# SCISSOR：时间不断变化的层次结构中的可扩展且高效的可达性查询处理

**摘要：**

随着时间的推移，时间演变层次结构（TEH）由层次结构的多个快照（一个或多个树的集合）组成。在层次结构的任意（可能是过去的）快照中测试给定顶点对之间的可达性通常很重要。虽然基于区间的索引已成为静态层次结构中可达性测试的常用策略，但由于过高的索引开销，将此策略直接扩展到TEH是不切实际的。在本文中，我们提出了SCISSOR（具有渐进式解决方案细化的选择性快照索引），据我们所知，这是用于在TEH中回答可达性查询的第一个时间和空间有效框架。这里的主要思想是仅为TEH快照的选择性散布子集维护索引。将通过利用临时附近的索引快照的索引并分析两个快照之间发生的结构更改来回答对非索引快照的查询。我们还提供了一个实验研究，展示了SCISSOR框架在索引成本和查询延迟方面的可扩展性和效率。

**1、介绍**

树和树集合（以下统称为层次结构）长期以来用于表示层次结构，例如面向对象软件中的XML文档和子类 - 超类关系。 但是，在许多现代应用程序中，层次结构不是静态的 - 它们会随着时间而变化。考虑管理多个大型XML文档版本的XML存储。 每次将新版本提交到XML存储时，对应于XML文档的层次结构都会更改。类似地，软件版本控制系统必须处理可以从一个版本改变到下一个版本的类层次结构，尤其是在软件开发周期的早期阶段。 这种随时间变化的层次结构称为时间演化层次结构（TEH）。 TEH由层次结构的快照序列组成，随着时间的推移而发展。

在这些应用程序中，通常需要测试特定快照（可以是过去或现在的任何快照）上另一个顶点的给定顶点的可达性（或下降）。 例如，这种测试对于有效地评估特定版本的多版本XML文档上的XPath查询是必要的。 类似地，对于许多软件管理任务（例如，错误跟踪），有必要找出给定类是否是特定版本的软件中的另一类的传递子类。 试图找出顶点w是否可以从TEH的给定快照中的另一个顶点v到达的查询被称为快速特定的可达性查询（SS-reachabilityquery）。

先前已经在诸如XML查询处理[3]和面向对象（OO）软件管理的几种上下文中研究了回答静态层次结构的可达性查询的问题。 这些查询可以通过层次结构的按需遍历来回答（以广度优先或深度优先的方式），但对于大型层次结构，查询响应时间较差。 研究人员已经证明，在关系数据库上索引层次结构可以有效地缓解这个问题。 在为此目的提出的几种索引技术[1,4]中，基于区间的索引[3]是最受欢迎的索引之一。  
 Shu-Yao Chien等。 [7]提出了一种管理XML文档的方案，该方案可以在多个版本上发展。 但据我们所知，他们没有解决回答可达性查询的问题。 在本文中，我们提出了一个可扩展，时间和空间有效且可调的索引框架，称为选择性快照索引，具有渐进式解决方案细化（SCISSOR），用于回答TEH的任何特定快照上的可达性查询。 据我们所知，SCISSOR是第一个能够有效回答TEH任何给定快照的可达性查询的框架。 SCISSOR框架的一个关键设计特性是它不需要为每个快照编制索引。

1. **概述和背景**

正如介绍中所提到的，TEH由树集合的一系列快照组成，随着时间的推移而发展。 假设TEH的每个快照满足作为树集合的条件（即，任何快照中的任何顶点最多具有一个父节点）。 假设每个边缘从父顶点指向子顶点。 出于索引目的，树集合通常通过添加虚构的超级根来转换为单个树，该超级根成为集合的各个树的根的父级。 为了简化讨论并且不失一般性，我们假设在每个快照上执行这样的转换。

考虑一个TEH T.让{T 1，T 2，...，T q，...，T r}成为TEH的不同快照。 令EDList（T q，T q + 1）表示在快照T q和T q + 1之间发生的变化。 请注意，任何两个快照之间的EDList可以表示为一组顶点加法，一组顶点删除，一组边缘添加和一组边删除的并集，其中任何一个都可能为空。 我们假设任何两个快照之间的EDList不违反每个快照是树集合的属性。 换句话说，假设顶点v是快照T q中的顶点u的子节点，并且EDList（T q，T q + 1）包括边缘（w，v）的相加，则假设EDList（T q，T q） +1）还包括删除边（u，v）。 我们还假设每个EDList遵守基本一致性条件。 例如，如果EDList包含顶点v的删除，则v上的任何传入和传出边缘也在同一EDList中被删除。

