1. 研究的导论

1.1. 基于选择的ε-相似Join

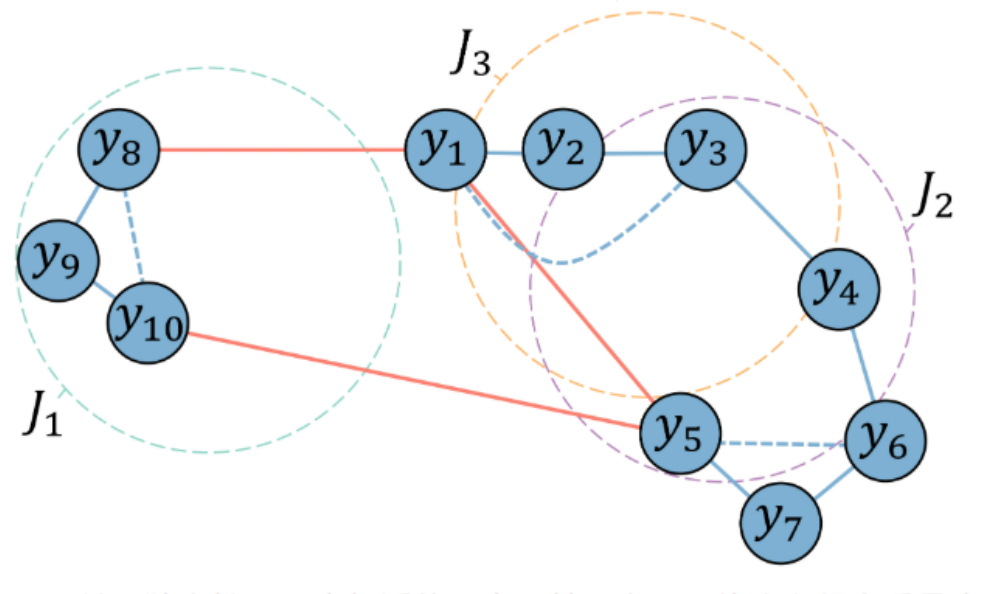
1 ε-相似Join的定义：对于查询集X及数据库Y

```
SELECT * FROM X JOIN Y ON distance(X.vector, Y.vector) <= ϵ
-- X和Y是两个向量数据库，其中X.vector和Y.vector存储的是嵌入向量
```

1. 逻辑视角：把X,Y各自中每点都算一下距离，把小于等于ε的向量对挑出来，即$X\bowtie_{\varepsilon}Y=\{(x_k,y_l)\in X\times Y\,|\,\delta(x_k,y_l)\leq\varepsilon\}$
2. 计算视角：为每个$x_i\in X$定义半径为ε的窗口$J_i=\{y_j\in Y\,|\,\delta(x_i,y_j)\leq\varepsilon\}$，则$X\bowtie_{\varepsilon}Y=\bigcup\limits_{x_i\in X}(\{x_i\}\times J_i)$($\times$表示配对)

2 关于基于选择的方法

1. 算法的流程：将每个x_i在Y的搜索都看作独立的过程
   - 预处理：在数据库Y上预构建一个k-NN图
   - 查询：遍历X中的每个x_i(遍历顺序任意)，通过在Y的k-NN图上贪心搜索，试图找到x_i在Y上所有的ε-邻居(即窗口J_i)
2. 存在的问题：如下图蓝线是2-NN图的边，x点未标出而仅画出其窗口



   - 关于独立性：X中相近的两点，其两窗口可能会有很多重叠点，但将每个x_i看作独立查询，就无法利用起这种重叠
   - 关于连通性：Y上k-NN的连通性很差，如上例假设$J_4=\{y_1,y_8\}$，在Y的2-NN图中永远不可能从y_1搜索到y_8(或反之)
   - 贪心的局限：会错误剪枝，如图$J_2=\{y_3,y_4,y_5\}$从y_4开始贪心搜索先到达y_6，然后就这条搜索路径被剪枝无法搜到y_5
3. 问题的解决：通过X两点窗口的滑动打破独立性，用邻接图替代k-NN图以增强连通，换用更全局更鲁棒的搜索策略

