KO I OS：Top-k语义重叠集合搜索

Pranay Mundra罗切斯特大学
pmundra@ur.rochester.edu

Jianhao Zhang§*Acho S oft wareInc.*jianhao@acho.io

Fatemeh Nargesian罗切斯特大学
fnargesian@rochester.edu

Nikolaus Augsten萨尔茨堡大学
nikolaus.augsten@plus.ac.at

arXiv:2304.10572v1 [cs.DB] 20 Apr 2023

摘要——**我们研究了基于语义重叠度的 Top-k 集合相似性搜索问题。普通重叠要求集合元素之间完全匹配，而**语义重叠度 **允许语法不同但语义相关的元素匹配，从而提高重叠度。语义重叠度是二分图的最大匹配得分，其中两个集合元素之间的边权重由用户定义的相似度函数确定，例如嵌入之间的余弦相似度。令牌索引等常见技术无法用于语义搜索，因为相似元素在字符层面可能毫无关联。此外，验证候选项的开销很大（对于语义重叠度而言，其复杂度为立方级，而语法重叠度仅为线性级），因此需要高度选择性的过滤器。我们提出了**KOIOS**，这是首个用于语义重叠搜索的精确且高效的算法。**KOIOS **利用复杂的过滤器，最大限度地减少所需的图匹配计算次数。实验结果表明，对于中型到大型集合，只有不到 5% 的候选集合需要验证，而且其中超过一半的集合可以在无需执行开销高昂的图匹配计算的情况下进一步剪枝。我们在四个真实数据集上展示了该算法的效率，并证明与普通集合相似性搜索相比，语义集合相似性搜索能够提升结果质量。**

索引术语—**集合相似度、语义重叠、语义搜索、二分图匹配、语义连接**

I. 引言

集合相似性搜索是许多应用中的一项核心任务，例如数据清洗 [1]–[3], 数据集成 [4],[5], 文档搜索 [6], 和数据集发现 [7], [8]。两个集合的相似度通常通过普通重叠度（两个集合中相同元素的数量） [8], [9] 或对重叠度进行某种归一化 [7], [10]来评估。在开放世界词汇表和数据质量具有暂时性这一背景下，对于字符串集合而言，普通重叠度被证明是无效的，因为它仅考虑集合元素之间的精确匹配。为了解决这一问题，Fast-Join [11], [12]和 SilkMoth [13] 等模糊集合相似性搜索技术，将集合相似度与应用于字符串集合元素的基于字符的相似度函数相结合，例如编辑距离或基于元素词元的 Jaccard 相似度。两个集合的模糊重叠度是一个二分图的最大匹配得分，该图以集合元素作为节点，并以元素两两之间的字符相似度作为边权重。遗憾的是，模糊搜索只能处理集合元素中的拼写错误和较小的不相似性，而无法处理语义等价或语义相似但在字符层面毫无关联的元素。由于模糊集合搜索技术高度依赖元素词元之间的精确匹配，因此无法将其扩展到语义相似度度量。

§在罗切斯特大学期间完成的工作。

示例 1： 考虑查询集 Q 以及图 1 中的候选集合 $\mathcal{L} = \{C_1, C_2\}$ 。目标是在 $\mathcal{L}$ 中找到与 Q 最相似的前1个集合。(1) 普通重叠仅考虑集合元素 洛杉矶 的精确匹配，以评估集合之间的两两相似度。拼写错误（Blaine 与 Blain）、同义词（BigApple 和 纽约市）或元素之间的其他关系（例如，查尔斯顿 和 哥伦比亚 是南卡罗来纳州（SC）的两座城市这一事实）均被忽略。(2) 模糊相似度搜索允许在语法相似的集合元素之间进行匹配。通过基于三元组的 Jaccard 相似度，可以检测到 Blaine 与 Blain 之间的关系（图中展示了三元组及其相似度）。然而，BigApple 和 纽约市 并未对集合相似度产生贡献；相反，由于这些术语在字符层面的相似性，BigApple 被匹配到了 Appleton（威斯康星州的一座城市）。集合元素之间的其他关系也无法被检测到。因此，C_1 被排在前1，尽管 C_2 与查询更加相似：C_2 与 C_1 一样，在 洛杉矶 和 Blaine 上实现了匹配，但除此之外，还包含同义词和语义相关的元素。□

本文提出了语义重叠度以及 KOIOS 算法，该算法利用这一新颖度量解决 top-k 集合相似性搜索问题。语义重叠度是对普通重叠度和模糊重叠度的泛化，使得字符层面并不相关、但语义上相互关联的集合元素也能够对整体集合相似度做出贡献。给定一个查询集 Q 和一个候选集合C，我们构建一个加权二分图，其中节点是 Q 或 C 中的元素，而 Q 中的元素与 C 中的元素之间的边则根据它们的语义相似度赋予权重。最大二分图匹配会选择一个子图，使得任意两条边都不共享节点，并且子图中所有边的相似度之和最大化。遵循模糊集合搜索领域的相关研究 [11], [13],，集合元素之间的映射是一种可选的一对一映射。语义相似度通过用户指定的函数进行量化，例如元素嵌入之间的余弦相似度。

示例 2： 接续示例 1，我们使用语义重叠度，在 $\mathcal{L} = \{C_1, C_2\}$ 中针对 Q 执行前1搜索。图 1（右侧）展示了集合元素之间最低相似度为 0.7 的语义相似关系（虚线），以及使一对一匹配最大化的子图（实线），该子图定义了语义重叠度；匹配是可选的，因为并非所有元素都需要匹配。除了字符层面完全相同或相似的元素（Blain、Blaine）之外，语义相关的元素（例如，查尔斯顿 和 SC）也会对语义重叠度产生贡献；尽管 Appleton 在字符层面上】【。

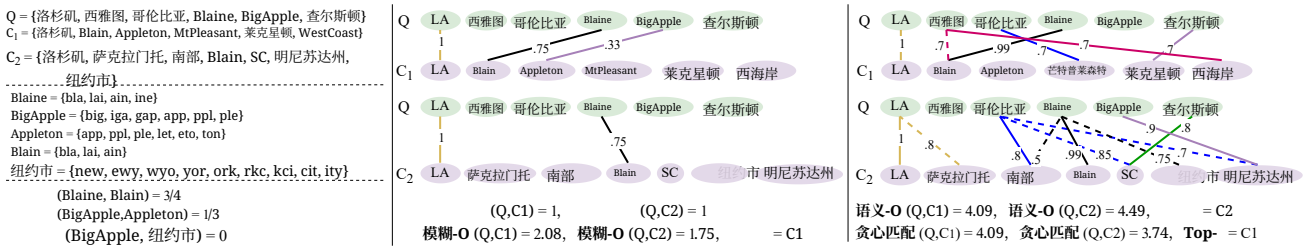


图 1：使用普通重叠、模糊重叠和语义重叠进行前1搜索。模糊重叠中的元素相似度采用三元组的 Jaccard 系数。

与 BigApple 的相似度，由于在语义上无关，因此不计入。C_2 如预期般排名前1。需要注意的是，按照权重降序匹配边的贪心匹配方法并非最优方法，且无法将 C_2 排在 C_1 之前。□

语义重叠适用于各种各样的任务。例如，在存在脏数据以及由不同标准、格式和组织生成的数据时，语义重叠搜索可以帮助查找可连接的数据集。普通重叠已在表连接搜索中得到广泛研究 [7][8], [10]。SEMA-JOIN 探讨了语义连接的概念 [14], ：给定两个可连接的列，目标是利用从大型表格语料库中获得的单元格值在行级和列级上的统计相关性，找到两个列中值的最优映射方式。除了发现可连接的列（这并非 SEMA-JOIN 的重点）之外，在没有大型单元格值映射数据语料库的情况下，KOIOS 还能够根据单元格值的语义相似度，找到其最优映射方式。

解决语义集合重叠搜索问题需要应对若干挑战。首先，计算语义重叠度需要对两个集合进行二分图匹配，这一过程开销较大，对于基数为 n [15] 的集合，需要花费 $\mathcal{O}(n^3)$ 时间。值得注意的是，尽管贪心匹配的复杂度较低，但它并不能始终获得最优的 top-k 结果。其次，存储库中集合的数量庞大，因此需要采用强力过滤器来排除不具备潜在价值的集合。KOIOS 通过一种新颖的过滤验证框架解决了这一问题。KOIOS 的精化阶段尽可能避免对集合进行开销较大的图匹配，并将精确匹配推迟到后处理阶段。KOIOS 为集合的语义重叠度定义了用于过滤的上下界，并以增量和迭代的方式不断精化这些界限。第三，由于实际存储库中的集合数量庞大且词汇表规模较大，过滤器需要频繁更新。为了降低这一开销，KOIOS 支持易于更新的过滤器，这些过滤器基于候选集的动态划分运行。最后，根据数据分布的不同，许多集合可能需要进行后处理。KOIOS 按照集合成为 top-k 结果的潜在可能性，对待后处理的集合进行排序。此外，它还根据已完成后处理的集合的历史记录，采用一种专门的过滤器来提前终止图匹配算法，从而进一步提高剪枝能力。至此。

总而言之，我们作出以下贡献。

- 我们提出一种名为语义重叠的新型集合相似度度量，通过考虑集合元素的语义相似度来推广普通重叠度量，而元素的语义相似度由用户定义的相似度函数量化。
- 我们基于语义重叠对 top-k 集合相似度搜索问题进行形式化，并提出一种名为 KOIOS 的新型过滤验证框架来解决该问题。
- 我们提出了高效且易于更新的过滤器，可在细化阶段和后处理阶段积极剪枝集合。
- 我们在真实数据集上对过滤器的剪枝能力、响应时间和内存占用进行了全面分析。实验结果表明，KOIOS 的内存占用较小，速度至少比不使用所提过滤器的基线快 5.5 倍，最高可快 740 倍。对于中型到大型查询集，KOIOS 表现尤为出色，能够剪枝超过 95% 的候选集。

