇÃO DE CONFIGURAÇÃO DO SISTEMA

A aplicação de configuração do sistema tem como objectivo permitir a criação do modelo de descrição do sistema federado sem ser necessário conhecer os detalhes do mesmo ou as especificidades da linguagem OWL. Também se pretende optimizar certas partes da criação do modelo de forma a diminuir o trabalho do utilizador. A existência de uma aplicação deste género permite também reduzir os erros na construção do modelo, ao proporcionar auxílios visuais, como a apresentação apenas da informação válida num contexto e a validação de consistência no momento da inserção de novos elementos.

Tal como o modelo de dados, a aplicação está organizada em três componentes (figura 5.1):

1. Definição dos modelos de dados parciais (área 1);
2. Definição do modelo de dados global (área 2);
3. Mapeamento entre os modelos de dados (área 3).

5.1 Acesso à taxonomia

O acesso à taxonomia é realizado de diferentes formas, consoante as necessidades do acesso e as características da informação a aceder. Distingue-se desta forma:

1. um acesso a um elemento específico do modelo de um acesso a elementos resultantes de uma pesquisa;
2. um acesso à componente fixa do modelo (esquemas parciais e funções do esquema global) de um acesso à componente variável (estrutura do

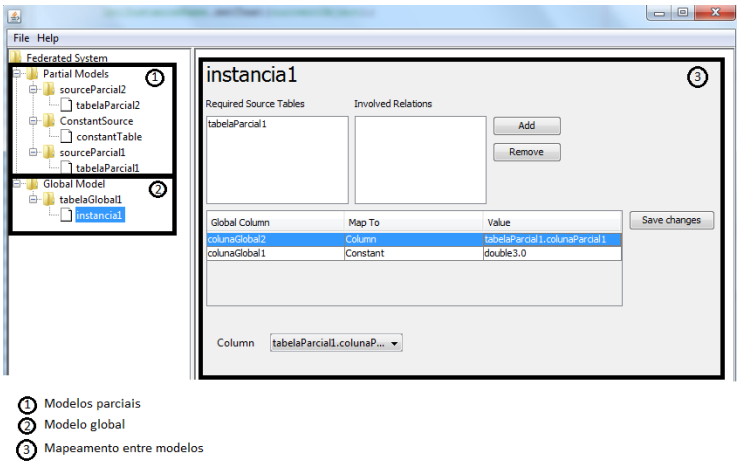


Figura 5.1: Organização geral da aplicação

esquema global).

O acesso à componente fixa do modelo é realizado através de código originalmente gerado pelo *Protégé* e modificado consoante necessário. O código gerado é constituído por um conjunto de classes equivalentes às classes definidas no modelo (figura 5.2) e contém métodos de leitura e escrita nas diversas propriedades das classes do modelo. A base de todos os classes geradas é a classe *AbstractCodeGeneratorIndividual* que forma uma camada de abstracção de acesso à interface *OWLAPI*. Para além disso, é também disponibilizada uma *factory* para a criação de novos elementos do modelo. A listagem 21 mostra como se pode criar uma coluna e adicionar valores às suas propriedades.

```
1  //Create a factory for a OWLModel object
2  FisFactory factory = new FisFactory(ontology.getOwlModel());
3
4  //Create the column
5  Column col = factory.createColumn(strColumnName);
6
7  //Set the column properties
8  col.setLength(lenght);
9  col.setScale(scale);
10 col.setColumnAccess(columnAccess);
```

Listagem 21: Criar uma coluna no modelo e dar valores às suas propriedades

O acesso à componente variável do modelo é realizado directamente através da interface *OWLAPI*, já que a geração de código não é viável para as classes que só são conhecidas em tempo de execução. A listagem 22 mostra a criação de uma tabela.

```
1  public boolean addGlobalTable(String table)
2  {
```