1.2. 本文的ε-相似Join概述

1 一些基本概念

1. Join滑动窗口：个半径为ε的球，以x_i为圆心时覆盖的Y中的点就是J_i
2. Join窗口顺序：即滑动的策略
   - 依赖关系：是一个MST(最小生成树)，可从父节点滑动到子节点；被存储为(κ_i,x_i)的配对集($\kappa_i\in X\cup\{\varnothing\}$为$x_i$父节点)
   - 处理顺序：通过遍历依赖树得到的线性的处理顺序(线性列表)，记作$\aleph$
3. Join的成本：$C\left(X\bowtie_{\epsilon}Y\right)=\sum\limits_{x_i\in X}\begin{cases}c_{\epsilon}\left(x_i\right) & (\varnothing,x_i)\in\aleph\\ c_{\kappa}\left(\kappa_i,x_i\right) & \text{otherwise}\end{cases}$
   - 起始成本：此时$\kappa_i=\varnothing$只能强制对x_i执行一次ε-范围搜索(同基于选择的方法)，成本记作$c_{\varepsilon}(x_i)$
   - 滑动成本：从父节点κ_i滑动到x_i的成本记作$c_{\kappa}(\kappa_i,x_i)$，每次滑动的成本都将被累加

2 本文研究概述

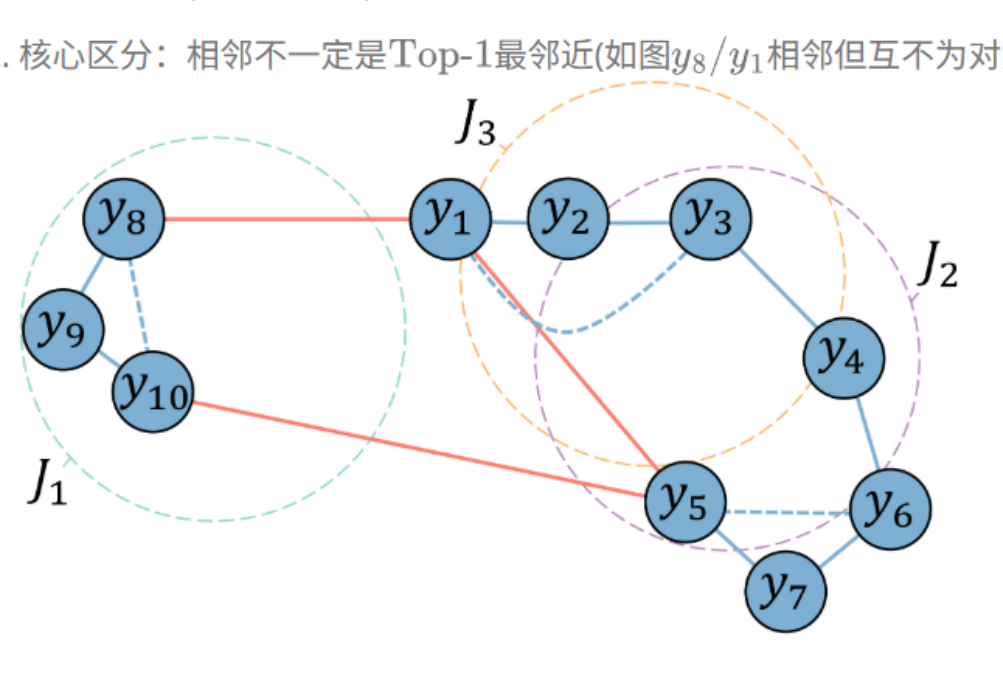
1. 核心贡献：包括两项技术，即如何用**Join窗口滑动**复用上一步计算结果，如何通过**Join窗口顺序选择**选择处理X的最优顺序
   - 窗口滑动：提出了邻接(Adjacent)图作为窗口滑动的理论基础，但在实践中用邻近图作为在高维空间中邻接图的替代
   - 最优顺序：通过预估滑动成本来最小化滑动的总耗时，给出"按什么顺序滑"的理论最优解
2. 可扩展性：将基于范围的ε-相似Join，扩展成基于Top-k的k-相似Join，并应用在图的动态维护上
3. 实验方面：相比近似算法速度提升一个数量级/结果更优，相比精确算法速度提示两个数量级/结果损失微乎其微

