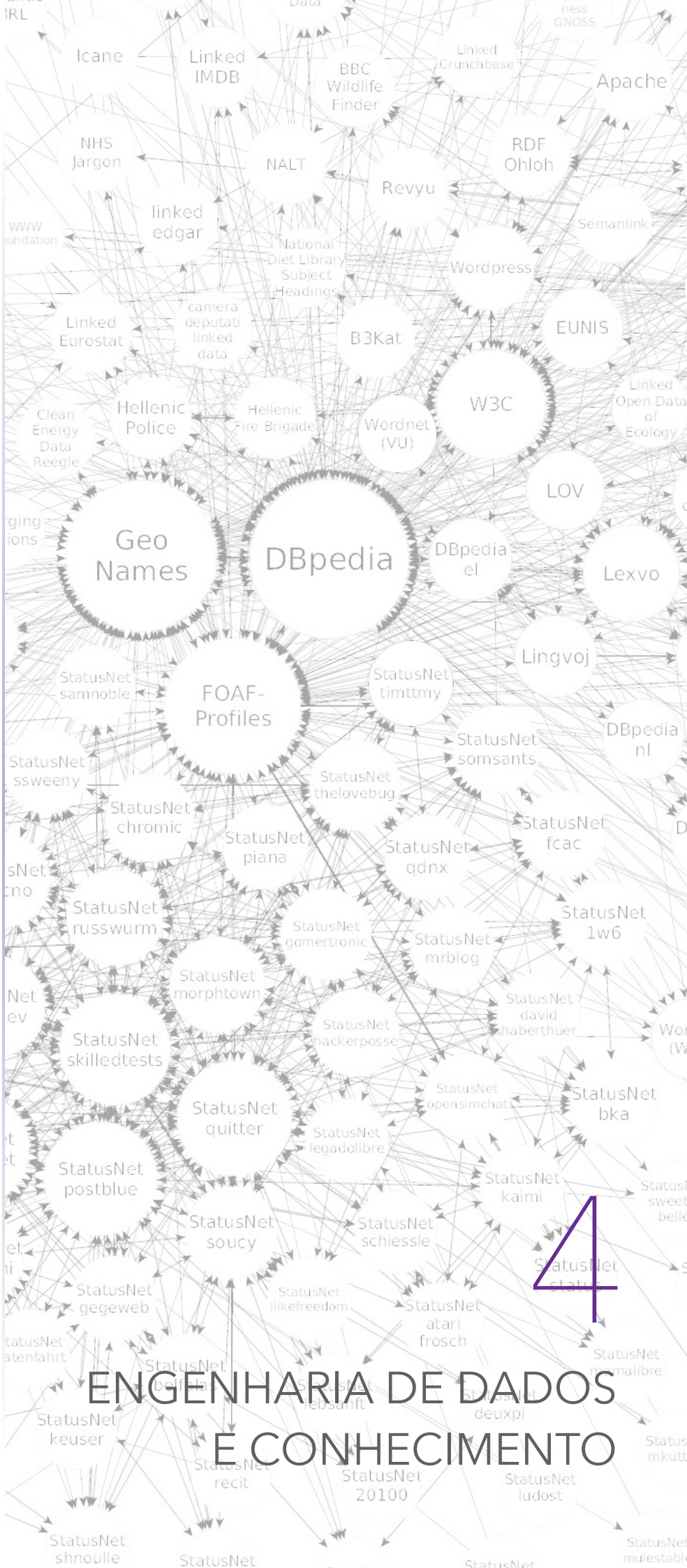
Listening to the data is important… but so is experience and intuition. After all, what is intuition at its best but large amounts of data of all kinds filtered through a human brain rather than a math model?

Steve Lohr



4

ENGENHARIA DE DADOS E CONHECIMENTO

universidade de aveiro
theoria poiesis praxis

Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte**. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Engenharia de Dados e Conhecimento, tal como foi lecionada, no ano letivo de 2017/2018, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

Até à data, nas várias disciplinas estudadas, os dados de que fomos tratando possuíam uma forma bastante **estruturada**, isto é, estavam organizados de uma forma regular e sobre a qual, as demais entidades que dela fossem tratar ou manipular, já saibam que tipo de dados receber *a priori*, a sua hierarquia e outras definições. Nesta disciplina, por outro lado, abordaremos outro tipo de manipulação de dados, entre não-estruturados a semi-estruturados, como objeto da área de **engenharia de dados**.

estruturada

engenharia de dados

Mais à frente, iremos abordar as demais relações que existem entre vários dados e informações produzidas, no produto que se concebe como **conhecimento**. O conhecimento torna-se assim possível, sabendo criar relações entre vários membros, como relações de tipo e subtipo, pertença, entre outros, como iremos verificar mais no fim deste documento.

conhecimento

1. Estrutura de Dados e sua conversão em informação

A informática por si só carateriza-se por ser a área de estudo que pretende transformar quantidades de **dados** em **informação**. A vária informação que hoje em dia circula e sobre a qual trabalhamos pode sempre ser dados (ou matéria-prima) para nova informação. Nesta cadeia de transformação de dados é natural que a complexidade destes se torne cada vez maior, por vias de entropia — enquanto que os primeiros dados da cadeia são bastante estruturados, consoante avançamos nela os dados começam a perder a sua **estrutura**, uma vez que estamos sempre a considerar novos dados, como a informação que foi produzida através de outros.

dados, informação

Dados semi-estruturados

Algures no meio da cadeia de processamento de dados em informação é normal que parte dos dados que se consideram como nova matéria-prima para novas manipulações, tenham uma estrutura em parte regular, outra não tanto. Este tipo de dados não podem ser entregues de uma forma clara a Sistemas de Gestão de Bases de Dados, uma vez que as suas formas de manipulação são muito rígidas e incompatíveis com este tipo de dados.

Esta incompatibilidade deve-se, no fundo, ao facto de não ser possível criar um modelo de dados que sirva para todo o caso e não poder ser criada uma linguagem comum de pesquisa por todos os dados que as SGBDs servem, o que leva a que não seja possível criar formas de otimização e, por conseguinte, bons sistemas de armazenamento da mesma.

Como referido, os **dados semi-estruturados** têm assim um conjunto algo *sui generis* de caraterísticas como possuidores de um esquema que não é fornecido *a priori* — encontrando-se muitas vezes inerente aos próprios dados, pelo que são **descritivos** —, são parciais, evoluem rapidamente com a quantidade de dados fornecidos e podem ter tamanhos consideravelmente grandes, em comparação com os dados originais.

dados semi-estruturados

descritivos

Existem algumas linguagens que permitem a fácil manipulação de dados semi-estruturados, como a JSON ou a XML. Neste documento iremos focar-nos essencialmente na XML, como teremos oportunidade de verificar um pouco mais à frente. Por enquanto, ficamos com um pequeno exemplo de XML para exemplificar um caso de dados semi-estruturados, como podemos ver no Código 1.1.

código 1.1
exemplo de dados semi-estruturados

```
<biblioteca>
  <livro ano="2004">
    <titulo>The Dark Tower</titulo>
    <autor>
      <apelido>King</apelido>
    </autor>
    <editora>Grant</editora>
  </livro>
  <livro ano="1991">
    <titulo>The Waste Lands</titulo>
    <autor>
      <nome>Stephen</nome>
      <apelido>King</apelido>
    </autor>
    <isbn>9788466300223</isbn>
  </livro>
</biblioteca>
```

Em XML os dados estão organizados numa estrutura de grafos, pelo que existem nós (onde eles se encontram), sendo que o esquema final se descreve pelas ligações entre estes. Isto permite que seja fácil descobrir dados novos e carregá-los, tal como integrar dados heterogéneos ou até mesmo pesquisar dados sem ter noção do seu tipo. No entanto, isto também é uma vantagem, uma vez que todos os dados em XML perdem informação sobre o seu tipo, podendo ser interpretados de forma errada numa aplicação cliente, por si só, também, tornando a tarefa de otimização também muito difícil.

Introdução ao XML

De facto, o XML é uma linguagem que permite a aglomeração de dados de uma forma fácil e bem estruturada com a sua lógica de grafo. Mas como é que tudo isto começou? O que é que motivou a sua criação?

Com os tempos, esses que ainda hoje se vivem, a *World Wide Web* (WWW) permitiu que várias quantidades e qualidades de dados fossem criados e fossem tornados acessíveis a grande parte das pessoas. Com estas quantidades exacerbadas de dados tornou-se difícil a tarefa de transmitir, armazenar e apresentar tais informações. Claro que por esta altura já existiam Sistemas de Gestão de Bases de Dados (SGBDs), mas estes também não são os melhores sistemas para receberem como seu argumento de entrada uma quantidade de texto muito grande e sem tratamento que soubesse designar que tipos de dados estão em si inseridos e de que entidades estamos a tratar.

Por esta altura, no entanto, já existia uma tecnologia que permitia fazer precisamente o que se precisava — o HTML. Este tipo de dados transmitia, em texto aberto, todos os tipos de dados de forma indiscriminada e de forma a que uma aplicação cliente pudesse interpretar. No entanto, há um grave problema com o HTML, para que este possa servir para qualquer tipo de aplicação — o número de marcadores disponíveis em HTML são fixos e não designam propriamente a semântica dos dados dos seus elementos. Nasce assim o XML, como uma forma de tirar todos os benefícios do HTML, mas com permissões de criar marcadores diversos, para que se pudesse atribuir as semânticas necessárias para cada elemento.

Por exemplo, enquanto que em HTML a identificação de uma pessoa passaria por descrever uma listagem de textos com a própria legenda em si inserida (Código 1.2), em XML já poderíamos ter cada legenda representada no marcador (Código 1.3).

código 1.2

```
<html>
  <body>
    <h3>Person's Identification</h3>
    <ul>
      <li>Name: Han Solo</li>
      <li>Gender: Male</li>
      <li>Born: After 30 BBY</li>
      <li>Height: 1.80</li>
      <li>Died: 34 ABY</li>
    </ul>
  </body>
</html>
```

código 1.3

```
<person>
  <name>Han Solo</name>
  <gender>Male</gender>
  <born>After 30 BBY</born>
  <height>1.80</height>
  <died>34 ABY</died>
</person>
```

Como podemos verificar em ambos Código 1.2 e Código 1.3 houve uma separação da estrutura de apresentação e de alguns comportamentos específicos da aplicação, sendo agora possível efetuar uma atribuição do valor semântico aos dados e ser feita uma apresentação dos mesmos (tal como uma pesquisa) baseada apenas no conteúdo [3].

Os ficheiros XML, para além da sua extensão que nos sistemas UNIX é meramente descritivo, devem conter uma primeira linha que indica tanto que se trata de um ficheiro é

XML, como qual é a sua versão, codificação ou outras configurações próprias do ficheiro, que pretendemos que uma aplicação cliente saiba antes de manipular o conteúdo do mesmo. Esta linha tem a forma do Código 1.4.

código 1.4

```
<?xml version="1.0" encoding="UTF-8"?>
```

Existem também algumas regras a que todos os elementos deverão ser regidos. Sempre que um elemento é aberto, como por exemplo `<elemento>`, deverá haver um par, no final do bloco, que feche o mesmo, dando uso ao carater `'/'` em `</elemento>`. Por vezes pode ser útil usar **elementos vazios**, isto é, elementos cujo corpo não se descreve pela forma de um bloco, mas antes por uma linha. Um exemplo de elemento vazio poderá ser visto no Código 1.5.

elementos vazios

código 1.5

```
<img align="left" src="static/figure1.jpg"/>
```

Estes elementos também poderão ter **atributos**, isto é, pequenas classificações que se fornecem para acrescentar valor ao elemento semântico XML. Tal atributo, da mesma forma que em HTML, descreve-se na abertura de um elemento, tendo a forma `nome_do_atributo="valor"`.

atributos

Validação de ficheiros XML (XML Schema Definition)

Os ficheiros XML são ficheiros de texto sem nenhuma particularidade especial na perspetiva do sistema operativo, pelo que o seu conteúdo terá apenas sentido se for lido por uma aplicação cliente que seja capaz de interpretar o seu conteúdo. Sendo assim, então como é que podemos saber se um determinado ficheiro se encontra capaz de transmitir a informação da forma pretendida, ou seja, como é que podemos **validar** um ficheiro XML?

validar

Mais do que um formato para a escrita de documentos em texto aberto, o XML é um conjunto de ferramentas que permitem a sua manipulação de forma fácil e intuitiva. Uma primeira ferramenta que iremos abordar neste documento será a **XSD** (*XML Schema Definition*) que permite que se elabore uma moldura ou máscara para o nosso ficheiro e depois que um processador faça a combinação entre o ficheiro XSD e o XML para conferir a sua validação.

XSD

Mas vamos por passos. Um ficheiro XSD (com extensão `*.xsd`) é tal que possui um modelo de dados a ser apresentados num ficheiro XML. Por outras palavras, define os elementos que vão aparecer num documento, especificando quais são os seus elementos-filho e o número destes que existe, a qualidade destes elementos (se são vazios e que tipo de dados é que podem possuir) e a obrigatoriedade dos mesmos.

Vejamos, inicialmente, um exemplo muito simples, reutilizando parte do Código 1.3, criando assim um XSD correto em Código 1.7 para o XML em Código 1.6.

código 1.6

```
<person>
  <name>Han Solo</name>
  <birth>06/12/1977</birth>
  <height>1.80</height>
</person>
```

código 1.7

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3schools.com"
           xmlns="http://www.w3schools.com"
           elementFormDefault="qualified">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="birth" type="xs:date"/>
        <xs:element name="height" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Olhando bem para o Código 1.7 podemos ver que inicialmente temos um elemento `<xs:schema>`. Este elemento designa logo o facto de que estamos a começar um ficheiro que modela um sistema de dados XML e é onde apontamos a versão do XSD passando esta informação a quem vai processar este ficheiro. De seguida, temos a informação de que existe um elemento denominado `person` e que é um tipo complexo, formado por uma sequência de três elementos, entre os quais `name` (que é uma *string*), `birth` (que é uma data) e `height` (que é uma quantidade em precisão simples).