SS可达性查询SSReach（v，w，q）试图找出顶点w是否可以从TEH的第q个快照中的顶点v到达。 如果w可以从T q中的v到达，则答案应为TRUE，否则为FALSE。

**2.1静态树中可达性分析的基于区间的索引**

对静态层次结构中的可达性查询的有效回答存在相当大的兴趣。 广度优先遍历，深度优先遍历和传递封闭是回答可达性查询的最早方法之一。 但是，它们不能很好地扩展。 近年来一直采用的另一种方法是在关系数据库上维护层次结构的某些索引信息。 已经提出了几种索引方案，例如基于区间的索引和HOPI索引，用于回答可达性查询[3,4]。 基于区间的索引是最受欢迎的方案之一，因为它简单，高效，并且能够在索引成本和查询时间之间提供良好的平衡。 一些研究人员提出了在一般有向图中扩展基于区间的可达性测试索引的方案[5,6,8]。

图1示出了基于区间的索引方案。 每个顶点左边的数字是它的预订值，而右边的数字是它的后序值。 有了这个索引，当且仅当w的预序值在v的前序和后序值之间（v pre <w pre <v post）时，可达性查询SSReach（v，w）才为真。 注意，E pre（= 5），B pre（= 1）和B post（= 8）满足条件，因此E可以从B到达。

**2.2 朴素的方法及其缺点**

**2.2.1索引所有快照（IAS）**

回答SS可达性查询的直接方法是使用基于区间的索引来索引TEH的每个快照，并使用第2.1节中描述的条件检查包含。然而，这种简单方法存在许多缺点。首先，索引每个快照的计算开销将非常高，因为它需要遍历每个快照。其次，由于需要保存每个快照的索引值，因此存储开销也会很高。随着层次结构的增加和更频繁的变化，计算和存储成本都会加剧。第三，在快照的某些部分上可能只有很少的查询，在这种情况下，每个版本的索引在存储和计算方面都是浪费的。但是，还应该注意的是查询分发是不知道的。第四，使用这种方法，使用可达性测试框架的应用程序几乎无法控制索引成本与查询效率之间的权衡。换句话说，应用程序无法调整系统以减少索引开销，即使它们可以容忍查询延迟的小幅增加。

**2.2.2无索引快照（INS）**

另一种直接的方法是不维护与任何快照相对应的索引。 查询将通过按需遍历层次结构的相应快照来回答。 此方法的问题是由层次结构的查询时间遍历引起的非常高的查询延迟。

理想的方法不仅可以平衡索引成本和查询延迟之间的权衡，而且在应用程序应该能够控制权衡的意义上也是可调的。

1. **SCISSOR方法 中的可达性查询处理**

在不失一般性的情况下，在SCISSOR框架中回答可达性查询的问题形式化如下。

假设快照q和（q + b）被索引（即，预订和后序索引可用于快照q和快照（q + b））。 还假设q和（q + b）之间的所有中间版本的编辑列表都是可用的。 现在的问题是回答SSReach（v，w，q + a），即，是否可以从快照q + a中的v到达w，其中0≤a≤b。如果a = 0或者a = b，则查询在索引快照上，并且可以通过检查v pre <w pre <v post是否在相应的索引中来回答。 另一方面，假设0 <a <b。回想一下，在这种情况下，我们的策略是首先测试时间上最接近的索引快照上的同一对顶点之间的可达性（即，我们将查询修改为SSReach（v，w，q）），然后分析 在版本q和（q + a）之间进行编辑，以检查v和w之间的可达性是否受这些编辑的影响。 通过这种分析，我们将能够回答SSReach（v，w，q + a）。在整个讨论过程中，我们使用图2中描述的TEH作为运行示例。 在此图中，仅在Snapshot-1个快照中建立索引。

问题是我们如何设计一种技术来有效地分析快照q和（q + a）之间发生的编辑的影响？分析编辑的一种简单方法是按照它们在编辑列表中出现的顺序处理版本q和（q + a）之间发生的所有编辑，并分析这些编辑对来自v的w的可达性的累积影响。 然而，由于两个原因，效率不高。 首先，对于大多数查询，很可能编辑列表中的大部分编辑完全不相关（例如，发生在层次结构的非常不同的部分），并且处理它们会增加不必要的开销。 其次，处理编辑需要加载树结构的相应部分并根据编辑进行更新。 因此，处理每个编辑都会产生高内存开销。