2. SimJoin算法: ε-相似Join

2.1. 第一步: 在Y上建立邻接图

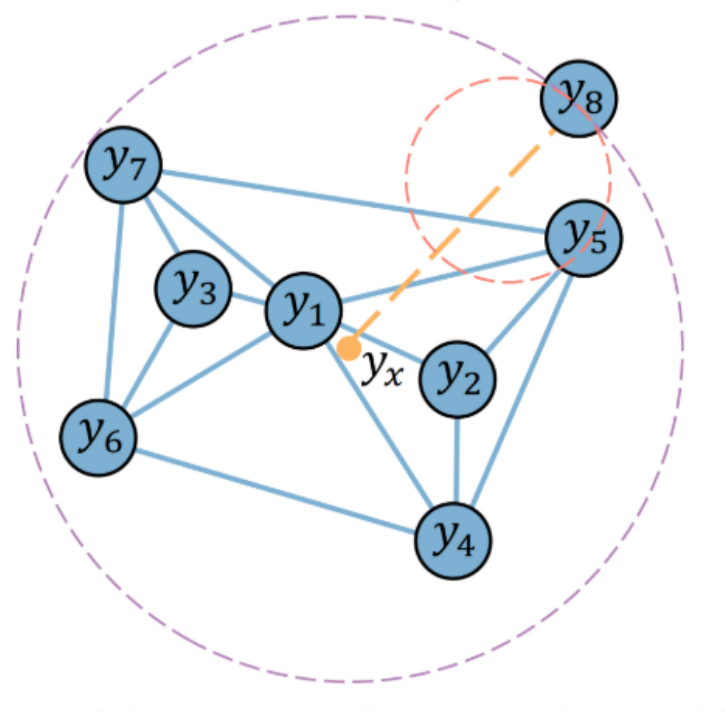
1 什么是相邻(Adjacent)点

1. 基本定义：(空心球法则)点A和点B是相邻的，**当且仅当**能找到一个空心球(球内无点)，它的球面可以穿过A和B
2. 核心区分：相邻不一定是Top-1最邻近(如图y_8/y_1相邻但互不为对方的Top-1邻近)，但是Top-1最邻近一定相邻



2 Join窗口的连续性

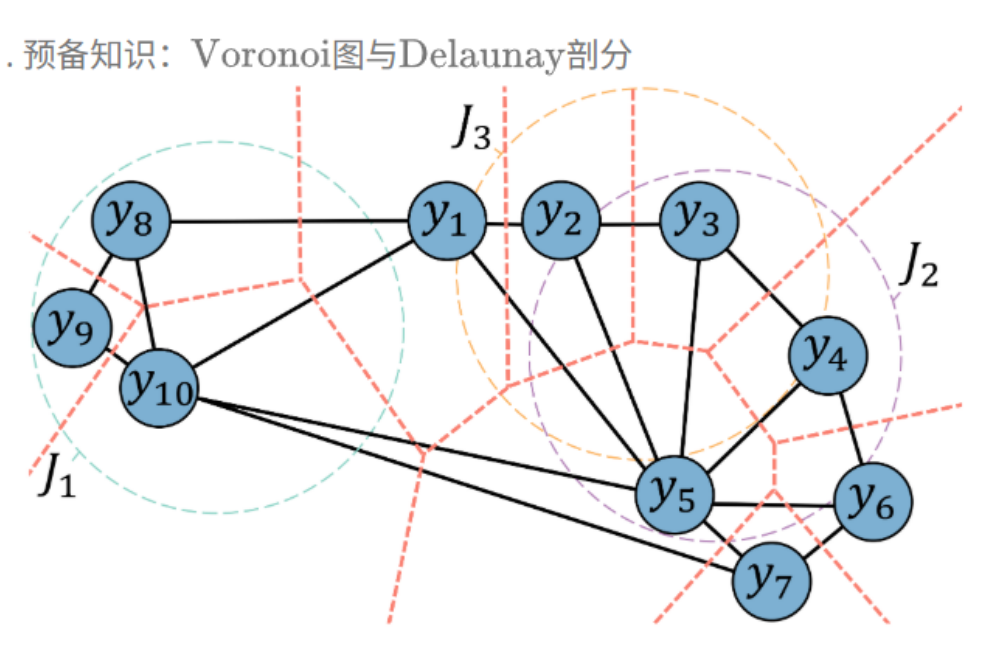
1. 连续窗口：考虑一滑动窗口J内的所有点，如两点相邻则连接二者(邻接图)；如果经过连接后所有点都连通，则窗口连续
2. 核心结论：画个半径固定的圆(如半径为ε的Join窗口J)，让圈内点若两点相邻就连接⇒连接后的图全连通(构成窗口连续)