O XSD é constituído assim por dois tipos de componentes. Primeiramente temos os **componentes principais**, como os tipos simples, os tipos complexos, os atributos e os elementos. Destes, já vimos dois deles no exemplo do Código 1.7, mas olhemos agora para o Código 1.9 que define o XML do Código 1.8.

componentes principais

```
<lastname lang="en">Solo</lastname>
<age>50</age>
<vehicle>Millennium Falcon</vehicle>
```

código 1.8

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3schools.com"
           xmlns="http://www.w3schools.com"
           elementFormDefault="qualified">
  <xs:element name="name" type="xs:string">
    <xs:attribute name="lang" type="xs:string" use="optional"/>
  </xs:element>
  <xs:element name="age" type="xs:int"/>
  <xs:element name="vehicle">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Millennium Falcon"/>
        <xs:enumeration value="X-Wing Starfighter"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

código 1.9

No exemplo acima, do Código 1.9 podemos então interpretar o seguinte modelo de dados: devemos ter um ficheiro XML que possui: um elemento `name` (que é uma *string*) e tem um atributo `lang`, que é opcional; um elemento `age` (que é um inteiro); e um elemento `vehicle` que é um tipo simples de dados, sendo necessariamente uma *string* entre as opções `Millennium Falcon` e `X-Wing Starfighter`.

Neste pequeno excerto de código temos uma quantidade grande de exemplos também dos **componentes secundários** do XSD, como grupos de atributos e restrições de identidade — embora também existam grupos de modelos e notações. [3]

componentes secundários

Caso a caso, em XSD podemos ter **elementos simples**, que são elementos que só possuem texto, não podendo ter mais elementos no seu interior, sendo que o texto poderá assumir qualquer tipo acessível pela norma XSD. Estes elementos têm a forma do Código 1.10.

elementos simples

```
<xs:element name="…" type="xs:…" default="…" fixed="…"/>
```

código 1.10

No Código 1.10 temos alguns atributos possíveis de um elemento, entre os quais o `default` que define o valor por omissão, caso nenhum seja atribuído, ou o `fixed`, que define um e um só valor por omissão, não podendo ser alterado nunca.

Se um elemento (não confundir com a especificação de um elemento em XSD) tiver atributos, então não poderá ser considerado como um elemento simples, mas antes composto, uma vez que terá mais um componente em si inserido, neste caso, um ou mais `attribute`. Os atributos têm a forma do Código 1.11.

```
<xs:attribute name="…" type="…" use="…"/>
```

código 1.11

Todos os atributos são opcionais por omissão, mas note-se o atributo XSD `use` que poderá assumir o valor de `optional` ou `required`, caso seja necessário.

Um **elemento complexo** é, então, um elemento que tem outros elementos e/ou atributos no seu interior. Existem apenas quatro tipos de elementos complexos: elementos vazios, elementos que contêm somente outros elementos, elementos que contêm somente texto e elementos que contêm outros elementos e texto. No Código 1.7 podemos reparar que temos um elemento complexo com nome `person`. Tal elemento contém um `sequence` dentro. Este marcador permite que se indique uma ordem de elementos a serem apresentados, quer estes sejam simples ou complexos.

elemento complexo

Existem outros marcadores possíveis de implementar em XSD que poderás verificar na sua página oficial[1].

Espaços de nomes XML (XML Namespaces)

No início da secção anterior referimos que um determinado ficheiro, que dá pelo nome do esquema XML, permite que seja criada uma estrutura modelo de um ficheiro XML. Para o definir estudámos alguns dos seus componentes e tivemos oportunidade de verificar que já existem um conjunto de marcadores específicos para a sua manipulação. Sendo o XML uma linguagem que permite, virtualmente, a criação de qualquer marcador, porque é que só usamos o marcador `element` quando pretendemos especificar um elemento e não outro qualquer, que pudesse dar uma semântica semelhante?

Existe uma pequena configuração que pode ser aplicada nos ficheiros XML que é o **espaço de nomes** (em inglês e mais vulgarmente denominado de *namespace*). O que o espaço de nomes permite fazer é especificar um conjunto de nomes de marcadores que já possuem uma semântica a si associada, pelo que inequivocamente, o nome `element` dará apenas e somente para referir um novo elemento. Note-se, no entanto, que o espaço de nomes, para ser usado, deverá ser atribuído a um nome e explícito em cada uma das operações que sejam feitas com ele. Por exemplo, no caso do Código 1.7 (agora replicado no Código 1.12 com destaque nestas partículas) temos que o espaço de nomes para o XSD foi atribuído ao nome `xs`, sendo sempre usados nomes próprios do XSD com o prefixo `xs`. [4]

espaço de nomes

código 1.12

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3schools.com"
           xmlns="http://www.w3schools.com"
           elementFormDefault="qualified">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="birth" type="xs:date"/>
        <xs:element name="height" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Caso não seja usado qualquer prefixo algures no Código 1.12 para especificar um marcador `element`, então será considerado um marcador sem semântica aparente com nome `element`, em nada relacionado com um marcador XSD. Se pretendermos aplicar um determinado espaço de nomes a um documento por completo, então podemos aplicar um **default namespace**, isto é, declarar um espaço de nomes sem o atribuir a um prefixo.

default namespace

O código XSD, depois de feito, pode ser interpretado e processado por um variado conjunto de ferramentas já incluídas nos próprios IDEs ou até mesmo *online*, onde se têm de fazer acompanhar de um ficheiro XML para testar a sua validação.

Transformações de ficheiros XML (XSLT)

A criação da linguagem XML partiu, como já referimos, da necessidade de arranjar um bom meio de transporte de grandes quantidades de dados em forma de texto aberto,

[1] Ver mais em https://www.w3schools.com/xml/schema_elements_ref.asp.

entre várias máquinas. Uma primeira sugestão de solução para este problema era o uso do HTML, mas dado o limitado número de marcadores que possui, criou-se uma alternativa ao mesmo denominada de XML, que partilhava muitas das suas propriedades. Na verdade, tanto o XML como o HTML herdam propriedades de uma linguagem-mãe: a SGML, sigla inglesa para *Standard Generalized Markup Language*. Por esta mesma razão, não deverá haver grandes problemas em tentar comutar entre o HTML e o XML, perdendo apenas o apoio dos vários marcadores que o XML fornece. Mas como é que podemos fazer esta transformação?

Da mesma forma que foi criado o XSD para a validação de ficheiros XML, houve uma ferramenta que foi criada para a transformação destes, denominada de **XSLT** (*Extensible Stylesheet Language Transformations*). Esta ferramenta permite não só a transformação de um ficheiro XML para HTML (ou vice-versa), como uma transformação de um qualquer formato *markup* para um outro formato *markup* (ou igual), como a fusão entre dois XML para dar um XML. [5]

XSLT

Esta ferramenta possui uma quantidade algo elevada de elementos que a caracterizam, no entanto, neste documento só iremos referir alguns que se podem considerar como essenciais, tentando sempre mostrar a proximidade que tentaram implementar nela às linguagens de programação imperativas.

Uma tarefa muito habitual em linguagens de programação imperativa, quando se necessitam de mais conteúdos já criados em elementos exteriores ao código atual, é a **importação** de funções. Esta importação, em XSLT, também pode ser feita através do elemento `<xsl:import>`, que inclui outra folha de estilo em si, sendo que as definições e regras do modelo no documento têm precedência sobre as regras do modelo e definições na folha de estilo importada. Para indicar a folha de estilo em particular usa-se o atributo de `<xsl:import>` denominado por `href`. No Código 1.13 podemos ver um exemplo de aplicação, onde consideramos que existe um ficheiro XSLT já definido com um determinado estilo, com o nome `DiscCatalog.xslt`.

importação

código 1.13

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="DiscCatalog.xslt"/>

  <xsl:template match="/">
    <xsl:apply-imports/>
  </xsl:template>
</xsl:stylesheet>
```

O Código 1.13 descreve assim um ficheiro XSLT que por si importa um outro ficheiro XSLT onde estão definidos estilos. Depois, juntamente com o verbalismo de um novo elemento denominado de `<xsl:template>` aplica os estilos importados a partir do ponto `"/"` da árvore XML do ficheiro onde se aplicar estas transformações.

Note-se que, como já foi referido, sendo que um ficheiro A importa B terá uma precedência de importação, da maior para menor, de A B, então se um ficheiro A importar B e C, sendo que B importa D e C importa E, então teremos uma precedência final de A C E B D.

A instrução `<xsl:template>`, como referido, serve então para construir um modelo de representação final da estrutura dos dados. Claro está que todos os elementos dentro de um *template* serão abrangidos pelas alterações que sejam feitas sobre este, no entanto, caso o atributo da instrução do *template*, `apply-templates`, `select` seja usado, então apenas os membros selecionados serão abrangidos pelas alterações, como mostra o Código 1.14.

código 1.14

```
<xsl:template match="/">
  <table>
    <xsl:apply-templates select="/sales/domestic"/>
  </table>
</xsl:template>
```

O ficheiro final onde as várias alterações se refletem é descrito e indicado com a instrução `<xsl:output>` sendo que a sintaxe do mesmo encontra-se definida no Código 1.15.

código 1.15

```
<xsl:output method="xml|html|text|name"
            version="string"
            encoding="string"
            omit-xml-declaration="yes|no"
            standalone="yes|no"
            doctype-public="string"
            doctype-system="string"
            cdata-section-elements="namelist"
            indent="yes|no"
            media-type="string"/>
```

Das várias opções de atributo permitidas na sintaxe do `<xsl:output>`, como podemos ver no Código 1.15, temos três que são muito usados: o atributo `method` especifica o tipo do ficheiro de saída (sendo por norma XML, embora dependa do valor da transformação); o atributo `encoding` que permite designar a codificação sobre a qual o valor final é traduzido; o atributo `indent` que especifica se se pretende que o objeto final tenha, ou não, indentação à esquerda da árvore. [3]

Mais uma vez, e de forma semelhante a uma linguagem de programação imperativa, também temos a possibilidade de criar variáveis e parâmetros, sendo que uma variável é um nome que se pode vincular a um valor. Existem dois elementos que podem ser usados para elementos: eles são o `<xsl:variable>` (suporta apenas um valor padrão) e o `<xsl:param>` (permitem a passagem de parâmetros que são usados em lugar o valor padrão). No entanto, tanto um como o outro possuem um atributo `name` que é obrigatório e que especifica o nome da entidade. Vejamos o exemplo no Código 1.16, onde por um lado criamos um parâmetro que não é nada mais que um ponteiro para um ficheiro de nome `starwars.xml` e por outro criamos uma variável com nome `releaseDate` que usa o ficheiro instanciado pelo parâmetro para chegar ao campo `releaseDate` com valor 6 de dezembro de 1977.

código 1.16

```
<xsl:param name="saga" select="document('starwars.xml')"/>
<xsl:variable name="releaseDate" select="$starwars/episode/[releaseDate=06121977]"/>
```

Outros elementos importantes são por exemplo a `<xsl:value-of>` que permite que se crie um nó de resultados, sendo que o atributo `select` permite designar a forma sobre a qual se pretende o resultado (fazer uma seleção do que se quer). O objeto resultante é convertido para uma *string* uma vez que o ficheiro final estará em texto aberto e perderá a capacidade de distinguir o seu tipo de dados. Este nó também só se criará caso exista algo com que se possa preencher o seu espaço. Vejamos um exemplo de aplicação no Código 1.17 onde selecionamos o valor de um campo `name` de um ficheiro XML.

código 1.17

```
<xsl:value-of select="$starwars/episode/name"/>
```

A instrução do Código 1.17, por si só, se houver vários episódios do Star Wars (que existem) não terá muita lógica se eu não pretender retirar o valor de cada um dos vários episódios. Para poder percorrer os dados como se de uma lista se tratasse, isto é, para poder iterar sobre cada um dos episódios, mais em particular, teríamos de usar a instrução `<xsl:for-each>` do XSLT. Neste caso, e consoante a seleção feita para iteração sobre o campo do atributo `select`, conseguiríamos abranger os nomes de todos os episódios, como podemos ver no Código 1.18.

código 1.18

```
<xsl:for-each select="$starwars/episode">
  <xsl:value-of select="name"/>
</xsl:for-each>
```

Dos elementos principais de uma linguagem imperativa já só falta mesmo referir como fazer uma avaliação de uma condição *if*. Para isso podemos usar as instruções de `<xsl:if>` e a `<xsl:choose>` para escolhermos uma opção entre um conjunto delas possíveis (análogo a um *switch*).

A sintaxe de uma cláusula *if* é então a seguinte do Código 1.19.

código 1.19

```
<xsl:if test="expression">
  <!-- content -->
</xsl:if>
```

No Código 1.19, a `expression` é tal que tem a forma de uma expressão Booleana convencional, como `a > b`, e caso esta seja considerada verdadeira, então o conteúdo do seu bloco será executado.

Uma vez que não existe nenhum bloco para um caso *else*, podemos usar a instrução `<xsl:choose>` que, de forma algo análoga a um *switch...case* ou mais especificamente a um *if...else* com possibilidade de avaliação de muitas condições, permite que se escolha uma entre várias alternativas de transformação, como podemos ver no caso explícito no Código 1.20.

código 1.20