II. 问题定义

我们假设集合中的元素两两可比较，即用户可以定义用于比较元素的相似度函数。

定义 *1*（语义重叠度）。给定两个元素集合Q 和 C 以及相似度阈值 $\alpha > 0$，假设 $M: Q \to C$ 是一种可选的一对一匹配，它为每个$q_i \in Q$ 确定与之匹配的 $M(q_i) \in C$ 或不匹配任何元素。设 *sim(., .)* 为对称相似度函数，对于相同元素返回 *1*，对于不同元素返回 $[0, 1]$ 中的一个值。我们定义 $sim_\alpha(x, y) = sim(x, y)$，当 $sim(x, y) \geq \alpha$ 时；否则为 *0*。Q 和 C 的语义集合重叠度为：

$$\mathcal{SO}(Q, C) = \max_{M} \sum_{q_i \in Q} \mathrm{sim}_\alpha(q_i, M(q_i))$$

当集合元素为字符串时，我们将集合元素称为词元。根据上下文明确的情况，我们使用 sim 表示 sim_α，使用 $\mathcal{SO}(C)$ 表示 $\mathcal{SO}(Q, C)$。语义重叠度由匹配关系 $M: Q \to C$ 定义，该匹配关系使 M 中各对元素的两两语义相似度聚合达到最大值。语义重叠度是一种对称度量。sim 的选择取决于具体上下文，例如，如果集合词元是字符串，那么其嵌入向量的余弦相似度 是比较元素的一种常见方法。其他纯粹基于字符的函数包括词元 [13] 中单词的 Jaccard 相似度，以及

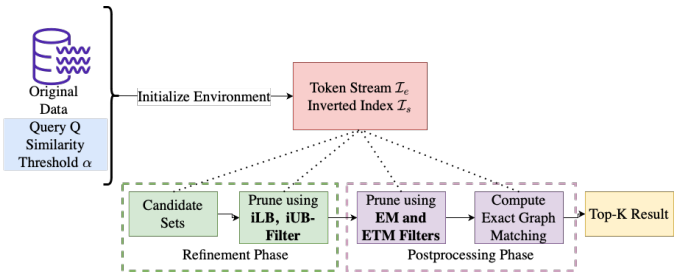


图 2：KOIOS 框架。

词元 [13], [16] 的编辑距离。普通重叠是语义重叠的一种特殊情况，其中 sim 通过评估元素是否相等来计算：$\text{sim}(q_i, c_j) = 1$ 当 $q_i = c_j$ 时成立，而 $\text{sim}(q_i, c_j) = 0$，否则。

引理 1. 普通重叠是语义重叠的下界：$|Q \cap C| \leq \mathcal{SO}(Q, C)$。

证明。 $|Q \cap C| > \mathcal{SO}(Q, C)$ 不可能，因为我们总能构造一个匹配 M ： $Q \to C$ ，将 $Q \cap C$ 中的所有元素映射到与之相同的元素。该映射的相似度总和为 $|Q \cap C|$。□

两个集合的语义重叠可以表述为一个最大二分图匹配问题。对于集合 C 和Q，我们定义一个加权二分图 $G = (V, E)$， $V = \{Q, C\}$，其中 C 和 Q 是 G中互不相交且相互独立的节点集合，并且边 $(q_i, c_j) \in E \subseteq Q \times C$ 表示 $\text{sim}(q_i, c_j) \neq 0$。最大二分图匹配的边权重之和称为该匹配的得分。寻找最大权重二分图匹配称为指派问题，其时间复杂度为$\mathcal{O}(n^3)$，其中 n 是集合 [15]的基数。

定义 **2** (***Top-k*** 语义重叠搜索)。给定一个查询集 Q 和一个集合族 $\mathcal{L}$, $k \leq |\{C \in \mathcal{L} \mid \mathcal{SO}(Q, C) > 0\}|$，从中找出一个包含 k 个不同集合的子集合族 $\omega \subseteq \mathcal{L}$，使得：

1) $\mathcal{SO}(Q, C) > 0 \ \forall C \in \omega$, *and*
2) $\min\{\mathcal{SO}(Q, C),\ C \in \omega\} \geq \mathcal{SO}(Q, X) \ \forall X \in \mathcal{L} \setminus \omega$

平局通过任意方式打破，使结果的大小为 k [17]。

我们的解决方案是一种过滤器验证算法，称为 KOIOS。在细化阶段，KOIOS 使用易于应用的过滤器识别候选集，并积极剪枝不具有潜在可能性的集合。在后处理阶段，KOIOS 通过仅部分计算精确匹配，进一步提升剪枝效果。图 2 展示了 KOIOS 的基本框架。

III. 基本过滤

我们将按照语义重叠度降序排列的结果集列表称为*top-k* 结果列表，并将 θ_k^* 表示为 top-k 结果中语义重叠度排名第 k 小的结果集的语义重叠度。当然，只有在计算出 top-k 结果后，才能确定 θ_k^* 的值。假设有一种算法，按照 $\mathcal{L}$ 中的顺序遍历各个集合，并维护一个当前 *top-k* 列表，其中包含目前为止发现的最佳集合：当前 top-k 可以包含 $\mathcal{L}$ 中任意一个语义重叠度非零的 k 子集。

重叠度。我们将当前 top-k 列表中集合的最小语义重叠记为 θ_k；根据定义， $\theta_k \leq \theta_k^*$。

根据定义 1，任何包含至少一个元素的集合 $C \in \mathcal{L}$，只要该元素与 Q 中某个元素的相似度高于 α，就具有非零的语义重叠度，因此属于候选集合。显然，如果 $\mathcal{SO}(C) < \theta_k$，则可以剪枝集合 C，因为至少存在 k 个具有更优 $\mathcal{SO}$ 得分的集合。然而，由于计算 $\mathcal{SO}(Q, C)$ 的开销很大（其复杂度关于集合基数呈立方级增长），因此我们在细化阶段的目标是在不计算精确匹配的情况下进行剪枝。为此，我们计算候选集语义重叠度的下界和上界。通过将这些界限与当前 θ_k 进行比较，可以帮助剪枝集合，从而减少精确图匹配计算的次数。

UB过滤器：我们定义了语义重叠度的一个上界。

引理2。 给定查询集 Q 和候选集合 C，则$\mathcal{SO}(C) \leq |Q| \cdot$最大值$_{q_i \in Q, c_j \in C}\{$相似度$(q_i, c_j)\} = UB(C)$。

证明。集合 Q 和 C 的语义重叠二分图中一条边（q_i， c_j）$\in Q \times C$的权重，受这两个集合中任意两个元素之间的最大相似度所限定。 M 在 Q 和 C之间的任意匹配的大小受 $|M| \leq min(|Q|, |C|) \leq |Q|$所限定。根据定义 1，$\mathcal{SO}(C) \leq UB(C)$。 □

由于 $UB(C) \leq \mathcal{SO}(C)$，任何满足 $UB(C) < \theta_k$ 的集合 C 都可以使用引理 2 安全地剪枝。

下界过滤器：给定一个二分图，在每次迭代中，贪心匹配算法都会选取未匹配节点之间权重最高的边，直到无法找到连接未匹配节点的边为止。最大匹配的贪心算法的复杂度为 $\mathcal{O}(n^2 \cdot \log n)$，其中 n 是集合的基数。如例 2 所示，贪心算法无法找到最优解。然而，已有研究证明，贪心匹配的得分至少达到最优得分的一半 [18]。

引理3。 设 Q 和 C 是两个语义重叠度为$\mathcal{SO}$（C）的集合。在对应的二分图中，令 LB（C）为以下两者中的较大值：***(a)*** 最大边权重；***(b)*** 贪心匹配得分。则， LB（C）$\leq \mathcal{SO}$（C）。

证明。 (a) 我们总能构造一个一对一匹配$M = \{(q_i, c_j)\}$，其中包含权重最大的边 (q_i, c_j)。(b) 由于二分图的贪心匹配是最优匹配的下界， $\mathcal{SO}(C)$以下界为贪心匹配得分。

□

如果已知 θ_k^* 的值，我们就能在细化步骤中执行最大限度的剪枝。将 θ_k 初始化为接近 θ_k^* 的值，可以提升 UB过滤器的剪枝能力。一种方法是计算一组样本集合的 $\mathcal{SO}$，并选出排名前 k 的集合，以此初始化 top-k 列表。在这种情况下，UB过滤器剪枝能力的提升是以图匹配计算为代价的。为避免这一代价，我们利用 θ_k 的一个下界进行剪枝。

引理4。 设 $\mathcal{R}$ 为当前运行的前k 个列表， θ_{lb} 为 $C \in \mathcal{R}$的最小下界 $LB(C)$。那么， $\theta_{lb} \leq \theta_k^*$。

证明。根据定义，θ_{lb} 是 $\mathcal{R}$中集合的最小 LB，而θ_k 是 $\mathcal{R}$中集合的最小精确 $\mathcal{SO}$ 。由于对于任意集合 $C \in \mathcal{L}$，$LB(C) \leq \mathcal{SO}(C)$，$\theta_{lb} = min_{C\in\mathcal{R}}\{LB(C)\} \leq min_{C\in\mathcal{R}}\{\mathcal{SO}(C)\} = \theta_k$，其中 $\theta_{lb} \leq \theta_k \leq \theta_k^*$。 □

