屹立在关系数据库上的语义网

都听说语义网是Web的未来，但你还是不知道它怎么能让你的应用变得更好。它能加快开发速度吗？能应付复杂的数据结构吗？面向对象的原则还要吗？

■ 文 / Patrick van Bergen

语义网是W3C在Tim Berners Lee引领下建立的一个框架，它的基本前提是认为数据应该以一种全局的方式实现自描述。也就是说，数据表达的不仅是数字、日期、文字等，还应该明确地表达各字段与其对象的关系类型。采用语义网的统一数据结构，在不同服务器间交换数据将更为便利，最重要的是，数据可被全世界的搜索引擎所访问。

很好，就只有这些吗？提供一套数据到RDF的导入/导出工具不就行了吗？有什么内在理由非得把整个数据结构都建立在语义网之上？

在本文中，我将说明我们在Procurios是如何实现语义网的概念的，实现背后的理论依据何在，以及语义网相对传统的关系数据库有何优势。首先我要说明我们如何在一个关系数据库（MySQL）里实现语义网，并在其上叠加一层面向对象层，甚至为它建立了一个数据版本控制系统。

三元组（Triple）

在传统的关系数据库里面，数据是存在记录里的。每一条记录包含若干字段。字段里的数据可能属于某个对象。字段及其所属对象的关系没有在数据库里用数据表现出来，只是作为元数据表现为列的形式（列名、数据类型、排序规则、外键）。对象没有被明确地建模，只是透过一系列表之间的关联有所体现。

语义网是相互联系的许多三元组（“主—谓—宾”三元组）构成的网络，在这些三元组里，谓词也是数据本身的一部分。而且，每个对象都有各自的标识符，这些标识符并非仅在数据库内部才有意义一个整数，而是在全球范围内都绝无二义的URI。

一个三元组是包含了三个值的一条记录：形式为(uri, uri, uri)或者(uri, uri, value)。前者将一个对象关联到另一个对象，例如像“Voxinc. is a supplier”这样一句事实陈述（“Voxinc.”、“is a”还有“supplier”都是有各自uri标识的语义主体）。后者将一个常量关联到一个对象，例如“Voxinc's phone number is 0842 020 9090”。简单直接的实现会是这个样子：

```
1.CREATE TABLE `triple` (
2.  `subject`   varchar(255) NOT NULL,
3.  `predicate` varchar(255) NOT NULL,
4.  `object`    longtext,
5.);
```

这张表完整实现了语义网，但放到任何实际的应用里，它都实在太慢了。有不少方法可以优化这个简单的实现，但据我所知，还没有什么所谓的“最佳实践”。我们必须解决几个问题：

- 如何唯一地标识一个三元组？（如果你的应用有此必要。主、谓、宾的组合本身没有唯一性。）
- 给定主、谓的情况下，如何快速地搜索？（已知这一组人，告诉我他们的名字）
- 已知谓、宾的情况下，如何快速地搜索？（找出名字以“Moham”开头的人）

为了解决以上问题，我们需要做一些修改：

- 建立一个仅存储三元组ID的索引表。
- 为用到的每一种主要数据类型（varchar(255)、longtext、integer、double、datetime）分别建立一个三元组表。
- 三元组表通过外键引用索引表。
- 为两种查表方式增加两个额外的索引：一个主—谓组合键、一个谓—宾组合键。

这是三元组索引表（我们用MySQL）：

```
1. CREATE TABLE `triple` (
2.   `triple_id` int(11) NOT NULL auto_increment,
3.   PRIMARY KEY (`triple_id`)
4. ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

这是“datetime”类型的三元组表，其他数据类型与此类似：

```
1.  CREATE TABLE `triple_datetime` (
2.    `triple_id`    int(11) NOT NULL,
3.    `subject_id`   int(11) NOT NULL,
4.    `predicate_id` int(11) NOT NULL,
5.    `object`       datetime NOT NULL,
6.    `active`       tinyint(1) NOT NULL
        DEFAULT '1',
7.    PRIMARY KEY (`triple_id`),
8.    KEY (`subject_id`, `predicate_id`),
9.    KEY (`predicate_id`, `object`),
10.   CONSTRAINT `triple_datetime_ibfk_1`
      FOREIGN KEY (`triple_id`)
      REFERENCES `triple` (`triple_id`)
      ON DELETE CASCADE
11. ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