```
<xsl:choose>
  <xsl:when test="price > 10">
    <xsl:value-of select="artists"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="artist"/>
  </xsl:otherwise>
</xsl:choose>
```

O XSLT, tal como o XSD, necessita que exista um processador próprio para que possa ser devidamente executado. Para a sua execução recomendamos processadores como o *lxml*[2] ou o *Saxon*[3].

Expressões delimitadoras da árvore XML (XPath)

No uso do XSLT muitas vezes aplicou-se uma lógica de endereçamento de conteúdos num documento XML semelhante ao do sistema UNIX para os diretórios e ficheiros. Na verdade, por cada vez que usamos o atributo `select` na linguagem do XSLT estamos a usar uma outra ferramenta do XML que se chama **XPath** e que permite que se faça do endereçamento de uma região específica do ficheiro XML. [6]

XPath

Na verdade, uma expressão XPath permite que nós sejam referidos tanto pela sua **posição absoluta**, isto é, indicando todo o caminho até uma determinada região de código através da raiz, como pela sua **posição relativa**, todo o caminho até uma determinada região de código desde o ponto atual de posição no tratamento de dados. Como já tivemos oportunidade de verificar, esta é a ferramenta que permite ao XSLT realizar cálculos simples como numeração, referências cruzadas de figuras, tabelas e equações. Mais, estas manipulações por parte do XSLT são feitas sem que este conheça de forma prévia a estrutura do documento em que trabalham.

posição absoluta
posição relativa

Com o XPath há, de facto, uma diferença em comparação aos SGBDs, que necessitam sempre de saber previamente a estrutura dos dados que gere, a todo o instante. Isto deixa um pouco a desejar em termos de otimização e capacidades para efetuar comparações e ordenações, mas deixa espaço para que tais capacidades possam ser descobertas e evoluídas. Por exemplo, se usarmos as duas ferramentas que já estudámos anteriormente (XSD e XSLT) podemos melhorar em termos de otimização (sabendo o esquema de um ficheiro XML, caso este seja grande, podemos ter um conhecimento algo prévio da estrutura dos dados) e em termos de comparações e ordenação (pelo que sabendo o esquema dos dados também conseguimos saber que tipos é que eles têm).

Um documento XML, como já devemos saber, é então constituído por um conjunto de nós. Na verdade, podemos definir o documento como uma **árvore**, sendo que possui sempre um e um só nó raiz e conjuntos de nós-filhos deste, com relações entre si. Na linguagem

[2] http://lxml.de.

[3] http://saxon.sourceforge.net.

XPath existem 7 tipos de nós: nós de elemento (os elementos da linguagem XML que temos vindo a referir até agora, identificados por marcadores), nós de atributo (atributos aos elementos), nós de texto (conteúdo de cada elemento, desde que não seja elemento também — **valores atómicos**), nós de espaço de nomes (nós que identificam o contexto dos marcadores que identificam os elementos e atributos), nós de instrução (nós que possuem caraterísticas de linguagens de programação imperativa), nós-raiz e nós de comentário (nós que não acrescentam valor ao documento XML, mas que permitem que algum texto seja inserido para comentar a estrutura ou valores).

valores atómicos

Um dos conceitos mais importantes de perceber na tecnologia XPath é o **contexto**. Por contexto, pretendemos referir o local atual de leitura de um ficheiro XML. Consideremos o Código 1.18, onde no bloco `for-each` entramos em `$starwars/episode`. Se repararmos bem, dentro do bloco queremos o valor do campo `name`. Tal campo, contrariamente ao que se possa pensar, não está a exprimir uma posição absoluta, mas antes uma posição relativa, sendo que na verdade estamos a tentar obter o valor de `$starwars/episode/name`, para cada valor de `episode`.

contexto

Como também podemos ver pelo caminho referido anterior (`$starwars/episode/name`), o XPath também possui uma sintaxe própria. Na verdade, o XPath possui uma forma específica para designar cada um dos tipos de nós que discrimina (à exceção dos nós comentário, de instrução e de texto, que designa de forma diferente). Abaixo, podemos ver o significado da sintaxe do XPath para a seleção de nós:

- `person` — seleciona todos os elementos `person`, dentro de um contexto;
- `/people` — seleciona o elemento-raiz de nome `person`[4];
- `people/person` — seleciona todos os elementos `person`, filhos de `people`;
- `//people` — seleciona todos os elementos `people` independentemente do contexto;
- `people//lastname` — seleciona todos os elementos `lastname`, independentemente do contexto, desde que filhos do elemento `people`;
- `//@id` — seleciona todos os atributos denominados `id`.

Claro está que, por exemplo, o último caso "seleciona todos os atributos denominados `id`" não fará muito sentido de ser aplicado na prática, mas antes se for todas as pessoas com ID `@id`. Para fazer este tipo de expressões em XPath podemos usar combinações como as seguintes:

- `/people/person[1]` — seleciona o primeiro[5] elemento `person`, filho de `people`;
- `/people/person[last()]` — seleciona o último elemento `person`, filho de `people`;
- `/people/person[last()-1]` — seleciona o penúltimo elemento `person`, filho de `people`;
- `/people/person[position() < 3]` — seleciona os dois primeiros elementos `person`, filhos de `people`;
- `//person[@id]` — seleciona todos os elementos `person` que possuam o elemento `id`;
- `//person[@id='p1234']` — seleciona todos os elementos `person` que possuam o elemento `id` igual a `p1234`;
- `/people/person[@born > 1929]` — seleciona todos os elementos `person` que possuam o elemento `born` superior que `1929`, filhos de `people`;
- `/people/person[@born > 1929]/lastname` — seleciona todos os elementos `lastname`, filhos de elementos `person` que possuam o elemento `born` superior que `1929`, filhos de `people`;

Em XPath também poderá haver a necessidade de selecionar nós desconhecidos. Se quisermos selecionar todos os elementos filho do elemento `people` então podemos escrever `//people/*`, todos os elementos no documento com `//*` e com atributos `//[@*]`.

---

4 Note-se que sempre que um caminho começa por um carater "/" isto significa que estamos a iniciar a designação de um caminho absoluto para um elemento.

5 As listas em XPath, contrariamente à grande maioria das linguagens, identifica o primeiro elemento de uma lista com o número 1.

Bases de dados em XML e consultas em XQuery

Os ficheiros XML, por si, são bastante bons a organizarem dados, de forma a que estes continuem a possuir e explicitar relações entre os vários objetos nele designados e de forma a que estes possam permanecer em texto aberto. Já referimos também, que uma das grandes vantagens do uso de XML é que é usado por muitas aplicações clientes, pelo que facilmente se consegue arranjar dois ou mais sistemas que permitam a troca de mensagens usando esta norma de documento.

No entanto, há alguns problemas que, em comparação com bases de dados relacionais já nossas conhecidas como MySQL ou SQL Server deixam o XML, por si só, um pouco aquém das nossas perspetivas e da sua utilidade real, enquanto suporte para a salvaguarda de dados e consequente informação. Um desses problemas está na manutenção da base de dados, pelo que um ficheiro XML não possui (ou não consegue garantir sozinho) qualquer serviço ou atividade que designe **transações** ou **cópias de segurança**. Por esta mesma razão, criarão-se, com os tempos e as necessidades, **bases de dados XML**, que permitem a implementação deste tipo de mecanismos.

transações, cópias de segurança, bases de dados XML

Note-se, porém, que haver uma base de dados em XML, dado que o ficheiro XML raiz é muito verboso, também terá um tamanho consideravelmente grande e não será necessariamente compatível com Sistemas de Gestão de Base de Dados mais comuns de se usar.

Então mas como é que tais bases de dados poderão preservar XML? Ora, dado que os documento XML possuem uma estrutura intrinsecamente em árvore, então estes dados facilmente poderão ser sujeitos a correspondências diretas com bases de dados relacionais (onde cada grau de elemento poderá formar uma entidade — técnica denominada de **internal schema parsing**), poderão ser guardados como texto XML dentro de uma instância de uma tabela de uma base de dados relacional, como um mero **sistema de ficheiros** ou mesmo num tipo especial para bases de dados XML denominado de **NXD** (sigla inglesa para *Native XML Database*). [7]

internal schema parsing
sistema de ficheiros
NXD

Mais em particular, uma base de dados NXD permite que as várias ferramentas XML consigam trabalhar em uníssono sobre os dados, de forma contínua, isto é, garantindo que as entradas, saídas e os formatos guardados são todos iguais, suportando transações e permitindo a validação com XSD. Uma vez que foi feita mesmo para a manipulação de documentos XML, é normal que o desempenho deste mecanismo seja bastante bom.

Exemplos de gestores de bases de dados NXD são a BaseX, a eXist ou a Sedna.

Da mesma forma que existem **consultas** (em inglês *queries*) nos SGBDs, no caso da NXD também é possível criar consultas à base de dados através de mais uma ferramenta suportada e criada especificamente para o XML — o **XQuery**. O XQuery é uma linguagem normalizada para combinar documentos, bases de dados, páginas Web, entre outros, com uma *interface* bastante fácil de utilizar.

consultas
XQuery

Por outras palavras, o XQuery é capaz de extrair informação de uma base de dados NXD em uso num serviço Web, gerar relatórios em dados preservados numa base de dados, pesquisar parcelas textuais na Web com termos relevantes e compilar os resultados, selecionar e transformar dados XML para XHTML (para ser publicado na Web) ou dividir um documento XML que representa múltiplas transações em múltiplos outros documentos.

O XQuery está criado sobre expressões XPath, pelo que partilham o mesmo modelo de dados e suportam ambos as mesmas funções e operações. Contrariamente ao desenvolvimento da linguagem XSLT, o XQuery tem a sua base em linguagens funcionais, uma vez que cada consulta que com ela é criada, uma nova expressão é avaliada, podendo, estas, ser combinadas entre si para formar expressões ainda maiores. Esta linguagem é constituída por expressões de caminho (*path expressions*), construtores de elementos, expressões **FLWOR** (acrónimo de *For*, *Let*, *Where*, *Order by*, *Return*), expressões de ordenação, condição e quantificadoras, funções e operações e tipos de esquemas XML.

FLWOR

Vejamos, inicialmente, alguns dos literais desta nova linguagem. Os inteiros, definidos pelo tipo `xs:integer`, têm a forma `1`, `-2` ou `+3`. Em termos de vírgula flutuante, temos dois tipos, um definido por `xs:decimal` (precisão simples) e outro definido por `xs:double`.

Caso não queiramos representar uma quantidade numérica, também temos o tipo definido por `xs:string`, que se usa através de apóstrofes como em `'Hello World'`, ou através de aspas, como `"<p>Hello World</p>"`. Os comentários, por sua vez, são feitos começando uma linha por `(:` e terminando com `:)`, como podemos ver no Código 1.21.

```
(: This is a comment line :)
```

código 1.21

Em termos de **funções** existem principalmente duas que consideramos bastante importantes: a `document()` e a `collection()`. A função `document()` permite que se retorne um documento XML completo, sendo que o seu nome (caminho) é passado como seu argumento: por exemplo `document("starwars.xml")`. Já a função `collection()` permite que se retorne uma **coleção** associada com um caminho, pelo que geralmente se usa para identificar uma base de dados numa consulta. Por exemplo, para ver um conjunto de elementos `video` filhos de `videos` numa base de dados `blockbuster`, temos uma expressão como `collection("blockbuster")/videos/video`.

funções

coleção

As expressões FLWOR são quase como o núcleo de toda a linguagem XQuery, sendo o que lhe dá necessidade na sua utilização. Trabalhando de forma semelhante aos `SELECT FROM WHERE` em SQL, uma expressão FLWOR junta variáveis a valores em cláusulas `for` e `let`, criando novos resultados. Inerente à própria natureza do XQuery, uma combinação do uso de junções de variáveis pelas cláusulas `for` e `let` é chamada de **tuplo**.

tuplo

Vejamos então cada uma das expressões FLWOR em ação. A primeira expressão é a `for` e é usada para iterar sobre cada elemento de uma sequência XPath. Por exemplo, olhando para o Código 1.22 temos um caso em que com uma cláusula `for` iteramos sobre um tuplo para imprimirmos os seus valores internos.

```
for $index in (100, 200, 300)
return <tuple>{ $index }</tuple>

(: resultado da expressão acima :)
<tuple>100</tuple>
<tuple>200</tuple>
<tuple>300</tuple>
```

código 1.22

A segunda expressão FLWOR a analisar é a `let`, onde se cria uma nova variável com um valor específico, retornando sempre em forma de tuplo. No Código 1.23 podemos ver um código semelhante ao Código 1.22 onde ao invés da cláusula `for` usamos uma `let`. Note-se que no caso do uso de `let`, deixamos de usar a palavra `in` para usar a atribuição denotada por `:=`.

```
let $index := (100, 200, 300)
return <tuple>{ $index }</tuple>

(: resultado da expressão acima :)
<tuple>100 200 300</tuple>
```

código 1.23

Continuando temos agora a instrução `where` que através de uma condição usando XPath filtra dados. Por exemplo, considerando que temos uma base de dados que preserva os dados de uma biblioteca, podemos listar os livros cujo número de autores seja superior a 2, através da função nativa `count()`, como podemos ver no Código 1.24.

