Horae: A Graph Stream Summarization Structure for Efficient Temporal Range Query

Introduction

- 现有摘要结构无法存储图流中的时间信息，因此无法支持时间查询。设计一种支持图流上时间范围查询的方案极具挑战性。

- 一方面，直接在现有图流摘要结构中添加时间信息会导致时间范围查询延迟极高，无法对时间范围查询高效处理，现有结构（如 *TCM*）需对范围内的每个时间点独立执行查询，查询延迟随范围长度线性增长，实际应用中成本极高。另一方面，传统数据结构（如区间树 、线段树 ）无法处理图流上的时间范围查询。

- 本文提出 *Horae*，一种高效支持时间范围查询的新型图流摘要结构。*Horae* 的查询逻辑基于 **二进制范围分解算法（*BRD*）** 和 **多层存储结构**，核心是将任意时间范围查询高效分解为多个子窗口查询，确保对数级的查询延迟。

Horae Structure

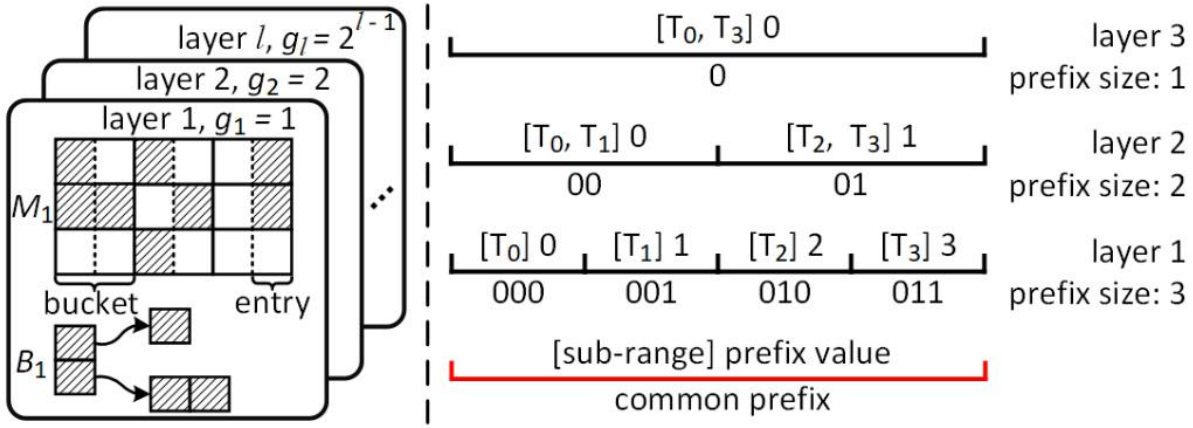
- Horae 结构包含$\lceil \log_2(T_u+1) \rceil + 1$ 层，它最初从单个层开始，每当图流的当前时间片增加到更大的 2 次幂时$(T_u = 2^i, i \geq 0)$

就会创建一个新层。 Horae 根据不同的前缀大小排列各层，每层通过对应的前缀大小聚合完整的图流数据。

形式上，第p层通过子区间 $\{[0, 2^{p-1}-1], [2^{p-1}, 2\cdot 2^{p-1}-1], ...\}$ 聚合图流数据，这些子区间具有相同的前缀大小l-p-1。第p层的所有子区间长度均为2^{p-1},我们将第p层中每个子区间的相同长度定义为该层的粒度。简而言之，Horae的第p层表示粒度为2^{p-1}的图流摘要。

每层中，Horae包含一个$m \times m$的矩阵M_p和一个邻接数组$\mathrm{B_p}$，以对应粒度对图流进行摘要。矩阵的每个桶包含b个条目，每个条目存储一对指纹和一个权重值，邻接数组采用VOV格式，便于结构扩展和数据访问。W_p^q表示第p层中前缀为q的窗口。

$$\mathrm{W}_p^q = [\mathrm{q} \times 2^{p-1}, (\mathrm{q}+1) \times 2^{p-1} - 1]$$