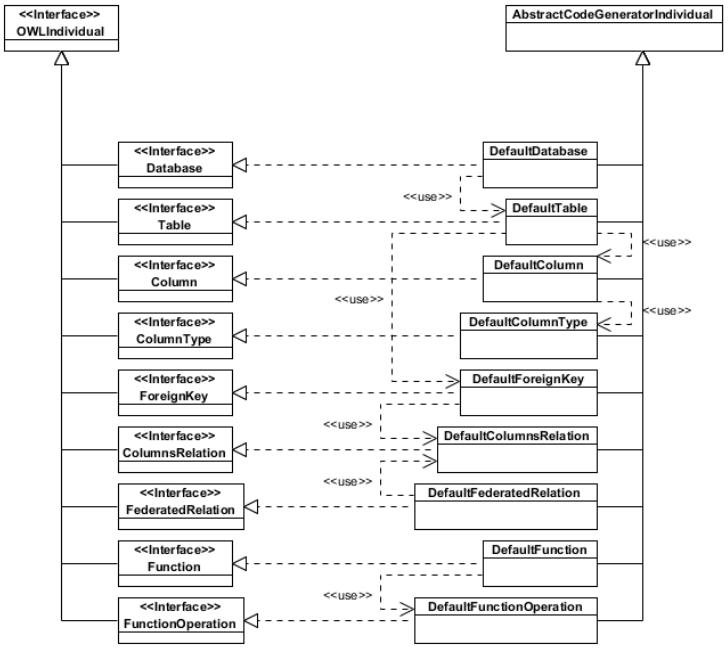


Figura 5.2: Classes utilizadas para o acesso aos elementos fixos da taxonomia

```
3     //get active OWL model
4     OWLModel owlModel =
          Global.getCurrentProject().getOntology().getOwlModel();
5
6     OWLNamedClass newTable = null;
7     OWLNamedClass fedEntity = null;
8
9     try{
10        //get resource name for the class fm:FederatedEntities,
              superclass of global tables
11        String fedEntityString =
              owlModel.getResourceNameForURI(getNsFm() +
              "FederatedEntities");
12
13        //get class fm:FederatedEntities
14        fedEntity = owlModel.getOWLNamedClass(fedEntityString);
15
16        //create table
17        newTable = FISOntologyUtils.createOWLClassSafe(owlModel,
              getNsFm() + table);
18
19        //add superclass
20        newTable.addSuperclass(fedEntity);
21    }
22    catch(Exception ex){
23        ...
```

```
24      }
25      return true;
26  }
```

Listagem 22: Criar uma tabela no modelo global

Por fim, o acesso através de pesquisas é realizado através da interface *Jena*, recorrendo às linguagens *SPARQL* e *SPARQL-DL*. As duas linguagens têm sintaxes iguais, mas a linguagem *SPARQL-DL* é executada sobre um motor de inferência, o que permite processar a semântica adicional adicionada pelo *OWL*. No entanto, devido a limitações conceptuais, não é possível realizar interrogações sobre a TBOX de uma taxonomia utilizando a variante *DL*. Como essas interrogações são necessárias para obter informações sobre as tabelas e colunas do modelo de dados global, as duas variantes são utilizadas.

Os recursos obtidos através destas interrogações podem depois ser processados através das duas técnicas apresentadas anteriormente: código gerado pelo *protégé* ou interface *OWLAPI*.

5.2 Modelos de dados parciais

Os modelos de dados parciais podem ser inseridos de forma manual ou através de um componente que tenta a importação automática a partir de uma ligação à fonte de dados.

A inserção manual é realizada através de um conjunto de ecrãs que disponibilizam acesso a campos correspondentes à taxonomia *relational.owl* alterada. Nomeadamente, são disponibilizados os ecrãs correspondentes a:

- *dbs:database* - correspondente a uma fonte de dados parcial, permite inserir novas tabelas e definir os campos *dbs:uri*, *dbs:provider*, *dbs:username* e *dbs:password*.
- *dbs:table* - correspondente a uma tabela de uma fonte de dados parcial, permite inserir novas colunas e aceder aos seus campos *dbs:columnAccess* e *dbs:columnType*.