```
for $book in collection("library")//book
let $authors := $book/author
where count($authors) > 2
return $book/title
```

código 1.24

As expressões da cláusula `where` também podem ser de um outro tipo admitido pelo XQuery — tipo quantitativo. Através das construções `every $variável in <conjunto> satisfies <condição>` ou `some $variável in <conjunto> satisfies <condição>`, podemos identificar e filtrar casos em que a `condição` se aplica todas as vezes ou algumas, respetivamente, num conjunto cujos filhos são elementos. Por exemplo, considerando que temos uma base de dados de filmes, podemos escolher todos aqueles que possuem género classificado, pelo

menos, como sendo de animação, através de uma cláusula com `some`, como podemos ver no Código 1.25. Já no Código 1.26 podemos ver o caso em que todos os filmes selecionados têm exclusivamente o género classificado como animação.

código 1.25
```
for $movie in collection("blockbuster")//movie
where some $genre in $movie/genres/genre = "Animation"
return $movie/title
```

código 1.26
```
for $movie in collection("blockbuster")//movie
where every $genre in $movie/genres/genre = "Animation"
return $movie/title
```

Se quisermos ordenar os resultados de uma consulta, então temos que usar a cláusula `order by`. Para ordenar os resultados do Código 1.24 por ordem alfabética apenas temos de juntar `order by`, como podemos ver no Código 1.27.

código 1.27
```
for $book in collection("library")//book
let $authors := $book/author
where count($authors) > 2
order by $book/title
return $book/title
```

Também nos é permitido, já fora das expressões FLWOR, criar instruções que desviam o curso natural de execução das consultas — cláusulas *if…else*. Estas cláusulas têm a forma de `if (condição) then {valor} else {valor}`.

Existem ainda outras funções importantes, como a `contains()` para saber se o segundo argumento da função pertence ao conjunto indicado como primeiro argumento, a `data()` para retirar os valores presentes nos elementos atómicos, a `text()` para retornar os conteúdos dos elementos atómicos como texto, a `distinct-values()` para retornar apenas o conjunto sem repetições formado por uma lista entregue como argumento, entre outras…

Tal como é permitido usar funções, também nos é permitido criar novas funções, denominadas de **UDF**s (sigla de *User Defined Functions*), que possuem a sintaxe do Código 1.28.

UDF

código 1.28
```
declare function <prefixo>:<nome_da_função>($parâmetro as tipo_de_dados)
as tipo_de_retorno {
  (: code block :)
};
```

No Código 1.29 temos um exemplo de uma UDF que lista livros, de uma base de dados de uma biblioteca, de um autor, ordenando-os por ordem alfabética de título.

código 1.29
```
declare function local:books_by_author($authorName)
as element()* {
  for $book in collection("library")//book
  where contains($books/author[1], $authorName)
  order by $book/title
  return $book/title
};
```

Para executar a UDF do Código 1.29 apenas temos de executar `local:books_by_author("Exupéry")`.

Até agora temos visto como consultar dados numa base de dados, mas como é que fazemos para os alterar? Ora, para isso existe mais uma ferramenta, anexa ao XQuery, denominada de **XUpdate**, que serve para fazer as diversas atualizações de campos numa base de dados XML.

XUpdate

Por **atualizações** pretendemos referir tanto a adição de um novo nó, a sua eliminação, uma mera modificação de um dos seus valores ou até mesmo a cópia de um nó modificado para um novo com nova identidade.

atualizações

Em termos de sintaxe, e dado que o XUpdate está anexo ao XQuery, os seus comandos serão entregues como retorno das consultas feitas.

Para fazermos uma inserção devemos assim respeitar a seguinte sintaxe (Código 1.30).

```
(…) return insert node <novo_nó> into <nó_existente>
```
código 1.30

No local de `into` podemos colocar modificadores como `before` ou `after`.

Para fazermos uma eliminação de um nó, devemos seguir a seguinte sintaxe (Código 1.31).

```
(…) return delete node <nó_a_eliminar>
```
código 1.31

Para inserirmos um atributo novo num nó temos de seguir a seguinte sintaxe do Código 1.32.

```
(…) return insert node attribute <nome_atributo> { "novo_valor" } into <nó_em_XPath>
```
código 1.32

Da mesma forma que podemos inserir um atributo também o conseguimos apagar, como podemos ver no Código 1.33.

```
(…) return delete node $<local_do_nó>/@<atributo_a_apagar>
```
código 1.33

Trocar um nó é fazer uma operação de `replace`, como podemos ver no Código 1.34.

```
(…) return replace node <nó_a_trocar> with <nó_antigo>
```
código 1.34

Para trocar o valor de um nó apenas temos de acrescentar `value of` antes de `node` na sintaxe do Código 1.34, mas ainda assim podemos querer mudar a identidade de um nó, através de uma operação de renomeação, como podemos ver no Código 1.35.

```
(…) return rename node <nó_para_mudar_nome> as "novo_nome_do_nó"
```
código 1.35

Emissão de Feeds

Como já tivemos oportunidade de referir, os documentos XML foram feitos para que se pudesse transmitir entre duas ou mais máquinas uma quantidade de informação algo considerável, mas de forma fácil, organizada e em texto aberto. Por volta dos anos 1999 este tipo de documentos começou a ser criado orientado à troca de mensagens, de forma mais regular e atualizada, sobre notícias de domínio geral.

Para fazer deste serviço um serviço atualizado e regular, é importante que as empresas que servem tal documento estejam a fazê-lo de forma recorrente, isto é, por vias de uma **emissão** de dados constante ao longo do tempo. Nasce assim o **RSS** (sigla de *Really Simple Syndication*), como uma forma de sumariar conteúdo visível na Web para um ficheiro pequeno e muito simples de carregar.

emissão, RSS

Ao longo dos tempos começaram a surgir grandes páginas que liam vários **feeds** RSS, isto é, **agregadores** de serviços, como o Google News®.

feeds
agregadores

O tipo de ficheiro RSS contém uma estrutura muito regular e fixa, como podemos ver no Código 1.36.