**3.1、观测和算法概述**

我们设计了一种技术，通过分析那些可能改变SSReach（v，w，q）的计算值的编辑来克服这些限制，因为层次结构从版本q演变为版本q + a。 问题是我们如何正确地找出哪些编辑会影响w的可达性？ 乍一看似乎无关的编辑实际上可能会影响可达性，因为其他按时间顺序编辑后续编辑。 例如，在图2中，处理查询SSReach（B，E，3）时，编辑Add（C，G）似乎与查询无关。 但是，此编辑与Add（G，E）一起改变层次结构的版本1和版本3之间从B到E的可达性。

我们做了两个重要的观察，这些观察将帮助我们准确地识别可能影响从v到w的可达性值的编辑，因为图形从快照q演变为（q + a）。

• 首先，假设SSReach（v，w，q）为TRUE。 只有通过删除w或w祖先上的边缘直到顶点v，才能改变这种可达性状态（w使得v无法到达）。

• 如果假设SSReach（v，w，q）的计算结果为FALSE，那么在版本q和q + a之间发生的任何可以改变可达性状态的编辑（即，使得可以从v到达）将包含至少一个编辑， 向w或其祖先之一添加一个内边缘，直到（并包括）w当前树的根。

实际上，这两个观察结果甚至适用于中间阶段（在编辑子集之后）。 我们通过称为影响列表的独特概念将这些观察结果纳入我们的算法。 节点对（v，w）的影响列表（表示为ImList（v，w））是动态变化的顶点列表，具有以下重要属性。 在算法的任何一点，如果任意顶点u没有出现在ImList（v，w）中，那么可以保证涉及顶点u的编辑不会影响算法中的可达性状态。ImList（v，w）在每次处理编辑时都会更新，因此是动态的。但是，ImList在算法的每个阶段都满足以下两个不变量。如果w当前可以从v到达，则ImList包含w和w的所有祖先，直到顶点v。如果w当前不能从v到达，则ImList包含w的所有祖先，直到w的树的根。 请注意，这两个不变量是我们前面提到的两个观察的直接含义。 在我们的算法中重复使用ImList来动态地从编辑列表中选择下一个编辑以进行处理。 ImList（v，w）是基于SSReach（v，w，q）的结果使用版本q的可用索引构造的，并且每次处理编辑时都更新它。 （如下面的算法描述中所述）。 考虑版本q和q + a之间存在编辑。 在最坏的情况下，需要处理所有编辑以回答查询，因此回答查询的复杂性随着编辑的数量线性增长。

**3.2可达性测试算法**

我们现在概述我们的算法，以有效地处理编辑。首先，我们解释三个重要的符号，我们将使用我们的算法描述。版本q + a的累积编辑列表（表示为CEDList（q + a））包含在快照q（在时间上最接近的索引快照）和快照（q + a）之间发生的所有编辑。 Reachability\_Status是一个变量，它在算法逐步处理各种编辑时保存v和w之间的可达性状态。在算法的任何一点上的任意顶点u的当前祖先列表（表示为CANList（u））表示u的祖先列表，它们在算法的那一点以它们到u的距离的递增顺序（即，考虑到已经处理的编辑的效果）。请注意，在我们的方案中，大多数顶点的CANLists不需要显式存储 - 可以使用快照q的索引来计算它们。仅当顶点的CANList由于迄今为止已处理的编辑而发生了变化时，才必须显式存储顶点的CANList。

算法1显示了我们算法的伪代码。在算法的初始化阶段（第1-2行），我们首先通过连接快照（q + 1）到（q + a）的编辑列表来计算CEDList（q + a）。还预处理CEDList（q + a）以消除彼此否定的任何编辑对。在第3-11行中，我们处理查询的目标节点w在快照q中不存在时的特殊情况。此节点可能已添加在快照q和（q + a）之间（即，CEDList（q + a）将包括Add（w））或w可能仍然不存在于快照（q + a）中（即， CEDList（q + a）将不包括Add（w））。如果在快照q和（q + a）之间添加了w，（第4-7行）Reachability\_Status被设置为FALSE，ImList（v，w）被初始化为仅包括w并且算法前进到迭代阶段。另一方面，如果CEDList（q + a）不包括Add（w），则算法立即终止，Reachability\_Statu设置为FALSE（第7-9行）。这是因为查询的目标节点甚至不存在于快照中（q + a）。在第12-19行中，我们处理了快照q中存在w的情况。在这种情况下，我们评估SSReach（v，w，q）并将其分配给Reachability\_Status。如果SSReach（v，w，q）为TRUE，那么ImList（v，w）被初始化为w，其后w的祖先以w与w的距离递增的顺序导致v（即w后跟其立即父母等等直到v）。如果SSReach（v，w，q）为FALSE，则ImList（v，w）被初始化为w，其后w的所有祖先以它们与w的距离的递增顺序。