Operations of Horae

- Insert

Horae将输入元素(s,d,w,t)插入到所有层中，根据t得到该输入元素所属时间片，再根据所属时间片除以当前层粒度得到前缀，将s->d与前缀相连根据哈希函数得到哈希值，根据该哈希值的后F位得到指纹，根据哈希值的剩余位得到要在矩阵中存储的行列值，如果插入矩阵失败，则将该边的指纹和权重值存入邻接数组

- Extend

具体而言，当$\mathrm{r}(t) = 2^{l-1}$时（即当前时间片为 2 的幂次），Horae 触发扩展：在第$l+1$层构建新矩阵M_{l+1}和邻接数组B_{l+1}，并分别复制M_l和B_l的所有数据，以确保各层数据完整性。

Operations of Horae

- Temporal range query

本质上，Horae 可直接在每层中查询某个窗口的数据（简称 “窗口查询”）。我们进一步设计了 BRD 算法，将任意时间范围查询快速分解为多层窗口查询的组合。

对于任意时间范围，可能出现三种情况，需采用不同分解策略:

1）完全对齐

此时$\mathrm{L} = 2^{\mathrm{m}}$，$[T_b, T_e] = W_{m+1}^{\lfloor T_b/L \rfloor}$

2）半对齐

时间范围与第m+1层窗口的左边界与右边界对齐，分别称为左对齐与右对齐。此时，范围分解等价于将L分解为多个2的幂次之和

3）无对齐

时间范围与第m+1个窗口的左右界均不对齐，此时首先找到不大于Te的最大窗口左端点，将范围划分为[Tb,bp-1]和[bp,Te]，划分后的两个子范围分别右对齐和左对齐，可进一步根据半对齐的算法进行分解

Operations of Horae

- Window edge query

给定边s->d和窗口W_p^q，Horae将边与前缀q拼接，在第p层执行查询，计算该边对应的指纹对与行列地址，定位到特定桶，检查桶内所有条目是否存在匹配的指纹对，若匹配返回对应权重，否则在邻接数组中查找，存在则返回权重，不存在返回0

Window node query

给定源节点v和窗口W_p^q，Horae需检查第p层的矩阵和邻接数组，计算该节点对应的指纹和行地址，遍历对应行的所有桶条目，累加指纹匹配的权重，否则在邻接数组中查找该节点的哈希值，并累加对应边数组的权重

1.压缩版Horae

为降低内存开销，我们提出压缩版 Horae（Horae-cpt）。标准 Horae 的每层矩阵存储完整的图流数据且尺寸相同，虽能实现对数级查询时间，但空间效率有待提升。

