**摘要**

当用户向数据库发出查询时，他们会对结果有所了解。如果数据库中的搜索不可用，系统将返回空结果，或者更糟糕的是，错误的匹配结果。 我们称这个问题为MisMatch问题。 在本文中，我们在XML关键词搜索的环境中解决了MisMatch问题。 我们的解决方案基于我们介绍的两个新概念：目标节点类型和可区分性。 目标节点类型表示查询结果所匹配的节点类型，而可分辨性用于衡量查询关键词的重要性。 利用这些概念，我们开发了一种低成本的后处理算法，用于评估结果，使得MisMatch可以向用户生成有用的建议。

我们的方法有三个值得注意的特点：（1）对于使用MisMatch问题的查询，它生成解释，建议的查询及其样本结果作为输出给用户，帮助用户判断MisMatch问题是否在没有读取所有查询结果的情况下得到解决; （2）它是可移植的，因为它可以使用任何低级的基于共同祖先的匹配语义（对于没有ID引用的XML数据）或最小的基于Steiner树的匹配语义（对于具有ID引用的XML数据），其返回树结构作为结果。 它与所采用的结果检索方法的选择是正交的;（3）它在整个查询评估时间中占据很小比例的方式是轻量级的。 对三个真实数据集的大量实验验证了我们方法的有效性，效率和可扩展性。 已经构建了一个名为XClear的搜索引擎，可以在http://xclear.comp.nus.edu.sg上找到。

1、简介

当用户向数据库发出查询时，他们对结果有所期待。如果他们搜索的内容在数据库中不可用，由于产品从货架上移除，衣服大小不可用，他们寻找的结果可能找不到 在这种情况下，系统可能返回一个空结果，或者更糟糕的是返回错误结果。我们称之为MisMatch问题。

例如，用户想要搜索笔记本电脑。 她希望模特Vaio W颜色为红色。 如果数据库中的笔记本电脑Vaio W不能使用红色，那么无论数据的组织方式如何，或者是什么类型的查询，用户显然都无法得到她想要的东西。

MisMatch问题是一个自然而常见的问题。 它可以在任何形式的信息检索中发生在任何结构的数据上，即，可以是结构化查询或关于结构化，非结构化和半结构化数据的关键词查询。 这种问题在结构化数据的结构化查询环境中吸引了大量的研究工作[6,17,29-31]，其中包括失败查询和非回答查询等描述。 但是，在半结构化数据的关键词搜索环境中没有做过这样的工作。 这是一个需要解决的重要领域。 根据我们在[41]提供的XML关键词搜索引擎XClear上进行的实验，用户在27％的查询中遇到了这样的问题。

我们可以提供什么来帮助用户？ 理想情况下，如果我们与人类进行互动，我们可以获得以下帮助：

1.通知：“对不起，我们没有这样的产品。”

2.解释：“因为Vaio不能使用红色W.”

3.建议：“你可以选择其他一些颜色：白色，蓝色和粉红色。”

当在结构化数据（关系表）上发出结构化查询时，MisMatch问题（即，用户搜索的内容在数据库中不可用）导致空结果。 检测问题是微不足道的，因为空的结果是显而易见的。 将向用户提供消息（通知部分）。 一些现有的作品[6,17]试图通过精确定位导致空结果的约束来解释非答案查询（解释部分）。 一些作品[29-31]专注于生成一些替代约束来提出一些建议的查询（建议部分）。

当通过非结构化数据（在Web搜索中）发出关键词查询时，MisMatch问题将导致不匹配结果列表。首先检测问题甚至很困难，因为返回的结果很可能不是空的。情况可能是出现在一个文档中的查询关键词彼此相距很远而且与语义无关。例如，对于关键词查询'Vaio W red'，如果笔记本电脑Vaio W没有红色，则仍然可以返回许多网页，其中“Vaio W”出现在网页的一部分，而“红色”出现在网页的另一部分。它会导致不匹配的结果。因此，我们需要分析关键词在结果中是否“语义上”相关。这种分析具有挑战性，因为数据是非结构化的。对问题的一部分（仅建议生成部分）的有限解决方案是从查询日志[19,42]挖掘一些类似且流行的查询并将其显示给用户（建议部分）。但缺点是这种流行的查询并不能保证得到合理的结果。

在这项工作中，我们专注于在半结构化数据（即XML数据）的关键词搜索环境中识别和解决MisMatch问题。 大多数关于XML的研究都是基于没有ID引用的XML数据完成的。 在本文中，我们也专注于解决XML关键词搜索的MisMatch问题，而不考虑ID引用。 但在后面的Sect. 6，我们将我们的MisMatch问题解决方案扩展到XML关键词搜索，该搜索考虑了ID参考，其中匹配语义不同。 在本文的其余部分，没有说明，我们默认是在没有ID引用的情况下讨论XML。 我们将仅在Sect. 6.及之后讨论有ID引用的XML数据。

现在，让我们看一下MisMatch问题在XML关键词搜索的上下文中的行为。

**示例1**：图1中的XML数据树描述了在线购物中心的项目信息。 假设用户想要购买笔记本电脑。 她更喜欢索尼的红色Vaio W，并想知道它有多少。 然后，她可以发出查询Q = {'Vaio'，'W'，'red'，'price'}来搜索笔记本电脑。 不幸的是，没有笔记本电脑可以满足她的所有要求。 Vaio W只有三种颜色：白色，蓝色和粉红色。 现有的关键词搜索方法，如LCA（最低共同祖先）[33]，SLCA [38]，ELCA [12]甚至LCA的最新变种[22]，仍然可以找到包含所有查询关键词的一些结果。 其中一个查询结果是以商店为根的子树：0.0.0，其中关键词“red”与一台笔记本电脑匹配，而其余关键词与另一台笔记本电脑匹配。 显然，用户不期望以商店为根的子树，因为它包含太多无关信息，即所有笔记本电脑。

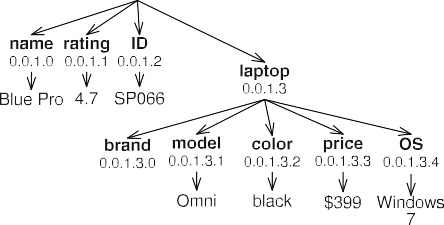
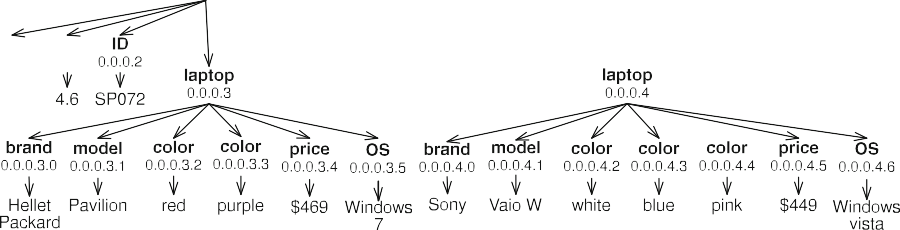


Fig. 1 Sample XML document about an online shopping mall

更糟糕的是，可能有数百家商店出售Vaio W，因此返回了数百个不匹配的结果。 在这种情况下，想象一下如果用户与销售人员进行交互，她将被告知产品不可用，并建议使用笔记本电脑Vaio W的其他一些可用颜色。

我们可以看到，XML关键词搜索中的MisMatch问题也会导致错配结果列表。 它为搜索引擎带来了三个挑战：（1）如何设计一种检测方法来区分具有MisMatch问题的查询和没有问题的查询; （2）如何解释查询导致不匹配结果的原因; （3）如何找到好的建议，以及将它们呈现给用户的好方法。

我们对MisMatch问题的解决方案是在查询评估结束时运行一个小的后处理作业，由两个组件组成，即检测器和建议器。前者解决了上述第一个挑战，后者解决了其余两个问题。

我们的不匹配检测技术的核心思想是基于目标节点类型的概念（正式定义见第2节）。 直观地，目标节点类型表示查询结果r想要匹配的节点的类型。 我们在架构级别计算它。 同时，通过现有技术在数据级别计算结果r的实际根。 如果r的根与其目标节点类型不匹配，我们声称r错过了目标。 我们可以对查询Q的所有结果执行类似的检查。如果查询Q的所有结果都错过了他们的目标，那么我们说Q具有MisMatch问题。

一旦检测到不匹配，我们就会提出一个称为可区分性的概念，以便在原始查询中找到“重要”关键词，并使用这些概念来解释不匹配的原因并建议可能的放松。 区分性受到IR [32]中提出的tf \* idf评分指标的启发，同时考虑了XML数据的结构特性。 然后根据每个查询结果r，我们尝试查找一些“近似”查询结果，其中包含这些“重要”查询关键词，并且在结构上与r一致，同时合理地替换其余“不太重要”的查询关键词。 最后，可以从近似结果推断出解释和建议的查询。 为了进一步改善用户体验，我们的建议甚至在不评估查询Qj的情况下为每个建议的查询Qj生成样本结果，这有助于用户判断Qj是否有用。