```
<rss>
  <channel>
    <item></item>
    <item></item>
    <item></item>
  </channel>
  <channel>
  </channel>
</rss>
```
código 1.36

Da mesma forma que a estrutura-base tem sempre de ser igual (um conjunto de canais, cada um com um conjunto de itens), o conteúdo de um item também precisa de ter alguns elementos como um título, um *link*, uma descrição (que na verdade é o próprio con-

teúdo), mas também elementos opcionais, como um idioma, marcação de direitos de propriedade, uma data de publicação, uma categoria ou até um *time-to-live*, para saber de quanto em quanto tempo é que o *feed* é atualizado.

Uma vez que o RSS por si é bastante limitado, por várias razões, sendo a primeira a questão da estrutura fixa, em 2005 surgiu uma alternativa ao RSS, definida no RFC 4287, chamada de **Atom**. O Atom, por exemplo, permite que o tipo de dados sobre os vários elementos que nele são designados possam ser diferentes, uma vez que aceita a escolha de diferentes espaços de nomes. Isto acontece porque o Atom tira partido das várias ferramentas XML que foram surgindo ao longo dos tempos, algo que o RSS nunca poderia ter feito, uma vez que foi criado antes das próprias ferramentas.

RFC 4287

Atom

Pela mesma razão, o Ator permite **modularidade**, isto é, permite que vários outros documentos sejam reutilizados fora do contexto de um *feed* Atom.

modularidade

2. Engenharia de Conhecimento

Uma vez processados os dados em informação, isto é, uma vez gerando alguma utilidade de um conjunto de dados que fora processado, então devemos ser capazes de criar e gerir **relações** entre as várias informações. À criação e gestão de relações entre várias informações e possíveis gerações de dados é dado o nome de **conhecimento**.

relações

conhecimento

De forma análoga ao conhecimento que é aprendido por um ser humano, nas disciplinas de manipulação de dados (como é o caso de *data science*), o conhecimento é então um fenómeno de consolidação de informações, estas, que surgiram da tradução de dados retirados de uma determinada fonte.

As regras e as várias aplicações deste conhecimento são determinadas pela **engenharia do conhecimento**, que procura sempre conseguir uma melhor solução com o maior rendimento possível em termos de como atingir o conhecimento a partir de um determinado conjunto de dados ou informação.

engenharia do conhecimento

Redes semânticas

Como já tivemos oportunidade de verificar, uma das fontes de recolha de dados que foi criada ao longo do tempo foi a **WWW** (sigla inglesa de *World Wide Web*) que surgira em 1989 pelas mãos de Tim Berners-Lee no CERN (acrónimo francês de *Conseil Européen pour la Recherche Nucléaire*). Esta rede de fornecimento de dados provém da necessidade de partilha destes e do seu rápido acesso entre todas as equipas constituintes do CERN.

WWW

Esta invenção do WWW, com o sucesso que teve dentro das instalações do CERN, levou Tim Berners-Lee a imaginar a sua aplicação fora do *campus*, querendo assim uma rede que permitisse a partilha de dados com a forma de conteúdos em texto, transações entre duas ou mais partes, ou multimédia. Estas trocas de dados permitiriam assim que várias máquinas pudessem comunicar entre si. Foi assim que surgiu um novo conceito — o de **rede semântica** — sendo que, quando duas máquinas partilharem dados entre si, ambas terão de partilhar também um contexto de aplicação dos dados, para que uma transação possa ocorrer sem qualquer tipo de problemas (como uma má aplicação dos dados). Esta definição também já foi estudada em Introdução à Inteligência Artificial (a3s1).

rede semântica

Hoje em dia, este sonho que outrora fora o de Tim Berners-Lee, é quase o nosso dia-a-dia em curso, uma vez que por cada vez que acedemos a uma página na WWW não só nós aprendemos algo com a nossa visita, mas também a nossa máquina comunica com um servidor em busca de uma página, confirmando se aquilo que esta lhe dá é o que foi pedido (teste de contexto), através de **metadados**, compreensíveis pelas máquinas, acerca das páginas e das relações entre elas. [9]

metadados

Contudo, a aplicação atual das redes semânticas possui um problema — esta só existe entre os conceitos de páginas e das suas relações, mas não entre os seus conteúdos. Hoje em dia, por exemplo, se precisarmos de preparar uma viagem, teremos que visitar um grande conjunto de páginas em busca do que realmente precisamos para a fazer: precisamos de verificar páginas para o transporte, depois para o hotel, para o aluguer do car-

ro, entre outros... Este tipo de tarefa acaba por ser bastante exaustiva, uma vez que os locais onde vamos não possuem qualquer tipo de informação de que realmente estamos à procura de um serviço como assistência a uma viagem (por exemplo, um aluguer de um carro não precisa de ser relacionado com uma viagem, mas antes com alguma necessidade por ter o próprio numa oficina).

Mesmo que tenhamos um serviço que faça esta pesquisa exaustiva por nós (exemplos de serviços que já existem para pesquisar informações de hoteis, viagens e alugueres tudo numa página só) — páginas vulgarmente denominadas de **mashup** — estas terão de efetuar pesquisas entre vários outros serviços menores e independentes que não se têm de reger a qualquer tipo de norma para a sua interface para aplicações externas (APIs), pelo que precisam de saber, *a priori*, como contactar com uma determinada página em busca de uma informação.

mashup

Este tipo de coisas acontece por não termos uma rede semântica instituída sobre os conteúdos das nossas páginas *web*, mas antes uma **rede sintática**, isto é, uma rede que possui e estabelece relações apenas entre os elementos que constituem a sua estrutura como página *web*. Alguns destes elementos são o `href` que permite que uma página garanta o acesso a uma segunda a um utilizador (por vias de um *link*). Isto significa que as relações entre os demais conteúdos ficam por fazer para o utilizador.

rede sintática

Para que uma rede semântica possa então ser montada há que criar uma formalidade especial entre várias máquinas, para que estas possam comunicar entre si, de forma totalmente inequívoca. Esta formalidade deverá passar por uma linguagem para descrever os dados e as suas ligações, regras que permitam às máquinas a extração de informação dos dados (para classificar, pesquisar, entre outros...), tecnologias específicas e ferramentas eficientes para o fazer e a criação de ontologias para descrever os vários tipos de dados. Uma **ontologia** é uma forma de descrever algo que existe. No nosso caso isto acaba por ser algo fulcral uma vez que só desta forma é que podemos partilhar a forma como cada máquina poderá organizar os seus conhecimentos — qual a estrutura de cada objeto conhecido e de cada relação.

ontologia

No conhecimento humano as ontologias são bastante usadas, mas sem que o nosso pensamento o perceba. Por exemplo, quando lemos uma frase como "A Maria gosta do João" conseguimos perceber logo de imediato que estamos a falar de duas pessoas e a estabelecer uma relação de carinho de uma para outra. Isto acontece porque temos, em nós, já inserido um modelo (conceito) de uma pessoa e sobre que forma é que a representamos textualmente — sabemos que não é à toa que usamos uma letra maiúscula para escrever o nome de uma pessoa e conhecemos todo um conjunto de nomes possíveis de serem atribuídos — e percebemos que "gostar" é um verbo que mostra afeição, porque algures no tempo aprendemos isso e criámos um modelo do que significa "gostar". Só nesta frase podemos investigar o nosso conhecimento em muitos aspetos, analisando a sua semântica em muitos níveis (desde ao que cada partícula de conhecimento significa, até à sua compreensão num todo através do conhecimento prévio da ortografia e gramática do português).

Num mundo da *web* várias partículas semelhantes às presentes na frase anterior são divulgadas, embora as ontologias não o sejam — da mesma forma que humanos falam entre si (ninguém explica todas as palavras do que irá dizer a seguir, numa conversa). Isto acaba por ser uma desvantagem aos olhos da engenharia do conhecimento, uma vez que se escondem o significado dos dados o que dificulta (e por vezes impossibilita) a sua integração noutras aplicações.

Para combater este problema, ao longo dos tempos foram sido criadas formas novas de representar os dados, sem que a sua estrutura fosse perdida nos seus transportes. Fazem-se assim acompanhar os dados das suas legendas para os vários campos (propriedades) descritas. Uma das primeiras formas que surgiram para o fazer foram os **modelos tabulares**, isto é, uma forma que por si já era muito familiar à maioria dos utilizadores, o que possui a grande vantagem de ler facilmente legível por humanos. Na Figura 2.1 podemos ver um exemplo de aplicação de um modelo tabular para a descrição de restaurantes.

modelos tabulares

Como podemos ver na Figura 2.1 existe uma semântica associada a cada dado nela inserido. Por exemplo, o dado "Pizza Planet" sabemos que é o nome de um restaurante

que fica em Sunnyside, cuja especialidade de cozinha é Italiana, com um preço moderado e aberto de segunda a sexta. Isto acontece porque a primeira linha da tabela possui uma legenda para a forma como cada dado se encontra representado — é uma ontologia. [10]

figura 2.1

restaurante	morada	especialidade	preço	aberto
Los Pollos Hermanos	Albuquerque	Fast-Food	$	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
The Three Broomsticks	Hogsmeade	Bar	$	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
The Mos Eisly Cantina	Tatooine	Bar	$$	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
Pizza Planet	Sunnyside	Italiana	$$	Seg, Ter, Qua, Qui, Sex
Monk's Cafe	Nova Iorque	Saladas	$$$	Ter, Qua, Qui, Sex
MacLaren's Pub	Nova Iorque	Bar	$$	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
Krusty Burguers	Springfield	Bar	$	Seg, Ter, Qua, Qui, Sex, Sáb, Dom

Embora este modelo esteja suficientemente bem feito para que humanos o possam ler, a forma como os programas o poderão manipular traduzir-se-á por leitura constantes dos vários valores de forma linear, pelo que a sua manutenção é bastante limitada e rígida para ser usado numa aplicação de *software*.

Por questões de complexidade algorítmica, e como vimos na disciplina de Bases de Dados (a3s1), em 1969 Edgar Codd criou um modelo denominado de **modelo relacional**, baseado na lógica de predicados de primeira ordem. Este modelo tira proveito das vantagens de um modelo tabular, mas permite que sejam criadas relações entre as várias entidades que nela são criadas. Estas entidades, cada uma delas disposta sobre uma tabela com tuplos (conjuntos de dados organizados num registo de linha). Uma representação do conteúdo da Figura 2.1 sob um modelo de dados relacional poderá ser visto na Figura 2.2.

modelo relacional

⊙ **Edgar Codd**

figura 2.2

Restaurante

ID de restaurante	nome	morada	ID de especialidade	preço
1	Los Pollos Hermanos	Albuquerque	1	$
2	The Three Broomsticks	Hogsmeade	2	$
3	The Mos Eisly Cantina	Tatooine	2	$$
4	Pizza Planet	Sunnyside	3	$$
5	Monk's Cafe	Nova Iorque	4	$$$
6	MacLaren's Pub	Nova Iorque	2	$$
7	Krusty Burguers	Springfield	2	$

Especialidade

ID de especialidade	nome
1	Fast-Food
2	Bar
3	Italiana
4	Saladas

Horário

ID de restaurante	dia	abre às	fecha às
1	segunda	11	16
1	terça	11	16
1	quarta	11	16
…	…	…	…
1	domingo	10	22
…	…	…	…
5	terça	9	3
5	quarta	9	3
5	quinta	9	3
5	sexta	9	4
6	segunda	11	16

Num modelo relacional podemos verificar com mais facilidade a forma como cada elemento é descrito, uma vez que a necessidade de saber descrever um elemento também é maior. Isto acontece porque é clara a relação entre os conceitos de várias entidades. Como podemos ver na Figura 2.2 um restaurante possui uma relação com uma especialidade culinária (entre várias que existem) e faz parte de uma tabela de horários. Ainda assim, os Sistemas de Gestão de Bases de Dados (SGBDs) continuam sem ter um modelo daquilo que é um restaurante, uma especialidade ou um horário, contudo é possível questionar-lhes caraterísticas dos mesmos. [11]

Este tipo de modelo de representação de dados é bom quando sabemos a estrutura de uma base de dados e quando esta se mantém fixa durante o tempo. No caso dos dados reolhidos na *web* é impossível garantir que a forma como os dados são transmitidos é sempre igual, uma vez que não existe qualquer tipo de norma para o mesmo. Neste caso temos mesmo que assumir que a estrutura dos dados está sempre a mudar, assim como o conjunto de operações sobre os mesmos.

No entanto há que considerar que as bases de dados sob um modelo relacional também possuem alguma flexibilidade, pelo que se pretendermos adicionar, por exemplo, um bar a um restaurante (fora os restaurantes que são bares), podemos adicionar o bar a uma tabela exclusiva para bares e depois fazer a correspondência entre um restaurante e um bar numa nova tabela, intermediária da relação. Esta solução acaba por ser pouco intrusiva, dado que a inserção dos dados é feita da mesma forma que antes, mas a relação fica instituída na própria conceção da tabela intermediária. Isto, por si, também acaba por ser pouco eficiente uma vez que, por exemplo, se pretendermos saber os estabelecimentos existentes num determinado lugar, então teremos de pesquisar duas tabelas para o efeito — o que não é ideal numa base de dados relacional, pelo que forma dados redundantes.

Como solução para este problema o que se costuma fazer é uma **normalização** dos dados (ou da estrutura destes), o que significa que o modelo de dados já existente terá de ser alterado (solução intrusiva) — trará custos ao nível da sua programação. Por este mesmo motivo esta solução não pode ser transportada para a manipulação de dados preservados e provenientes da *web*. Preferindo assim a solução não intrusiva apresentada antes, em termos de aplicação na *web*, aqui os dados serão inseridos até que a sua gestão, manutenção e utilização sejam consideradas impraticáveis. Isto acontece porque com as bases de dados relacionais em ambientes como os da *web* os níveis de complexidade tornam-se bastante elevados, uma vez que não havendo um formalismo de estrutura nos dados provenientes desta, a aplicação de regras de relações para inserção de novos dados na base de dados torna-se uma tarefa mais dificíl, e a criação de novas tabelas de relação aumentam o peso de análise da base de dados. [12]

normalização

Uma solução considerada hipotética, que surgiu depois do modelo relacional, teria uma representação tal que fosse flexível o suficiente para acomodar tipos de dados em permanente mudança, com um nível de legibilidade consideravelmente bom. Neste novo modelo teríamos então uma representação com a forma `[Entidade, Propriedade, Valor]`, onde a `Entidade` seria a designação de uma determinada instância de uma entidade, `Propriedade` seria um campo a ser preenchido (como um nome, uma morada, um preço, ...) e `Valor` seria o valor em respeito a uma determinada `Propriedade`. Note-se que esta hipótese de representação de dados não é recomendável para ser aplicada por questões de otimização em pesquisa e inserção de dados, no entanto é bastante flexível e boa para contextos *web*, uma vez que permite que outro tipo de entidades sejam facilmente acomodadas, por muitas propriedades variáveis que estes possam ter. Criamos assim um modelo de dados denominado de **modelo chave-valor**.

modelo chave-valor

Esta implementação de um modelo chave-valor, como foi referido, não é, de todo, ótima, no entanto num contexto *web* não será mau uma vez que responde ao informalismo que existe no meio. A má eficiência e desempenho deste modelo de representação deve-se especialmente ao facto de se terem que fazer operações de junção (*join*) repetidamente para se chegar a um determinado resultado pretendido. Um exemplo de tradução da tabela de restaurantes da Figura 2.2 para um modelo de chave-valor poderá ser visualizado na Figura 2.3, onde temos um esquema `[Identificação de Entidade, Chave, Valor]`.

identificação de entidade	chave	valor
restaurante1	nome	Los Pollos Hermanos
restaurante1	morada	Albuquerque
restaurante1	especialidade	Fast-Food
restaurante1	preço	$
restaurante1	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
restaurante2	nome	The Three Broomsticks
restaurante2	morada	Hogsmeade
restaurante2	especialidade	Bar
restaurante2	preço	$
restaurante2	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
...	...	...
restaurante7	nome	Krusty Burguers
restaurante7	morada	Springfield
restaurante7	especialidade	Bar
restaurante7	preço	$
restaurante7	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom

figura 2.3

Seguindo o modelo demonstrado na Figura 2.3 cada dado é diretamente descrito através de uma propriedade que o define, sendo que as relações semânticas que antes seriam inferidas pelos nomes das tabelas e das colunas, são agora fornecidos como dados dentro da própria tabela. Na verdade, na Figura 2.3 não temos qualquer tipo de designação semelhante ao anterior nome da tabela, mas podemos fazê-lo facilmente, juntando, para cada registo de uma entidade, uma nova propriedade que designe o seu tipo, como podemos ver na Figura 2.4.

identificação de entidade	chave	valor
restaurante1	**tipo**	**restaurante**
restaurante1	nome	Los Pollos Hermanos
restaurante1	morada	Albuquerque
restaurante1	especialidade	Fast-Food
restaurante1	preço	$
restaurante1	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
restaurante2	**tipo**	**restaurante**
restaurante2	nome	The Three Broomsticks
restaurante2	morada	Hogsmeade
restaurante2	especialidade	Bar
restaurante2	preço	$
restaurante2	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
...	...	...
restaurante7	**tipo**	**restaurante**
restaurante7	nome	Krusty Burguers
restaurante7	morada	Springfield
restaurante7	especialidade	Bar
restaurante7	preço	$
restaurante7	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom

figura 2.4

Representação de dados em modelos chave-valor

Na secção anterior chegámos a um consenso de que os dados, pela sua instabilidade de estrutura e forma provenientes da *web*, estão melhor organizados numa base de dados que siga uma representação dos mesmos segundo um modelo de chave-valor. Com isto vimos entretanto uma representação de uma tabela que fomos construindo ao longo da secção, com a qual chegámos a uma representação onde constavam as descrições de várias entidades com a forma de `[Identificação de Entidade, Chave, Valor]`. Então, mas que forma é

esta de representação de um dado, através de um esquema `[Identificação de Entidade, Chave, Valor]`?

A tabela da Figura 2.3 e da Figura 2.4 possui uma forma de mostrar o seu conteúdo através de três campos `[Identificação de Entidade, Chave, Valor]`, pelo que vulgarmente se denomina de **triplo**. Mais rigorosamente, este triplo descreve-se como um tuplo de três elementos com o esquema `(sujeito, predicado, objeto)`, onde um `sujeito` representa uma entidade identificada por um elemento de identificação (ID), um `predicado` representa uma propriedade que pretende representar uma parte de uma entidade (à qual está anexada) e um `objeto` representa ou um valor literal para uma determinada propriedade (como um texto ou uma representação numérica) ou uma entidade que por sua vez pode ser sujeito noutro triplo. Basicamente o que isto pretende descrever é que o cruzamento de dados poderá ser feito então dentro de uma mesma tabela, sendo que o valor de um determinado campo (objeto) poderá querer apontar para um outro campo de um outro dado residente na mesma estrutura de dados. Veja-se a Figura 2.5.

triplo

identificação de entidade	chave	valor
restaurante1	tipo	restaurante
restaurante1	nome	Los Pollos Hermanos
restaurante1	morada	Albuquerque
restaurante1	especialidade	**especialidade1**
restaurante1	preço	$
restaurante1	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
restaurante2	tipo	restaurante
restaurante2	nome	The Three Broomsticks
restaurante2	morada	Hogsmeade
restaurante2	especialidade	**especialidade2**
restaurante2	preço	$
restaurante2	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
...	...	...
restaurante7	tipo	restaurante
restaurante7	nome	Krusty Burguers
restaurante7	morada	Springfield
restaurante7	especialidade	**especialidade2**
restaurante7	preço	$
restaurante7	aberto	Seg, Ter, Qua, Qui, Sex, Sáb, Dom
especialidade1	tipo	especialidade
especialidade1	nome	Fast-Food
especialidade2	tipo	especialidade
especialidade2	nome	Bar
especialidade3	tipo	especialidade
especialidade3	nome	Italiana
especialidade4	tipo	especialidade
especialidade4	nome	Saladas

figura 2.5

Como podemos agora verificar na Figura 2.5 cada restaurante que antes possuía uma designação da sua especialidade, passou agora a ter uma indicação da especificação de uma especialidade. Isto é possível uma vez que como objeto ao predicado `especialidade` de um restaurante, está a ser indicada uma entidade que, por si, há de possuir um valor literal (um texto com o nome da especialidade, neste caso de aplicação).

Preservar triplos de dados

Então, mas depois da descrição de triplos como é que os podemos guardar, gerir e pesquisar dentro deles? Para o fazer devem-se criar **triplestores**, isto é, armazenamentos de triplos que geralmente são efetuados sob a forma de grafos.

triplestores

Para o efeito, consideremos assim a seguinte implementação em Python para uma unidade de manipulação de *triplestores*. Existem várias implementações disponíveis *online*, mas preferimos mostrar esta implementação para que se tenha um primeiro contacto com a forma como podemos gerir e pesquisar entre uma conjugação de triplos de uma base de dados sob um modelo de chave-valor.

Esta implementação é baseada em dicionários e faz a indexação cruzada dos três termos de um triplo, por forma a permitir o acesso direto aos triplos através de qualquer um dos termos. Criamos assim uma classe denominada de `TripleGraph` que se inicializa pela criação de três dicionários (um `spo` para a organização de um triplo pela ordem `(sujeito, predicado, objeto)`, um `pos` para a organização de um triplo pela ordem `(predicado, objeto, sujeito)` e um `osp` para a organização de um triplo pela ordem `(objeto, sujeito, predicado)`. Note-se que, depois de usado, qualquer um destes índices consistirá de um dicionário de dicionários com conjuntos, isto é, com a forma `spo = {subject:{predicate:set([object])}}`. Veja-se, então, o Código 2.1 onde construímos a classe e o seu construtor.

código 2.1

```
class TripleGraph:
  def __init__(self):
    self._spo = {}
    self._pos = {}
    self._osp = {}
```

A nossa classe terá então sete métodos que permitirão a total manipulação destes triplos. Num primeiro método podemos inserir todos os termos de um triplo a um índice, caso estes ainda não estejam presentes (Código 2.2).

código 2.2