借助引理4，如果$UB(C) < \theta_{lb}$，我们可以安全地剪枝候选集合。

IV. 精化：候选选择

在精化阶段，我们使用两种索引结构：令牌流 $\mathcal{I}_e$ 和倒排索引 $\mathcal{I}_s$。设 $\mathbb{D} = \cup_{C_j\in\mathcal{L}, c_i\in C_j} c_i$为 $\mathcal{L}$的词汇表。令牌流 $\mathcal{I}_e$ 是 $\mathbb{D}$ 中所有元素构成的流，这些元素按照与任意查询元素 $q_i \in Q$的最大相似度降序排列。令牌流 $\mathcal{I}_e$是由元组$(q_i, c_j, \text{sim}(q_i, c_j))$构成的序列，其中 $q_i \in Q$和 $c_j \in \mathbb{D}$。如果 $\text{sim}(q_i, c_j) < \text{sim}(q_l, c_m)$，则元组$(q_i, c_j, \text{sim}(q_i, c_j))$将在$\mathcal{I}_e$中排在$(q_l, c_m, \text{sim}(q_l, c_m))$之后。当不存在满足$\text{sim}(q_i, c_j) \geq \alpha$且 $q_i \in Q$的剩余令牌 $c_j \in \mathbb{D}$ 时，该流终止。倒排索引 $\mathcal{I}_s$ 将 $c_j \in \mathbb{D}$映射到 $\{C_1, \ldots, C_m\} \subseteq \mathcal{L}$，使得 $\forall\, C_i, 1 \leq i \leq m$：$c_j \in C_i$。从 $\mathcal{I}_e$中读取元组$(q_i, c_j, \text{sim}(q_i, c_j))$后，会查询索引 $\mathcal{I}_s$，以获取所有包含 c_j的集合。这将产生一个由 $\mathcal{L}$中的集合构成的流，这些集合按照集合元素与某个查询元素之间的最大相似度降序排列。显然，如果某个集合出现在该流中，则其语义重叠度非零，因此属于候选集合。首次在该流中观察到 C 时，$(q_i, c_j, \text{sim}(q_i, c_j))$实际上是$\max\{\text{sim}(q_i, c_j)\}, c_j \in C, q_i \in Q$。因此，根据引理2和3，我们可以初始化 $UB(C) = min(|Q|, |C|) \cdot \text{sim}(q_i, c_j)$以及 $LB(C) = \text{sim}(q_i, c_j)$。KOIOS使用从 $\mathcal{I}_e$ 中获取的前 k 个集合来初始化当前的top-k 列表和 θ_{lb}。随着获取的集合不断增加，可能会发现 $LB(C)$较低的集合，从而导致当前的top-k 列表和 θ_{lb}更新。

算法1展示了KOIOS精化阶段的伪代码。该算法的优势在于将精确匹配计算推迟到后处理阶段，同时在此阶段积极地对集合进行剪枝。在每次迭代中，KOIOS从 $\mathcal{I}_e$ 读取元组，并更新候选集的边界。任何满足 C 和 $UB(C) < \theta_{lb}$ 的集合都可以被安全地剪枝。满足 C 和 $LB(C) \leq \theta_{lb} \leq UB(C)$的集合则处于待定状态，直到读取更多元素并收集到有关该集合边界的更多证据。可能发生的变化包括：$UB(C)$减小、$LB(C)$增大，或 θ_{lb} 增大。索引 $\mathcal{I}_e$ 和 $\mathcal{I}_s$使我们能够按照初始上界和下界的降序获取候选集。通过按此顺序处理候选集，搜索算法能够及早识别有希望的集合，更新top-k 列表，并改进 θ_{lb}，从而实现较高的剪枝率。

根据引理2，$UB(C)$ 与其相对于某个查询元素的最大元素相似度相关联。为了根据初始 UB 和 LB对集合进行排序，我们需要按照词汇表 $\mathbb{D}$ 中各元素与某个查询元素的相似度降序检索这些元素。为了避免计算词汇表元素与查询元素之间的所有成对相似度，对于给定的 sim，任何能够实现高效基于阈值的相似度搜索的索引都适用。例如，当 sim 为余弦相似度时对于词元的词嵌入或元素词元集合的 Jaccard 相似度，可分别将 Faiss 索引 [19] 或最小哈希 LSH [20] 插入该算法。这使 KOIOS 能够独立于相似度函数的选择执行语义重叠搜索。这使 KOIOS 有别于现有的模糊集合相似性搜索技术 [13], [21]–[23], ，后者依赖针对特定相似度函数设计的过滤器。

为了按上界降序检索候选集，一种简单的方法是扩展倒排索引的思路，使用一个映射将每个查询元素关联到词汇表中所有元素的列表。列表中的元素按其与查询元素的相似度降序排列。该索引的大小随 Q 的基数线性增长。现在，设 $r: Q \to \mathbb{A}$ 由 $r_j(q) = t^j$ 给出，其中 $j \in 1, \ldots, |\mathbb{D}|$，也就是说，$r_j(q)$ 返回与 $\mathbb{D}$ 最相似的第 j 个元素。请注意，对于 Q 中的元素而言，第 j 个最相似的元素不一定是与 Q 中任一元素具有最高第 j 位相似度的那个元素，$\arg\max_{t^j}(\text{sim}(t^j, q_i)\ \forall i \in [Q])$。例如，$r_{i-1}(q)$ 和 $r_i(q)$ 的相似度都可能高于 $r_1(q')$。函数 r 可以通过一个在 $\mathbb{D}$ 上共享的索引 $\mathcal{I}$，以及一个大小为 $|Q|$ 的优先队列 $\mathcal{P}$ 来实现，该队列跟踪与 Q 中元素最相似的元素。我们将 $\mathcal{I}$ 和 $\mathcal{P}$ 称为令牌流 $\mathcal{I}_e$。

给定元素 q，索引 $\mathcal{I}$ 会返回 $\mathbb{D}$ 中与 q 最相似的下一个尚未见过的元素。在初始步骤中，我们使用 Q 中的所有元素探测 $\mathcal{I}$，并将结果加入优先队列。此时，$\mathcal{P}$ 中有 $|Q|$ 个元素，每个元素都是与某个查询元素最相似的元素。队列会记录 $\mathcal{P}$ 中每个元素对应的查询元素。队列的顶部始终会给出查询元素与相似度最高的尚未见过元素之间的映射。与 Q 第二相似的元素已缓存在 $\mathcal{P}$ 中。为保持队列大小，当我们弹出顶部元素时，只需使用与弹出元素对应的查询元素探测 $\mathcal{I}$，因为其余查询元素最相似的元素仍存在于 $\mathcal{P}$ 中。请注意，当提供非零 α 阈值时，我们会在取回第一个相似度小于 α 的元素时停止探测 $\mathcal{I}$。

此外，我们构建了一个倒排索引，将 $\mathbb{D}$ 中的每个元素映射到 $\mathcal{L}$ 中对应的集合。由于 $\mathcal{I}_e$ 按照与 Q 的相似度降序返回元素，因此从 $\mathcal{I}_s$ 中检索到的新集合也会按照 UB过滤器 的降序排列。

V. 优化：高级过滤器

如果通过索引获取的集合未使用 UB过滤器进行剪枝，则会将其添加到候选集合中。我们引入两个高级界限 $iUB(C_i)$ 和 $iLB(C_i)$，并介绍如何增量更新这些界限。iUB 和 iLB 基于集合的部分二分图贪心匹配。对于候选集合中的一个元素，$\mathcal{I}_e$ 可能会提供多个匹配元素（二分图中的所有边）。但是，我们只考虑有效边，即那些匹配未匹配元素的边。**iLB：** 在 iLB 中，我们假设所有不属于部分贪心匹配的边的相似度均为零。

算法1 KOIOS（精炼）

输入： $\mathcal{L}$：一个存储库，$\mathcal{I}_e$：令牌流，$\mathcal{I}_s$：$\mathcal{L}$ 中集合的倒排索引，Q：一个查询集，k：搜索参数，α：词元相似度阈值
输出： $\mathcal{U}$：候选集

$\mathcal{U} \leftarrow \emptyset$，$\mathcal{P} \leftarrow \emptyset$ // 候选集，剪枝集
$sim \leftarrow 1$，$\mathcal{L}_{lb} \leftarrow \emptyset$ // 初始相似度，top-k 下界列表
当 $sim \geq \alpha$ **时执行**
  $t, sim \leftarrow$ get_next_similar_token($\mathbb{Q}, \mathcal{I}_e$)
  如果 $sim \geq \mathcal{L}_{lb}$.bottom() **则** $Cs \leftarrow$ get sets(t, $\mathcal{I}_s$)
    对于 $C \in Cs$ 和 $C \notin \mathcal{U}$ 和 $C \notin \mathcal{P}$ **执行** $\mathcal{U}$.add(C)
  对于 $C \in \mathcal{U}$ **执行**
    UB(C).update(t, sim)
    如果 UB(C) $< \mathcal{L}_{lb}$.bottom() **则**
      $\mathcal{U}$.remove(C)，$\mathcal{P}$.add(C)
    $LB(C)$.update(t，sim)
    如果 $LB(C) > \mathcal{L}_{lb}$.bottom() **则** $\mathcal{L}_{lb}$.update(C)
返回 $\mathcal{U}$

引理 5. 给定候选集合 C 和查询 Q，Q和 C 的二分贪心匹配中任意边子集的权重总和都是 $\mathcal{SO}(C)$的下界。

证明。令 $G' = (V, E')$ 为贪心匹配 M'：$Q \to C$ 的二分图，其得分为 s'。假设 M''：$Q \to C$ 是一个匹配，满足 $G'' = (V, E'')$，其中 $E'' \subseteq E'$；令 s'' 为 M'' 的得分。由于 $E'' \subseteq E'$，我们有 $s'' \leq s'$。根据引理3，s' 是 $\mathcal{SO}(C)$ 的下界，因此，$s'' \leq s' \leq \mathcal{SO}(C)$。 □