压缩版 Horae 选择性存储部分数据：**第一层仍存储所有窗口数据，其余层仅保留前缀为奇数的窗口**，即每层（除第一层外）仅存储约半数图流数据。

我们将第p层（p>1）的矩阵大小设为第一层的一半，使内存成本降低 50%。

1）插入操作

2）查询操作

2.并行插入

3.多候选位置

4.查询的局部优化

5.逐出策略

Experiment

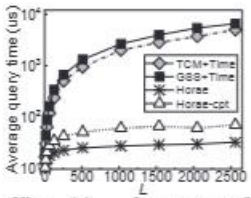


Fig. 11. Latency of temporal edge query on lkml

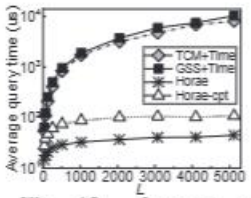


Fig. 12. Latency of temporal edge query on wiki

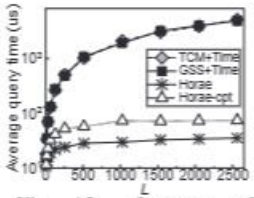


Fig. 13. Latency of temporal edge query on stackoverflow

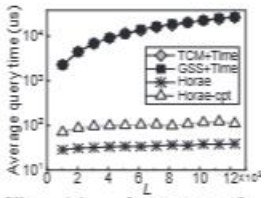


Fig. 14. Latency of temporal edge query on IP flow

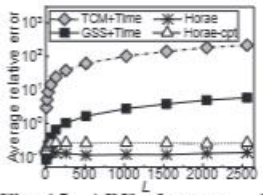


Fig. 15. ARE of temporal edge queries on lkml

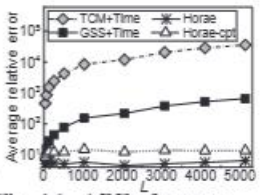


Fig. 16. ARE of temporal edge queries on wiki

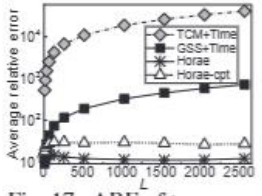


Fig. 17. ARE of temporal edge query on stackoverflow

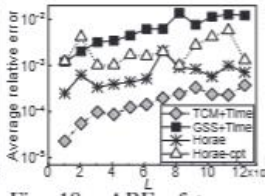


Fig. 18. ARE of temporal edge query on IP flow

时间边权重查询

Experiment

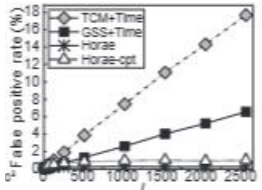


Fig. 19. FPR of existence query on lkml

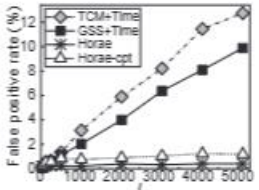


Fig. 20. FPR of existence query on wiki

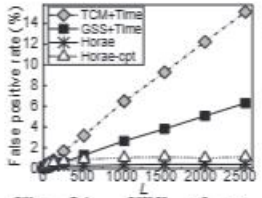


Fig. 21. FPR of existence query on stackoverflow

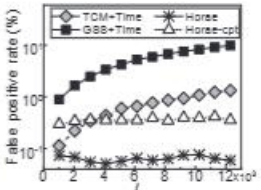


Fig. 22. FPR of existence query on IP flow

时间边存在性查询

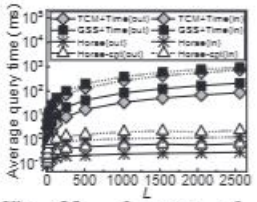


Fig. 23. Latency of temporal node queries on lkml

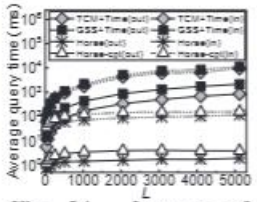


Fig. 24. Latency of temporal node queries on wiki

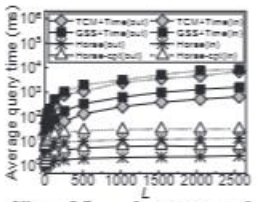


Fig. 25. Latency of temporal node queries on stackoverflow

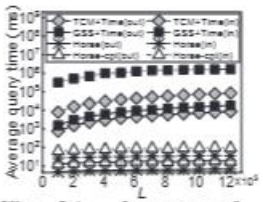


Fig. 26. Latency of temporal node queries on IP flow

时间节点查询

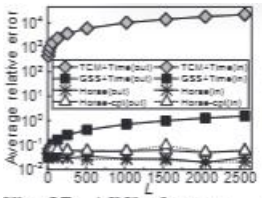


Fig. 27. ARE of temporal node queries on lkml

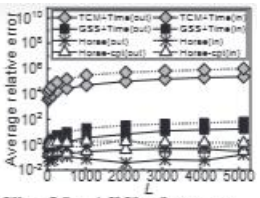