然后算法进入迭代阶段（第20-42行），其中处理CEDList（q + a）中的编辑。在任何迭代中要处理的编辑的选择由Reachability\_Status的值和ImList的当前组成（v，w）确定。假设在第i次迭代开始时Reachability\_Status为TRUE，算法执行以下动作（第22-30行）。对于正在考虑的顶点，我们检查CEDList（q + a）以查看是否存在正在考虑的顶点的删除编辑作为要删除的边的目标。假设x是ImList（v，w）上具有Delete编辑的第一个顶点。我们通过执行以下操作来处理此编辑。首先，从ImList（v，w）中移除超出x（不包括x）的所有顶点。其次，对于顶点x的每个后代，如果尚未实现，我们实现其CANList。我们还更新x的每个后代的CANList以反映编辑（即，我们从每个CANList移除顶点x之外的顶点）。最后，将Reachability\_Status设置为FALSE，并从CEDList（q + a）中删除编辑本身。

如果在第i次迭代开始时Reachability\_Status为FALSE，则执行伪代码中的第31行到第40行。再次从左到右扫描ImList（即，以它们到w的距离的递增顺序考虑顶点）。但是现在，对于每个被考虑的顶点，我们检查CEDList（q + a）以查看是否存在添加编辑，其中正在考虑的顶点作为要添加的边的目标。假设y是ImList上的第一个这样的顶点。设添加的边是（z，y）。为了处理这个编辑，我们首先检查y是否在ImList中有父元素（即，y不是根）。如果是这样，必须有一个未处理的删除编辑，其中y作为目标（因为假定每个版本都是树的集合）。我们发现编辑，将其从CEDList（q + a）中删除，并从当前ImList中删除y右侧的所有顶点。接下来，我们检查z当前是否可以从v访问（这可以通过检查CANList（z），如果实现或通过对快照q的包含检查来完成）。如果z不能从v到达，我们将z和z的所有父项按照它们到z的距离的递增顺序附加到ImList。由于z本身无法从v访问，因此Reachability\_Status保持为FALSE。另一方面，如果可以从v访问z，则Reachability\_Status设置为TRUE。我们还将z和z的所有父对象以它们到z的距离的递增顺序附加到顶点v到ImList。 y的所有后代的CANLists被实现并更新以反映y的新祖先

如上所述，我们的算法在两个不同的条件下终止。 他们是

1. Reachability\_Status为FALSE，并且没有涉及ImList中顶点的Add编辑;

要么

1. Reachability\_StatusisTRUEandthereareno删除涉及ImList中顶点的编辑。

终止时，Reachability\_Status的值保存SSReach（v，w，q + a）的结果。 请注意，在算法期间，Reachability\_Status的值可能会翻转多次。