把这些放在一起，我们有完整的算法。 输入是搜索引擎返回的所有结果的（排名）列表。 对于具有MisMatch问题的用户查询，我们的算法输出由三部分组成：

1. 向用户显式通知：“您搜索的内容是无法使用的”。
2. 解释查询中的哪些关键词导致不匹配的结果。
3. 一些数据驱动的保证有合理的结果的建议查询。

请注意，查询结果的质量始终是关键词搜索问题的圣杯，特别是对于那些存在歧义问题的查询。由于没有现有的方法可以彻底解决模糊查询[2]，我们的解决方案不能保证MisMatch问题的检测也百分之百正确，这实际上也已在我们的实验中进行了研究。 这也解释了为什么我们采用后处理而不影响原始查询的结果，即原始查询的结果将与我们生成的建议一起返回。

给定查询有许多可能的松弛，其中许多本身也可能是空的（导致不匹配）。 重要的是要确保所给出的建议至少具有一些结果并且不会自行不匹配。

正如下面相关工作部分所讨论的，在查询放松和生成部分匹配答案方面有大量工作。 这些系统虽然很有价值，但并未解决我们上面描述的所有三个挑战，因此不适合我们的问题背景。 特别是，它们中的许多产生了大量可能的部分匹配答案的列表，用户必须通过这些答案才能意识到存在不匹配。

总之，我们的主要贡献包括：

1. 我们在XML关键词搜索中识别MisMatch问题。 我们通过调查查询结果并推断每个查询结果的目标节点类型来检测MisMatch问题。 它是可移植的，因为它可以与任何基于LCA的匹配语义一起使用，并且与结果检索方法的选择正交。
2. 我们设计了一种数据驱动方法，通过查找近似查询结果来生成解释和建议查询，其中包含原始查询Q中的重要关键词，同时与Q的结果具有一致的结构。我们提出可区分性，即结构 - 了解tf \* idf评分指标，量化关键词的重要性。
3. 我们提出了一种新的基于位图的标记方案，以加速寻找近似结果。 因此，MisMatch检测器和建议器是轻量级的：它只占整个查询处理时间的4％.
4. 我们进一步扩展检测器和建议器以处理带有ID引用的XML数据，其中数据不再是树结构，并且搜索语义/方法是不同的，即最小的基于Steiner树的搜索语义/方法。 这导致了解决MisMatch问题的通用框架，该问题是可移植的，数据驱动的和轻量级的。
5. 我们构建了一个名为XClear的搜索引擎[41]，它嵌入了上面的MisMatch问题检测器和建议器，能够处理任何带/不带ID引用的XML数据。 大量实验验证了我们方法的有效性，效率和可扩展性。

我们首先解决了没有ID引用的XML数据的失配问题。 第2节检测失配问题。第3节讨论如何查找解释和推荐的查询。 第4节介绍了我们用于有效近似结果检测的标签方案。 第5节介绍了索引和算法。 然后我们在第6节进一步将我们的错误匹配解决方案扩展到具有ID引用的XML数据。第7节是实验。第8节是相关工作。我们还在第8节中说明了这项工作与我们以前的工作[39]之间的区别。 第9节总结，展望未来。

1. 检测没有ID引用的XML数据上的失配问题

在本节中，我们将介绍如何在没有ID引用的情况下检测XML数据上的MisMatch问题。首先，我们对XML文档和查询结果进行建模（第2.1节）。然后我们将讨论检测器如何根据查询结果推断用户可能的搜索目标（第2.2节）。 最后，通过检查推断的搜索目标和查询结果（第2.3节），可以很容易地检测到MisMatch问题。

2.1准备工作

2.1.1数据模型

我们将没有ID引用的以数据为中心的XML文档建模为一个有根的，被标记的和有序的树。树的每个节点对应于XML数据的一个元素，并且它具有标签名和（可选地）某个值。在不失一般性的情况下，我们只需使用“node”一词来表示XML树中的节点。为了加速关键词查询处理，所有现有工作都采用杜威标签方案[37]。如图1所示，对于节点n，其dewey标签由一系列组件组成，这些组件隐式地包含从文档根到n的路径上的所有祖先节点。例如，从laptop：0.0.0.3，很容易发现它的父标签是0.0.0

**定义1**（节点类型）XML树中节点n的类型，表示为n.type，是从root到n的标记名路径。

在本文的其余部分，如果不引起歧义，则标签名称n用于表示n的节点类型。

**定义2**（关键词匹配节点）如果标签名称或n的值部分包含k，则节点n被称为关键词k的匹配关键词的节点。

**定义3**（子树包含）如果n等于m或者存在从n到m的有向路径，则认为节点n子树包含另一个节点m。n也被认为子树包含 m的标签名或值中的关键词。

例如，在图1中，laptop:0.0.0.3的节点类型是online\_mall / electronics / shop / laptop；color：0.0.0.3.2是关于关键字'red' 的关键字匹配节点；laptop：0.0.0.3子树包含节点color：0.0.0.3.2; laptop：0.0.0.3也被称为子树包含关键字'red'，因为'red'是颜色的值：0.0.0.3.2。

2.1.2查询结果格式

要定义表示查询结果的通用格式，让我们先看一下现有的匹配语义。 到目前为止，现有的匹配语义，如SLCA [15,38]，ELCA [12]，基于实体的SLCA [26]都基于最低共同祖先（LCA）的概念。 设lca（m1，...，mn）是节点m1，...，mn的最低共同祖先。 对于给定查询Q = {k1，...，kn}和XML文档D，Li表示ki的倒排列表。 然后将文档D中查询Q 的LCA定义为LC A（Q）= {v | v = lca（m1，...，mn），mi∈Li（1≦ i ≦n）}。 SLCA和ELCA都定义了LC A（Q）的子集，第8节详细描述了SLCA和ELCA，以及它们与LCA的关系。

**定义4**（查询结果格式）对于关键字查询Q = {k1，...，kn}，我们将查询结果的格式定义为：

r = (vlca, {m1, m2,..., mn} )

其中mi是关键字ki（i∈[1，n]）的关键字匹配节点，vlca是节点m1，...，mn的最低共同祖先，即vlca = lca（m1，...，mn）。

定义4在两个方面非常通用：（1）它与搜索引擎采用的任何现有的基于LCA的匹配语义兼容，因为节点v是查询Q的SLCA（或ELCA）节点的一个必要条件是：v必须是一组关于Q的关键字匹配节点mi 的最低共同祖先；（2）我们的查询结果格式构成“返回路径”和“返回子树”的骨架（看英文）。这一现象很好的解释了我们的解决方案的可移植性特征，以便稍后在3.4节中检测和解决MisMatch问题。

2.2推断查询结果的目标节点类型

在本节中，我们将根据查询结果推断用户的搜索目标。 很明显，用户的搜索意图不依赖于数据的可用性。 通常，用户发起关键字查询时知道自己的搜索目标，但不知道数据的可用性。 因此，应该通过分析查询关键字的语义来推断用户的搜索意图。 然而，现有的基于LCA的结果检索方法纯粹在数据级别生成查询结果，而根本不会推断用户的搜索意图。 因此，我们建议使用节点类型来模拟关键字匹配节点的语义，这些节点由查询关键字匹配。通过分析这些节点类型，我们将在模式级别推断用户的搜索意图。

由于关键字可以匹配不同类型的节点，因此对于一个特定查询用户的搜索目标可能是各种各样的。 例如，关键字“price”可以匹配图1中所有者的名称（0.3）或产品价格（0.0.0.3.4）。但一个确定的查询结果r对应于唯一一个搜索目标，因为每个查询关键字在给定的查询结果r中具有唯一的对应关键字匹配节点。 因此，我们引入了一个称为目标节点类型（TNT）的概念来表示查询结果r要匹配的节点类型。

为了推断结果r的TNT，我们建议使用节点类型来模拟每个关键字匹配节点的语义。

**示例2**：对于示例1中的查询Q = {'V aio'，'W'，'red'，'price'}，如果用户正在与销售人员进行交互，则销售人员将知道用户正在寻找笔记本电脑，因为销售人员知道每个查询关键字的含义。这里的XML关键字搜索结果是r = (0.0.0,{0.0.0.4.1,0.0.0.4.1,0.0.0.3.2,0.0.0.4.5})。 我们使用节点类型来模拟每个关键字匹配节点的语义。不同关键字匹配节点的节点类型是（'Vaio'和'W'匹配同一节点）：

0.0.0.4.1 :{online\_mall/electronics/shop/laptop/model}

0.0.0.3.2 :{online\_mall/electronics/shop/laptop/color }

0.0.0.4.5 :{online\_mall/electronics/shop/laptop/ price}.