```
def _addToIndex(self, index, a, b, c):
  if a not in index:
    index[a] = {b : set([c])}
  else:
    if b not in index[a]:
      index[a][b] = set([c])
    else:
      index[a][b].add(c)
```

Esta função `addToIndex()` permite que uma segunda `add()` possa adicionar todas as permutações dos termos do triplo a todos os índices criados no Código 2.1 (Código 2.3).

código 2.3

```
def add(self, subject, predicate, object):
  self._addToIndex(self._spo, subject, predicate, object)
  self._addToIndex(self._pos, subject, predicate, object)
  self._addToIndex(self._osp, subject, predicate, object)
```

Da mesma forma que adicionamos, também podemos remover, daí termos funções análogas para a remoção, como podemos ver no Código 2.4.

código 2.4

```
def _removeFromIndex(self, a, b, c):
  try:
    bs = index[a]
    cset = bs[b]
    cset.remove(c)
    if len(cset) == 0:
      del(bs[b])
    if len(bs) == 0:
      del index[a]
  except KeyError:
    pass

def remove(self, index, subject, predicate, object):
  triples = list(self.triples(subject, predicate, object))
  for (deleteSubject, deletePredicate, deleteObject) in triples:
    self._removeFromIndex(self._spo, deleteSubject, deletePredicate, deleteObject)
    self._removeFromIndex(self._pos, deleteSubject, deletePredicate, deleteObject)
    self._removeFromIndex(self._osp, deleteSubject, deletePredicate, deleteObject)
```

filtragem

Para fazer a **filtragem** de triplos podemos usar uma função denominada de `triples()`, pelo que esta filtra os triplos existentes, dado um triplo padrão. Basicamente o que deverá fazer é, caso exista um sujeito, um predicado e um objeto como entregues à função, se o objeto estiver num determinado predicado, então este será entregue, caso contrário será

entregue uma referência para o próximo identificador de entidade, em busca do mesmo objeto (Código 2.5).

código 2.5

```
def triples(self, subject, predicate, object):
  try:
    if subject != None:
      if predicate != None:
        if object != None:
          if object in self._spo[subject][predicate]:
            yield (subject, predicate, object)
        else:
          for returnObject in self._spo[subject][predicate]:
            yield (subject, predicate, returnObject)
      . . .
```

Os outros dois métodos implementados nesta classe são o `load()` e `save()`, que permitem que os dados sejam carregados e guardados, respetivamente, sob a forma de um ficheiro CSV.

fusão

Uma das vantagens de usar triplos é que se permite a **fusão** entre dois ou mais grafos, desde que estes possuam, pelo menos, um elemento igual, isto é, desde que dois ou mais grafos possuam um sistema consistente de identificadores para os sujeitos e para os objetos.

pesquisa

Uma das tarefas mais importantes de serem feitas sobre os triplos é a **pesquisa**. Esta tarefa pode ser feita através da utilização do método `triples()`. Por exemplo, se possuirmos uma base de dados sobre filmes em que temos registos que estabelecem relações sobre um determinado filme, com a forma de `filmeA, nome, Nome do Filme` ou `filmeA, realizado por, realizadorA` e registos que estabelecem relações sobre um determinado realizador, com a forma de `realizadorA, nome, Nome do Realizador` ou `realizadorA, participa em, filmeA`, então podemos fazer uma pesquisa pela lista de atores dirigidos por um realizador procurando inicialmente pelo triplo `filmeA, realizado por, realizadorA` e depois pelo triplo `filmeA, onde participa, atorA`, sendo que no primeiro triplo pretendemos que varie o `filmeA` e no segundo varie o `atorA`, como podemos ver no Código 2.6.

código 2.6

```
for subject, predicate, object in _g.triples(None, "realizado por", realizadorID):
  print "No filme ", _g.valueObject(subject, "nome"), " orientou: "
  for subject2, predicate2, object2 in _g.triples(subject1, "onde participa", None):
    print "Ator ", _g.valueObject(object2, "nome")
```

Claro está que para efetuarmos a pesquisa conforme se verifica no Código 2.6 é necessário ter um conhecimento prévio de como é que as várias estuturas se apresentam numa base de dados. Mais, o processamento indicado no Código 2.6 foi feito segundo um módulo já existente para processar triplos, pelo que não estamos a usar qualquer norma que possa existir para navegar entre os registos de uma base de dados por triplos. Para resolver ambos estes problemas podemos usar linguagens de pesquisa que permitam a abstração dos processos de filtragem entre largos conjuntos de triplos.

inferências

A utilização de linguagens para pesquisa permite também que seja facilitada a capacidade de fazer **inferências** sobre a informação. Uma inferência é um processo de derivar nova informação através de outra que já existe, mas de forma automática. Estes processos poderão ser simples e determinísticos (por exemplo, se uma pedra pesa um quilograma, então infere-se que a mesma tenha um peso de mil gramas), baseados em regras (por exemplo, se uma determinada pessoa tem menos de 18 anos, então não poderá consumir bebidas alcoólicas), por classificações (por exemplo, se uma empresa se localiza numa cidade junto à costa, etnão infere-se que tal empresa é classificada como sendo do litoral), por julgamentos (por exemplo, se uma pessoa tem mais de 1.80 metros, então infere-se que esta é alta) ou por serviços *online* (por exemplo, sabendo o endereço de um restaurante, podemos usar um geocodificador para se conseguir as coordenadas num mapa).

As inferências por si são usadas quando precisamos de relacionar um triplo que termina com uma URI (sigla inglesa para *UniformResource Identifier*), com um segundo triplo que possui uma nova referência para um terceiro ou um valor literal para o elemento pretendido.

Normas das redes semânticas

Ao longo do nosso estudo de redes semânticas surgiram alguns problemas que ficaram entretanto sem resposta, um dos quais — o mais claro de todos — sendo a carência de **normas** para efetuar manipulações sobre os dados presentes nestas. Estas normas são importantes para que se possam especificar sobre que forma é que um predicado poderá existir, que tipos de dados se podem encontrar, porque é que uma entidade aparece representada de uma forma e não de outra, como é que outros poderão saber que representações foram usadas (partilha de ontologias) e qual a melhor forma de guardar e partilhar dados semânticos em termos visuais (vírgulas, tabulações, espaços, ...).

normas

Em suma, é importante que existam normas subjacentes a dados que são transmitidos de máquina em máquina, para que se possa saber de um eventual contexto de informação e de como é que duas máquinas poderão tratar uma mesma informação (ou informações diferentes) da mesma maneira, isto é, tendo a perceção correta sobre o tipo e o valor dos dados que são partilhados e geridos.

Sendo que nesta manipulação e gestão dos dados precisamos de uma forma de os identificar, uma forma de os representar e partilhar, e uma forma de os pesquisar, ao longo do tempo chegou-se a uma norma que criou o conceito de **URI** (*Uniform Resource Identifier*) para a identificação de uma entidade com grau absoluto de certeza, o conceito de **RDF** (sigla inglesa para *Resource Description Framework*) para a representação, descrição e partilha de dados com uma semântica adjacente, e o conceito de **SPARQL** para a pesquisa entre dados semânticos, respetivamente. Analisemos com mais cuidado, de seguida, cada uma das partes desta norma para redes semânticas.

URI

RDF

SPARQL

Como já sabemos, a organização estrutural de uma rede semântica baseia-se num grafo onde cada nó necessita de um identificador único, de forma a que este possa ser referenciado de uma forma consistente, através de todo o conjunto de triplos que descrevem as relações do grafo. Mais atrás, demos identificadores para alguns objetos da entidade Restaurante, com a forma de `restaurante1`, `restaurante3`, ... No entanto, com esta forma não temos como garantir que tal atribuição seja unívoca e absoluta. Para isso, criou-se um conceito no universo das redes semânticas em que todos estes tópicos deverão ser considerados como **recursos**, isto é, quer um objeto seja considerado material ou imaterial, é um recurso aos olhos das redes semânticas. A cada um destes recursos atribui-se assim um URI, uma *string* que univocamente atribui uma identificação a um objeto, qualquer que este seja.

recursos

A utilização de URI poderá confundir-se com a utilização dos URLs (sigla para *Uniform Resource Locator*), que em redes permite localizar um determinado recurso. Isto acontece porque os URLs são um subconjunto dos URI, daí a sua estrutura e forma serem equivalentes.

Tendo uma identificação para um recurso, cabe-nos agora saber descrevê-lo dentro de uma rede semântica, isto é, ter a capacidade de apontar as suas caraterísticas e de que forma é que um determinado objeto poderá, ou não, ter relações com outros nós presentes numa rede. Surge assim o RDF, sendo um modelo padrão para expressar dados semânticos, usando declarações na forma de triplos. Aqui existem três pontos importantes a reter, em relação à lógica de apresentação de um triplo `Sujeito, Predicado, Objeto`: um **sujeito** poderá ser um recurso ou ser desconhecido; um **predicado** terá obrigatoriamente que ser um recurso; e um **objeto** poderá ser um recurso ou um literal.

sujeito

predicado

objeto

A identificação de um recurso pertencente a um RDF poderá ser simplificada da seguinte forma: consideremos que temos um URI para um recurso `http://tables.com/#rest1`. Este URI poderá ser dividido em duas partes — uma primeira, onde está descrito o URI base (`http://tables.com/#`) e uma segunda, onde está descrito o RDF (`type`). Para não termos que estar sempre a escrever o URI para a RDF, podemos especificar o seu **prefixo**, dizendo que `nome_de_rdf = http://tables.com/#`. Agora, cada vez que precisarmos de especificar um RDF pertencente ao `nome_de_rdf`, podemos simplesmente escrever `rdf:type`.

prefixo

Atrás indicámos que um sujeito poderá ser **desconhecido** — denomina-se mais corretamente por sujeito **anónimo** (ou vazio). Isto pode acontecer uma vez que, por alguma razão, um valor para o sujeito ainda está a ser produzido ou simplesmente não existe. Nestes casos, a identificação em termos de linguagem de um nó anónimo é feita indicando um prefixo com o carater '`_`'. Por exemplo, para um RDF `type` anónimo, teríamos de indicar `_:type`. Este RDF, geralmente, tem o seu nome automático e internamente gerado, para ser válido apenas num grafo.

desconhecido
anónimo

Os **literais** que temos vindo a falar poderão existir nos objetos, para terminar o encadeamento de novos nós no grafo. Podendo este ser um valor numérico ou um texto, em relação ao texto a norma RDF permite a especificação de um idioma (português, inglês, ...) e/ou um tipo (`integer`, `boolean`, ...) ao valor. No caso dos tipos de dados, geralmente usam-se os tipos já especificados no espaço de nomes do XML Schema, e no caso dos idiomas usam-se as normas ISO 639 (idenficação de idiomas com apenas duas letras, como `pt`, `en`, `fr`, ...).

literais

ISO 639

O RDF entretanto, poderá ser representado sob várias formas, entre as quais as mais utilizadas, a N-Triples, a N3, a RDF/XML e a RDFa. Para podermos contextualizar melhor as noções patentes entre as várias representações possíveis consideremos que possuímos uma base de dados para um grafo que possui pessoas como entidade dos nós e cujas relações são, de facto, relações entre pessoas. Neste cenário, temos então informação acerca de um indivíduo qualquer Marty, isto é, uma série de informação que ajudará a identificar um indivíduo de forma única (informação de contexto, para conseguir distinguir um Marty de muitos outros com a mesma designação de nome). Estes procedimentos, por si, também ajudarão a obter um URI para um indivíduo, que poderá ser usado a partir daí.

Este exemplo que estamos a dar, na verdade, é algo que já existe, para descrever entidades e relações entre pessoas. Referimo-nos assim ao **FOAF** (acrónimo inglês para *Friend of a Friend*), sendo que este é um exemplo de RDF muito presente na *web* e que serve, explicitamente, para descrever relações entre entidades que descrevam pessoas. No entanto, com o passar dos tempos, o FOAF tem vindo a ser estendido para descrever um número de objetos e outras entidades também comuns.

FOAF

Consideremos assim a nossa primeira representação RDF que dá pelo nome de **N-Triples**. O formato N-Triples é uma forma simples de representar um RDF, aproximando-se inclusive ao formato CSV. Assim, por cada linha, temos uma única declaração (um triplo) contendo sempre um sujeito, um predicado e um objeto, terminando sempre por um ponto final, como podemos ver no Código 2.7.

N-Triples

```
<URI_de_sujeito> <URI_de_predicado> [<URI_de_objeto>|"literal_de_objeto"] .
```

código 2.7

Como podemos ver no Código 2.7, os sujeitos, predicados e objetos são sempre expressos com URIs absolutos entre parênteses angulares '`<`' e '`>`'. No entanto, os objetos também são os únicos que poderão não possuir parênteses angulares, somente quando se pretende que estes triplos terminem com um literal. Mais, note-se que caso estejamos a trabalhar com nós anónimos, então estes deverão possuir a desginação do RDF anónimo, tal como referimos anteriormente, com `_:type`, sendo `type` uma palavra alfanumérica começada por uma letra.

Já o caso dos objetos, tal como abrigado pelas normas da RDF, estes poderão receber alguns atributos, esses, que sendo um idioma então são representados pela concatenação ao literal de `@lang` (onde `lang` se refere a um idioma cuja representação é ditada pela norma ISO 639) e, por outro lado, sendo um tipo, então são representados pela concatenação ao literal de `^^xsd:type` (sendo `type` o tipo de dados que se pretende aplicar, como `integer`, `boolean`, entre outros apresentados no espaço de nomes do XML Schema).

No Código 2.8 podemos ver um exemplo de aplicação da norma N-Triples.