   - 辅助引理：在y_x及其滑动窗口J中，令y_k为按与y_x距离从近到远排名第k个点，则至少存在一个$y_j(j<k)$使y_k和y_j相邻
   - 引理实例：图中y_8恒在小圆边上，将小圆从y_8向x滑动同时缩放半径，y_8必定能与其它更"内部"的点共圆(相邻)
   - 证明思路：可为最远点y_k找到离y_x更近的相邻点y_j，然后又为y_j找到更近的相邻点，以此类推....所有点都可找到相邻点
3. 一些补充：窗口剪枝规则与连续滑动
   - 窗口剪枝：在寻找窗口J中所有点时，如果$y_p\in J$但是y_p某一个相邻点$y_q\notin J$，则无需再从y_p出发探索y_q的任何相邻点
   - 连续滑动：对于窗口J_1和J_2如果$y_1\in J_1$和$y_2\in J_2$两点是相邻的，则从J_1滑动到J_2的过程叫做连续滑动

3 邻接图的构建方法

1. 预备知识：Voronoi图与Delaunay剖分

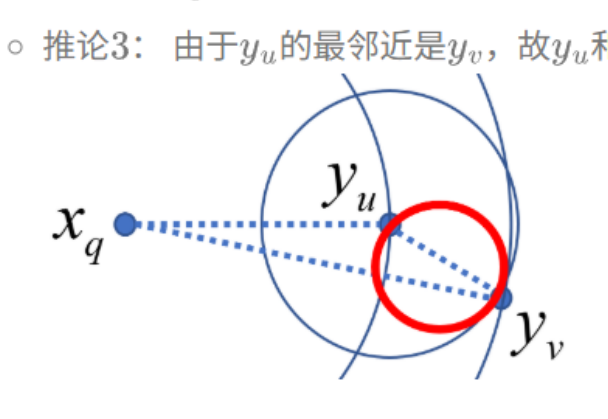


结构	含义	实例
Voronoi图	给定k个质心(每质心有一区域/即Voronoi单元)，使该区域的任意点到该区域质心最近	图中红线
Delaunay剖分	如果两点所在的Voronoi单元相邻，则直接连接这两点	图中黑线

2. 构建方法：邻接图=Delaunay三角剖分
   - 暴力遍历法：暴力遍历Y中每一对可能的点组合(y_p,y_q)，检查是否有空心球同时穿过它俩
   - 空间划分法：画出Y的Voronoi图，将Y中每点都分配在一个Voronoi单元，如果y_p和y_q的Voronoi单元接壤则两点相邻
   - 三角剖分法：所谓的Y中点的邻接图，就是这些点的Delaunay三角剖分

4 工程实现的折中

1. 存在的问题：当维度达到100及以上时，相邻定义的空心球法则极易被满足，使得几乎任意两点都相邻
   - 原因在于：高维空间中体积增长速度极快，使得数据点非常稀疏
   - 带来后果：邻接图几乎变成全连接图，使得邻接图不能提供任何有效信息
2. 解决方案：用Y上的k-NN图近似代替Y上的邻接图，注意主流数据库中Y的k-NN图在数据输入或插入时，就已经被构建好了
3. 理论保证：为何这种"偷梁换柱"是靠谱的，或者说两邻近点有多大可能是相邻的
   - 定理：若$x_q\in X$在Y中Top-1最邻近是y_u，考虑$y_v,y_w\in Y$，如果$\delta(y_u,y_v)<\delta(y_u,y_w)$则$y_v$有更高的概率离$x_q$更近
   - 推论1：既然离y_u越近就有更高的概率更靠近x_q，则y_u的最邻近y_v最有可能最靠近x_q，即最有潜力成为x_q的2-NN
   - 推论2：x_q的最近邻y_u(1-NN)的最近邻y_v，有极大概率是x_q的2-NN
   - 推论3：由于y_u的最邻近是y_v，故y_u和y_v必定相邻，而y_v又极有可能是x_q的2-NN，故x_q的1-NN和2-NN极有可能相邻