假设 $iLB_l(C)$ 是 C 的部分贪心匹配当前得分。当读入 C 中一个与未匹配查询元素相似度为 s_{l+1} 的未匹配元素时，根据引理 5，将下界更新为 $iLB_{l+1}(C) = iLB_l(C) + s_{l+1}$。由于我们从 $\mathcal{I}_s$ 获取部分贪心匹配的边，因此始终考虑得分最高的部分匹配，从而计算出最大的 iLB。通过跟踪 Q 与每个 C 之间迄今已匹配的元素集合，可以直接判断某个 s_l 是否必须包含在匹配中。更新候选集合的下界可能导致更新 top-k 列表和 θ_{lb}。由于 θ_{lb} 始终递增（否则我们不会进行更新），更新 θ_{lb} 会根据上界剪枝更多集合。

由于集合 C 可能包含与查询元素相同的元素，即 $C \cap Q \neq \emptyset$，且索引按降序返回元素，因此 C 的下界会缩小为它与查询的重叠元素数量 $|Q \cap C|$，再加上 Q 和 C 中剩余元素的贪心匹配得分。因为下界会更新 θ_{lb}，而更紧的 θ_{lb} 能增强我们方法的剪枝能力，所以我们选择将集合的下界初始化为其普通重叠度（相同元素的数量）。为此，算法在首次探测 $\mathcal{I}_e$ 时，始终将查询元素本身纳入结果。采用这一策略，我们可以处理词汇表外的元素。例如，如果 sim 表示词元嵌入向量的余弦相似度，而某些词元未被嵌入语料库覆盖，那么只要查询中包含相同词元，我们仍会在语义重叠计算中考虑它们。

iUB-Filter: iUB 假设任何不属于部分贪心匹配的边都具有可能的最大相似度，即目前从 $\mathcal{I}_e$观察到的最小相似度。

引理 6. 给定查询 Q 和集合 C_i 的基数为 l、得分为 S_i的部分贪心匹配。在从 $\mathcal{I}_e$获取 (q_m, t, s)后，其中 $q_m \in Q$，元素 t 属于 $\mathcal{L}$中的任意集合，且s 是下一个最大的元素对相似度，我们有上界 $iUB(C_i) = S_i + min(|Q| - l, |C_i| - l) \cdot s \geq \mathcal{SO}(C_i)$。

证明。由于 l 个 C_i 元素已与 Q 的元素匹配，因此集合 C_i 最多还剩 $m_i = min(|Q| - l,\ |C_i| - l)$ 个元素待匹配。读取到一个元素 t时，其相似度次高，即 s，无论它属于哪个集合 t，根据定义，C_i 中任何尚未见到的元素的相似度都小于或等于 s。存在两种情况：元素 $t \in C_i$ 或 $t \notin C_i$。如果 $t \in C_i$ 已与查询中的某个元素匹配，我们便丢弃该元素，并知道任何未匹配元素与查询元素的相似度不超过 s。否则，观察到元素 t 后，C_i 中任何未匹配元素的上界可收紧至 s，并且有 $iUB(C_i) = S_i + m_i \cdot s$。

□

更新 $iUB(C_i)$ 可能会导致集合被剪枝，如果 $S_i + m_i \ . \ s \leq \theta_{lb}$。一种简单的方法是在每次从 $\mathcal{I}_e$ 检索到新元素时，更新所有集合的上界。然而，这会导致大量细微的更新，其中许多更新不足以降低上界，因而无法剪枝更新后的集合。为了解决这一问题，我们提出了一种技术，根据集合中未见元素的数量（m）将集合分组到不同的桶中。只有包含新检索元素的集合才需要更新，并被移动到桶 $m-1$ 中。其他所有集合无需更新，但当其上界低于 θ_{lb} 时，仍会立即被剪枝。

KOIOS 将集合划分到 m 桶 $B_m = \{(C_i, S_i) | m_i = m\}$ 中。新元素到达时，其相似度为 s，任何满足 C_i in B_m 且匹配元素总数为 S_i 的集合，如果其更新后的上界小于 θ_{lb}，即满足 $S_i + m \cdot s \leq \theta_{lb}$，就应进行剪枝。由此可知，如果 $S_i \leq \theta_{lb} - (m \cdot s)$，我们就可以安全地对 C_i 进行剪枝。由于桶中的所有集合剩余元素数量相同，且 s 对所有集合均为固定值，因此该剪枝不等式的右侧对桶中的所有集合都相同。

我们将成对的 $(C_i,\ S_i)$ 按 S_i 值升序排列存储在一个桶中。当一个相似度为 s 的元素到达时，我们扫描每个桶中有序的集合列表。如果某个对满足条件 $S_i \leq \theta_{lb} - (m \cdot s)$，我们就剪枝该集合。一旦找到一个集合 S' 不满足该条件，我们就可以断定其余集合也不满足该条件，因而无法剪枝，因为其余集合的 S_i 值必定大于 S'。现在，假设桶 B_m 中的一个集合 C_i 包含新到达的元素：我们首先从其桶中移除成对的$(C_i,\ S_i)$，更新 S_i，然后将该对插入 B_{m-1}。将更新限制在包含某个元素的集合上，并使用元素相似度来剪枝部分集合，可以使我们在每次迭代中减少大量更新。我们的实验表明，维护桶不会带来很大的开销。

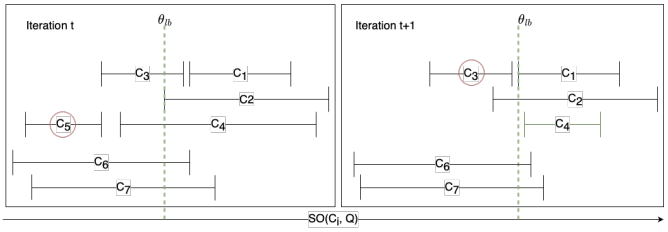


图3：使用 $k = 2$的上界和下界进行剪枝的集合。每个集合都用其下界和上界表示。红色圆圈表示被剪枝的集合，绿色线条表示下界和上界的最新更新。

由于即使对于相同的词汇表外元素，相同元素的相似度也为 1（例如 iLB），KOIOS 会将新获得的集合 C_i 及其 S_i 的上界初始化为其普通重叠度。

示例 3： 考虑图 3 中的集合 C_1、.....、C_7。假设我们要搜索与某个查询的语义重叠度最高的前 2 个集合。每个集合都用其相对于该查询的下界和上界构成的区间表示。最初，θ_{lb} 根据 C_1 和 C_2的下界计算得出，因为它们在所有集合中拥有前 2 个最高下界。这意味着，在迭代 t开始时，集合 C_5 被剪枝，因为 $UB(C_5) < \theta_{lb}$。其余集合仍保留在候选集合中，因为它们的下界都小于 θ_{lb}，且上界都大于 θ_{lb}。假设在迭代 $t+1$中，通过从令牌流读取一个元素 C_4，$LB(C_4)$ 和 $UB(C_4)$ 得到更新（见图 3 右侧）。此时，C_4 是下界最高的集合，因此 θ_{lb} 的值更新为 $LB(C_1)$。这样我们就可以安全地剪枝 C_3，因为 $UB(C_3) < \theta_{lb}$。在精化阶段结束时，算法将剩余集合（C_1、C_2、C_4、C_6和 C_7）传递给后处理阶段。□

VI. 后处理阶段

所有在细化过程中未被剪枝的候选集都需要进行验证。由于基数较大，即使其中元素的相似度相对较低，某些集合仍可能进入这一阶段。KOIOS 在后处理阶段应用过滤器，以尽可能减少精确匹配计算的次数以及完成计算所需的时间。

No-EM： 令 $\mathcal{U}$ 为尚未被剪枝的候选集，并令 θ_{ub} 为针对 $C \in \mathcal{U}$ 的第 k 大 $UB(C)$。

引理 7. 具有 $\theta_{ub} \leq LB(C)$ 的集合 $C \in \mathcal{U}$ 保证出现在前-k 个结果 ω中。

证明。 计算 a $C \in \mathcal{U}$ 的最大图匹配永远不会增加 θ_{ub}，因为 $\mathcal{SO}(C) \leq UB(C)$。如果集合$C \in \mathcal{U}$ 满足 $\theta_{ub} \leq LB(C)$，根据 $LB(C) \leq UB(C)$ 和$\theta_k^* \leq \theta_{ub}$，可以保证 $\theta_k^* \leq \mathcal{SO}(C)$，且 C 位于前 k 个结果中。 □

该引理使我们能够跳过对某些集合的精确语义重叠计算。由于与查询的距离为 θ_k^* 处存在并列情况，因此 top-k 搜索问题可能存在多个解。所有解的 θ_k^* 值均相同。

算法 2 KOIOS（后处理）

输入： Q：一个查询集，k：搜索参数，$\mathcal{U}$：未剪枝集，k：搜索参数，$\mathcal{L}_{lb}$：前-k 下界列表
输出： $\mathcal{L}_{ub}$：前-k 个结果

```
Q_ub ← init pq UB(U) // 未剪枝集上的优先队列
L_ub ← init_topk_UB(U) // 未剪枝集上的 top-k 上界列表
当 ¬L_ub.all checked() 时
    C ← L_ub.select next unchecked()
    如果 LB(C) ≥ UB(L_ub.top()) 那么
        C.checked ←真
        继续
    精确语义重叠(C)←计算_精确语义重叠_提前_终止(C,Q)
    下界(C),上界(C)←精确语义重叠(C), C.checked ←真
    if SO(C) < L_ub.bottom() 则 L_ub.remove(C)
        if SO(C) ≥ L_lb.bottom() 则 Q_ub.add(C)
    if SO(C) ≥ L_lb.bottom() 则 L_lb.update(C)
    当 L_ub.len() < k 且 ¬Q_ub.empty() 时执行
        C ← Q_ub.pop()
        如果 L_lb.bottom() < UB(C) 那么 L_ub.add(C)
return L_ub
```