Fig. 28. ARE of temporal node queries on wiki

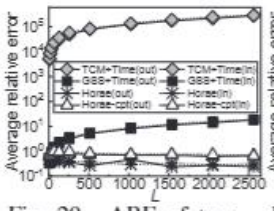


Fig. 29. ARE of temporal node queries on stackoverflow

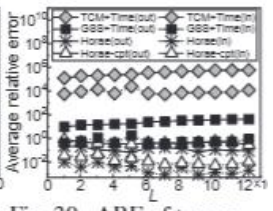


Fig. 30. ARE of temporal node queries on IP flow

- GSS：
- 三种查询原语：
- 1. 边查询：给定边e = {s,d}，若该边存在于图中则返回其权重w(e)，否则返回-1；
- 2. 1跳后继查询：给定节点v，返回从v出发一跳可达的节点集合，若无此类节点则返回{-1}
- 3. 1跳前驱查询：给定节点v，返回通过一跳可到达v的节点集合，若无此类节点则返回{-1}
- 通过这些原语，我们可以重构整个图：首先在哈希表中查找所有节点 ID，然后对每个节点执行 1 跳后继查询或 1 跳前驱查询以找到所有边，边的权重可通过边查询获取。由于图可被重构，所有类型的查询和算法均可支持。实际上，在许多情况下无需重构整个图，只需根据具体算法在需要时使用原语获取信息。

  （1）节点查询：计算某个节点的所有出边权重之和（或入边权重之和）
- （2）可达性查询：通过多次 1 跳查询迭代遍历图草图 Gh，模拟广度优先搜索（BFS）或深度优先搜索（DFS）

- TCM:
- 边查询：在多个草图对应的邻接矩阵中获取根据边节点哈希值定位的权重值，取其中的最小值
- 节点查询：例如出度聚合权重，在多个草图对应的邻接矩阵中根据节点的哈希值定位到某行并将该行的权重累加起来，取其中的最小值
- 路径查询：a是否可到达b.（1）在每个草图上调用现成的reach算法，判断映射节点hi(b)是否可由映射节点hi(a)到达（2）合并多个草图的结果，运用与运算，仅当所有草图上都可达，估计可达性才为true
- 子图查询：（1）在每个草图上调用现有算法subgraph(Q)查找子图匹配，并计算聚合权重（2）取各个草图上子图聚合权重的最小值。

HIGGS: HIerarchy-Guided Graph Stream Summarization

Xuan Zhao[1,3], Xike Xie[2,3], Christian S. Jensen[4]
[1]School of Computer Science, University of Science and Technology of China (USTC), China
[2]School of Biomedical Engineering, USTC, China
[3]Data Darkness Lab, MIRACLE Center, Suzhou Institute for Advanced Research, USTC, China
[4]Aalborg University, Aalborg, Denmark
zhaoxuan2118@mail.ustc.edu.cn, xkxie@ustc.edu.cn, csj@cs.aau.dk

introduction

- Horae的多层结构存在以下缺点：1.全局哈希冲突：将查询分解到不同层会导致每层的全局哈希冲突，从而降低查询精度。2.缺乏时间层次结构：缺乏用于组织时间信息的明确层次结构会影响插入和查询效率。3.不适应流的不规则性：平坦、均匀的网格状结构不适应图流的不规则性。
- 图流的不规则性源于两个因素。1）顶点度分布的偏斜：顶点度遵循幂律分布，这意味着少数高度顶点（头部顶点）连接到大量低度顶点，如图 2 所示。2）头部顶点的影响：头部顶点的变化会导致其众多入射边的显著改变，从而引起图流的大幅变化。
- 我们提出了 HIGGS，这是一种新颖的基于项的自下而上的层次数据结构，用于对包含时间信息的图流进行摘要。HIGGS 的功能类似于聚合 B 树，其中每个树节点对应一个特定的时间间隔，并集成了一个压缩矩阵，该矩阵表示其子树的摘要图数据。在底层，它自适应地将不规则流边分布到存储桶中，优化了空间利用率并降低了树的高度。与依赖全局结构信息的现有方法不同，HIGGS 利用其层次结构对存储和查询处理进行本地化，并将变化和哈希冲突限制在小且可管理的子树中。