然后我们就可以知道用户输入了三种信息：笔记本电脑型号，笔记本电脑颜色和笔记本电脑价格。用户的搜索意图，即笔记本电脑，对应于节点类型“online\_mall / electronics / shop / laptop”，其与上述三种节点类型密切相关。

遵循类似的LCA理念——找到连接所有查询关键字的最低/最小节点作为最相关的结果，我们定义最低节点类型，它在模式级别连接上述所有节点类型作为目标节点类型，其中正式定义将在稍后介绍。它是连接用户输入信息的最相关的节点类型。 例如，在示例2中，online\_mall / electronics / shop / laptop是在模式级别连接了笔记本电脑型号，笔记本电脑颜色和笔记本电脑价格的最低节点类型，即使没有笔记本电脑可以满足数据级别的所有要求。

然而，实际上XML文档在一个tb类型的节点可以子树包含多少个ta类型的节点上面存在一些限制。例如，一个笔记本电脑节点（节点类型为online\_mall/electronics/shop/laptop）只能子树包含一个笔记本电脑模型节点（节点类型为online\_mall/electronics/shop/laptop/model），而它可以子树包含多个笔记本电脑颜色节点（节点类型为online\_mall/electronics/shop/laptop/color）。同样，一个商店节点（节点类型为online\_mall/electronics/shop）可以子树包含多个笔记本电脑节点，因此，商店节点也可以子树包含多个笔记本电脑模型节点。

当我们试图在模式级别找到连接用户输入信息的最低节点类型时，这些约束条件将影响我们推断目标节点类型。

**示例3**：假设一个用户想要找到一家出售两种笔记本电脑型号的商店，即由Hewlett Packard生产的Pavilion和Omni型号，那么她可以在图1中发起一个查询Q = {‘Hewlett’, ‘Packard’, ‘Pavilion’, ‘Omni’} ，其中包含两个不同的笔记本电脑型号名称。如果用户与销售人员互动，销售人员会知道用户不是在找特定的笔记本电脑，而是与两种不同的笔记本电脑相关的东西，例如，一家出售这两种笔记本电脑的商店。在XML关键字搜索中，查询结果是以电子节点：r=（0.0，0.0.0.3.0，0.0.0.3.0，0.0.0.3.1，0.0.1.3.1）为根的子树。不同关键字匹配节点的节点类型为（“‘Hewlett”和“Packard”匹配同一节点）：

0.0.0.3.0 :{online\_mall/electronics/shop/laptop/brand}

0.0.0.3.1 :{online\_mall/electronics/shop/laptop/model}

0.0.1.3.1 :{online\_mall/electronics/shop/laptop/model}.

用户的关键字描述了一个笔记本品牌和两种不同的笔记本型号，即用户输入两个不同的笔记本电脑名称，匹配两种不同的笔记本型号。在这种情况下，假设在数据中一台笔记本电脑只能有一个笔记本电脑型号名称，则连接上述三种节点类型的最低节点类型不再是“online\_mall/electronics/shop/laptop”，因为这里有两个不同的笔记本电脑型号节点，而一个笔记本电脑节点只能子树包含一个笔记本电脑型号节点。相反，连接上述三种节点类型的最低节点类型是“online\_mall/electronics/shop”，因为一个商店节点可以子树包含多个笔记本电脑型号节点。因此，我们可以推断出用户的搜索意图是找到一家出售两种笔记本电脑型号的商店，而不是找到一种特定的笔记本电脑型号。

我们可以从XML文档的模式轻易地推断出不同类型的节点之间的包含约束条件。例如，DTD是一种常用的XML模式语言，其中运算符\*（出现零次或多次），+（出现一次或多次），？（出现零次或一次）用于指定特定类型节点下的子元素或属性的出现次数约束。令 t1.maxContain(t2) 为t1类型节点可以子树包含的t2类型节点的最大数量，范围是[0, +∞]。例如，在图1中，如果一个笔记本电脑节点最多可以子树包含一个笔记本电脑型号节点，那么laptop.maxContain(model) = 1; 如果一个商店节点可以子树包含多个笔记本电脑节点，我们得到shop.maxContain(laptop) = +∞; 此外，由于商店节点是笔记本电脑节点的父节点，笔记本电脑节点是笔记本电脑型号节点的父节点，我们可以通过乘以上述两个值进一步推断出shop.maxContain(model) = +∞。 这种计算可以基于模式离线完成。 如果XML文档的模式不可用，我们可以通过扫描XML文档总结DataGuide[11]来近似地推断出这些约束条件。

现在我们需要计算关键字匹配节点的每种不同节点类型出现的次数。设**T** = {t1, t2,...,tx }是关键字匹配节点的一组不同的节点类型。由于一些关键字匹配节点可能是相同的节点类型，令count(ti) 为 ti 类型的关键字匹配节点的数量。例如，对于示例3中的查询结果，有三个不同的关键字匹配节点：其中两个是online\_mall / electronics / shop / laptop / model类型，另外一个是online\_mall/electr onics/shop/laptop/brand类型。在这种情况下，**T** = {brand, model}, count(brand) =1 且count(model) = 2。

我们试图找到连接到用户输入信息的最相关的节点类型作为TNT，即找到可以连接到查询关键字匹配的所有节点类型的最低节点类型。接下来，我们将正式定义结果r的目标节点类型。

首先，TNT应该与**T**中的每个节点类型相关并连接，即TNT应该是**T**中所有节点类型的公共前缀。其次，目标节点类型的节点应该能够子树包含**T**中每个节点类型的所有出现。最后，TNT应该尽可能的低，以便它可以尽可能近地连接到**T**中的每个节点类型。所以我们正式定义扩展过的TNT如下：

**定义5**（单个查询结果的目标节点类型（TNT））：在XML文档D上给定查询Q = {k1，k2，...，kn} 和查询结果r = (vlca，{m1，m2，...，mn)}，令**T** = {t1，t2，...，tx} 为m1到mn的不同节点类型的集合，结果r的目标节点类型TNT (r) 定义为：

TNT (r) = t，其中t满足以下3个条件:

条件1: t ∈ commonPref i x(t1, t2,..., tx );

条件2: t.maxContain(ti) ≥ count(ti),i ∈ [0, x];

条件3: t不存在满足条件1和条件2的后代t’。

其中 commonPref i x(t1, t2,..., tx ) 表示一组节点类型的所有可能的公共前缀；t.maxContain(ti) 表示t型节点可以子树包含的ti型节点的最大数量；count(ti)表示m1到mn中ti类型的不同关键字匹配节点的数量。

TNT是可以连接到查询关键字匹配的所有节点类型最低的节点类型。不管用户搜索的目标是否存在于XML文档的数据中，TNT是在模式级别使用节点类型定义的。为了计算一个给定结果的TNT，我们从底向上检查**T**中每个节点类型的前缀，查看它是否满足条件2。

**示例4**：对于示例1中的查询Q = {'Vaio', 'W', ’red’, ‘price’}，其中一个结果为r = (0.0.0,{0.0.0.4.1,0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5}),，其中不同关键字匹配节点的节点类型是（'Vaio'和'W'匹配同一节点）：

0.0.0.4.1: {online\_mall/electr onics/shop/laptop/model}(表示为t1)，

0.0.0.3.2: {online\_mall/electr onics/shop/laptop/color}表示为t2)，

0.0.0.4.5: {online\_mall/electr onics/shop/laptop/price}(表示为t3).

不同节点类型的集合**T** = {t1，t2，t3}，其中count(t1) = 1, count(t2) = 1, count(t3) = 1。

然后我们检查**T**中所有节点类型的前缀。最低的是t = “online\_mall/electr onics/shop/laptop”。假设我们有以下约束条件（从XML模式推断或通过扫描XML文档）：一个笔记本节点可以子树包含一个模型节点，一个价格节点和多个颜色节点，然后它将满足：t.maxContain(t1) = 1 ≥ count(t1) = 1, t.maxContain(t2) = +∞ ≥ count(t2) 和 t.maxContain(t3) = 1 ≥ count(t3)。

因此，TNT (r) = t =“online\_mall / electronics / shop / laptop”，即使没有笔记本电脑可以满足用户在数据级别的所有要求。它是可以连接到查询关键字匹配的所有节点类型的最低节点类型。

**示例5**：对于示例3中的查询Q = {‘Hewlett’,‘Packard’,‘Pavilion’,‘Omni’}，由于图1中没有同时销售这两种型号的商店，因此返回的结果不是用户预期的商店。其中一个结果是r = (0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1})，其中不同关键字匹配节点的节点类型是（'Hewlett'和'Packard'匹配同一节点）：

0.0.0.3.0 :{online\_mall/electronics/shop/laptop/brand}(表示为t1)

0.0.0.3.1 :{online\_mall/electronics/shop/laptop/model}(表示为t2)

0.0.1.3.1 :{online\_mall/electronics/shop/laptop/model}(表示为t2).

不同节点类型的集合**T** = {t1, t2}，其中count(t1) =1，count(t2) = 2。

然后我们检查**T**中所有节点类型的前缀。最低的是t = “online\_mall/electr onics/shop/laptop”。假设我们有以下约束条件（从XML模式推断或通过扫描XML文档）：一个笔记本节点可以子树包含一个品牌节点和一个型号节点，然后我们得到：t.maxContain(t1) = 1 ≥ count(t1) = 1，但是t.maxContain(t2) = 1 ≯ count(t2) = 2 。正如我们所见，t不是TNT，因为笔记本电脑节点不能子树包含两个型号节点。

然后我们检查另一个前缀t’= “online\_mall/electronics/shop”，假设我们在XML文档中有以下约束条件：一个商店节点可以子树包含多个品牌节点和多个模型节点，然后我们得到：t’.maxContain(t1) = +∞ ≥ count(t1) = 1，t’.maxContain(t2) = +∞ ≥ count(t2) = 2 。

因此，r的目标节点类型是TNT (r) = t =“online\_mall / electronics / shop”。它是可以连接到查询关键字匹配的所有节点类型的最低节点类型。

我们的解决方案假设没有被提供的外部语义，因为通常XML数据不存在这样的信息，因此我们使用节点类型来模拟语义，其中相同类型的两个节点将具有相同的语义。如果我们确实有外部语义，比如词库，本体等，我们可以进一步改进我们的方法，这样我们甚至可以告诉节点类型“/ laptop / color”和“/ notebook / color”具有相同的语义而节点类型“/ owner / name”和“/ product / name”具有不同的语义。这将是我们未来的工作之一。