在此阶段，KOIOS 维护三种数据结构：1）按上界 ($\mathcal{L}_{lb}$) 排序的集合列表，其中上界为前 k 个；2）按上界 ($\mathcal{L}_{ub}$) 排序的集合列表，其中上界为前 k 个；3）按上界 ($\mathcal{Q}_{ub}$) 排序的集合优先队列。维护 $\mathcal{L}_{lb}$ 和 $\mathcal{L}_{ub}$，使我们能够分别快速访问 θ_{lb} 和 θ_{ub}。根据引理 7，算法只应计算满足 $UB(C) \geq \theta_{ub}$ 的集合的二分图匹配。因此，KOIOS 优先对上界较高的集合进行精确图匹配计算。直观来说，上界较高的集合有可能具有较高的语义重叠度。为加快处理速度，$\mathcal{L}_{ub}$ 中的所有集合都会进入队列，并在后台并行评估。集合 C 的精确匹配完成后，我们更新 $LB(C) = UB(C) = \mathcal{SO}(C)$。这会产生两种影响。首先，更新 $UB(C) = \mathcal{SO}(C)$ 可能导致 θ_{ub} 降低。因此，如果 $\mathcal{SO}(C) \geq \theta_{ub}$，该集合仍保留在 $\mathcal{L}_{ub}$ 中。如果 $\mathcal{SO}(C) < \theta_{ub}$，则将该集合加入 $\mathcal{Q}_{ub}$，因为算法之后可能会发现，$\mathcal{SO}(C)$ 高于当前位于 $\mathcal{L}_{ub}$ 中、其语义重叠度尚未计算的集合。将集合插入 $\mathcal{Q}_{ub}$ 会导致 $\mathcal{L}_{ub}$ 中有 $k-1$ 个集合。查询 $\mathcal{Q}_{ub}$ 可找到下一个上界排名第 k 的集合，并将其加入 $\mathcal{L}_{ub}$。

其次，$LB(C)$ 的更新可能导致 θ_{lb} 增大，并可能剪枝某些集合。该算法采用惰性策略，考虑 $\mathcal{L}_{ub}$ 中的集合进行此类剪枝，直到获得一个满足 $UB(C) \leq \theta_{lb}$ 条件的集合 C。当 $\mathcal{L}_{ub}$ 中的所有集合均满足条件 $\theta_{ub} \leq LB(C)$ 时，后处理阶段终止，并将该列表作为最终搜索结果返回。

EM-提前终止过滤器：尽管进行了大量剪枝，精确匹配的计算仍然开销很大。考虑在 C 和 Q 上构建的二分图 $G(V, E)$，其中 $V = Q \uplus C$，且元素 $q_i \in Q$ 和 $c_j \in C$ 之间边的权重为 $w(q_i, c_j) = \text{sim}(q_i, c_j)$。可选一对一匹配 $M \subseteq E$ 的权重为 $w(M) = \sum_{(q_i,c_j)\in M} w(q_i, c_j)$。如果对于每个 $v \in V$，都存在某个与 v 相 incident 的 $e \in M$，则称匹配 M 为完美匹配。匈牙利算法 [24] 将节点标记视为函数 $l : V \to \mathbb{R}$。可行标记满足 $l(q_i) + l(c_j) \geq w(q_i, c_j)\ \forall q_i \in Q, c_j \in C$。等式子图是子图 $G_l = (V, E_l) \subset G = (V, E)$，固定

在标记 l 下，使得 $E_l = \{(q_i, c_j) \in E : l(q_i) + l(c_j) = w(q_i, c_j)\}$。匈牙利算法考虑一个有效的标记函数 l，并维护一个匹配 M 和一个图 G_l。它从 $M = \emptyset$ 开始。每次迭代中，算法要么扩展 M，要么改进标记 $l \to l'$，直到 M 在 G_l 上成为完美匹配。设 M' 是 G 中的一个完美匹配（不一定在 G_l 中）。一个众所周知的结果是，如果 l 可行，且 M 是 G_l 中的完美匹配，那么 M 是最大权匹配 [24]。根据 Kuhn-Munkres 定理的证明，我们有 $w(M') \leq w(M)$。该定理还表明，对于任意可行标记 l 和任意匹配 M，都有 $w(M) \leq \sum_{v \in V} l(v)$ [24]。

引理 8。如果匈牙利算法 *[24]* 分配的标签总和小于 θ_{lb}，则可以在与 Q 进行二分图匹配时安全地剪枝集合 C 。

证明。假设存在一个有效的节点标记函数 l 和一个相等子图 G_l。匈牙利算法生成的 G_l 上的完美匹配 M 确实是最大匹配，其权重 $w(M)$ 即为 Q 和 C之间的语义重叠度。现在，设 M' 是 G中的某个完美匹配。根据 Kuhn-Munkres 定理，我们知道 $w(M') \leq w(M)$，并且对于标记函数 l 和匹配 M，有 $w(M) \leq \sum_{v \in V} l(v)$，因此，我们有 $w(M') \leq \sum_{v \in V} l(v)$。由此，节点标记值之和构成了 $\mathcal{SO}(Q, C) \leq \sum_{v \in V} l(v)$ 的上界。在图匹配过程中，可以动态计算该上界。一旦 $\sum_{v \in V} l(v)$ 超过 θ_{lb}，即可剪枝集合 C。□

通过在集合 C 的图匹配过程中计算此上界，一旦 $UB(C) < \theta_{lb}$，就可以终止该过程，并安全地移除 C 。对于较大的集合，这种提前终止尤为重要：当一个大集合的匹配并行执行时，随着其他集合的处理完成，全局 θ_{lb} 会持续更新。

示例 4： 假设图 4 中的集合 $D_1, \ldots, D_6$ 正处于后处理阶段。假设我们正在搜索排名前 3 的集合。每个集合都用相对于查询的下界和上界区间表示。后处理从 $\mathcal{L}_{ub} = [D_2, D_1, D_6]$ 开始。根据 $LB(D_2) > \theta_{ub}$，我们知道无需计算 D_2 的精确匹配，因为我们知道 θ_k (θ_k^*) 的真实值永远不会大于 θ_{ub}。因此，任何语义重叠度高于 θ_k^* 的集合都必须包含在最终结果中。D_1 和 D_6 的精确匹配并行计算。假设 D_6 完成计算，其边界得到更新。这导致 θ_{lb} 增大，θ_{ub} 减小。因此，D_6 从 $\mathcal{L}_{ub}$ 中移除，并添加到 $\mathcal{Q}_{ub}$ 中。在探测 $\mathcal{Q}_{ub}$ 并考虑 θ_{lb} 后，D_3 被添加到 $\mathcal{L}_{ub}$ 中。假设使用提前终止规则，$UB(D_3)$ 被更新为一个低于 θ_{lb} 的值。算法立即停止对 D_3 的匹配，并将其从 $\mathcal{L}_{ub}$ 中移除。此时，D_6 被重新添加到 $\mathcal{L}_{ub}$ 中，且 $\theta_{ub} = SO(D_6)$。算法继续计算 $\mathcal{L}_{ub}$ (D_1) 中的下一个集合。□

为实现搜索的横向扩展，我们随机划分存储库，并在各个分区上并行运行 KOIOS。为了改进这种方法，

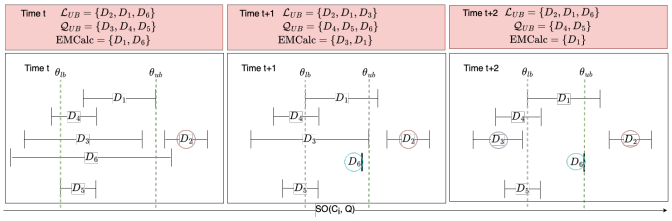


图4：后处理：利用上界、下界和精确语义重叠度对集合进行剪枝，适用于 $k = 3$。每个集合均用其下界和上界表示。此处红色圆圈表示 NoEM Filter，蓝色圆圈表示 EM，紫色圆圈表示 EM-ETM Filter。

剪枝能力，所有分区共享一个全局 θ_{lb} ，它是 θ_{lb} 的最大值。

VII. KOIOS 分析

A. 正确性

根据定义1，若集合 C 中至少有一个元素与Q的相似度大于 α ，则该集合与Q具有非零语义重叠度。令牌流会检索所有相似度大于 α的集合元素，而对于每个元素，倒排索引会返回包含该元素的所有集合。因此，所有具有非零语义重叠度的集合都会被纳入考虑。KOIOS 可在以下三种情况下对集合C 进行剪枝：(1) 当 C 因其某个元素 $c_j \in C$ 与某个查询元素 $q_i \in Q$的相似度 s 而首次出现时。由于 $\mathcal{I}_e$中的相似度按非递增顺序排列，$s = \max\{\text{sim}(q_i, c_j)\}$，$c_j \in C$，$q_i \in Q$。因此，根据引理2，可以计算上界 $UB(C)$；若 $UB(C) \leq \theta_{lb}$，则对 C 进行剪枝。请注意，将 θ_{lb} 纳入剪枝考量不会造成假阴性，因为如引理4所示，$\theta_{lb} \leq \theta_k \leq \theta_k^*$。(2) 在通过 iUB 过滤器进行细化时：引理6证明了一个基于任意集合元素相似度的上界，因此，任何被 θ_{lb} 剪枝的集合都不可能是假阴性。(3) 在后处理期间：引理8证明，当匈牙利算法的节点标签之和小于 θ_{lb}时，可以安全地对该集合进行剪枝。在此阶段，算法继续基于 UB过滤器进行剪枝，因为 θ_{lb} 正通过精确匹配计算不断改进。如果集合在算法终止前未在上述条件下被剪枝，则它会保留在 $\mathcal{L}_{ub}$ 中，并且实际上属于 top-k 结果。

B. 时间与空间复杂度

精确二分图匹配可在$\mathcal{O}(n^3)$时间内计算 [25], [26][1]。虽然 KOIOS 的最坏情况时间复杂度为 $\mathcal{O}(m \cdot n^3)$，其中 n 是最大集合基数，m 是 $\mathcal{L}$中的集合数，但得益于我们的过滤器，实际所需的比较次数远少于 m ，因此实际运行时间可缩短数个数量级。此外，EM 提前终止过滤器可以减少匈牙利算法的迭代次数。