2.2. 第二步: 确定X上的遍历顺序

2.2.0. 设计思想(理论层面的理想情况)

1 如何衡量滑动成本

1. 要干什么：已处理了$\overrightarrow{X}=\{x_p,\ldots,x_i,\ldots,x_q\}\subseteq X$，下一步要处理$x_j\notin\overrightarrow{X}$，该如何选择$x_i\in\overrightarrow{X}$作为跳板去处理$x_j$成本最低
2. 理想情况：如果两个点x_i和x_j的窗口的重叠区域$|J_i\cap J_j|$越大，则从x_i到x_j的滑动成本越低
3. 实际困难：计算J_j之前不知道J_j的位置和大小(无法计算$|J_i\cap J_j|$)，但计算了J_j后$|J_i\cap J_j|$就没有价值和意义了，一个死锁
4. 折中方案：抛弃用$|J_i\cap J_j|$衡量窗口的大小，而是用两窗口中心点之间的距离$\delta(x_i,x_j)$来近似衡量窗口的滑动成本

2 理论上如何得到处理顺序

1. 最小生成树MST：找到全局成本最低
   - 顶点集：包含X所有点，此外还需要包括y_0作为所有滑动路径的总起点(以y_0为入口在Y上滑动到第一个x_i范围内)
   - 边连接：将顶点全部两两全相连，连接边的权重就是两顶点间完成滑动的成本，在实际中用二者距离$\delta(u,v)$近似
   - 树生成：在这个图上运行类似Prime或Kruskal算法生成最小生成树，其包含了所有顶点/无环路/边权重总和最小
2. 最终的处理顺序：遍历(类似深度优先或广度优先)最小生成树，得到的列表$\aleph$就是对X的处理顺序
   - (补充)并行化：MST中处理顺序为父节点→子节点，可用主线程处理根节点+子线程处理其每个子节点，以此下推

2.2.1. 得到X上简化的近似全连接图

1 存在的问题：在$X\cup\{y_0\}$上建立全连接图G_C成本巨大

1. 计算每条边的权值的复杂度是$O(dn^2)$，对大规模数据库是不可承受之重

2 第一次简化：从全连接图G_C到稀疏ε-邻居图G_{ε}

1. 思考：反正很多边在构建MST时都没用，干脆一开始就别构建；执行层面只连接长度$\leq\varepsilon$的边，即稀疏ε-邻居图G_{ε}
2. 做法：通过改变ε值使稀疏ε-邻居图G_{ε}全连通(理论上使$\varepsilon\geq$MST中最长边)，在稀疏ε-邻居图G_{ε}上找到的最小生成树
3. 保证：可严格证明，这种做法和在全连接图G_C上找到的MST是一样的

3 第二次简化：从稀疏ε-邻居图G_{ε}到k-NN图

1. 难题：稀疏ε-邻居图G_{ε}(只连接长度$\leq\varepsilon$的边)的构建，仍然相对困难
2. 思考：k-NN图天然只会保留较短的边，通过设置不同的k可以让k-NN图尽可能地变成ε-邻居图的近似
3. 做法：直接用k-NN图作为稀疏ε-邻居图G_{ε}的近似，即近似稀疏ε-邻居图G_{ε}

2.2.2. 在近似“全连接图”上得到MST

1 对Kruskal算法的一些思考

1. Kruskal算法的流程
   - 初始化：对图$G=\{V,E\}$需要找到其MST即$T=\{V,E'\}$，初始化E'为空，并将E中所有边按权值从小到大排序
   - 贪心迭代：后从最小权值边开始依次遍历E中所有的边，假设当前遍历到的边为(u,v)
     - 如果此时$u\leftrightarrow v$已经连通了，则不能连接u,v不然会在树中形成环路
     - 如果此时$u\leftrightarrow v$没有连通，则连接u,v构成边(u,v)并加入E'，作为最小生成树$T=\{V,E'\}$中的一条边
   - 终止条件：一直到选够了$|V|-1$条边，即覆盖了所有$|V|$个点为止，MST构建完毕
2. 如果此处强行用Kruskal...
   - 离线时：事先对X的近似全连接图中所有边排序
   - 在线时：查询到来X全连接图变成$X\cup\{y_0\}$的全连接图，y_0的引入会带来$|X|$条新边，还是得进行一次全局的排序
   - 问题：全局排序的成本太大了，占到整个Join的20%