Methodology

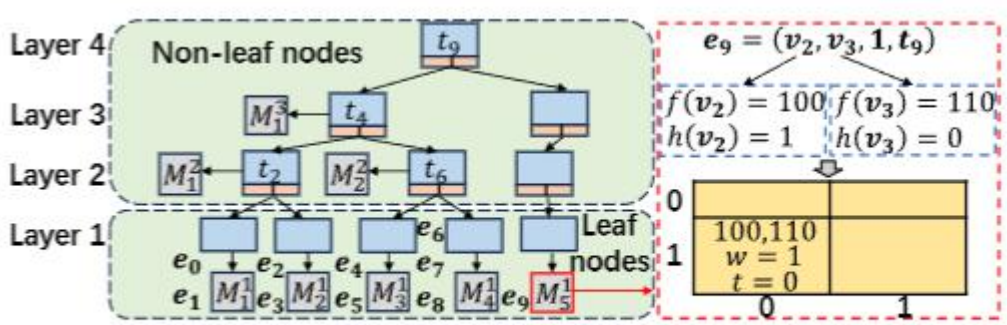


Fig. 6: Overview of the Architecture of HIGGS

•每个树节点对应一个时间间隔，最多包含 θ 个子节点，并附带一个压缩矩阵，用于汇总该时间间隔内的图流数据。

•一个具有 k 个子节点的非叶节点包含 k-1 个键（以时间戳作为分隔值），用于划分其子树的时间范围。所有叶节点均位于同一层（底层）。

•压缩矩阵中的每个桶包含一组条目，每个条目表示边ei的元组(f(si),f(di),ti,wi)

Operations of HIGGS

- 构建：HIGGS 的构建是基于项的自下而上过程，利用图流的到达顺序。当一条边到达时，它会以O(1)的时间插入到第一层的最新压缩矩阵中。如果矩阵已满，则创建一个新矩阵存储该边，并将其时间戳向上传播。如果根节点已满，则创建一个新根节点，将原根节点作为其子节点。填充节点（Filler nodes）确保与叶节点相连的矩阵保持在叶层，维持树的平衡。
- 插入：初始时，HIGGS 由单个根节点和一个空压缩矩阵 M 组成。当输入边ei=(si, di, wi, ti)到达时，首先使用哈希函数H(*)获取si和di的哈希值H(si)和H(di)。然后，取H(si)和H(di)的F1位后缀拼接成指纹对(f(si), f(di))，剩余位用于计算地址对(h(si), h(di))。根据计算出的地址，快速定位到桶M[h(si)][h(di)]，检查该桶中是否存在匹配的指纹对和时间戳。若存在，则累加权重；若不存在，则插入新条目。若所有条目均被占用且无匹配，则插入失败，创建新叶节点（链接相同大小的矩阵）重新插入，并将时间戳ti作为键传递给父节点用于查询。

Operations of HIGGS

- 聚合：父节点与子节点之间的聚合对提升查询精度和效率至关重要，采用自下而上的方式。叶节点：每个叶节点的压缩矩阵直接基于原始流项计算。非叶节点：上层节点的矩阵聚合其子节点的矩阵（涵盖子树内的所有后代节点）。

算法 2 描述了聚合过程的细节。对于第$i+1$层的节点，构建一个$\sqrt{\theta}d_i \times \sqrt{\theta}d_i$的矩阵，以聚合其第 i 层 θ 个子节点的$d_i \times d_i$矩阵。通过将指纹的高位移动到地址位，减少指纹存储开销，同时通过二进制移位操作实现数据聚合（矩阵大小为 2 的幂次，避免引入额外误差）。假设聚合过程中指纹存储减少 R 位，地址位增加 R 位，则矩阵大小变为原始的4^R倍（因此 θ 需为 4 的幂次）。第 i 层的指纹长度限制为$F_1 - (i-1)R$位，其中 θ 为节点的最大子节点数，d_i和F_i分别为第 i 层的矩阵大小和指纹长度。值得注意的是，聚合操作不涉及时间戳的存储。