回想一下，$\mathbb{D}$ 是 $\mathcal{L}$中所有不同词元构成的词汇表。Q 中的每个词元都可以具有大于 α 的相似度，与

[1]对于具有特定结构的图，使用戴克斯特拉算法和斐波那契堆可以实现更低的复杂度 [27]。

表 I：数据集特征。

	集合数	最大规模	平均规模	唯一元素数
DBLP	4,246	514	178.7	25,159
OpenData	15,636	31,901	86.4	179,830
Twitter	27,204	151	22.6	72,910
WDC	1,014,369	10,240	30.6	328,357

最多包含 $|\mathbb{D}|$ 个词元，因此令牌流 $\mathcal{I}_e$ 的空间复杂度为 $\mathcal{O}(|\mathbb{D}| \cdot |Q|)$。倒排索引 $\mathcal{I}_s$ 的空间复杂度与输入规模呈线性关系：它存储 $|\mathbb{D}|$ 个键，且所有列表的总长度最多为 $D^+ = \sum_{C \in \mathcal{L}} |C|$。在 $|\mathbb{D}| \leq D^+$ 且平均集合大小为 $\bar{C}$的情况下，$\mathcal{I}_s$ 的空间复杂度为 $\mathcal{O}\left(\mathcal{L} \cdot \bar{C}\right)$。在迭代过程中的任意时刻，两个 Top-$k$ 列表（ $\mathcal{L}_{lb}$ 和 $\mathcal{L}_{ub}$）最多存储 k 个集合，其空间复杂度为$\mathcal{O}(k)$。优先队列 $\mathcal{Q}_{ub}$ 的大小为 $\mathcal{O}(\mathcal{L})$。由此可得整体空间复杂度为 $\mathcal{O}\left(|\mathbb{D}| \cdot |Q| + \mathcal{L} \cdot \bar{C}\right)$。

VIII. 实验

我们在四个真实世界数据集上评估 KOIOS 过滤器的响应时间、内存占用和剪枝能力，并测试各种参数设置和查询基数。我们将 KOIOS 与一个基线方法以及一种最先进的模糊集合相似性搜索技术进行比较。通过与普通集合重叠搜索的结果进行比较，我们评估语义重叠度量对搜索的实用性。在实验中，我们使用 FastText [28] 的预训练向量[2] 来计算词元嵌入向量的余弦相似度，并将其作为函数 sim。

A. 实验设置

*1)*数据集：我们使用四个数据集：DBLP [29], OpenData [30],Twitter [31], 以及 WebTables（WDC）的公开语料库 [32]。对于 DBLP，我们考虑 2018 年和 2019 年发表的论文，并针对每篇论文，从其标题和摘要中提取由空格分隔的单词，生成一个词集。对于 Twitter 数据集中的每条英文推文，我们生成一个由推文中不同单词组成的集合，但不包括表情符号和 URL。OpenData 和 WDC 的集合则由每张表中每一列的不同取值构成。对于所有数据集，我们均移除数值，以避免偶然匹配。此外，我们对 OpenData 和 WDC 进行进一步筛选，丢弃预训练向量覆盖率低于 70% 的所有集合。提取出的集合的特征如表 I 所示。WDC 和 Twitter 中的大多数集合规模较小，但与 OpenData 和 DBLP 相比，它们包含更多集合。与 OpenData 不同，WDC 中存在一些出现频率极高的元素，这导致倒排索引中的倒排列表过大。因此，在细化阶段，WDC 中的候选集合数量较多，而且更新集合的上下界通常比在 OpenData 中开销更大。

2）基准测试：我们从每个数据集中生成了一个查询基准测试，即一组查询集。WDC 和 OpenData 中集合的基数分布极不均匀 [8]。为了评估查询基数对性能的影响，WDC 和 OpenData 的基准测试由从不同基数范围中选取的查询集组成。OpenData 的范围为：10 到 750、750 到 1000、1000 到

[2]https://fasttext.cc/docs/en/english-vectors.html

表 II：使用过滤器剪枝的集合平均百分比。

数据集	细化	后处理	
	iUB-Filter	EM-提前终止	无电磁干扰
DBLP	91%	5%	9.2%
OpenData	85.5%	2.1%	54.8%
Twitter	53.5%	0%	1.4%
WDC	89.2%	0.9%	9.8%

1500、1500 到 2500、2500 到 5k，以及 5k 到 32k；WDC 的区间为：10 到 250、250 到 500、500 到 750、750 到 1k，以及 1k 到 11k。对于每个区间，我们分别对 OpenData 和 WDC 使用均匀随机抽样，抽取 50 个和 100 个集合。按区间抽样可以避免基准测试偏向较小的集合。由于集合基数呈幂律分布 [7]，高基数集合使用较大的区间。由于 DBLP 和 Twitter 中基数分布不均的集合较少，我们不划分区间，而是使用均匀随机抽样抽取 100 个随机集合。我们报告每个基准测试和区间中各查询结果的平均值。

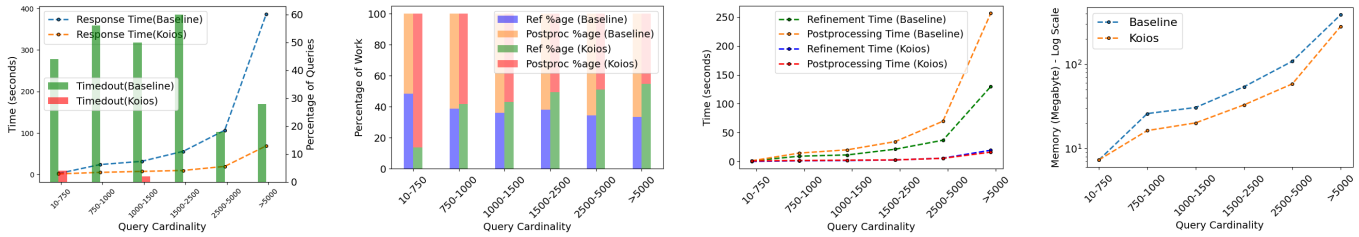
3) 实现：倒排索引（$\mathcal{I}_s$）按需计算，并存储在内存哈希映射中。为生成令牌流，我们使用 Faiss 索引 [19] 的 GPU 实现，在高维向量上执行 top-k 查询。我们以每批 100 个元素的方式查询 Faiss 索引。对于 DBLP、OpenData、Twitter 和 WDC，倒排索引的构建时间分别为 1.5、3.0、1.3 和 80 秒；Faiss 索引的构建时间分别为 3.6、9.5、3.8 和 12.5 秒。在精化阶段，我们缓存返回向量的相似度，以便在初始化图匹配所用的相似度矩阵时重复使用。对于图匹配，我们使用匈牙利算法 [33]的一种实现。为了在后处理阶段并行计算集合的图匹配，我们使用一种兼容 C++17 的线程池实现 [34]。除非另有说明，所有实验均使用以下参数：相似度阈值$\alpha = 0.8$、k = 10 和分区 = 10。

4) 基线方法：针对 top-k 语义重叠查询，基线方法遍历所有候选集并计算其二分图匹配。候选集是指至少包含一个与任意查询元素的相似度高于阈值 α 的元素的集合。我们使用令牌流获取候选集列表（基线的精化阶段），并使用线程池 [34] 并行计算所有候选集的二分图匹配（基线的后处理阶段）。鉴于 WDC 中集合数量庞大且元素出现频率较高，为所有候选集计算精确的二分图匹配是不可行的。例如，对于一个基数为 53 的查询集，我们有 190,679 个候选集。为了减少候选集的数量，我们启用 iUB-Filter 来辅助集合剪枝。这种方法称为基线方法+。

5) 系统规格：所有实验均在一台配备 2 颗 Intel® Xeon Gold 5218 @2.30GHz（64 核心）、512 GB DDR4 内存、一块 Samsung® SSD 983 DCT M.2（2 TB）以及 4 块 GPU（TU102，即 GeForce RTX 2080 Ti）的机器上进行。

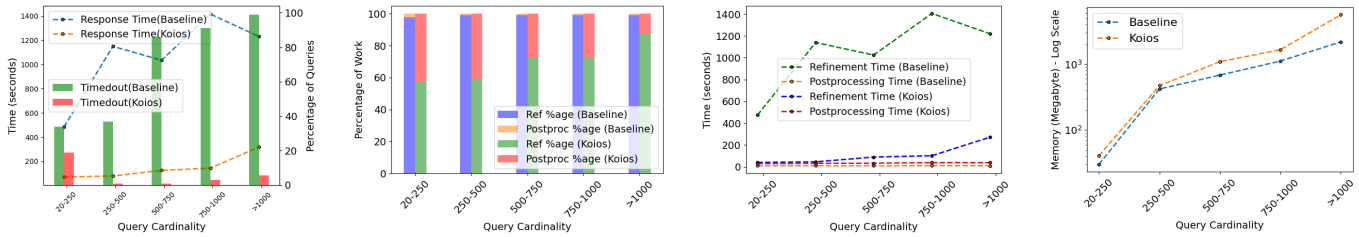
B. 响应时间

我们报告了表 III 中所有数据集在各项基准测试下的平均响应时间（单位：秒）（倒排索引和



(a) (b) (c) (d)

图 5：OpenData 结果：(a) 响应时间，(b)、(c) 阶段分解，(d) 内存占用 nt.