2 改进的算法(离线时)：更高效地生成MST

0. 补充点：构建X近似全连接图本质是构建X上的k-NN图，构建后X中每点都有一邻居列表，其中邻居已按距离排好序
1. 初始化：让图中每一个点都成为一个独立的连通块，这些连通块将在后续不断被合并
2. 主循环：为现有每个连通块找到一条边，要求能连接该连通块与其它连通块，且在满足这一条件基础上权重最小
   - 外循环：遍历图中每个节点u(u所在连通块是$\mathbf{s}$)，数组$E_{\min}$的第$\mathbf{s}$位$E_{\min}[\mathbf{s}]$记录已找到的连通块$\mathbf{s}$的最短对外边
   - 内循环：在u的有序邻居列表中右移，跳过与u在同在$\mathbf{s}$的点，直到遇见第一个不在同一连通块的点v后，在此处停下
   - 更新：对比边(u,v)及边$E_{\min}[\mathbf{s}]$的权值，前者更小时更新$E_{\min}[\mathbf{s}]$(否则忽略)，然后自此不再在u邻居列表中右移
3. 合并块：遍历数组$E_{\min}$以检查每个连通块的最短对外边(u,v)
   - u,v在一个连通块：直接忽略，因为例如连通块s_1/s_2最短边是$v_i\to v_j/v_j\to v_i$，处理s_1时v_i/v_j已并到一连通块
   - u,v在不同连通块：连接u和v，由此u所属的连通块+v所属的连通块$\xrightarrow{\text{合并为}}$一块更大的连通块
4. 算法终止：不断重复步骤2→3以不断合并连通块，合并到最后只有一个连通块了，可以**严格证明**这个连通块就是MST

3 改进的算法(在线时)：将起始点y_0塞入离线构建的MST

1. 数据结构：构建两个列表
   - 旧列表：X上最小生成树的边的集合，已经按照权重从小到大排序好了
   - 新列表：将y_0与X所有点相连，按权重将新边升序排序，只保留短于X中最长边的新边(仅这些边可撼动MST结构)
2. 初始化：构建空的列表T，用于存放$X\cup\{y_0\}$顶点集上的边，预计最后T中存放的就是$X\cup\{y_0\}$上最小生成树的边
3. 主循环：从新旧两个列表中，挑出有用的边构成$X\cup\{y_0\}$上新的最小生成树
   - 处理顺序：从左到右遍历旧列表中每个边(p,q)
     - 正式处理(p,q)前，先从左到右遍历新列表中所有权重比(p,q)小的边，遍历到的同时一并处理这些边
     - 遍历结束后，才处理掉(p,q)
   - 处理方式：不论边在新还是旧列表，都检测该边的两端点是否已成通路，未成通路则连接二者(将该边加入T)
4. 算法终止：遍历完旧列表后算法旋即结束，可**严格证明**以上两步(离线+在线)后建立的就是$X\cup\{y_0\}$上的最小生成树

4 $\aleph$列表的生成：对最终得到的MST深度优先遍历，遍历结果就是$\aleph$，即以y_0为起点的X中的处理顺序

2.3. 第三步: Join窗口滑动的JoinSlide算法

1 第一种情况：从源窗口J_i滑动到目标窗口J_j时$J_i\cap J_j=\varnothing$，需要"长途奔袭"