我们现在在图2所示的TEH上说明我们的算法，因为它通过快照1到3演变（只有快照-1被索引）。假设我们需要回答查询SSReach（B，E，3）。 Snapshot-1是最近的索引快照，因此它用作初始化点。 Reachability\_Status设置为TRUE，因为在对应于Snapshot-1的索引中B pre <E pre <B post。 CEDList（3）被初始化为{DEL（B，D），DEL（F，G），DEL（D，E），DEL（F，H），ADD（A，D），ADD（C，G），将ADD（G，E），ADD（D，H）}和ImList初始化为{E，D}。由于Reachability\_Status在第一次迭代开始时为TRUE，因此算法从左向右扫描ImList，以检查其中一个顶点上的内边缘是否被CEDList（3）中的DELETE条目移除。 E是具有DELETE编辑的最左侧顶点（DEL（D，E）是相应的编辑）。作为处理该编辑的结果，Reachability\_Status变为FALSE，ImList被截断为{E}并且DEL（D，E）被从CEDList（3）中移除。在第二次迭代中，算法从左向右扫描ImList，寻找CEDList（3）上的编辑添加入站边缘的顶点。该算法通过将ImList扩展为{E，G，F，A}并从CEDList（3）删除该条目来处理编辑ADD（G，E）。但请注意，Reachability\_Status仍然为FALSE，因为G尚未从B到达（编辑ADD（C，G）尚未处理）。因此，在迭代3中，我们继续在修改的ImList上寻找具有CEDList（3）上的相应ADD条目的更多顶点。现在找到编辑ADD（C，G）。然而，由于G已经具有父顶点，即F，因此条目DEL（F，G）应该存在于CEDList（3）中以便保持“树”属性。沿着ADD（C，G）发现并处理该editis。结果，ImList被修改为{E，G，C}并且Reachability\_Status被设置为TRUE（因为C可以从B到达）。条目ADD（C，G）和DEL（F，G）都从CEDList（3）中移除。算法进入第四次迭代，Reachability\_Status = TRUE，ImList = {E，G，C}和CEDList（3）= {DEL（B，D），DEL（F，H），ADD（A，D），ADD（ d，H）}。在该迭代中，算法从左向右扫描ImList，寻找在CEDList（3）上存在相应DELETE条目的顶点，该条目移除入站边缘。由于当前在CEDList（3）上没有这样的条目，因此算法终止，查询SSReach（B，E，3）的答案为TRUE（Reachability\_Status携带的最后一个值）。请注意，该算法仅处理原始CEDList（3）中的8个编辑中的4个（在算法开始时初始化）。这说明我们的算法不处理与查询明显无关的编辑，从而避免了不必要的开销。

1. **实验和结果**

我们现在讨论我们为评估SCISSOR框架而进行的实验。 我们已经用Java实现了SCISSOR框架。 实现以模块化方式完成，以便可以启用或禁用特定组件以进行评估。 我们将SCISSOR与第2节中讨论的另外两种方法进行比较，即索引所有快照（IAS）和索引无快照（INS）。 通过索引TEH的每个快照，可以在SCISSOR框架内实现IAS。 我们以两种方式实现了INS - 一种在查询的快照上使用广度优先遍历（BFT），而另一种使用深度优先遍历（DFT）。 这两种实现都是用Java进行的，它们对大多数实验产生了非常相似的结果。 除非另有说明，否则INS报告的结果对应于基于DFT的实现。

据我们所知，没有公开的TEH数据集。因此，我们不得不依赖合成数据集进行实验评估。然而，为了全面研究所提出的框架的行为，我们通过改变TEH的重要参数（例如，层次大小，高度，快照总数等）来生成许多数据集。生成器程序接受层次结构大小（就顶点数量而言），层次结构高度，快照数量以及快照之间的平均编辑数量作为输入参数。基于指定的层次结构大小和层次结构高度，我们生成一个树，每个非叶顶点具有大致相同数量的子节点，并将其指定为第一个快照。我们通过在先前快照上生成一定数量的编辑（由相应参数确定）来创建后续快照。在创建快照时，我们确保每个快照都是树或树集合。每个编辑生成如下。首先，我们需要决定编辑的类型。它可以是边添加编辑或边删除编辑。我们根据每个快照的边添加和边删除编辑的所需比率选择编辑类型。要生成添加编辑，将随机选择两个顶点，如果它们之间不存在边缘，则从第一个顶点添加边缘到第二个顶点，并删除第二个顶点上的任何入站边缘。要生成删除编辑，将从现有边集中随机选择边并删除。我们实验中使用的编辑列表具有大致相同的添加和删除编辑百分比。

查询工作负载以下列方式生成。 对于每个查询，源顶点，目标顶点和快照都是随机选择的。 但是，随机选择会导致很大一部分“无法访问”的查询（带有否定答案的查询）。 调整工作负载（通过删除一些随机选择的无法访问的查询）以具有大致相等的可访问和不可访问查询的百分比。