(a) (b) (c) (d)

图 6：WDC 结果：(a) 响应时间，(b)、(c) 阶段分解，(d) 内存占用。

表 III：平均响应时间和内存占用。

数据集	KOIOS 精化 (秒)	KOIOS 后处理 (秒)	KOIOS 响应 (秒)	KOIOS Mem (MB)	基线 响应 (秒)	基线 Mem (MB)
DBLP	0.3	0.44	0.83	16	211	11
OpenData	7.19	6.9	18.6	69.6	101	102.5
Twitter	0.2	0.45	0.7	10	518	10
WDC	109	34.3	147	1,775	1,062	885

令牌索引构建时间不计入 § VIII-A3。我们不报告超时查询的时间（2500 秒），因此，对于 WDC 和 OpenData 的某些区间，我们没有足够的数据。根据表 III，KOIOS 在所有数据集上相较于基线至少实现了 5 倍的加速，在 DBLP 和 Twitter 上则至少实现了 200 倍的加速。鉴于 WDC 和 OpenData 这两个数据集表现出较大的集合基数偏斜，我们根据查询基数对它们进行了进一步分析。

查询基数的影响：图 5a、5b、6a 和 6b 分别展示了 OpenData 和 WDC 的响应时间，以及各阶段耗时占比。报告的时间是每个区间内所有查询的平均值。由于 KOIOS 并行处理所有分区，我们报告每个区间内所有查询中各分区耗时占比的平均值。我们观察到，响应时间会随着查询基数增加而增长，这意味着 $\mathcal{I}_s$ 返回的相似元素和候选集合数量更多。WDC 在细化阶段的工作占比高于 OpenData，这是因为它的集合数量庞大，且元素出现频率较高。

与基线比较：图 5a 和图 6a 表明，在 OpenData 和 WDC 中，对于中等规模到大型查询，KOIOS 的性能尤其优于基线。这凸显了过滤器的剪枝能力。图 5a 和图 6a 还报告了超时查询的数量，基线的超时查询数量远高于 KOIOS，因为基线不会对大型低优先级.....

在精化阶段的潜在集合。对于小型查询，KOIOS 仅在约 5% 的 OpenData 查询和约 20% 的 WDC 查询中超时。这是由于较大的倒排列表导致后处理阶段产生了大量集合，这一点可从表 IV 和 V 中看出。我们还注意到，KOIOS 在某些中大型查询中也会超时，这是由代价高昂的图匹配所导致的。总而言之，KOIOS 在精化阶段的过滤开销显然是值得的，并显著缩短了相较于基线的运行时间。值得指出的是，KOIOS 能够以远低于超时限制的时间找到 k ×(分区数量) 个集合，而对于 WDC 的大多数大型查询，基线方法都会超时。尽管对于 OpenData 的较小型查询，KOIOS 与基线方法在响应时间上的差异并不明显，但 KOIOS 发现了 k ×(分区数量) 个集合，而基线方法则未能做到这一点。

与模糊搜索的比较：包括 Fast-Join [11], [12] 和 SILKMOTH [13] 在内的模糊搜索技术无法解决本文所研究的 Top-k 重叠度相似性搜索问题：（1）仅支持集合元素之间特定的字符级相似度（对元素中以空格分隔的词元计算 Jaccard 相似度，以及编辑距离）；无法应用词嵌入上的余弦相似度等语义相似度（正如我们在实验中所采用的方式）。（2）现有的模糊搜索算法均基于阈值：为了检索与给定查询最相似的 Top-k个集合，需要预先给定阈值 θ_k^*，但该阈值事先未知；事实上，这正是 Top-k 搜索面临的挑战之一，也是我们解决方案的一部分，即快速收敛到该阈值。

虽然模糊搜索算法不支持语义元素相似性，但我们的 KOIOS 算法支持所有语法元素相似度函数。我们将基于阈值的模糊搜索扩展为支持 top-k 搜索，具体如下：（1）选取

阈值 θ 是所有查询中的最小 θ_k^*。请注意，这让 SILKMOTH 占据了优势，因为我们传入了 θ_k^* 的真实值。(2) 使用阈值 θ 执行模糊搜索，并从查询结果中选出最相似的前 k 个集合（通过维护一个前 k 优先队列）。

我们的研究重点关注 SILKMOTH，该方法已被证明性能远胜于 Fast-Join [13]，并考虑 OpenData 和 WDC 数据集。由于这些数据集中的集合是从表中提取的，其中的大多数元素仅包含很少的词，因此在 SILKMOTH 中，非相同元素的边权重为零。因此，在实验中，对于 KOIOS 和 SILKMOTH，我们均将每个元素的三元组表示上的 Jaccard 相似度作为元素相似度。为生成令牌流，我们使用集合相似度连接技术 [9]，预先计算词汇表中与每个查询元素相似的元素，Jaccard 相似度阈值为 $\alpha = 0.8$。计算该基准测试的令牌流需要 8 秒。

我们比较了 SILKMOTH 的两个版本：第一个版本 SILKMOTH-semantic 对 SILKMOTH 算法进行了调整，以涵盖 KOIOS[3] 的大部分功能；这一调整由原始作者提出，旨在构建一个通用搜索框架，并排除了特定于相似度函数的过滤器 [13]。第二个版本 SILKMOTH-syntactic 使用了所有索引和过滤器，包括那些仅适用于特定相似度函数的索引和过滤器。我们在从 OpenData 的基准测试查询中随机抽取的 54 个查询上运行这些算法，以确保评估涵盖小型、中型和大型集合，并测量响应时间。KOIOS、SILKMOTH-syntactic 和 SILKMOTH-semantic 的平均响应时间分别为 72 秒、141 秒和 400 秒。在所有查询规模范围内，KOIOS 的性能均优于 SILKMOTH-semantic 和 SILKMOTH-syntactic，分别快一个数量级，即 6 倍（超时）和 2 倍。SILKMOTH 根据集合元素生成签名，并利用这些签名来帮助缩小候选空间。随着集合元素数量的增加，签名数量也会增加（此处通过划分为 q-gram 实现），而 SILKMOTH-syntactic 由于可行候选项数量过多而发生超时。KOIOS 的性能优于 SILKMOTH-syntactic，因为它基于相似度，在有序的集合元素配对流上进行操作，因此不受元素数量的影响。KOIOS 的性能优于 SILKMOTH-semantic，因为 SILKMOTH 的剪枝能力高度依赖于针对特定相似度函数专门设计的过滤器。

C. 过滤器的剪枝能力

表 II 报告了所有数据集中两个阶段所使用的不同过滤器的平均剪枝能力，而表 IV 和表 V 则分别详细展示了 OpenData 和 WDC 在不同查询基数区间内的剪枝能力。

精化阶段： 在 OpenData 和 WDC 中，候选集的数量会随着查询基数的增加而增加，因为更多必须读取来自 $\mathcal{I}_s$ 的倒排列表。iUB-Filter 的剪枝能力会随着查询基数的增加而增强。这有两个原因：首先，OpenData 和 WDC 中有大量小基数集合（表 I），这些集合会由倒排列表返回。其次，每个集合的 iUB 是根据剩余元素的数量计算的，因此，与查询集相比基数较小的候选集合，其 iUB 远小于 θ_{lb}，因而会被剪枝。如表 IV 和 V 所示，尽管候选集合的数量会随着查询基数的增加而增加，但需要 KOIOS 进行后处理的比例会随之下降。例如，对于 WDC 中的大型查询，KOIOS 只需对不到 5% 的候选集合进行后处理。

后处理阶段： 表 II 显示，与 EM-Early-Terminated 相比，No-EM 过滤器具有更强的剪枝能力，例如，在 OpenData 中剪除了超过一半的集合。请注意，报告的百分比是指在精化阶段未被过滤的集合。从表 IV 和 V 中，我们观察到，对于较大的查询，No-EM 与 EM-Early-Terminated 的组合具有最高的剪枝能力。对于 OpenData，该两种过滤器的组合比在 WDC 中剪除更多的集合。这是因为，在后处理阶段精确计算语义重叠度会增加 θ_k，从而导致剪除大量集合。

表 IV：OpenData：过滤器剪除的集合数量。

查询卡片	精化阶段		后处理阶段		
	候选项 Sets	iUB-Filter过滤后的	无能量最小化 (No-EM)	EM-Early 已终止	EM
$10-750$	1132	345	88	0	699
$750-1000$	2557	2422	85	2	48
$1000-1500$	2699	2571	83	4	41
$1500-2500$	3440	3328	84	2	26
$2500-5000$	3560	3451	82	4	23
>5000	5706	5502	79	5	120

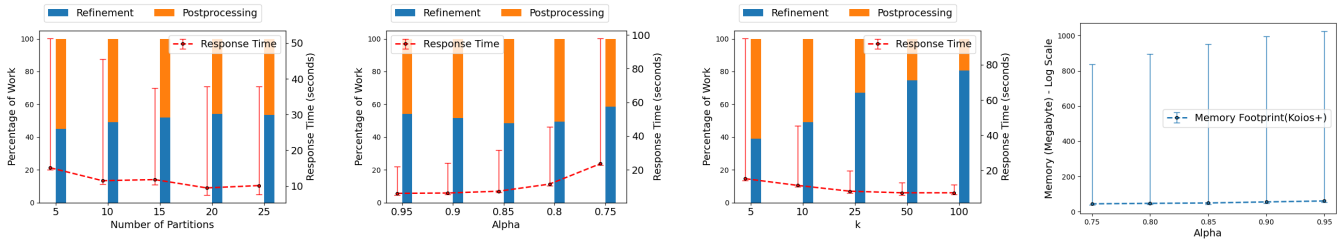
表 V：WDC：过滤器剪除的集合数。

查询卡片。	精化阶段 候选 iUB-Filter 过滤的 Sets		后处理阶段 无 EM	EM-早期已终止	EM
$20-250$	$124,217$	$60,196$	74	80	$63,867$
$250-500$	$189,665$	$186,512$	90	3	$3,060$
$500-750$	262，947	261，901	85	6	953
$750-1000$	274，695	$273,743$	83	26	843
>1000	$402,622$	$402,332$	84	3	203