A importação de fontes de dados foi implementada para fontes de dados relacionais (com interface *jdbc*) e ficheiros *xls* (criado com o programa *Microsoft Excel*). Tendo como entrada a *string* de conexão *jdbc* ou o url para o ficheiro *xls*, a aplicação obtém a estrutura dos dados e adiciona à taxonomia actualmente activa na aplicação. Desta forma facilita-se a configuração do sistema e reduz-se eventuais erros que são mais prováveis de acontecer nas configurações manuais.

O diagrama de classes do módulo de importação automática está na figura 5.3.

Este módulo de importação pode ser utilizado como exemplificado na listagem 23.

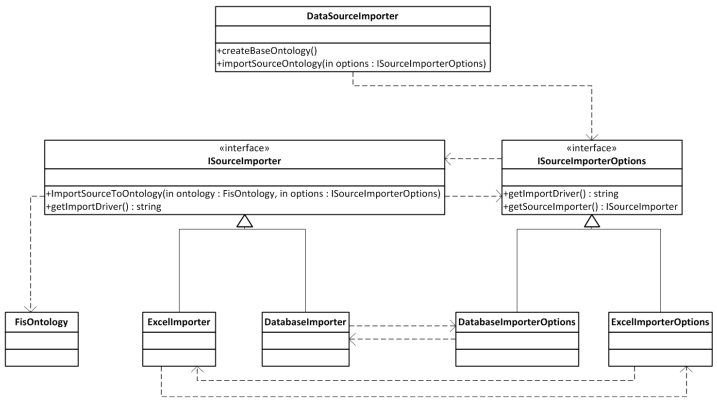


Figura 5.3: Diagrama de classes para a importação de fontes de dados

```
1  FISOntology ontology;
2  //initialize ontology as needed
3
4  DataSourceImporter sourceImporter = new
       DataSourceImporter(ontology);
5
6  //set options for importer
7  ISourceImporterOptions res = new DatabaseImporterOptions(driver,
       uri, schema, user, pass, importDriverName);
8
9  //import
10 sourceImporter.ImportSourceOntology(options);
```

Listagem 23: Utilizar importador de fontes de dados

5.2.1 Importação automática de modelos de dados parciais

A importação do modelo de dados é específica para cada tipo de fonte de dados suportado.

Para as fontes de dados JDBC, implementações da interface *DatabaseMetaData* [33] já disponibilizam os serviços necessários para aceder à meta informação. Este código é inspirado no código do *plugin* para o *Protégé DataMaster*[1] [35]. A listagem 24 mostra como é possível obter o nome das tabelas de uma base de dados de nome "dbteste" através desta interface *DatabaseMetaData*.

```
1  Connection conn = null;
2  DatabaseMetaData dbMetaData = null;
3
```

[1] Mozilla Public License [34]

```
4  try {
5    //get jdbc connection to source
6    conn = getConnection(conOptions);
7
8    //get metadata
9    dbMetaData = conn.getMetaData();
10
11   ResultSet tablesSet = dbMetaData.getTables(``dbteste'', null,
         null, null);
12
13   while(tablesSet.next())
14   {
15       String tableName = tablesSet.getString("TABLE_NAME");
16   }
17 }
18 catch(SQLException e) {
19     e.printStackTrace();
20 }
```

Listagem 24: Código Java para obter o nome das tabelas

Para obter a mesma informação das fontes de dados *xls* a estratégia é muito diferente, pois não existe uma classe com estes serviços e a estrutura dos dados não pode ser obtida de forma tão directa. A listagem 16 na secção 4.3 apresenta um exemplo de acesso a dados através da biblioteca POI. A importação automática destas fontes de dados assume que a primeira linha com células preenchidas é a linha dos cabeçalhos e faz uma recolha simples de colunas, sem importar chaves primárias ou estrangeiras.

5.3 Modelo de dados global

A definição do modelo de dados global tem de ser realizada de forma manual através da interface gráfica da aplicação. Para o efeito, são diponibilizados um conjunto de ecrãs que disponibilizam acesso à componente TBOX da taxonomia que descreve o modelo de dados global. Nomeadamente, são disponibilizados os ecrãs correspondentes a:

- tabelas - permite criar subclasses de *fm:FederatedEntities* correspondentes a tabelas no modelo de dados global.
- colunas - permite criar propriedades correspondentes a colunas no modelo de dados global.