Operations of HIGGS

- 对于任意指定的时间范围查询（包括边、顶点、路径和子图查询），可通过 HIGGS 上的边界搜索算法（算法 3）分解为一系列子范围查询。每个子范围查询在其对应的压缩矩阵上执行，从而将问题转化为跨矩阵查询。不同类型的查询原语（如边和顶点查询）对应不同的矩阵访问方式。边界搜索算法主要包括两个步骤：

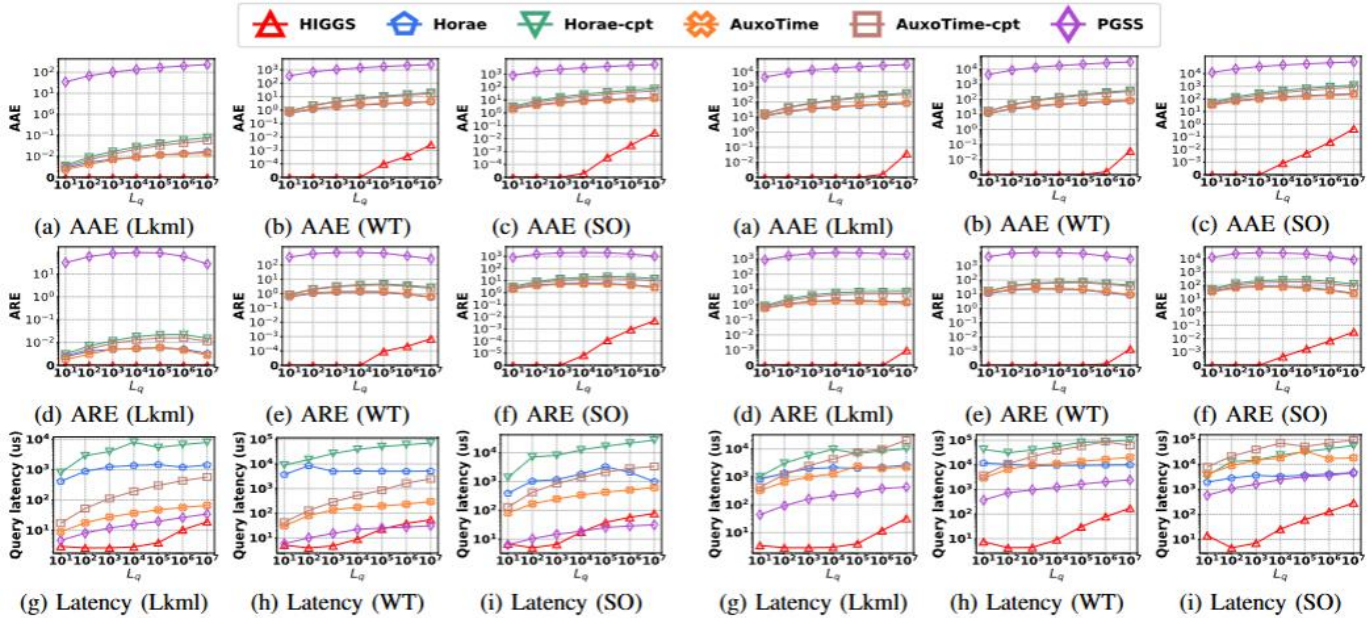
  范围覆盖识别：从 HIGGS 的根节点开始，识别完全包含在查询范围[ts, te]内的子节点，并将其加入查询列表 X。若[ts, te]落在某个子节点的范围内，则递归查询该子节点，直至ts和te不再位于同一子节点的范围内。