上面的表定义很直观，应该不难理解。只有“active”字段例外，我们暂时还不需要它，不过下一节就会用到了。

predicate_id 指向一张单独的“uri”表，那里存放了谓词的完整URI。这种做法不是必需的，完全可以把URI存放在triple_longtext表里。

两个组合键有一种奇妙的副作用：应用开发者以后都不用操心键的问题。关系效率的键已经在那里了。

手工编写SQL语句去查询这个三元组数据库（triplestore），可能令人望而生畏。需要一种特殊的查询语言才能高效率地完成这项工作。下面我们就来讨论这个话题。

查询一个给定对象的全部数据，可通过选取具有给定subject_id的所有三元组（每种数据类型的三元组表都要做一次查询）。看着就效率不高，事实也是如此：与一个对象一条记录的传统做法相比，三元组数据库总是比较慢的。然而，在较为复杂的情况下，关系数据库会逼着你连接多个表来获取全部数据。我们从三元组数据库获取全部对象数据，用了5个单独的查询（每种数据类型的表查询一次），实际上却比合五为一的单个联合查询还快。不管要取数据的对象有多少个，我们都同样是这5个查询。例如，要取得ID分别为12076、12077、12078的三个对象的全部数据，查询语句如下：

```
1. SELECT `object` FROM `triple_varchar`  WHERE
`subject_id` IN (12076, 12077, 12078);
2. SELECT `object` FROM `triple_longtext` WHERE
`subject_id` IN (12076, 12077, 12078);
3. SELECT `object` FROM `triple_integer`  WHERE
`subject_id` IN (12076, 12077, 12078);
4. SELECT `object` FROM `triple_double`   WHERE
`subject_id` IN (12076, 12077, 12078);
5. SELECT `object` FROM `triple_datetime` WHERE
`subject_id` IN (12076, 12077, 12078);
```

从这个例子可以看到，从库中获取对象数据，并不需要显式地提供类型或属性的信息。对象的类型保存在了其中一个三元组里。当存在对象继承的情况下，对象的确切类型只能在运行时判断，此时三元组数据库的这个特点就很有用。

数组和多语言

很多对象属性是数组类型（无序集）。用关系数据库来建模这样的属性，就要为每个这类属性设一个单独的表。查询若干对象的包括数组属性在内的全部属性，可不是一件容易的事。在三元组数据库里，无序集可以建模成一系列主、谓相同而宾不同的三元组。查询全部对象数据的时候，获取数组属性的值与获取其他标量值并没有什么不同。

多语言也是关系数据库难以处理的情况。每一个需要显示成不同语言的属性，其表结构都需要相应调整，很难避免数据重复。在三元组数据库里，多语言属性几乎可以完全当作数组元素来处理，只不过它的谓词不是完全相同。我们用这样的URI来表示一个属性在不同语言中的变体：

```
http://our-business.com/supplier/description#nl
http://our-business.com/supplier/description#en
http://our-business.com/supplier/description#de
```

（在表中这些谓词由相应的整数ID代替以加速查询。）

数据版本控制

程序员都很熟悉版本控制，我们用它保存过去版本的代码，可以跟踪代码的变化，可以退回到旧的版本，还可以多人同时处理同一个文件。可是对于数据来说，版本控制并不常见。我想主要是因为在传统的关系数据库中，建立那样的系统要付出太高的额外成本。

我们的框架有一项需求，它要能保留某种形式的数据交换历史。稍加考虑就会发现，如果我们用了三元组，其实并不难跟踪所有的数据修改。因为从版本控制的角度来看，每个版本所作的修改，无非是增加了一些三元组，又删除了一些三元组。