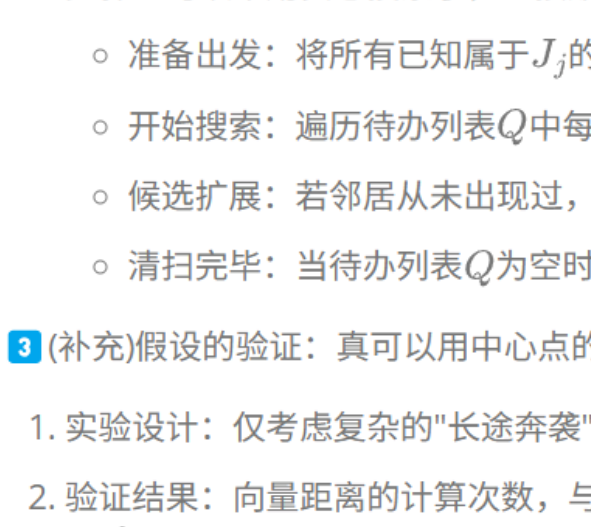
1. 目标：不求一蹴而就找到J_j中所有的内容，找到至少一个属于J_j的点
2. 步骤：在图上贪心地搜索
   - 准备出发：将源窗口J_i的所有点放入优先列表Q，并按照离目标窗口J_j的目标中心x_j的距离从小到大排序
   - 开始搜索：选取优先列表Q中最靠前(与x_j距离最小)的点作为**当前点**，探索**当前点**的所有邻居
   - 贪心决策：如果**当前点**的某个邻居比**当前点**更靠近x_j，则将这些邻居加入优先列表Q，对优先列表重新排序
   - 抵达目标：重复搜索+贪心的过程，一直到优先列表Q顶端出现了与x_j距离小于ε的点(J_j中的点)，"长途奔袭"结束
3. 补充：在需要进行"长途奔袭"时，算法隐含一个对比策略
   - 成本对比：用$\delta(x_i,x_j)$估计"长途奔袭"的成本，用$\delta(y_0,x_j)$估计“重开”的成本，选择成本较小者执行
   - 有何不同：如果“重开”则就不再用源窗口J_i的所有点初始化优先列表Q了，而是直接$Q=\{y_0\}$

2 第二种情况：从源窗口J_i滑动到目标窗口J_j时$J_i\cap J_j\neq\varnothing$，或者"长途奔袭"阶段结束后，需要"内部填充"

1. 目标：在已经找到了一个及以上J_j中的点时，从该已知点出发遍历整个图的这一块，以找到J_j中所有点
2. 保证：在一个窗口J_j中，所有的点都在邻接图中连通(k-NN图中也近似地连通)，所以从已知点出发必能找到J_j中所有点
3. 步骤：可以不用贪心搜索了，直接顺序遍历列表就行
   - 准备出发：将所有已知属于J_j的点放入待办列表Q，也不用做任何的排序了
   - 开始搜索：遍历待办列表Q中每个点作为**当前点**，探索**当前点**的所有邻居
   - 候选扩展：若邻居从未出现过，且其与x_j的距离小于ε，则加入最终结果集，以及加入待办列表Q(方便继续探索其邻居)
   - 清扫完毕：当待办列表Q为空时算法结束，最终结果集就是J_j的所有点

3 (补充)假设的验证：真可以用中心点的距离$\delta(x_i,x_j)$来估计J_i/J_j之间滑动的成本吗

1. 实验设计：仅考虑复杂的"长途奔袭"滑动过程，用向量距离的计算次数(搜索过程中最耗时的步骤)量化“实际”成本
2. 验证结果：向量距离的计算次数，与两窗口中心点距离$\delta(x_i,x_j)$呈强烈正相关



3. SimJoin算法扩展: k-相似Join

3.1. k-相似性Join概念

1 两种相似性Join：

类型	需要找到哪些点	剪枝难度	边界范围
ε-相似性Join $X\bowtie_{\varepsilon}Y$	每个$x_i\in X$在ε半径内圈住的Y中的点	低，需判断某点是否在x_i的ε半径外	固定为ε
k-相似性Join $X\bowtie_kY$	每个$x_i\in X$在Y中的Top-k最邻近	高，需判断某点是否在x_i的Top-k外	动态确定

2 k-相似性Join边界的特点

1. 边界的特点：由当前找到的x_i的Top-k点集确定，由于Top-k点集会随查询更新，故边界也动态变化但总体收缩
2. 带来的挑战：难以确定何时停止搜索，若某点在当前边界外，按理来说该放弃，但如果通过该点能导航到一离x_i更近的点呢

3.2. k-相似性Join算法

1 改进的点：仅需将SimJoin中的JoinSlide改成k-JoinSlide(如下)即可

1. JoinSlide的做法：定向的“长途奔袭”，采用贪心决策逐步靠近目标窗口，搜索到目标窗口的一个点时就终止
2. k-JoinSlide的做法：抛弃定向的搜索，而进行范围更广更彻底的搜索以探索所有可能
   - 初始化：将源窗口J_i的所有点放入优先列表Q，全部标记为未探索
   - 主循环：找出当前Q中距离x_j的最邻近的未访问点，将该最邻近点标记为已访问，并将其邻居全部加入Q
   - 终止：不断重复主循环，直到Q中的点全部被标为已访问，在当前的Q中找出x_j的Top-k，即为J_j中的点