与（示例2中）销售员理解客户请求的方式类似，应该通过分析用户查询关键字关于元数据和数据的语义来推断用户的搜索意图。目标节点类型的直观思想是使用节点类型（在元数据级别）来模拟每个关键字匹配节点的语义，其包含匹配的查询关键字（在数据级别）。通过分析这些节点类型，我们可以直观地推断出用户的搜索意图。

2.3根据目标节点类型检测失配问题

在推断出查询结果r的目标节点类型的情况下，检测器应该确定TNT与r的实际根即vlca之间是否存在不匹配。如果数据中存在用户搜索的内容，则这两者应该是一致的。

**定义6**：在XML数据D上给出查询Q = {k1，k2，...，kn}和查询结果r = (vlca,{m1, m2,..., mn})，如果vlca与TNT (r) 的节点类型不同，则查询结果r未命中目标。

对于示例4中的结果r，vlca.type = shop ≠ laptop = TNT (r)，所以我们说r未命中目标。现在，我们可以正式定义失配问题。

**定义7**（失配问题）：给定查询Q及其从关键字搜索引擎检索到的结果**R**，如果所有的r∈**R**都未命中目标，则Q具有失配问题。

正如我们前面在2.2节中提到的那样，对于某个查询不同的用户有不同的搜索目标。在这里，我们选择采取保守的方法：只有当所有可能的搜索意图都不匹配时，我们才判定查询存在失配问题。这种结论适用于具有不同意图的所有用户。 例如，对于示例4中的结果r，我们推断它未命中目标。以类似的方式，我们还将计算其他结果（如果有的话）的TNT。只有当所有结果都未命中相应的目标时，我们才会声称查询出现了失配问题。

此外，用户通常会从排名靠前的检索结果开始查看。因此，在不失一般性的情况下，我们也可以通过考虑Q的前K个检索结果来轻松扩展定义7，这样我们就可以通过改变K的值来调整方法的积极性，K可以像1那样具有攻击性，或者可以像查询结果的总数一样容忍。

检测器的时间复杂度是O(|**R**|)，这是非常高效的。正如之后的5.1节中所讨论的，我们在构建关键字倒排列表时存储每个节点的类型信息。假设查询中的关键字的数量和XML树的深度受到一些常数的限制，那么对于每个r∈**R**，TNT(r)都可以在O(1) 时间内被计算出来。

潜在直觉总结：

请注意，当用户指定结构化查询（如SQL）时，他们需要在select子句中指定他们的目标，并在where子句中指定谓词。作为类比，节点类型（定义1）是结构化查询中指定的谓词的隐式表示，而不同之处在于，在关键字查询中，您无法为关键字之间的结构关系指定约束。因此，我们尝试在关键字查询中收集可能的搜索谓词（以与每个查询关键字相关联的节点类型的形式），这引出了定义5。由于我们的主要目标是检测具有失配问题的查询，我们利用此“目标”（在结构级别）与彼目标（即内容级别的结果r的根节点）之间的不一致来证明是否发生了失配问题，这引出了定义6“未命中目标”。失配问题的直观概念（在定义7中）是：如果所有的查询结果的结构级别和内容级别之间都存在目标不一致的问题，那么我们可以断定用户搜索的内容在数据库中不可用。

1. 关于没有ID引用的XML数据的失配问题的解释和建议

正如第2节中所讨论的，失配问题的主要特征是：没有一个TNT节点子树包含所有的查询关键字。因此，查询关键字必须分布在多个TNT节点中，才会导致结果不匹配。总之，返回子树的根始终是用户期望的TNT节点的祖先。给定用户查询Q = {k1，k2，...，kn}和一个不匹配的查询结果r =(vlca，{m1，m2，...，mn})，其中mi是ki的关键字匹配节点，找到解释和一些有用的推荐的查询的基本思想可以分三个步骤来说明。

步骤1：由于r中的每个关键字匹配节点mi都可能包含Q中的几个关键字**K**，我们首先提出一种称为可区分性的tf \* idf启发式方法来评估这个**K**的重要性。

步骤2：然后我们尝试找到近似的查询结果，即r ’= (vlca’，{m1’，m2’，...，mn’})，它们是包含“重要”关键字的某些子树（由步骤1导出）。理想的近似结果r ’应满足以下特性：（a）r ’的节点类型应该与TNT(r) 相同；（b）对于原始结果r中的每个关键字匹配节点mi，总是存在节点mi’ 与mi (i ∈ [1, n]) 具有相同的节点类型。通过这些特性，至少可以确保r ’和r的结构彼此一致。

步骤3：然后，我们可以确定用户查询中的哪些关键字导致了结果不匹配，即未被近似结果包含的查询关键字。这是解释的部分。我们可以通过在步骤2中用与上述（近似结果中）mi’ 相关的关键字替换那些关键字，来进一步推断出建议的查询。

步骤1在第3.1节中说明，最后两步在第3.2节中说明。最后，在第3.3节中我们通过讨论如何对建议的查询进行排名来补充我们的建议。

3.1可区分性

在本节中，我们将提出一个概念来衡量查询关键字的重要性，即可区分性。我们发现查询关键字的重要性与它们匹配节点的类型密切相关。例如，在图1中，关键字“blue”可以匹配商店名称name:0.0.1.0或笔记本电脑颜色color:0.0.0.4.3。当它匹配商店名称时，它可能很重要，因为很少有商店名称包含关键字“blue”；当它匹配笔记本电脑颜色时，它可能不那么重要，因为许多颜色节点包含关键字“blue”。因此，我们提出了可区分性这个概念。

可区分性：D(**K**, t)表示当**K**匹配t类型节点时查询关键字**K**的重要性，这也意味着这个t型节点子树包含**K**中的每个关键字。较大的D(**K**, t) 表示对于t来说**K**是很重要的。

回想一下第3节中的第1步，在图3中，**K** 实际上表示从关键字匹配节点导出的查询关键字。 为了量化D(K, t)，我们提出了一个受术语频率\*逆文档频率(tf\*idf )[32]启发的评分测量方法，tf\*idf被广泛用于信息检索。

对于tf，我们可以简单地计算XML节点中的关键字频率。 在这项工作中，我们专注于以数据为中心的XML数据，其中每个XML节点都不包含长文本，在大多数情况下，关键字频率为1。同样的问题在[13]中也有指出，所以我们遵循[13]并且在公式中不考虑tf。

对于idf，它告诉我们被较少文档包含的关键字更重要。与idf类似，在XML的上下文中我们得到直觉1。令ft为t类型节点的数量，ftK为子树包含**K**中每个关键字的t型节点的数量。

**直觉1** idf (K, t)： 如果包含关键词**K**的t型节点的数量很少，则**K**对于节点类型t应该是重要的。形式上，与ft相比，ftK越小，idf (**K**, t) 应该越大。

正如有许多遵循直觉1的idf变体，我们定义idf (**K**, t) = 1 − ft**K** / ft。这样，idf (K, t)在[0,1]中归一化。

tf \* idf的前提是只有一种类型的（平面）文档，但在XML数据的上下文中，有多种类型的节点。节点类型本身也可能影响匹配节点的关键字的重要性。让我们先看看一个激励性的例子。

**示例6**：考虑图1中的关键字 'price'。它可以匹配一个所有者节点和所有的价格节点。当“price”与价格节点匹配时，它可能并不重要，因为有许多价格节点并且所有价格节点都包含关键词“price”。因此，idf ({‘price’},price)=0，因为ft**K** = ft。当它与所有者节点匹配时，它应该是重要的，因为在整个XML数据中只有唯一一个所有者。但由于ft**K** = ft = 1，同样的idf ({‘pprice’},owner)=0 。我们可以看到，仅仅通过tf \* idf，我们无法区分这两种情况（这两种情况中idf都为0）这是因为tf \* idf的方案假设只有一种类型的节点，而我们有不同类型的节点，我们需要考虑不同节点类型的权重。

所以我们提出直觉2来满足节点类型权重 (ntw)。

**直觉2** ntw(t)：节点类型t的权重与XML数据中的ft成反比。

因此，我们定义ntw(t) = 1 / ft。最后，我们可以定义D(**K**, t) 来实现可区分性的概念：

D(**K**, t) = idf (**K**, t) + ntw(t) = 1 - ft**K** / ft + 1 / ft  （1）

很容易验证可区分性的范围是 (0,1]。注意，可以有一些替代公式能够模拟上述的可区分性。研究这些可能性和它们之间的差异是我们的未来工作之一。

3.2 找出解释和建议的查询

为了找到解释和建议的查询，我们首先需要从初始查询的结果r中找到一些“重要的”查询关键字（就可区分性而言）。首先，我们需要设置一个阈值τ= 90％。 那些可区分性高于τ的关键词被认为是“重要的”，必须予以保留。此外，我们发现那些“重要”的关键字**K**实际上来自r的关键字匹配节点。因此，我们可能需要同时考虑两种独立的情况：