5.4 Mapeamento entre modelos

O mapeamento entre modelos é realizado através de instâncias das classes correspondentes às tabelas do modelo global. Cada instância corresponde a uma

partição horizontal dessa tabela. Ao seleccionar uma instância, é apresentado o ecrã da figura 5.4 onde podem ser introduzidas as informações do mapeamento.

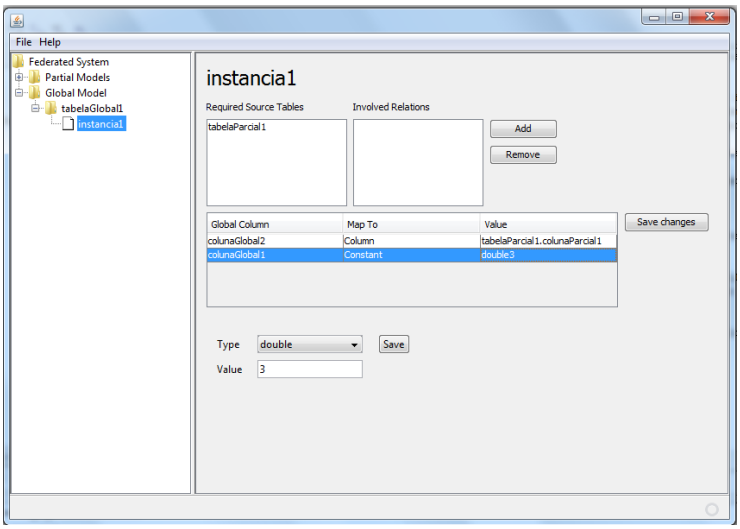


Figura 5.4: Ecrã para mapeamento de uma instância

O primeiro passo do mapeamento é indicar a tabela dos modelos parciais que irá ser utilizada para o mapeamento. Se mais de uma tabela for utilizada, é necessário indicar as colunas que vão definir a relação entre as tabelas (figura 5.5). A indicação da primeira tabela não tem qualquer influência no modelo, já que essa informação pode ser inferida do mapeamento das colunas. No entanto, essa indicação é utilizada para determinar as colunas que são apresentadas na interface gráfica. Para cada tabela para além da primeira, é inserido no modelo uma *FederatedRelation*, com uma *ColumnsRelation* para cada par de colunas indicado para a relação entre as tabelas.

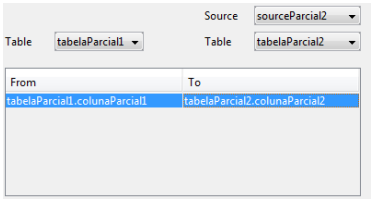
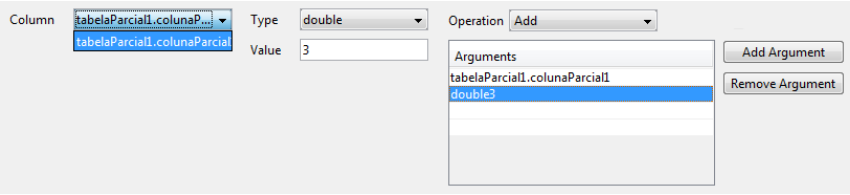


Figura 5.5: Ecrã para definição de relação entre tabelas

O mapeamento é realizado definindo os mapeamentos individuais das colunas do modelo global. Assim, cada coluna do modelo global pode ser mapeada para:

1. Uma coluna de um modelo parcial (figura 5.6a);
2. Uma constante (figura 5.6b);

3. Uma função (figura 5.6c).



(a) Coluna (b) Constante (c) Função

Figura 5.6: Ecrãs alternativos para mapeamentos de colunas

CAPÍTULO 6

APLICAÇÃO E AVALIAÇÃO

CAPÍTULO 7

CONCLUSÃO

APÊNDICE A

INTERROGAÇÕES SPARQL

A.1 Obter todas as instâncias de uma tabela global

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 PREFIX owl: <http://www.w3.org/2002/07/owl#>
4 PREFIX sparqldl: <http://pellet.owldl.com/ns/sdle#>
5
6 SELECT DISTINCT ?instance
7 WHERE {
8   ?instance rdf:type ?federatedTable
9 }
```

A.2 Obter a informação de acesso a uma instância de uma tabela global

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 PREFIX owl: <http://www.w3.org/2002/07/owl#>
4 PREFIX sparqldl: <http://pellet.owldl.com/ns/sdle#>
5 PREFIX dbs:
      <http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#>
6
7 SELECT ?federatedColumn ?column ?columnUri ?table ?source ?uri
      ?provider
8 WHERE {
```

```
9   ?instance rdf:type ?federatedTable .
10  ?federatedColumn rdfs:domain ?federatedTable .
11  ?instance ?federatedColumn ?column .
12  ?table dbs:hasColumn ?column .
13  ?source dbs:hasTable ?table .
14  ?source dbs:uri ?uri .
15  ?source dbs:provider ?provider .
16  ?column dbs:columnAccess ?columnUri
17 }
```

A.3 Obter a informação para juntar partições verticais de uma intância de uma tabela global

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX owl: <http://www.w3.org/2002/07/owl#>
4  PREFIX sparqldl: <http://pellet.owldl.com/ns/sdle#>
5  PREFIX dbs:
       <http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#>
6  PREFIX fm: <http://edu.ipl.isel.deetc.fis/#>
7
8  SELECT ?join ?left ?right ?cLeft ?cRight
9  WHERE {
10   ?entity fm:implicitJoin ?join .
11   ?join fm:TableLeft ?left .
12   ?join fm:TableRight ?right .
13   OPTIONAL
14   {
15     ?join dbs:relatedColumns ?columnList .
16     ?columnList dbs:fromColumn ?instLeft .
17     ?instLeft dbs:columnAccess ?cLeft .
18     ?columnList dbs:toColumn ?instRight .
19     ?instRight dbs:columnAccess ?cRight
20   }
21 }
22 ORDER BY ?join
```

A.4 Obter informações de mapeamento de uma coluna global para uma função

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX owl: <http://www.w3.org/2002/07/owl#>
4  PREFIX sparqldl: <http://pellet.owldl.com/ns/sdle#>
5  PREFIX dbs:
       <http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#>
6  PREFIX fm: <http://edu.ipl.isel.deetc.fis/#>
7
```

```
8  SELECT ?op ?args ?columnUri ?table ?source ?uri ?provider
9  WHERE {
10   ?instance ?federatedColumn ?column .
11   ?column fm:operation ?op .
12   ?column fm:arguments ?args .
13   ?table dbs:hasColumn ?args .
14   ?source dbs:hasTable ?table .
15   ?source dbs:uri ?uri .
16   ?source dbs:provider ?provider .
17   ?args dbs:columnAccess ?columnUri
18 }
```