所以，其实只需要增加两个表就够了：一个表用于跟踪修订数据，比如修改人、修改时间以及简短的描述供以后参考；另一个表用于跟踪版本里新增和删除的三元组。

```
1. CREATE TABLE `revision` (
2.   `revision_id`  int(11) not null auto_increment,
3.   `user_id`      int(11),
4.   `revision_timestamp`   int(11) not null,
5.   `revision_description` varchar(255),
6.   PRIMARY KEY  (`revision_id`)
7. ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
8.
9. CREATE TABLE IF NOT EXISTS
     `mod_lime_revision_action` (
10.  `action_id`   int(11) NOT NULL AUTO_INCREMENT,
11.  `revision_id` int(11) NOT NULL,
12.  `triple_id`   int(11) NOT NULL,
13.  `action`      enum ('ACTIVATE', 'DEACTIVATE')
       NOT NULL,
```

```
14.  `section_id`  int(11),
15.  PRIMARY KEY   (`action_id`),
16.  CONSTRAINT `revision_triple_ibfk_1`
     FOREIGN KEY (`revision_id`)
     REFERENCES `revision` (`revision_id`)
     ON DELETE CASCADE,
17.  CONSTRAINT `revision_triple_ibfk_2`
     FOREIGN KEY (`triple_id`)
     REFERENCES `triple` (`triple_id`)
     ON DELETE CASCADE
18. ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

每当用户修改了数据，就有一个新版本保存进数据库，库里还会记下新增及失效三元组的列表，另外还有对本次修改的简短说明。已经存在但处于失效状态的三元组会被激活，否则将创建新的三元组并激活。三元组绝不会真正被删除，只会被标记为失效。

查询三元组数据库的时候，因为里面其实包含了失效的数据，我们必须确保只有激活的三元组才会被查询到。

有了这样的设计，我们可以：

- 列出对数据所作的所有修订，显示何人何时做的修改，以及对修改的简短描述。
- 将修改退回到前一个版本，只需如此反方向执行修订：失效的三元组改为激活，激活的三元组改为失效。退回到任意的版本也是可行的，并不限于上一个版本。但要注意后续版本可能对前面的版本有依赖关系。
- 合作处理同一个对象。合并两个用户的修改可用各自起始及结束版本之间的差异。

对象数据库

我们在业务中都惯于使用对象的概念。数据构成的网必须经过重新组织才能用于一般的业务过程。为此我们需要在三元组数据库之上建立一层面向对象层。虽然 Web Ontology Language（OWL）就是为此而设计的，但我们没有选它，因为我们只需要其中一个很小的子集，而且我们希望在建模上保留最大的自由度，不然满足不了处理速度这项优先级很高的需求。我不能在这里一一介绍所有的细节，毕竟它是一个非常大的项目，不过有必要介绍以下特性：

- 数据库配置成一个Repository，应用开发者不允许进行直接的数据库访问。对象创建、修改、销毁、查询都通过Repository PI。这就保证了信息隐藏和模块化的OOP原则。
- 对象类型可与PHP类关联。不是必需的，但实践证明从PHP类生成对象类型非常简单。满足了多态性。
- 不仅简单对象被建模为对象（一组具有相同主词的三元组），对象的类型也被建模成对象。而且连类型的属性也被建模为对象。对象和它的类型可以用在同样的查询里。
- 对象类型可以有子类型。在三元组数据库里，可以简单直接地查询给定类型及其子类型的对象。
- 对象的属性也可以子类型化。因此你可以给类型树下游的子类型增加对其属性的数据类型约束，同时不影响类型树的上游。

这些特性都非常强大。仅仅用三元组这种数据结构，就能构造出真正的对象数据库。类型及属性都被看作普通的对象。也就是说同样的代码既可以处理普通的数据，又可以处理元数据。而且，用它来实现继承也较为简单，因为对象数据不用再硬塞到一行里。

查询语言

经过一段时间的使用，我们感觉只对这个对象数据库执行简单查询是很大的局限，我们希望查询能像SQL那样强大。另外由于我们还在继续使用普通的关系型的表，所以对象查询要能够结合对象数据库与关系型的表。因为上面的原因，专门的语义网查询语言SPARQL不能满足我们的需要。目前我们用一个查询对象和方法链（Method Chaining）去编写类似SQL的查询，然后由查询对象去构造实际的SQL语句。

提到这点是因为使用三元组数据库或者对象数据库的时候，确实有必要准备一种新的查询语言。背后的三元组数据库对普通的应用开发者来说太过简朴，底层的SQL查询会出现许多自连接将一个三元组的主词连接到另一个三元组的宾词，非常难理解。

结束语

我写这篇文章是因为感觉到这些新兴数据结构的细节知识还不够普及。能在一家鼓励向社区回馈知识的公司（Procurios！）工作也是值得骄傲的事情。来自Backbase的GerbertKaandorp曾经很认真地问我，为推广语义网做过些什么，仿佛这是一项神圣的任务。希望本文能有所贡献，启发读者自己动手去创建基于语义网的对象数据库。请务必告诉我您的想法！

作者简介

Patrick van Bergen住在荷兰Nijmegen，就职于Web开发公司Procurios。关注人工智能、软件架构和网络开发。有一位美丽的妻子Katja和三位千金Sky Sarah、Lila Sophia、Jade Selene。是漫画家Andreas的粉丝。他的个人网站patrickvanbergen.com。

责任编辑：郭晓刚（guoxg@csdn.net）