  边界节点识别：识别包含ts或te的子节点，并将包含在[ts, +infty]或[-infty, te]内的父节点的子节点加入 X。到达叶节点后，将包含ts或te的节点也加入 X，完成时间范围的分解。

  边查询：对于时间查询范围[ts, te]，对矩阵 M 执行边s->d的查询。利用公式 (1) 和移位操作推导指纹(f(s), f(d))和地址(h(s), h(d))，定位到桶M[h(s)][h(d)]。分两种情况：
  非叶节点：若 M 为非叶节点，检查是否有条目指纹匹配(f(s), f(d))，匹配则返回权重，否则返回 0
  叶节点：若 M 为叶节点，还需检查时间戳是否在[ts, te]内，指纹和时间戳均匹配时才视为有效条目

Optimization

- 多映射桶（MMB）：插入边 e 时，仅使用单个映射桶可能导致严重冲突，甚至因矩阵未充分利用而浪费空间。为此，采用多映射桶为每条边提供多个潜在插入位置。具体而言，使用线性同余法 为源顶点 s 生成地址序列{hi(s) | 1<= i <=r}，为目标顶点 d 生成地址序列{hj(d) | 1 <= j <= r}，其中 r 为顶点的映射位置数。源和目标的地址序列两两组合生成r * r个映射桶，并通过记录索引对(i, j)跟踪桶在地址序列中的位置。仅当所有映射桶均冲突时插入才会失败，显著降低失败概率并提高矩阵的空间利用率。
- 溢出块（OB）：在基础 HIGGS 框架中，若叶节点插入失败，则创建新叶节点存储边，并将时间戳作为键传递给父节点。但当多个同时间戳的边存在时，该键失效会导致错误。为此，引入溢出块将同时到达的边聚合到统一时间范围：若插入失败且边的时间戳与前一边相同，则在当前叶节点创建溢出块存储该边；否则创建新叶节点。溢出块本质上是小型压缩矩阵，可通过时间戳实现更精确的图流划分，从而提升查询精度。
- 并行化：为提升吞吐量，采用并行更新策略，为每层分配独立线程，仅更新最新节点。为维护子树的数据一致性，每个元素先由叶节点对应的线程更新，再由其他线程处理。由于仅需在元素级别保持顺序，并行效率较高，显著提高了 HIGGS 的吞吐量。



(a) AAE (Lkml) (b) AAE (WT) (c) AAE (SO)

(d) ARE (Lkml) (e) ARE (WT) (f) ARE (SO)

(g) Latency (Lkml) (h) Latency (WT) (i) Latency (SO)

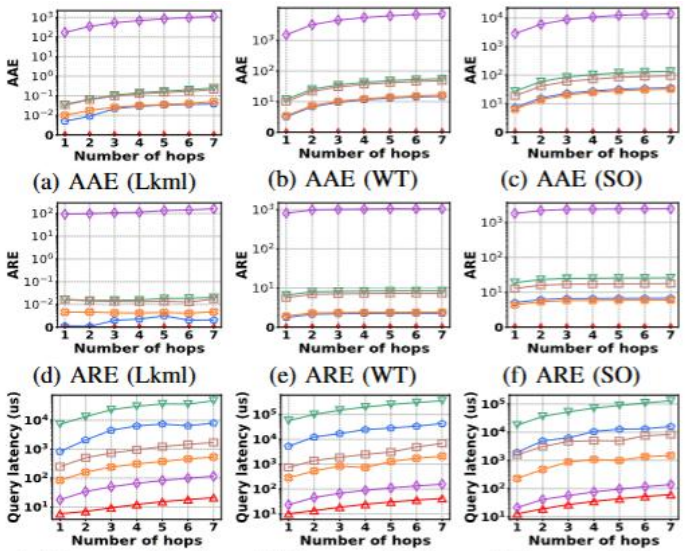
Fig. 10: Results on Edge Queries

(a) AAE (Lkml) (b) AAE (WT) (c) AAE (SO)