我们在两个主要性能方面评估SCISSOR，即索引开销和查询延迟。 我们使用摊销索引延迟作为量化索引开销的度量。 假设TEH具有n个快照，其中k被索引，并且索引所有k个的总时间是T个时间单位。 此业务情景的摊销索引成本为Tn。 为了更好地了解SCISSOR中的查询处理开销，我们测量了数据库查询引起的延迟以及编辑列表处理引起的延迟。 在每种情况下，我们都会报告所有查询的平均值。 我们还测量了我们的算法在回答特定查询时处理的编辑部分，以验证我们声称SCISSOR仅处理在查询和最近索引的快照之间发生的一小部分编辑。 摊销存储成本是在所有快照上存储索引和编辑列表所需的总磁盘空间与TEH（n）的快照数之比。

**选择性索引的好处**：在这组实验中，当TEH中的顶点数量从10K变化到500K时，我们量化了SCISSOR框架的优势。 所有层次结构的高度设置为10.此实验中每个TEH的快照总数为100.对于每个TEH，我们比较SCISSOR框架，IAS方案和INS方案的性能。 对于SCISSOR框架，我们通过执行10000个在所有快照中随机分布的查询来改变索引的快照的比例并测量系统的性能。

表1显示了INS，IAS和SCISSOR方案的索引和查询延迟。 INS具有最高的查询延迟，而IAS具有最高的索引延迟。 另一方面，SCISSOR（索引每100个版本）可以显着节省索引成本（500K节点TEH> 90％），与其他查询延迟相比，查询延迟略有增加（相同TEH <12％）方法。 在以下实验中，我们将展示SCISSOR如何平衡查询和索引延迟。

图3显示了SCISSOR和IAS方案的摊销索引成本，从10K到500K顶点变化顶点。 X轴表示在SCISSOR框架中索引的快照百分比的对数比例。 正如我们之前提到的，当索引快照的分数设置为1时，SCISSOR框架的行为与IAS方案完全相同。 正如预期的那样，IAS具有最高的摊销索引成本（每个快照大约62秒）。 对于SCISSOR框架，摊销的索引成本几乎可以通过索引快照的一小部分来降低。 此行为完全是由于需要编制索引的快照数量。 另一方面，INS的索引成本为零（图中未显示），因为它不会索引任何快照。

图4显示了同一实验的平均查询延迟。另一方面，IAS和SCISSOR的结果可能看起来很直观。正如所期望的那样，IAS具有最少的查询延迟值，因为它可以通过发布适当的数据库查询来获得答案，因为所有版本都已编制索引，并且由于需要处理的编辑，SCISSOR的延迟值显示边际增加。为了进一步阐明关于查询延迟的观点，图5中的堆叠直方图表示三种类型的延迟，首先是查询最近的索引快照所涉及的延迟，然后是在最近的索引快照和查询的快照之间获取编辑所涉及的延迟以及处理编辑所涉及的延迟。 X轴上的数据点（10K-1/100）表示数据集的大小（10K），后跟索引的快照的一小部分（每100个快照）。图5中值得注意的一点是（1）编辑列表处理延迟是平均查询延迟的一个非常小的贡献因素（2）编辑列表处理延迟随着索引快照的比例减少而增加。

由于SCISSOR仅为快照的子集编制索引，因此还可以节省大量磁盘空间。 由于需要为所有快照存储索引，因此IAS的分摊存储成本最高。 另一方面，对于SCISSOR，我们的实验表明，摊销存储成本与指数频率几乎呈线性变化。

1. **相关工作**

静态树的可达性查询的有效处理以及更一般地，对于静态图，一直是研究的活跃主题。 GRIPP [5]，DualLabeling [6]和GRAIL [8]是最近基于区间的方法，用于回答图中的可达性查询。 它们中的每一个以不同的方式扩充基本的基于区间的方法，以考虑非树边缘提供的附加连接。 DualLabeling使用传递链接表（TLT）增强基于区间的索引，而GRIPP使用可达性实例集来递归遍历由非树边组成的路径。 另一方面，GRAIL维护与多个生成树相对应的索引。 而Schekel等人。 讨论HOPI索引的增量维护[4]，它只能用于回答当前DAG上的查询，而不能回答以前的快照。 另一方面，SCISSOR可以回答任何快照上的可达性查询。

1. **结论**

对于许多现代应用程序来说，在时间推移层次结构中对可达性查询进行高效且可扩展的处理非常重要。 在本文中，我们提出了一个可调，时间和空间有效的框架，称为SCISSOR，用于测试TEH任何给定快照上给定顶点对之间的可达性。 SCISSOR背后的主要思想是有选择地索引TEH快照的子集，并使用这些快照索引来回应TEH的所有快照上的可达性查询**。**