（1）**K**来自r的单个关键字匹配节点；

（2）**K**来自r的多个关键字匹配节点，即，梳理来自多个关键字匹配节点的关键字可以实现高可区分性。

然后剩下的任务就是找到近似的结果，每个结果都包含重要的关键字**K**，我们可以从近似结果推断出建议的查询。

3.2.1第1阶段：基于单个关键字匹配节点

在阶段1中，我们从单个关键字匹配节点派生重要关键字，并找到如下近似结果：

给定用户查询Q和失配查询结果r = (vlca,{ m1, m2,..., mn})，每个关键字匹配节点mi包含一些Q中的关键字**K**i。对于每个不同的mi，我们计算可区分性D(**K**i，mi.type)。如果它大于阈值，那么我们试图找到包含mi的TNT节点作为近似结果。令从vlca到mi的路径为(vlca/p1/p2/.../p j /mi)，其中p1，p2，...，p j是vlca和mi之间的节点。然后我们继续从p1向下到mi遍历每个节点vlca’（即vlca’∈{p1，p2，...，p j，mi}），并验证以vlca’为根的子树是否可以形成关于r的近似查询结果r’= (vlca’, {m1’, m2’, ..., mn’}) 。

**定义8**（近似结果）：给定查询Q的查询结果r = (vlca，{m1，m2，...，mn})，如果r’ = (vlca’，{m1’，m2’，...，mn’}) 具有以下两个属性，则r’是近似结果：

P1: vlca’.type = TNT (r)

P2: 对于初始结果r中的∀mi，在r’中∃mi’， 使得mi’.type = mi .type，其中i∈[1, n]。

我们指定P1属性以确保近似结果应与我们在第2节中推断出来的基于结果r的目标节点类型一致。换句话说，P1是为了确保r’ 的vlca’ 应该与结果r要匹配的TNT具有相同的节点类型（但是不能这样做）。P2是为了确保r 和r’ 的内部结构的一致性，r 的关键字匹配节点中出现的每个节点类型也必须出现在r’ 的关键字匹配节点中。直观地说，每个关键字匹配节点的节点类型隐式地反映了用户期望的查询结果的约束。因此，我们需要将它们都保存到近似结果中。作为类比，它是结构化查询中指定的谓词的隐式表示，而不同之处在于，在关键字查询中，您无法为关键字之间的结构关系指定约束。请注意，如果有可能，mi’ 和mi应该是同一节点，因为我们更喜欢尽可能少地改变关键字。这也是我们通过检查从 vlca 到mi 的路径寻找近似结果的原因。只有不在以 vlca’ 为根的子树中的mi才会被另外的节点 mi’ 替换。

近似结果的构建从以 vlca’ 为根的子树开始，然后检查该子树是否满足上述属性。

建议的查询和示例查询结果：在找到近似查询结果后，可以通过以下方式轻松地推断出解释和建议查询：1）对于每个与mi’不是同一节点的关键字匹配节点mi，mi中的查询关键字是导致结果失配的原因；2）可以通过将mi中的关键字替换为mi’ 的相关值来生成建议的查询。此外，近似查询结果将用作相应建议查询的示例查询结果。

**属性1**：对于存在失配问题的给定查询，由我们的失配建议器生成的每个建议查询都保证至少有一个结果不会错过目标。

**证明**：给定查询结果r = (vlca，{m1，m2，...，mn}) 及其近似结果r’= (vlca’，{m1’，m2’，...，mn’})，令从m1 到mn 的不同节点类型的集合是**T** = {t1，t2，...，tx} 以及从m1’ 到mn’ 的不同节点类型的集合是**T’** = {t1’，t2’，...，tx’}，根据定义8中的P2，我们可以得到ti = ti’ (i ∈ [1, x])。然后根据定义5，我们可以得到TNT (r) = TNT (r’)。根据定义8中的P1，我们可以得到 vlca’.type = TNT (r)。所以r’不会错过目标。

接下来，我们将使用两个运行示例来说明我们如何找到建议的查询和示例查询结果。以下两个运行示例分别对应于示例4和5中的查询。

示例7：对于例4中的查询Q = {'Vaio'，'W'，'red'，'price'}，一个查询结果为r = (0.0.0，{0.0.0.4.1,0.0.0.4 .1,0.0.0.3.2,0.0.0.4.5})，其中只有三个不同的关键字匹配节点。因此，我们计算关于这三个关键字匹配节点中的查询关键字的三个可区分性值：D({‘Vaio’, ‘W’}, model) = 100 %, D({‘red’}, color) = 68.2 %, D({‘price’}, price) = 0.5%.

由于D({‘Vaio’, ‘W’}, model) >τ= 90％，因此它是重要的且必须被保留。然后我们检查从shop：0.0.0(vlca) 到model：0.0.0.4.1(mi) 的路径，即shop:0.0.0 / laptop:0.0.0.4 / model:0.0.0.4.1。我们从示例4中知道TNT (r) = laptop，所以我们检查以laptop为根的子树：0.0.0.4。对于原始结果r中的每个关键字匹配节点mi，在以0.0.0.4为根的子树内，我们总能找到具有相同类型的节点mi’。例如，对于r中的关键字匹配节点0.0.0.3.2，我们可以找到具有相同节点类型的节点0.0.0.4.2：(0.0.0.4.2).type = color = (0.0.0.3.2).type。因此，m’节点的集合是：{0.0.0.4.1, 0.0.0.4.1, 0.0.0.4.2, 0.0.0.4.5}。 这样，我们就可以建立一个近似查询结果r’：

r’= (0.0.0.4, {0.0.0.4.1, 0.0.0.4.1, 0.0.0.4.2, 0.0.0.4.5})

与r相比，关键字匹配节点color : 0.0.0.3.2变成了color : 0.0.0.4.2。节点color : 0.0.0.3.2包含关键字“red”，而color:0.0.0.4.2的内容是“white”。因此，用户查询中的关键字“red”会导致结果不匹配。通过将“red”更改为“white”，我们可以推断出建议的查询是 {'Vaio'，'W'，'white'，'price'}，r’ 是其对应的示例结果。同样的，我们也可以通过将“red”更改为“blue”或“pink”来查找建议的查询。

**示例8**：对于示例5中的查询Q = {'Hewlett'，'Packard'，'Pavilion'，'Omni'}，用户想要搜索同时销售笔记本电脑型号 'Pavilion' 和 'Omni' 的商店。 然而，没有这种同时出售这两种笔记本电脑型号的商店。一个查询结果是一个以electronics节点为根的子树：r = (0.0，{0.0.0.3.0,0.0.0.3.0,0.0.0.3.1,0.0.1.3.1})，其中只有三个不同的关键字匹配节点。所以我们计算关于这三个关键字匹配节点中的查询关键字的三个可区分性值：D({‘Hewlett’, ’Packard’}, brand) = 75.5%，D({‘Pavilion’}, model) = 100%，D({‘Omni’}, model) = 100%.

由于D({‘Pavilion’}, model) 和D({‘Omni’}, model) 都大于阈值τ（90％），因此它们都很重要。因此，我们将检查以下两条路径以找到近似结果：从electronics:0.0(vlca) 到 model:0.0.0.3.1 (mi)；从electronics:0.0(vlca) 到 model:0.0.1.3.1 (mi)。这里我们将以第一条路径，即electronics:0.0 / shop:0.0.0 / laptop:0.0.0.3 / model:0.0.0.3.1为例来说明如何检查路径。我们从示例5中知道TNT (r) = shop，因此我们检查以 shop 为根的子树：0.0.0。对于初始结果r中的每个关键字匹配节点mi，在以0.0.0为根的子树内，我们总能找到具有相同类型的节点mi’。例如，对于r中的关键字匹配节点0.0.1.3.1，我们可以找到具有相同节点类型的节点0.0.0.4.1：(0.0.0.4.1).type = model = (0.0.1.3.1).type。因此，m’节点的集合是：{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.0.4.1}。这样，我们就可以建立一个近似查询结果r’：

r’ = (0.0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.0.4.1})

与r相比，关键字匹配节点model:0.0.1.3.1变成了model:0.0.0.4.1。节点model:0.0.1.3.1 包含关键字“Omni”，而model:0.0.0.4.1的内容是“Vaio W”。因此，用户查询中的关键字“Omni”会导致结果不匹配。通过将“Omni”更改为“Vaio W”，我们可以推断出建议的查询是 {'Hewlett'，'Packard'，'Pavilion'，'Vaio'，'W'}，因为同时销售这两种型号的商店是存在的。r’ 是其对应的示例结果。

请注意，如果我们将阈值τ设置为一个非常低的值，比如说零，这意味着所有关键字都具有可接受的高可区分性，那么我们将检查包含至少一个关键字匹配节点的所有TNT节点。这可以涵盖所有可能性，但也会消耗更多时间。我们将在实验（第7节）中显示，这很可能不是必要的。

3.2.2第2阶段：基于多个关键字匹配节点

当重要的关键字来自于多个关键字匹配节点mi时，即，梳理来自多个关键字匹配节点的关键字可以实现高度的可区分性。为了计算可区分性，我们需要计算这些mi的最低共同祖先，并用v表示。与阶段1相比，这是唯一的区别。令**K**为被v子树包含的查询关键词。然后其余工作类似于阶段1，我们计算D(**K**, v.type)，如果它是可接受的高值，我们将检查从vlca到v的路径以找到近似结果。 有关我们的两阶段解决方案的详细信息，请参阅5.2中的算法1。

但是，它需要2n次计算才能获得n个关键字匹配节点的任何子集的所有可能的最低共同祖先。但我们发现属性2有助于我们在线性时间内完成它。

**属性2**：设M = {m1，m2，...，mn}是查询结果的不同关键字匹配节点的集合（如果i≠j 那么mi≠mj），按其Dewey标签排序。然后M的任何子集S的所有可能的最低共同祖先（LCA）都在集合 {lca(m1, m2), . . . ,lca(mi, mi+1), . . . ,lca(mn−1, mn)} 中，其中| S | ≥ 2。