A.5 Obter réplicas de uma instância

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX owl: <http://www.w3.org/2002/07/owl#>
4  PREFIX sparqldl: <http://pellet.owldl.com/ns/sdle#>
5  PREFIX fm: <http://edu.ipl.isel.deetc.fis/#>
6
7  SELECT ?replic
8  WHERE {
9    ?instance fm:replic ?replic
10 }
```

BIBLIOGRAFIA

[1] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated information systems: Concepts, terminology and architectures. Technical Report 99-9, Technische Universitat Berlin, 1999.

[2] Dennis McLeod and Dennis Heimbigner. A federated architecture for database systems. In *AFIPS '80: Proceedings of the May 19-22, 1980, national computer conference*, pages 283–289, New York, NY, USA, 1980. ACM.

[3] Thierry Barsalou, Niki Siambela, Arthur M. Keller, and Gio Wiederhold. Updating relational databases through object-based views. *SIGMOD Rec.*, 20:248–257, April 1991.

[4] Ravi Krishnamurthy, Witold Litwin, and William Kent. Language features for interoperability of databases with schematic discrepancies. *SIGMOD Rec.*, 20:40–49, April 1991.

[5] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. Schemasql: An extension to sql for multidatabase interoperability. *ACM Trans. Database Syst.*, 26:476–519, December 2001.

[6] Diplom Informatiker, Ulf Leser, Vom Fachbereich Informatik, Dr. Ing, and Prof Dr. Weber. Query planning in mediator based information systems vorgelegt von, 2000.

[7] F. Mougin, A. Burgun, O. Loreal, and P. Le Beux. Towards the automatic generation of biomedical sources schema. *Studies in Health Technology and Informatics*, 107:783–787, 2005.

[8] Daniela Nicklas, Matthias Grossmann, Thomas Schwarz, Steffen Volz, and Bernhard Mitschang. A model-based, open architecture for mobile, spatially aware applications. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 117–135, London, UK, 2001. Springer-Verlag.

[9] Hai Zhuge, Yunpeng Xing, and Peng Shi. Resource space model, owl and database: Mapping and integration. *ACM Trans. Internet Technol.*, 8:20:1–20:31, October 2008.

[10] Boris Motik, Bernardo Cuenca Grau, and Ulrike Sattler. Structured objects in owl: representation and reasoning. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 555–564, New York, NY, USA, 2008. ACM.

[11] Agustina Buccella, Alejandra Cechich, Ra Cechich, and Nieves R. Brisaboa. An ontology approach to data integration. *Journal of Computer Science and Technology*, 3:62–68, 2003.

[12] Microsoft. Sql server 2008 books online: Understanding federated database servers. `http://msdn.microsoft.com/en-us/library/ms187467.aspx`, October 2009.

[13] L. Haas and E. Lin. Ibm federated database technology. http://www.ibm.com/developerworks/db2/library/techarticle/0203haas/0203haas.html.

[14] Enterprise data integration: The case for federated systems solutions using sybase® adaptive server® enterprise 12.5. White paper, Sybase, 2001. Available online.

[15] Graham Bent, Patrick Dantressangle, David Vyvyan, Abbe Mowshowitz, and Valia Mitsou. A dynamic distributed federated database. *ACITA*, September 2008.

[16] Cristian Pérez de Laborda and Stefan Conrad. Relational.owl: a data and schema representation format based on owl. In *APCCM '05: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, pages 89–96, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[17] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*, chapter 16. Cambridge University Press, 2003.

[18] ISO. *Information technology – Database languages – SQL – Part 1: Framework (SQL/-Framework)*. ISO - International Organization for Standardization, 1999.

[19] Antlr parser generator. http://www.antlr.org/.

[20] Sqljep - java sql expression parser. http://sqljep.sourceforge.net/.

[21] Javacc home. https://javacc.dev.java.net/.

[22] Apache derby. http://db.apache.org/derby/.

[23] Eclipse public license 1.0. http://www.eclipse.org/legal/epl-v10.html.

[24] Eclipse data tools platform (dtp) project. http://www.eclipse.org/datatools/.

[25] Gnu affero general public license, version 3. http://www.gnu.org/licenses/agpl-3.0.html.

[26] Pellet: The open-source owl reasoner. http://clarkparsia.com/pellet.

[27] The owl api. http://owlapi.sourceforge.net/.

[28] Jena – a semantic web framework for java. http://jena.sourceforge.net/.

[29] Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In *In 3rd OWL Experiences and Directions Workshop (OWLED-2007*, 2007.

[30] Apache license, version 2.0. http://www.apache.org/licenses/LICENSE-2.0.

[31] Apache poi - the java api for microsoft documents. http://poi.apache.org/.

[32] Merge sort - sorting algorithm animations. http://www.sorting-algorithms.com/merge-sort.

[33] Interface databasemetadata javadoc. http://java.sun.com/j2se/1.3/docs/api/java/sql/DatabaseMetaData.html.

[34] Mozilla public license version 1.1. http://www.mozilla.org/MPL/MPL-1.1.html.

[35] Datamaster - protégé wiki. http://protegewiki.stanford.edu/wiki/DataMaster.