```
<http://plp.com/marty.rdf> <http://xmlns.com/foaf/0.1/homepage> <http://plp.com> .
<http://plp.com/marty.rdf> <http://xmlns.com/foaf/0.1/nick> "Mar" .
<http://plp.com/marty.rdf> <http://xmlns.com/foaf/0.1/name> "Marty McFly" .
<http://plp.com/marty.rdf> <http://xmlns.com/foaf/0.1/mbox> <mailto:m@outatime.com> .
```

código 2.8

```
<http://plp.com/marty.rdf> <http://w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
<http://plp.com/marty.rdf> <http://xmlns.com/foaf/0.1/knows> _:doc_brown .
<http://plp.com/marty.rdf> <http://xmlns.com/foaf/0.1/knows>
    <http://plp.com/biff_tannen.rdf> .
```

Como podemos verificar no Código 2.8 temos um conjunto de triplos que nos permitem descrever o Marty. Vejamos assim, linha-a-linha, como é que o descrevemos.

Inicialmente temos que o sujeito é o URI do Marty, com um predicado de *homepage* a indicar `http://plp.com` como objeto (URI que neste caso é um URL). De seguida temos que o Marty possui um *nickname* Mar e que o seu nome é Marty McFly. Já na quarta linha descrevemos que a sua caixa de correio é um URI (em forma de URL) para o endereço de correio eletrónico `m@outatime.com`. Na quinta linha, com a ajuda do tipo especificado na norma de sintaxe do RDF como predicado, temos que o Marty possui um tipo de `Person` (pessoa), conhecendo uma pessoa com identificação `doc_brown` (desconhecido no nosso domínio de informação) e uma pessoa com identificação `biff_tannen` (dentro do nosso domínio de informação).

Note-se que, nas últimas duas linhas do Código 2.8 indicamos que Marty possui uma relação de conhecimento de uma segunda pessoa porque o predicado `knows` (tal como os outros especificados à exceção da quinta linha) pertence à norma FOAF[6], que permite o estabelecimento e manipulação de normas ao nível de relações interpessoais e de descrição de pessoas por si só. Por outro lado, na quinta linha usamos um predicado da norma de sintaxe do RDF para indicar o tipo do sujeito, ao qual atribuímos o valor do objeto FOAF que descreve um tipo de pessoa (`Person`).

N3, Notation3

Um outro tipo de representação RDF, muito semelhante e frequentemente confundido com o N-Triples, é o **N3**. Proveniente de **Notation3**, o N3 permite que a informação que é digitada na linguagem N-Triples seja condensada, evitando repetições de dados excusados em N-Triples. Por exemplo, uma repetição clara é o prefixo do RDF que é repetido necessariamente em N-Triples — este ponto é removido no N3, pelo que se usa um prefixo comum, conforme a norma RDF suporta, havendo um reaproveitamento dos nós que participam em múltiplos triplos. Para criarmos um prefixo precisamos somente escrever no início uma especificação para o mesmo através de uma linha `@prefix nome_de_prefixo: <URI> .` (não esquecer do ponto final). Dando um nome de prefixo, ao invés de escrevermos um URI da seguinte forma `http://plp.com/marty.rdf` podemos antes escrever `plp:marty.rdf`, onde `plp` é o nosso prefixo.

Mas o reaproveitamento de dados não fica por aqui. Antes pelo contrário, o próprio sujeito e predicado poderão ser ambos reaproveitados. Um sujeito é reaproveitado quando novos triplos são representados através da adição de duplos separados por um ponto e vírgula `';'`. Já o predicado pode ser reaproveitado quando se representam triplos adicionando apenas objetos, separados por uma vírgula `','`.

A simplificação, de longe, fica por aqui. Na verdade, também nos é permitido fazer abreviações para entidades anónimas (como previsto pela norma RDF), colocando a secção de manipulação de entidades anónimas entre parênteses retos `'['` e `']'`. Por fim, as abreviações também se aplicam quando ao invés de usar certos predicados mais especiais como a definição de tipo (como vimos no Código 2.8) podemos escrever apenas `'a'`, quando ao invés de escrevermos um predicado de igualdade (`sameAs`) segundo a norma RDF podemos escrever apenas `'='`, quando ao invés de escrevermos um predicado de implicação de A para B (`implies`) segundo a norma RDF podemos escrever apenas `'=>'`, ou quando ao invés de escrevermos um predicado de implicação de B para A (`implies`) segundo a norma RDF podemos escrever apenas `'<='`.

No Código 2.9 podemos ver o mesmo exemplo que o visto no Código 2.8, mas agora na linguagem N3.

código 2.9

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix plp: <http://plp.com/> .
```

6 Podemos visitar a documentação FOAF para consultar outros predicados possíveis de serem aplicados, em http://xmlns.com/foaf/spec/.

```
plp:marty.rdf a foaf:Person ;
              foaf:homepage <http://plp.com/> ;
              foaf:knows plp:biff_tannen ,
                         [ a foaf:Person ;
                           foaf:name "Doc Brown"] ;
              foaf:mbox <mailto:m@outatime.com> ;
              foaf:name "Marty McFly" ;
              foaf:nick "Mar" .
```

A leitura de uma descrição em N3 deverá ser mais simples que a leitura de uma descrição em N-Triples. Para já, note-se que cada vez que uma linha termina com um ponto e vírgula (significando que de seguida se vai designar um duplo), o duplo parte na linha seguinte alinhado com o predicado da linha anterior, e quando uma linha termina com uma vírgula (significando que de seguida se vai designar um único termo), o termo objeto seguinte parte na linha próxima alinhado com o objeto da linha anterior. Isto é uma mera convenção para facilitar a leitura da descrição, pelo que não é necessário segui-la para que a representação se encontre correta.

O Código 2.9 começa por especificar os vários prefixos que a descrição irá necessitar, de forma semelhante a uma inclusão numa linguagem de programação imperativa. De seguida a descrição de Marty (pertencente ao prefixo `plp`) inicia com a indicação do seu tipo (note-se que o `a` é uma abreviação do tipo especificado no *schema* RDF, daí a inclusão do prefixo `rdf`) sendo uma pessoa como especificado no FOAF. Dado que a linha termina com ponto e vírgula, então significa que ainda não terminámos de designar o Marty como pessoa, pelo que de seguida temos que a sua *homepage* é um determinado URI, conhece uma pessoa identificada no mesmo RDF por `biff_tannen.rdf` e uma pessoa num RDF indeterminado com nome Doc Brown. Mais, sabemos também que Marty possui uma caixa de correio eletrónico com a designação `m@outatime.com`, tem o nome Marty McFly e um *nickname* Mar.

RDF/XML

Uma terceira forma de representar RDF é através do método **RDF/XML** (lê-se RDF sobre XML). Esta representação foi originalmente usada pelo consórcio W3C na recomendação do RDF, motivo pelo qual ainda é uma das mais usadas atualmente (novembro de 2017). No entanto, para nós que acabamos de verificar como é que a representação em N3 funciona, esta acaba por ser uma representação severamente criticada devido à dificuldade de legibilidade que apresenta para uma pessoa. O que acontece é que esta linguagem é feita de pequenas descrições, traçando cada uma delas um caminho através do grafo RDF[7].

Aqui, então, cada caminho inicia-se sempre com o elemento `<rdf:Description>`, sendo que a referência de um URI é indicada num atributo `rdf:about`. No caso de nós anónimos (isto é, sem URI), então o seu identificador (ID) é indicado no atributo `rdf:nodeID`, sendo que os predicados são especificados como elementos-filho do elemento `<rdf:Description>`.

A condensação que se ganhou com o N3 perante o N-Triples, aqui, mantém-se em alguns aspetos. Por exemplo, o predicado `type`, da norma de sintaxe do RDF, por exemplo, pode ser escrito de duas formas: uma primeira indicando em cascata os elementos `<rdf:Description rdf:about="URI_pretendido">`, `<rdf:type>` e `<foaf:Person>`; e uma segunda indicando somente o elemento `<foaf:Person rdf:about="URI_pretendido">`.

No Código 2.10 podemos ver um exemplo do mesmo RDF que no Código 2.9, mas agora com uma representação de RDF sobre XML.

código 2.10

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <foaf:Person rdf:about="http://plp.com/marty.rdf">
    <foaf:name>Marty McFly</foaf:name>
    <foaf:homepage rdf:resource="http://plp.com/"/>
    <foaf:nick>Mar</foaf:nick>
    <foaf:mbox rdf:resource="mailto:m@outatime.com"/>
    <foaf:knows>
      <foaf:Person>
        <foaf:name>Doc Brown</foaf:name>
      </foaf:Person>
      <foaf:Person rdf:about="http://plp.com/biff-tannen.rdf">
        <foaf:name>Biff Tannen</foaf:name>
```

[7] Quando existe mais que um caminho através do grafo RDF, então o elemento raiz deverá iniciar obrigatoriamente com `<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>`.

```
      </foaf:Person>
    </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

Como podemos ver no Código 2.10, então, começamos por ter uma raiz onde especificamos os vários espaços de nomes que vamos usar, seguido da entidade que realmente pretendemos descrever — neste caso uma pessoa, segundo a norma FOAF. Esta pessoa, possui então um nome Marty McFly, com uma *homepage* `http://plp.com` e com um *nickname* Mar. Possui também um endereço de correio eletrónico `m@outatime.com` e conhece duas pessoas, descritas ambas por FOAF: uma primeira, de nome Doc Brown, cuja denominação é indeterminada (nó anónimo); e uma segunda, de seu nome Biff Tannen, descrita segundo FOAF e pertencente ao mesmo domínio de Marty McFly.

RDFa

Por último, existe um método de representação de RDF denominado de **RDFa**. RDFa, nome proveniente de RDF Attributes, considera-se frequentemente como uma forma de anotar páginas *web* XHTML com dados RDF, sendo que o pretendido é publicar o conteúdo apenas uma vez, misturando conteúdos para as pessoas e conteúdos para as máquinas. No fundo, RDFa usa um conjunto de atributos que são adicionados aos marcadores XHTML, por forma a especificar semântica por detrás da informação que é apresentada.

CURIEs

Uma grande diferença visível com RDFa é que, para a identificação de sujeitos, predicados e objetos não usamos URIs, mas antes **CURIEs** (abreviatura para *Compact URIs*), de forma a reduzir o número de marcadores a usar. Por exemplo, se definirmos o espaço de nomes `xmlns:music="http://sounds.com/music"` podemos depois usar `music:country` ou ainda `music:pop/90s`. Mais, os CURIEs também permitem a utilização de partes locais que começam com números, sendo que `xmlns:amazonisbn="http://amazon.com/exec/item/ASIN"` pode ser usado como `amazonisbn:0345391802`.

Para escrever RDFa e com esta linguagem descrever uma entidade em RDF é importante conhecer que atributos referenciar para os sujeitos, para os predicados e para os objetos. Para um sujeito é importante usar o atributo `about`, pelo que este usa um URI, sendo que o URI base da página é o URI raiz para todas as declarações. Para um predicado podemos usar o atributo `rel` para usar um CURIE que expresse relações entre dois recursos, o atributo `property` para usar um CURIE que expresse relações entre recursos e literais e o atributo `rev` para usar um CURIE que expresse relações inversas em dois recursos. Já para um objeto, o atributo `content` permite que se use uma *string* (representando um literal), o atributo `href` permite que se use um URI para expressar um objeto (clicável na mesma linha), o `src` permite que se use um URI para expressar um objeto (embutido na mesma linha) e o `resource` permite que se use um URI para expressar um objeto quando este não está visível na página.

Em termos das abreviações que eram possíveis fazer em N3 e que, de alguma forma, também existem com um aspeto semelhante em RDF/XML, em RDFa também é possível usar o atributo `datatype` para especificar um tipo de dados de um literal (aplicável a um objeto com literal) e o atributo `typeof` para especificar o tipo de um sujeito (aplicação de `rdf:type`). No Código 2.11 podemos ver a mesma descrição da pessoa Marty McFly, agora em RDFa.

código 2.11

```
<div xmlns:foaf="http://xmlns.com/foaf/0.1/"
     about="http://plp.com/marty.rdf" typeof="foaf:Person">
  Nome: <span property="foaf:name">Marty McFly</span>
  <br/>
  Nickname: <span property="foaf:nick">Mar</span>
  <br/>
  Homepage: <a rel="foaf:homepage" href="http://plp.com/">People.com</a>
  <br/>
  Amigos:
  <br/>
  <ul rel="foaf:knows">
    <li about="http://plp.com/biff_tannen.rdf" typeof="foaf:Person"
        property="foaf:name">
      Biff Tannen
    </li>
    <li typeof="foaf:Person">
      <span property="foaf:name">Doc Brown</span>
    </li>
```

```
  </ul>
</div>
```

Estas são as várias formas de representação do RDF passíveis de serem aplicadas. Mas como vimos no início desta secção, há uma terceira e última tarefa que é importante que possa ser feita e que fornece todo uma lógica de utilização aos grafos de recursos — as **pesquisas**. Para fazermos pesquisas sobre RDFs, como também já foi referido anteriormente, usamos **SPARQL** (lê-se *sparkle* e possui um acrónimo recursivo de SPARQL Protocol and RDF Query Language), que é uma linguagem RDF que nos permite efetuar consultas nos grafos, da mesma forma que o SQL permite que sejam executadas consultas a uma base de dados comum. Na verdade, o próprio SPARQL possui uma sintaxe bastante similar com a SQL. [13]

pesquisas

SPARQL

Existem quatro formas de iniciar uma pesquisa com SPARQL. Uma primeira forma é usando o método `SELECT` — esta forma permite que seja feita a junção de variáveis e o retorno destas — e as outras são a `ASK` — esta forma permite que seja testada uma cláusula para aferir se um determinado padrão de consulta possui uma solução ou não —, a `DESCRIBE` — esta forma permite que seja devolvido um novo RDF, com dados sobre os recursos (os resultados provenientes serão todos URIs) — e a `CONSTRUCT` — esta forma permite que seja devolvido um único grafo RDF especificado por um único modelo de grafo. No caso do `CONSTRUCT`, o resultado será um grafo RDF formado pegando em cada solução de consulta (na sequência de soluções), substituir pelas variáveis no modelo de grafo e combinar os triplos num único grafo RDF através de uma operação de união de conjuntos.