**证明**（归纳法）：步骤1：当n = 2时，该属性显然成立。步骤2：我们假设当n = k-1时，Mk-1 = {m1，m2，...，mk-1} 的任何子集的所有LCA都在 {lca(m1, m2), lca(m2, m3), . . . ,lca(mk−2, mk−1)} 中。我们将证明对于k个节点的集合Mk = {m1，m2，...，mk-1，mk}，所有可能的LCA都在集合L = {lca(m1,m2),lca(m2,m3),...,lca(mk-1，mk)} 中。假设Dewey(mk-1) = a1.a2...aj.aj+1... 和Dewey(mk) = a1.a2...aj.a’j+1...，令m’ = lca(mk-1, mk)，则Dewey(m’)=a1.a2 ... aj。由于节点是按Dewey标签排序的，因此在Mk中不存在另一个节点mi，使得lca(mi, mk ) 是m’ 的后代；否则，Dewey(mi)的形式应为a1.a2...aj.a’j+1...且mi应出现在mk-1和mk之间，这显然是不可能的。因此，对于包含mk的任何子集，即{m1’, m2’,..., mk}，它们的LCA不可能是m’ 的后代。如果LCA节点等于m’，则它在L中；如果LCA节点是m’的祖先，我们可以得到以下结果，因为找到LCA等于找到一组节点的Dewey标签的最长公共前缀：lca({m1’, m2’, ..., mk}) = lca({m1’, m2’, ..., m’}) = lca({m1’, m2’, ..., mk-1})，根据假设LCA也在L中。此外，对于不包含mk的子集，根据假设，它们的LCA也将在L中。

对于具有失配问题的查询，通过属性2我们最多只需要进行n - 1次的LCA计算，就能找到所有可能的近似结果。我们将使用示例9来说明我们如何推断阶段2的建议查询。

**示例9**：假设用户想要找到一台品牌为Hewlett Packard的操作系统为windows vista的紫色笔记本电脑。她可能会尝试在图1中发出查询Q = {'Hewlett'，'Packard'，'purple'，'windows'，'vista'}。其中一个查询结果是r = (0.0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.3, 0.0.0.4.6,0.0.0.4.6})。通过定义6，我们知道TNT (r) = laptop，但结果是一个以shop节点为根的子树，因此它未命中目标。

假设Hewlett Packard只有两种紫色的型号。关键字匹配brand:0.0.0.3.0，color:

0.0.0.3.3和OS:0.0.0.4.6在第1阶段不具有高可区分性（90％）：D({'Hewlett'，'Packard'}, brand)) = 75.5％; D({'windows', 'vista'}, OS))) = 42.5%; D({'purple'}, color)) = 80.7%。现在在阶段2中，通过属性2，关键字匹配节点的所有可能的最低共同祖先是0.0.0.3和0.0.0。以0.0.0.3为例，我们会发现laptop:0.0.0.3子树包含的关键字具有很高的可区分性：

D({'Hewlett'，'Packard'，'purple'}, laptop) = 98.4%

请注意，上述三个关键字实际上来自两个关键字匹配节点，即brand:0.0.0.3.0和color:0.0.0.3.3。

然后类似于阶段1，我们将尝试在从r的vlca到laptop:0.0.0.3的路径中找到近似的查询结果，即（shop:0.0.0/laptop:0.0.0.3）。最后，我们找到以laptop:0.0.0.3为根的近似结果，并通过将关键字'windows vista'更改为'windows 7'来获得建议的查询。

3.3对建议的查询进行排名

在生成所有的建议查询之后，我们构建初步的排名模型，来使用以下因素判断建议查询的质量分数：

1. （在原始查询中）需要更改的关键字数量表示为cn。cn越大，得分越低。
2. 近似查询结果的根vlca’ 和初始查询结果的根vlca之间的距离表示为dt（dt等于他们的Dewey标签的长度差异）。dt越大，得分越高，因为更紧凑的子树是首选。
3. 需要更改的关键字的可区分性的总和表示为D。D越大，得分越低，因为我们不希望替换具有高可区分性的关键字。

总结上述排名因素，我们通过计算它们的乘积来获取排名得分：

score = 1 / e^cn × (1 − 1 / e^dt)× 1 / e^∑D （2）

3.4我们方法的特征摘要

总而言之，我们的失配探测器和建议器具有以下特征。第一点，它是可移植的：通过在定义查询结果的格式（定义4）中捕获现有搜索语义中的LCA共性，我们的方法可以和任何基于LCA的匹配语义一起使用（回顾第2.1.2节）；由于我们的方法是查询评估的后处理，因此它与所采用的结果检索方法正交。第二点，它是由结果驱动的：我们的方法接收初始查询的结果作为输入，回顾第3.2节，建议器从每个结果中找到重要的关键词（保留在建议查询中），以保证建议的实践质量。第三点，它是轻量级的：正如之后第4节中所讨论的那样，它占据了整个查询评估时间的很小一部分。因此，对于没有失配问题的用户查询，用户将不会因检测器花费的额外时间而烦恼很多。最后，建议附带原始用户查询的结果，因此对于满足原始查询当前结果的用户，他们可以简单地忽略我们的建议并浏览其原始查询的结果。

1. 在没有ID引用的XML数据上通过一种新的标签方案高效地处理失配问题

回顾定义8，为了检查TNT节点是不是近似查询结果，核心操作是验证P1和P2两个属性是否成立。检查P1是微不足道的，因此我们的目标是通过设计一个新的节点标记方案以及相应的逻辑运算来实现对P2的有效检查。

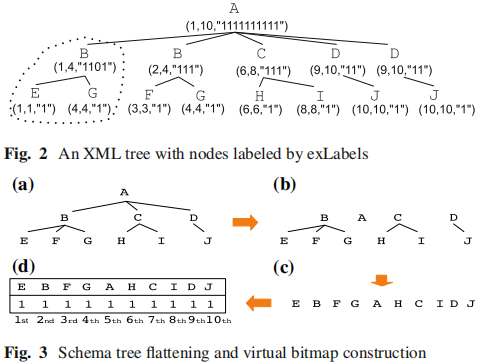
4.1节点标记

由于我们的建议器经常需要在查找建议查询的过程中访问节点类型，因此我们首先收集XML数据中的所有节点类型。**通过简单地扫描XML文件，我们可以使用DataGuide [11]获得包含所有节点类型的模式树**。例如，对于图2中的XML数据，我们可以构造如图3a所示的模式树，其中模式树中的每个节点表示唯一的节点类型。请注意，图3中的每个节点都应该是一个表示为路径的节点类型（根据定义1），但为了简单起见，因为没有歧义所以我们使用标记名称代替。

然后，我们使用位图来表示模式树中的所有节点类型，其中位图中的每个位都对应于一种特定类型。如下，我们有目的地决定哪个位对应于哪种类型：

1、以自上而下的方式逐级展平架构树。假设节点n具有k个子节点，则n将被插入其第⌊k / 2⌋个和第⌊k / 2⌋+ 1个子节点之间的位置。这样，n将保持在其邻居和邻居的孩子之间的位置。图3a-c显示了这种扁平化过程。

2、构造一个如图3d所示的虚拟位图。每个不同的节点类型在位图中都有唯一的位置编号。例如，F的位置编号是3。



这种位到类型的映射具有很好的属性：出现在XML中的特定子树中的所有节点类型的位将保持在一起。我们稍后会看到，此属性有助于确保标签尺寸尽可能的紧凑。

对于XML树中的节点n，以n为根的子树可以包含不同类型的节点。为了指示哪些节点类型出现在其子树STn中，我们为n分配了一个名为exLabel的标签label(a, b, bm)。这里，a是出现在STn内的节点类型的最小位置编号（在位图中）; 类似地，b是出现在STn内的节点类型的最大位置编号。bm是（模式树的）位图中从位置a到b的子序列，指示在以n为根的子树中可以找到哪种类型的节点。 具体来说，如果在位图中a + i - 1处的节点类型出现在以n为根的子树中，则bm[i] = 1；否则，bm[i] = 0 (i∈[1, b-a+1])。

**示例10**：在图2中，对于以虚线圈出的以节点B为根的子树，它包含E，B和G这3种类型的节点。根据图3d中的位图，E的位置编号为1，B为2，G为4。在从位置1到4的四种节点类型中，节点B的bm指示这四种节点类型中的哪一种出现在B的子树STB中。结果，bm = 1101，因为第三种节点类型F没有出现在STB中，并且B的exLabel = (1, 4, '1101')。注意，B的exLabel是紧凑的，因为表示E，B和G的位保持在一起，这是上述bit-to-type映射的益处。

4.2逻辑运算

与节点标记类似，对于查询结果r = (vlca, {m1, m2,..., mn})，我们可以特意构建一个exLabel来表示它的节点类型信息，即使它根本不是一个节点。设a’（b’）是mi的节点类型的最小（最大）位置编号，查询结果的标签表示为(a’, b’, bm’)。

有了查询结果标签 (a’, b’, bm’) 和子树根标签 (a, b, bm)，我们可以通过检查以下包含关系来验证属性P2：(a’, b’, bm’) ⊆ (a, b, bm)。仅当a ≤ a’ ≤ b’ ≤ b且出现在bm’中的所有位也出现在bm中时，此关系才成立。这可以通过bm’ 和bm上的逻辑AND运算来有效地完成。