2 一些补充点：k与ε的设置

1. 含义：k定义的是数量，ε定义的是范围，但是二者的对应关系与数据集疏密有关(不固定)，选择取决于需求
2. 选择：k的设置直接就是要返回的结果数量，ε的选择可以根据经验启发式地调整，也可以利用领域知识从特定数据集中学习

3.3. k-相似性Join应用

1 传统的索引的维护

1. 插入：为所有新点构建一套独立索引，暂不合并到原有主索引；检索时两套索引都需要检索，然后合并二者的结果
2. 删除：将被删点的出邻居及入邻居直接相连
3. 弊端：当数据变化(尤其是插入量太大)时，必须得从零开始将新旧数据点重新构建一套索引，成本巨大

2 SimJoin的解决方案

1. 插入：将新数据X和旧数据Y合并成一张新的k-NN图
   - 核心计算：将X中每一点与Y中Top-k相连，即完成一次$X\bowtie_kY$
   - 反向检查：对于上一步中被X连接到的Y中的点，尝试更新这些点的邻居列表，看看X中的点是否能成为其Top-k中的点
2. 删除：当某点y_p被删除时，只精确修复受影响的点
   - 受影响者：即y_p的每个入邻居y_x(即有向边$y_x\to y_p$)，因为删除y_p后每个y_x的邻居列表就残缺了
   - 恢复动作，让y_x在$Y\backslash\{y_x,y_p\}$上执行一次k-NN搜索，重新找到Top-k邻居(附带更新邻居列表+连接边)
   - 实现方式：基于k-JoinSlide算法，让y_x自己向自己滑动以自我修复
     - 初始化：将y_x(当下还残缺的)邻居列表中所有的点放入优先列表Q，全部标记为未探索
     - 主循环：找出当前Q中距离目标中心y_x的最邻近的未访问点，将该最邻近点标记为已访问，并将其邻居全部加入Q
     - 终止：不断重复主循环，直到Q中的点全部被标为已访问，再在当前的Q中找出y_x的Top-k

4. 实验及结果概述

1 实验设置

1. 数据集：选取了8个涵盖图像/音频/文本等场景的真实数据集+两种人造数据集
2. 测试方法：将每个数据集X随机一分为二X_1,X_2，再进行自我Join(即$X_1\bowtie X_1$)，以及普通的相似Join(即$X_1\bowtie X_2$)
3. 对比基线：VBASE(经典的基于选择的算法)/XJoin(SOTA的机器学习+基于选择)/FGF-Hilbert(SOTA的精确算法)
   - 补充：为公平起见是SimJoin和VBASE都采用NSG图索引，但注意FGF-Hilbert无需图索引，XJoin则基于LSH
4. 评测指标：用Recall衡量性能，此外还计量运行时间(但是除FGF-Hilbert外三个近似算法中，图构建耗时不计入运行时间)
5. 测试环境：每个测试都是单线程的，运行5次后取平均

2 核心结果

1. 模型性能：
   - 时间与召回：在所有的场景下SimJoin都能在更短的时间内达到比基线更高的召回
   - 计算成本：以向量距离计算次数作为指标，SimJoin计算成本远低于基线，说明了滑动窗口避免了海量冗余计算
2. 可扩展性：
   - 基数扩展性：当ε增大，即查询结果数量(基数)扩大时，相比基线SimJoin的运行时间增长很微小
   - 数据量扩展性：SimJoin的运行时间，与处理的数据的规模呈线性或者次线性增长
3. 组件性能：
   - 换掉NSG：比如将底层图索引结构换成Vamana/HNSW后SimJoin依旧稳定运行，但是HNSW对模型性能有下降
   - 测试MST：将本文的MST方法与其他方法对比，本文的方法耗时最短，而且在SimJoin整个过程中占比极低
4. 可迁移性：
   - 对k-相似Join：在该任务上，依然能用更短的时间达到同样甚至更高的召回率
   - 索引维护实验：用k-相似Join维护索引，速度吊打全局重建以及FreshDiskANN，且质量(用维护的图去检索的质量)更高