(d) ARE (Lkml) (e) ARE (WT) (f) ARE (SO)

(g) Latency (Lkml) (h) Latency (WT) (i) Latency (SO)

Fig. 11: Results on Vertex Queries

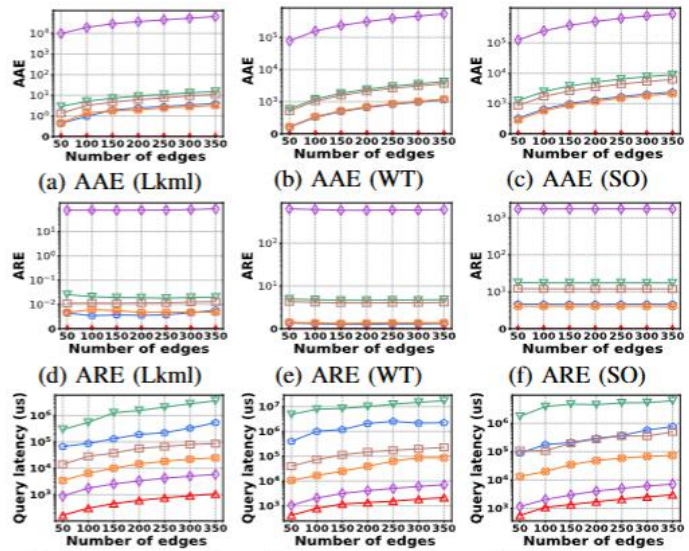


(a) AAE (Lkml) (b) AAE (WT) (c) AAE (SO)

(d) ARE (Lkml) (e) ARE (WT) (f) ARE (SO)

(g) Latency (Lkml) (h) Latency (WT) (i) Latency (SO)

Fig. 12: Results on Path Queries

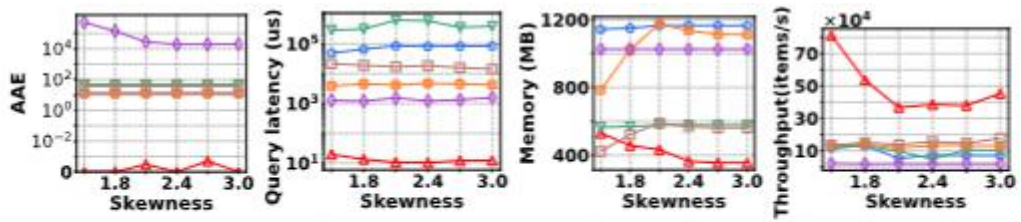


(a) AAE (Lkml) (b) AAE (WT) (c) AAE (SO)

(d) ARE (Lkml) (e) ARE (WT) (f) ARE (SO)

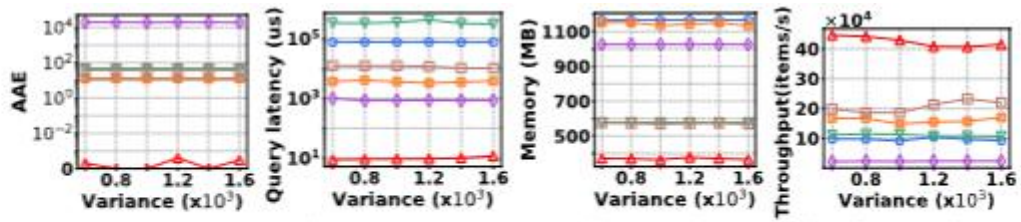
(g) Latency (Lkml) (h) Latency (WT) (i) Latency (SO)

Fig. 13: Results on Subgraph Queries



(a) AAE (b) Latency (c) Space Cost (d) Throughput

Fig. 14: Vertex Queries and Update Cost by Skewness



(a) AAE (b) Latency (c) Space Cost (d) Throughput

Fig. 15: Vertex Queries and Update Cost by Variance

- 为pgss增加指纹