De todos os métodos SPARQL, o mais usado é, de facto, o método `SELECT`, razão pela qual iremos focar-nos mais no seu detalhe e funcionamento. No Código 2.12 podemos começar por ver a estrutura-base de uma consulta com `SELECT`.

código 2.12

```
BASE <URI>

PREFIX nome_do_prefixo: <URI>

SELECT nome_de_coluna1, nome_de_coluna2, …, nome_de_colunaN
FROM grafo_a_pesquisar
WHERE {
  # regras de padrão de pesquisa
}
ORDER BY modificador
```

Para nos facilitar a compreensão da aplicação de SPARQL nos nossos grafos RDF, consideremos o grafo RDF, representado em N3, do Código 2.13.

código 2.13

```
@prefix fb: <http://rdf.freebase.com/ns/>

fb:en.hollywood_homicide fb:film.film.directed_by fb:en.ron.shelton ;
                         fb:film.film.starring fb:en.harrison.ford ,
                                               fb:en.kurupt ,
                                               fb:en.robert.wagner ;
                         fb:film.film.initial_realease_date “2003” .
fb:en.back_to_the_future fb:film.film.directed_by fb:en.robert.zemeckis ;
                         fb:film.film.starring fb:en.michael.j.fox ,
                                               fb:en.christopher.lloyd ,
                                               fb:en.lea.thompson ;
                         fb:film.film.initial_realease_date “1985” .
fb:en.gran_torino fb:film.film.directed_by fb:en.clint.eastwood ;
                  fb:film.film.starring fb:en.clint.eastwood ,
                                        fb:en.bee.vang ,
                                        fb:en.christopher.carley ;
                  fb:film.film.initial_realease_date “2008” .
fb:en.coco fb:film.film.directed_by fb:en.lee.unkrich ;
           fb:film.film.starring fb:es.anthony.gonzalez ,
                                 fb:es.gael.garcia.bernal ,
                                 fb:en.benjamin.bratt ;
           fb:film.film.initial_realease_date “2017” .
fb:en.toy_story_3 fb:film.film.directed_by fb:en.lee.unkrich ;
                  fb:film.film.starring fb:en.tom.hanks ,
                                        fb:en.tim.allen ,
                                        fb:en.joan.cusack .
fb:en.lee.unkrich foaf:name “Lee Unkrich” .
```

Em SPARQL, as variáveis são designadas com um ponto de interrogação no primeiro carater, pelo que se queremos selecionar uma coluna denominada de `coluna`, então teremos

de escrever `SELECT ?coluna`. Nesta ordem de raciocínio, para determinarmos que realizador é que aparece no próprio filme, então temos de fazer a consulta do Código 2.14.

código 2.14

```
PREFIX fb: <http://rdf.freebase.com/ns/>

SELECT ?person ?movie
WHERE {
  ?movie fb:film.film.directed_by ?person .
  ?movie fb:film.film.starring ?person .
}
```

Basicamente, no Código 2.14 estamos a procurar correspondências entre pessoas e filmes onde um determinado filme possua uma determinada pessoa como entidade responsável pela relação `directed_by` e o mesmo filme possua a mesma pessoa como entidade responsável pela relação `starring`. Por outras palavras, a consulta poderá ser vista como se se pretendesse obter todos os registos onde um determinado filme aparece como sujeito de um triplo com predicado `directed_by` e objeto uma pessoa e onde o mesmo sujeito e objeto aparece num triplo com predicado `starring`. O resultado da consulta acima poderá ser visto na Figura 2.6.

figura 2.6

movie	person
fb:en.gran_torino	fb:en.clint.eastwood

Em SPARQL também podemos unir pesquisas, inclusive com partículas consideradas como **opcionais**, isto é, caso existam, então são consideradas. No Código 2.15 podemos ver um exemplo de consulta que usa um `OPTIONAL` para referir, precisamente, uma partícula de pesquisa que é considerada opcional, sendo que a consulta completa pretende capturar todos os filmes realizados por Lee Unkrich, se possível, com data de lançamento.

opcionais

código 2.15

```
PREFIX fb: <http://rdf.freebase.com/ns/>

SELECT ?movie ?release_date
WHERE {
  ?movie fb:film.film.directed_by fb:en.lee.unkrich .
  OPTIONAL {
    ?movie fb:film.film.initial_release_date ?release_date .
  }
}
```

No Código 2.15 podemos então verificar que se tenta capturar uma tabela final com duas colunas (uma com `movie` e outra com `release_date`), onde os registos que por lá apareçam sejam filmes realizados por Lee Unkrich e, se possível, com a indicação da data de lançamento, como podemos ver na Figura 2.7.

figura 2.7

movie	release_date
fb:en.coco	2017
fb:en.toy_story_3	

No entanto, mesmo com um parâmetro opcional, o SPARQL permite-nos que filtremos o resultado das nossas pesquisas através da cláusula `FILTER`. Por exemplo, pegando novamente no exemplo de consulta do Código 2.15, podemos agora aplicar-lhe um filtro no Código 2.16 para retornar apenas os registos que não possuem data de lançamento.

código 2.16

```
PREFIX fb: <http://rdf.freebase.com/ns/>

SELECT ?movie ?release_date
WHERE {
  ?movie fb:film.film.directed_by fb:en.lee.unkrich .
  OPTIONAL {
    ?movie fb:film.film.initial_release_date ?release_date .
  }
  FILTER (!bound(?release_date))
}
```

A aplicação do filtro `!bound(?release_date)` faz com que apenas queiramos ver todos os registos cujo valor de `release_date` seja infinito ou `NaN` (*not a number*), como é o caso do vazio. Com esta consulta obtemos assim o resultado visível na Figura 2.8.

movie	release_date
fb:en.toy_story_3	

Este filtro poderá ser aplicado com um largo conjunto de funções que poderão ser vistas na própria documentação do SPARQL[8]. Por exemplo, uma segunda aplicação possível são as expressões regulares, através do uso do filtro `regex()`. A função `regex()` leva como argumentos a variável para teste de expressão, a expressão regular em si, e uma regra para aplicação da expressão regular, relacionado com o facto de ser *case-sensitive* ou não (argumento opcional).

O uso de chavetas '`{`' e '`}`' na linguagem SPARQL acontece porque cada um dos blocos por elas formados diz respeito a um único **grupo-padrão**. Um grupo-padrão é uma zona onde se aplicam as conjunções de consultas, de forma restrita, pelo que o SPARQL, quando olha para um grupo-padrão vê-lo como uma tabela e não como um conjunto de triplos para seleção. Nesta linguagem, dada esta construção, é-nos permitido usar vários grupos-padrão numa só pesquisa. Por exemplo, se pretendermos uma conjunção dos nomes de realizadores e de atores (caso possuíssemos tal informação no nosso grafo RDF), então teríamos um código algo semelhante ao Código 2.16 (Código 2.17).

grupo-padrão

código 2.17

```
PREFIX fb: <http://rdf.freebase.com/ns/>

SELECT ?name
WHERE {
  {
    ?movie fb:film.film.directed_by ?person .
    ?person fb:type.object.name ?name
    FILTER regex(?name, "^…")
  }
  {
    ?movie fb:film.film.starring ?actor .
    ?actor fb:type.object.name ?name
    FILTER regex(?name, "^b")
  }
}
```

Da mesma forma, podemos usar o `UNION` para especificarmos a união de grupos-padrão, sendo que desta forma o `WHERE` não se aplicará aos valores fundidos do triplo resultante de cada grupo-padrão, mas antes do conjunto-união dos dois. Esta versão poderá ser vista no Código 2.18.

código 2.18

```
PREFIX fb: <http://rdf.freebase.com/ns/>

SELECT ?name
WHERE {
  {
    ?movie fb:film.film.directed_by ?person .
    ?person fb:type.object.name ?name
    FILTER regex(?name, "^…")
  }
  UNION
  {
    ?movie fb:film.film.starring ?actor .
    ?actor fb:type.object.name ?name
    FILTER regex(?name, "^b")
  }
}
```

O uso do `CONSTRUCT` no SPARQL permite que se construa um novo grafo RDF a partir de uma determinada consulta. Por exemplo, poderíamos criar um novo grafo, relativo a uma consulta semelhante à efetuada no Código 2.17, onde apenas quiséssemos triplos onde se relacionasse uma pessoa e o seu primeiro ano de trabalho (Código 2.18).

8 Acessível em http://www.w3.org/TR/sparql11-query

código 2.19

```
PREFIX fb: <http://rdf.freebase.com/ns/>

CONSTRUCT {
  ?person <http://employment.history/was_employed_in> ?year
}
WHERE {
  {
    ?movie fb:film.film.directed_by ?person .
    ?movie fb:film.film.initial_release_date ?year
  }
  UNION
  {
    ?movie fb:film.film.starring ?person .
    ?movie fb:film.film.initial_release_date ?year
  }
}
```

Já o `ASK`, como já referimos também, permite que seja verificada a validade de uma determinada declaração, obtendo sempre uma resposta lógica de 'verdadeiro' ou 'falso'. Por exemplo, podemos perguntar se existe um filme onde "Tim Allen" e "Tom Hanks" contracenam, como verificamos no Código 2.19.

código 2.20

```
PREFIX fb: <http://rdf.freebase.com/ns/>

ASK {
  ?movie fb:film.film.starring fb:en.tom.hanks .
  ?movie fb:film.film.starring fb:en.tim.allen .
}
```

O `DESCRIBE`, por fim, permite que se crie um grafo de recursos RDF, sendo uma descrição de um recurso do qual se conhece pouco ou nada. Os resultados desta consulta serão assim muito descritivos e cheios de indicações para URIs, o que se torna pouco legível sem ser para uma máquina.

Claro está que, como em SQL, aqui também nos é permitido haver atualizações sobre os dados já existentes, entre as quais operações de inserção e de eliminação. Para efetuar uma **inserção** apenas temos de usar as cláusulas `INSERT DATA` se querermos inserir algo de forma direta, isto é, inserir um triplo completo. Por exemplo, no Código 2.20 estamos a inserir um nome para uma referência de um ator.

inserção

código 2.21

```
PREFIX fb: <http://rdf.freebase.com/ns/>

INSERT DATA {
  fb:en.tim.allen foaf:name "Tim Allen" .
}
```

No entanto, para inserir os dados não precisamos de o fazer um a um, pelo que basta que encontremos um padrão, para que se possam inserir vários triplos regidos por este, de uma só vez. Por exemplo, podemos inserir um tipo de Ator a todos os triplos que possuem uma relação `starring`, como podemos ver no Código 2.21.

código 2.22

```
PREFIX fb: <http://rdf.freebase.com/ns/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INSERT {
  ?actor rdf:type fb:Actor .
}
WHERE {
  ?movie fb:film.film.starring ?actor .
}
```

Da mesma forma que podemos inserir dados, também os podemos **eliminar**, sendo que para isso usamos a cláusula `DELETE DATA` para apagar um (ou mais) triplo(s) em específico (analogamente ao visto no Código 2.20) ou a cláusula `DELETE` para apagar vários triplos sob um padrão de teste (analogamente ao visto no Código 2.21).

eliminar

E assim terminam os apontamentos de Engenharia de Dados e Conhecimento (a4s1).

33 ENGENHARIA DE DADOS E CONHECIMENTO

1. Estrutura de Dados e sua conversão em informação

2. Engenharia de Conhecimento

As referências abaixo correspondem às várias citações (quer diretas, indiretas ou de citação) presentes ao longo destes apontamentos. Tais referências encontram-se dispostas segundo a norma IEEE (as páginas Web estão dispostas de forma análoga à de referências para livros segundo a mesma norma).

[1] H. T. Zagalo, "Apresentação," 2017.

[2] H. T. Zagalo, "Introdução aos Dados Semi-Estruturados," 2017.

[3] H. T. Zagalo, "Introdução ao XML," 2017.

[4] W3C Consortium (2009, December 8th). *Namespaces in XML 1.0* [Web Article]. Available: https://www.w3.org/TR/xml-names/

[5] W3C Consortium (2017, June 8th). *XSL Transformations (XSLT) Version 3.0* [Web Article]. Available: https://www.w3.org/TR/xslt/

[6] W3C Consortium (1999, November 16th, 2017). *XML Path Language (XPath)* [Web Article]. Available: https://www.w3.org/TR/xpath/

[7] XML.com (2001, October 31st). *Introduction to Native XML Databases* [Web Article]. Available: https://www.xml.com/pub/a/2001/10/31/nativexmldb.html

[8] W3C Consortium (2017, March 21st). *XQuery 3.1: An XML Query Language* [Web Article]. Available: https://www.w3.org/TR/xquery-31/

[9] H. T. Zagalo, “Introdução à Web Semântica," 2017.

[10] H. T. Zagalo, “Semântica dos Dados," 2017.

[11] H. T. Zagalo, “Expressão da Semântica," 2017.

[12] H. T. Zagalo, “Standards da Web Semântica," 2017.

[13] H. T. Zagalo, “A linguagem SPARQL," 2017.

Apontamentos de Engenharia de Dados e Conhecimento
1ª edição - novembro de 2017

Autor: Rui Lopes
Agradecimentos: professor Hélder Zagalo

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.

apontamentos