**示例11**：在图2中，假设查询结果r = (vlca,{m1,m2})，其中m1.type = B，m2.type = G。那么r的exLabel是 (2,4,'101')。如果我们想检查一个近似查询结果是否存在于以图2中左边节点B为根的子树中，已知节点B的exLabel是 (1,4,'1101')，那么我们知道存在近似查询结果，因为 (2,4,'101') ⊆ (1,4,'1101')。

5 针对没有ID引用的XML数据的失配解决方案的索引构造和算法

5.1数据处理和索引构建

在XML文档解析阶段，我们收集所有不同的节点类型，并如4.1节中所述为每个节点类型生成位图代码。对于每个访问过的节点n，**我们将dewey标签deweyID[37]分配给n**；获得n的节点类型tn；为n构造一个exLabel标签。为了加快查询处理和细化，我们建立了三个索引。

第一个索引称为替换表，它是一个B+ 树，以(t, deweyID)为键存储每个节点。 这样的索引具有以下属性：通过向右扫描位置(t, deweyID)，我们可以在以deweyID为根的子树下找到所有类型为t的节点。回顾第3.2节，在我们找到一个近似的查询结果r’ 之后，我们需要具体化r’ 中的替换节点以推断出建议的查询。由于我们知道每个替换节点的类型t和r’ 的根节点的deweyID，使用替换表，我们可以通过调用getReplacement(t, deweyID) 轻松地找到所有这些节点。第二个索引是用来保存每个节点的exLabel和类型信息的。

为了加速可区分性的计算，特别是公式1中的参数ft**K**，我们构建了第三个索引倒排索引：对于（XML数据中）每个不同节点类型t和不同关键字k的组合，我们构建一个包含所有子树包含关键字k的t型节点的倒排列表。这些倒排列表按节点类型t分组。因此，**可以通过简单地计算节点类型t 下K中每个关键字的倒排列表的交集来计算ftK[23]**。算法getDist(deweyID, **K**) 返回关键词集合**K**关于具有deweyID的节点的类型的可区分性。

5.2算法

算法1中给出了主要步骤，其中输入是查询Q及其检索结果**R**。对于检测器，它检查Q的每个结果（第3行）并计算其TNT（第4行）。一旦其中一个结果命中目标，这意味着用户想要的是在检索到的结果中，它将终止该过程（第5行）。 否则，它会如第4.2节所述为查询结果构建一个exLabel（第8行）。

对于建议器，正如第3.2.1节中所述的第一阶段，它检查查询结果的每个关键字匹配节点nd（第10行）。如果可区分性大于阈值τ（第11行），则将检查从vlca到该节点的路径上的TNT节点，以便找到近似查询结果（第12行）。可以通过检查exLabel之间的包含关系来轻松检查近似查询结果是否存在（第13行），其中函数contains() 将在算法3中显示。如果存在近似查询结果，则将通过调用QuerySuggester() 推断出解释和建议的查询（第14行）。

对于第3.2.2节中所述的第二阶段，我们对关键字匹配节点进行排序（第16行），并根据属性2检查每两个相邻关键字匹配节点的LCA节点（第18行）。然后我们需要找到该LCA节点子树包含哪些查询关键字（第19行）。之后，我们完成与阶段1相同的步骤（第20-23行）。最后，它返回按照第3.3节中的排序公式排序的建议查询（第24行），附有每个建议查询的一个示例结果。

给定近似结果根和原始查询结果，算法2展示如何推断建议的查询。根据定义8中的属性P2，不在以vlca’ 为根的子树中的关键字匹配节点将被vlca’ 中具有相同的节点类型的节点替换（第2-4行）。对于需要更改的关键字匹配节点，可能有多个替换节点要替换它。可以通过调用函数getReplacement() 从索引中检索这些节点（第4行）。请注意，可能不止一个关键字匹配节点需要更改，因此将通过考虑所有可能的情况来推断建议的查询（第6行）。

算法3给出函数contains() 来检查两个exLabel之间的包含关系，即第一个包含第二个exLabel。正如第4.2节中所讨论的，关系成立的一个条件是第二个标签的范围应该包含在第一个标签中（第1-2行）。之后，我们需要确保第二个标签中出现的每一位都出现在第一个标签中。由于两者的位图长度可能不相同，我们将第一个位图缩小为与第二个位图相同的长度（第3行）。然后我们只需要在在两个位图上执行逻辑AND操作就可以完成位检查（第4行）。然后将相应地返回指示包含关系的布尔结果（第5行和第6行）。

7 实验

我们进行了大量实验，以验证我们的方法的有效性，效率和可扩展性。为了说明方便，我们将MisMatch 检测器和建议器称为MisMatch模块，并将我们的扩展过的MisMatch解决方案（使用ID参考的XML数据）称为扩展MisMatch模块。

7.1 实验设置

所有实验均在2.83GHz Core 2 Quad机器上进行，3GB RAM，32位windows 7操作系统。所有代码均以Java实现。Berkeley DB Java版[1] 用于存储算法的所有索引。

数据集：我们的MisMatch模块测试了三个真实的数据集：

1. IMDB4 90MB，我们的数据集中选择了近200年的近200,000部电影。 每部电影都包含标题，评级，导演等信息。
2. DBLP 520MB，其中包含自1990年以来的出版物。
3. INEX5的IEEE出版物90MB。

查询集合：我们的查询集包含针对这三个数据集中的每一个的18个查询，所有这些查询都是从我们系统的真实用户日志数据中收集的。 表1和表2中显示了针对IMDB和ACMDL的10个样本查询及其最佳3个建议查询（如果有）。此外，还为每个数据集生成了1000个随机查询（参见第7.5.2节），其中最大值 （平均）数量为2691（169）。

基本事实：对于每个数据集，我们雇用15名评估员来接收具有MisMatch问题的查询，并且他们的判断基于给出的查询和他们各自的结果。 如果15个评估员中至少有8个同意，我们通过判断查询来获得MisMatch问题，从而获得了基本事实。最终，IMDB（DBLP，IEEE，ACMDL）的18个查询中的9个（10个，10个，9个）存在MisMatch问题。

关键字搜索方法：对于没有ID引用的XML数据，我们选择SLCA [38]，这是迄今为止最有效的方法之一。 由于迄今为止没有提出基于SLCA的搜索方法具有结果排序组件，因此对于实验，我们采用了XRank的结果排序方案[12]。

7.2评估MisMatch问题

需要统计的数据：

1、**表1（XXX上的10个样本查询）**：对于某个 XML 数据库，统计 10 个查询对应的建议查询数量，排名前3的建议查询；

2、**表3（MisMatch探测器的灵敏度）**：对于每个 XML 数据库，统计两个查询集合A和B，A为评估员认为具有MisMatch问题的查询集，B是我们的检测器声称具有MisMatch问题的查询集；

3、**图6（前3个和前5个建议查询的平均分数）**：评估者对最佳3和最佳5建议查询的平均分数；

4、 **表4（不同阈值τ和不同排名因子情况下建议查询的分数）**：对于每个 XML 数据库，分别发起一个失配查询。分别统计不同排名因子情况下建议查询的分数：1、有所有的排名因子；2、没有cn；3、没有dt；4、没有∑D。分别统计不同阈值τ情况下建议查询的分数：1、0.9；2、0.6；3、0.3；4、0.0。（DBLP 差别较小）

5、**图8（查询处理时间）**：对于每个 XML 数据库（测三个），我们统计10个示例查询的SLCA查询、MisMatch检测器和MisMatch建议器的平均耗时。

6、**图9（数据大小对处理时间的影响）**：对于某个 XML 数据库（测两个），我们统计10个示例查询在不同数据大小情况下SLCA查询、MisMatch检测器和MisMatch建议器的平均耗时。

7、**图10（可区分性阈值τ对处理时间的影响）**：对于某个 XML 数据库（测一个），我们统计10个示例查询在不同可区分性阈值τ情况下生成的建议查询的平均数量和MisMatch模块的平均处理时间。

a: 建议查询数 - τ图; b: 时间 - τ图 或 时间 - 建议查询数图。

8、**图11（随机查询的可扩展性测试）**：对于某个 XML 数据库，随机生成100 个长度为2-5的具有MisMatch问题的查询，统计不同建议查询数区间的随机查询的占比，和MisMatch模块的平均处理时间。

7.2.1 MisMatch问题的频率（扯）

我们对15名人类评估员进行了调查。 每个评估员都需要在XClear [41]（一个XML关键字搜索引擎）中发出30个查询，以找到他们想要在IMDB数据集中捕获的电影。 要求每个评估员根据查询结果判断她的查询是否存在MisMatch问题。 在DBLP，IEEE数据集上也进行了相同的实验。我们发现平均用户在27％的查询中遇到了这样的问题。

7.2.2 MisMatch探测器的灵敏度

通过获得基本事实，我们研究了MisMatch探测器的精确度和召回率。设A是具有MisMatch问题的查询集。设B是我们的检测器声称具有MisMatch问题的查询集。然后精度= |A∩B | / | B |，同时召回= |A∩B | / | A |。 每个数据集的查询结果如**表3**所示。我们发现：

（1）我们的探测器实现了完美的召回率，即我们不会错过任何有MisMatch问题的查询。 这是因为在确定Q是否具有MisMatch问题之前，检测器会检查Q的所有结果（通过定义7）。

（2）非完美的精确度告诉我们可能会在没有MisMatch问题的情况下意外地识别某些查询。例如，对于在IMDB上发布的查询“Joel Ethan”，数据库中没有人具有这样的名称。 对于这样的查询，用户是否打算找到与两个人相关的电影，或者找到具有该名称的不存在的人是不明确的。 在这种情况下，我们的方法推断电影作为TNT，但有些用户可能认为是找到一个人，但名字错误输入。 请注意，实际上没有现有方法可以彻底解决模糊查询[2]。

（扯）此外，我们在MisMatch问题（定义7）的更一般设置中评估了我们的探测器的灵敏度，以便模拟不同用户的不同程度的攻击性，其中我们在其顶部K结果未命中时决定具有不匹配问题的查询目标（见第2.3节）。特别地，我们将K从1变为10.在精度和召回方面，结果与表3相同。原因如下。由于我们的探测器的输入是XRank [12]返回的前K个结果，其排序方案给出了更高分数以获得更紧凑的结果，并且那些非不匹配的结果比不匹配的结果更紧凑，非不匹配的结果将在排名列表中优先排序。因此，有两种情况需要考虑。情况1：如果查询Q具有至少一个不会错过目标的结果（通过定义6），则前1个结果将始终不会错过目标，因此通过定义7 Q将不会出现不匹配问题。情况2：如果查询Q的每个结果都错过了目标，那么我们当然可以推导出它的每个前K个结果也错过了目标（因为前K个结果是所有结果的子集）。

7.3有效性

我们首先看一下**表1**中实际查询的解释和建议是什么样的。对于Q8，'恐慌室'（'2001'）与类型标题（年）的节点相关联，但没有单个电影包含 所有关键字。 当然，一个建议是找到一个标题相同但年份不同的电影（例如，'2001'→'2002'），或找到同一年但不同标题的电影（例如，'恐慌室'→'承诺' 土地'）。请注意，我们不直接替换关键字，而是首先替换关键字匹配节点，然后将关键字派生为替换关键字（算法中）。 表1中括号内的术语表示涉及替换的节点的类型。 箭头的左侧是导致不匹配问题的关键字（解释部分）。Q3有3061条建议，因为Q3有大量结果，我们的建议器通过检查每个结果来生成建议（如果有的话）。

7.3.1评估方法（扯）

我们为每个数据集选择具有MisMatch问题的查询以进行用户研究。

为了进行公平的评估，我们意识到两件事。 首先，我们邀请专家和新手参与评分建议查询的任务。 对于DBLP，IEEE和ACMDL，我们要求三名CS研究生和三名本科生在其他院系; 对于IMDB，我们要求三位电影迷和三位非粉丝。 评估者显示每个查询的匹配结果，最佳5个建议查询以及相应的样本查询结果。 其次，要求评估员通过使用基于累积增益的评估（CG）度量[18]（从0到5点，5表示最佳而0表示最差）对每个建议查询的质量进行评分。 与采用二元判断（是或否）的传统度量（如精确度和召回）相比，CG意识到所有结果与用户不相关的事实。（评估员为室友、同学和朋友）

7.3.2整体质量评估

默认情况下，我们采用τ= 0.9。

最佳3和最佳5建议的平均分数如**图6**所示。我们可以发现对于具有MisMatch问题的查询，我们的方法能够为它们找到合理的建议查询，随后它会产生更有意义的结果; 最佳3个建议的得分总是高于最佳5个建议的得分，这也显示了我们的查询排名方案的效果。

虽然我们建议的查询可以带来更好的查询结果，但有些评估者仍然给出了一些低分数，因为新关键字和旧关键字在语义上并不相似，例如表1中的Q10替换。但是考虑到词汇语义超出了范围 这篇论文。

最有可能的是，最困难的3个建议的查询将由苦苦挣扎的用户查看。 因此，在本文的其余部分，当我们讨论建议查询的质量时，我们指的是最佳3个建议查询的平均分数。

7.3.3关于查询排名方案的研究

我们进一步研究了建议查询排名的排名因子如何影响建议查询的整体质量。 排名因子包括cn，dt和∑D，如3.3节中所述。每种情况的建议查询的分数如**表4**所示。请暂时忽略τ的选择。通过按列方式比较得分，我们发现：

（1）采用所有排名因子的模型总是优于任何错过三个排名因子之一的模型。

（2）在不考虑要替换的关键字（即D）的可区分性的情况下，建议的查询质量比没有任何其他两个因素的情况下降得更多。 它表明可区分性起着重要作用。

7.3.4关于可区分性阈值的研究

可区分性阈值τ的选择将决定在建议中保留哪些“重要”的关键词，从而可能导致建议查询Qs的不同候选者，这反过来可能影响QS的整体质量。因此，我们采用τ的4种选择，从强（0.9）到弱（0），如**表4**所示。

通过比较分数，我们可以看到即使我们设置较小的阈值τ，最佳的建议查询通常也不会改变。这是因为当我们设置一个像0.9这样的高τ时，我们就已经找到了最好的建议查询，正如第3节中所述，保留具有高可区分性的关键字更合理。稍后在第7.5.1节中我们还将研究τ对方法效率的影响。

7.4效率

对于表1中的每个查询，我们运行算法10次并收集热缓存的平均处理时间，如**图8**a所示。我们记录MisMatch模块使用的时间，我们从图8a得到三个观察结果：

（1）MisMatch模块仅占用整个查询处理时间的一小部分。平均而言，它大约为查询总耗时的4％。对于MisMatch模块花费少于1ms的查询，它太小而无法在图8a中显示。此外，平均而言，检测器花费约为建议器的1/40时间，因为它只需要检查结果的节点类型，如第4节所述。

（2）当生成更多建议查询时，MisMatch模块的处理时间相对较长。例如，正如我们在表1中看到的，Q3生成的建议查询多于其他查询，因此MisMatch Module会消耗更多时间。

（3）对于没有MisMatch问题的查询，与查询评估时间相比，MisMatch模块引入的时间可以忽略不计，因为一旦找到没有MisMatch问题的查询结果，它将终止。例如，对于打算由Warner Bros找到电影的Q4，由于存在这种类型的电影，Q4没有MisMatch问题，而我们的MisMatch模块只需0.05ms。

7.5可扩展性

回想一下，在得出查询存在MisMatch问题的结论之前，我们的检测器会检查查询的所有结果，对于每个查询结果，我们的建议器会尝试派生建议的查询。 因此，MisMatch模块的处理时间应取决于找到的建议查询的数量，而后者又取决于

- 要查询的XML数据的大小，以及

- 可区分性阈值τ的选择，和

- MisMatch Module处理的结果数量。

7.5.1示例查询

**图10（可区分性阈值τ的影响）**：表1中10个示例查询关于不同的可区分性阈值τ生成的建议查询的平均数量和MisMatch模块的平均处理时间。

首先，我们通过研究增加数据大小对MisMatch模块的影响来进行可扩展性测试。 我们在IMDB和DBLP上运行不同大小的查询。 由于BLINKS是一种内存方法，我们发现如果数据集大于45MB，它会在我们的机器中抛出内存不足错误.Acecentsurvey [7]也有类似的结论。因此，我们必须降级ACMDL的大小 此实验的数据集。 **图9**显示了数据集上一个查询的平均处理时间，其中我们有两个观察结果。

（1）MisMatch模块的处理时间关于数据大小线性增加，因为更大的数据大小可能会导致更多的结果，我们的MisMatch模块需要检查所有的结果以确定是否存在MisMatch问题并根据每个结果得到建议查询。

（2）查询的总处理时间也是随着数据大小的增加而增加，不论数据大小，MisMatch模块总是仅占整个查询处理时间的4％左右。

其次，我们研究了可区分性阈值τ对我们的MisMatch模块的处理时间的影响。**图10**显示了一个查询关于不同的可区分性阈值τ生成的建议查询的数量和MisMatch模块的处理时间。我们可以看到，τ越小，生成的建议查询就越多，MisMatch模块的处理时间也越长。因为阈值τ设置得更低，那么更多关键字将被视为具有可接受的可区分性，这样我们就可以检查更多的TNT节点，因此最终找出更多的建议查询。如第7.3节所述，最有可能的是，将τ设置为0.9就可以找到与τ为0.6,0.3甚至0.0时相同的最佳的建议查询。因此，我们将τ设置为0.9作为效率和有效性之间的平衡。总而言之，MisMatch 模块只占用了关键字查询处理时间的一小部分，同时可以为用户提供一些有用的建议查询，以解决可能的MisMatch问题。

7.5.2随机查询

除了真实的样本查询，我们还进一步研究了MisMatch模块对随机查询的性能。随机选择IMDB数据集中的关键字以形成长度为2-5的查询，并保留具有MisMatch问题的查询。我们记录前1000个这样的查询并计算我们的MisMatch模块输出的建议查询的数量。**图11**a显示了具有不同建议数范围的查询的分布，我们发现大多数失配查询的建议查询数量不大于500。图11b显示了MisMatch模块处理时间与随机查询的建议查询数之间的线性关系。

7.6总结

在本节中，我们已经在实验中验证了我们的后处理方法可以高精度地检测MisMatch问题并进行调用，并随后生成有用的建议以帮助用户。同时，它还具有可扩展性和轻量级，仅占整个查询处理时间的4％左右。