D. 内存占用

KOIOS 是一种内存算法。我们在表 III 中报告了所有数据集在各项基准测试中的平均内存占用。本文的扩展版本对 KOIOS 的内存占用进行了深入分析。图 5d 和 6d 展示了 KOIOS 和基线算法在 OpenData 和 WDC 上的内存占用。报告的数值是在每个区间内成功查询所对应的数据结构的平均内存占用。请注意，某些数据结构（例如倒排索引、令牌索引和 top-k lists）在所有查询区间内的大小都是固定的。为了全面展示内存占用情况，这些图和表 III 中报告的数值是数据结构占用量的总和。

[3]SILKMOTH-semantic 需要使用满足三角不等式的 度量词元相似度函数。KOIOS 只要求满足对称性，也可以处理词元嵌入上的余弦相似度，而余弦相似度并不是一种度量。



(a) (b) (c) (d)

图 7：KOIOS 在 OpenData 上的参数分析：运行时间与 (a) 分区数量、(b) 元素相似度（α）和 (c) 结果大小的关系；(d) 内存占用与 alpha 的关系。

精化阶段和后处理阶段所用的数据结构，尽管精化阶段所用的数据结构（top-k 下界列表（$\mathcal{L}_{lb}$）除外）会在该阶段结束时释放。表 III 显示，KOIOS 的内存利用率与基线相当。

查询基数的影响：在图 5d 和 6d 中，我们观察到内存占用随查询基数线性增长。这可以用候选集数量的线性增长来解释（表 IV 和 V），因此，查询相关数据结构的大小也会随着查询基数的增加而增大，这些数据结构包括令牌流、上界桶、下界数据结构和优先队列。

与基线比较：图 5d 和 6d 展示了基线方法和 KOIOS 的平均内存占用。对于 OpenData 和 WDC 上的小型及中型查询，两者的内存占用相近。对于大型查询，KOIOS 在 OpenData 上占用的内存少于在 WDC 上的内存。这是因为 iUB-Filter 会剪枝大多数候选集，从而减小后处理数据结构的大小。请注意，对于基线方法，我们没有足够的 WDC 大型查询数据，因为几乎所有查询都会超时。

E. 结果质量

为了评估语义重叠度带来的质量提升，我们使用普通集合重叠度，对比 OpenData 上排名前 k 的语义搜索结果与排名前 k 的语法搜索结果。图 8 报告了两个列表中排名第 k 的集合的得分。比较排名前 k 的语法列表中第 k 个集合的语法重叠度与排名前 k 的语义列表中对应集合的语法重叠度后，我们发现，语义重叠度能找到语法重叠度较低（精确匹配的元素更少）、但语义重叠度较高（语义相似的元素更多）的集合。这些集合通常包含语法上不规范的元素，例如 (松鼠, squirrell) 和 (konstantine, konstantin)，或者语法上不匹配但语义相似的元素，例如 (利兹, 谢菲尔德)。我们还研究了两种搜索返回的集合，并报告了结果交集的大小。图 8 显示，语义重叠度能找到普通重叠度无法发现的集合。特别是，对于最小区间，普通重叠度会漏掉 50% 的结果。这表明，如果只考虑语法，那么语义重叠度可以帮助找到那些不会出现在排名前 k 结果中的集合。

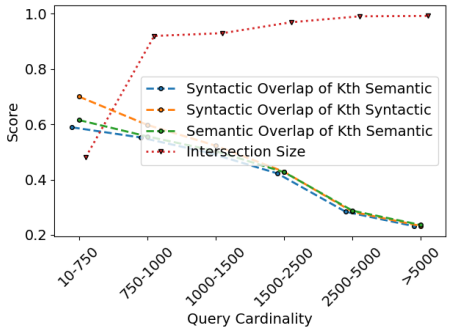


图 8：普通重叠度与语义重叠度的比较。

考虑了重叠度。请注意，只要索引返回精确结果，KOIOS 就会返回精确解。

F. 参数分析

我们的算法使用三个参数进行搜索：1）元素相似度阈值（α）、2）数据分区的数量，以及 3）用户提供的结果集数量（k）。我们对这些参数对 KOIOS 响应时间的影响进行了实证研究。在这组实验中，我们从 OpenData 的基准测试中随机选择了 100 个查询。由于数据存储库中的集合基数遵循 Zipf 分布 [7], [8],，从基准测试区间中进行随机采样可以避免基准测试过度偏向较小的查询集。图 7 展示了不同参数取值下的平均响应时间，以及查询细化时间和后处理时间所占比例的细分情况。

分区数量：在图 7a 中，我们固定 $k = 10$ 和 $\alpha = 0.8$，并改变分区数量。随着分区数量的增加，响应时间逐渐缩短。这是因为分区数量越多，每个分区需要处理的集合数量就越少。由于各分区并行处理，因此分区数量越多，响应时间就越短。由于集合是随机分配到各个分区的，因此各分区的集合数量在期望上相同。所有分区处理完成后，会对各分区的前 k 个结果进行归并排序。然而，与整体运行时间相比，归并开销可以忽略不计。

除了平均响应时间外，图 7a 还报告了一个分区在细化和后处理阶段的平均工作量百分比。一个有趣的观察结果是，随着分区数量的增加，后处理的百分比.....

处理中的集合变得更小。这是因为，当分区数量较多时，所有分区在单位时间内考虑的集合数量也会增加，并且 θ_{lb} 会快速增长。因此，iUB- Filter 具有更强的剪枝能力。

元素相似度阈值： 在图 7b 中，我们固定 $k = 10$（分区数量）为 10，并改变 α 的值。我们观察到，α 越高，查询的平均响应时间越短。较小的 α 意味着词汇表中有更多元素可能会被纳入与元素查询的匹配，因此算法会考虑更多候选集。虽然精炼过滤器的剪枝能力与 α 的值无关，但图匹配的成本会随二分图中边的数量增加而上升：较小的 α 值会导致图中边的数量增多，因此匹配时间也会更长。图 7d 还显示，增大 α 会使 KOIOS 的内存占用略有增加。这是因为增大 α 后，令牌流会变短，候选集的数量也会减少。我们最终得到较小的 θ_{lb} 值，从而使更多集合进入后处理阶段；这会产生更大的后处理数据结构，进而增加内存占用。

结果集基数： 在图 7c 中，我们将分区数固定为 10 和 $\alpha = 0.8$，并改变 k 的值。对于图中给定的 k 值，每个分区都会执行一次 top-k 搜索，并合并各分区返回的最终结果，即对于 $k = 50$，我们合并来自十个分区的十个 top-50 列表。随着 k 增大，平均响应时间反而降低，这一观察结果有悖直觉，因为较高的 k 意味着较低的 θ_{lb}，以及 iUB-Filter 较弱的剪枝能力。然而，响应时间之所以降低，是因为随着 k 增大，平均后处理工作量减少。

通过考察被纳入考虑的集合数量以及进入后处理阶段的集合数量，可以解释响应时间和后处理工作量的变化。当 k 较大时，由于 iUB-Filter 会剪除大量集合，进入后处理阶段的集合最终会出现在 top-k 结果中，其中许多集合会通过 EM-Early-Terminated-Filter 进行过滤，并迅速添加到 top-k 结果中。

IX. 相关工作

***(*模糊*)*集合相似性搜索** 集合相似性搜索领域的大多数研究都考虑词元集合上的包含度和 Jaccard 等语法度量，并采用阈值搜索 [9], [35]。为避免计算集合之间的成对相似度，常见做法是采用过滤—验证范式，其中的核心步骤是前缀过滤 [36], [36],[37]。PPJoin 在前缀过滤的基础上加入位置过滤技术，利用词元顺序信息进一步缩小候选集合 [38]。此外，也有基于划分的搜索研究，其目标是对数据进行划分以实现更快的搜索，例如 SSJoin 和 GreedyPlus [39], [40]。

表连接搜索 大多数现有的连接搜索技术都考虑等值连接，并使用属性值集合重叠度的（归一化）基数作为可连接性度量 [7], [8], [10], [41]–[43]。JOSIE 结合了集合相似性搜索领域的过滤技术，利用普通重叠 [8]解决连接搜索问题。获取近似连接搜索结果的一种常见方法，是使用最小哈希等哈希函数生成集合签名，并据此构建局部敏感哈希索引 [7], [10], [20]。

语义技术 PEXESO 是一种基于普通重叠扩展的基于阈值的集合相似性搜索技术 [44]。在 PEXESO 中，如果两个元素的相似度高于用户指定的阈值，就视为“匹配”。集合相似度定义为查询集中至少有一个匹配元素的元素数量，再除以查询基数。也就是说，元素之间可以形成多对多匹配，因此普通重叠无法表示为所提出度量的特例。SEMA-JOIN 将连接列中的两组值作为输入，并预测连接关系（基于单元格值的多对一连接） [14]。为此，SEMA-JOIN 会找出使成对语义相关性的总和最大的连接关系。该方法依赖大型表格语料库（来自网络的 1 亿张表），并将统计共现用作语义相关性的代理指标。其基本思路是：如果两个值在语料库的同一行中存在显著的统计共现（行共现分数），那么它们就可以进行语义连接（例如，GE 和通用电气）。此外，值对还必须在语义上兼容，即它们应在语料库的同一列中共现（列级共现）。最后，为连接两列，SEMA-JOIN 力求最大化成对列级和行级相关性的总和。与 KOIOS 不同，KOIOS 解决的是查找可与查询集连接的集合这一搜索问题，而 SEMA-JOIN 则在发现之后寻找连接列元素的最佳方式。

X. 结论

我们定义了语义重叠度量。为了解决 top-k 语义重叠搜索问题，我们提出了 KOIOS，这是一种精确且高效的过滤验证框架，采用了功能强大且易于更新的过滤器，可将图匹配计算量降低至候选集的 5% 以下。我们通过在四个不同数据集上进行实验，证明了 KOIOS 具有较短的响应时间和较低的内存占用。在未来的工作中，我们计划将语义重叠扩展到具有多对一映射的实例，以涵盖查询中的噪声或拼写变体。例如，United States of America 和 United States 均可映射到 USA，并具有相同的相似度。

致谢

本研究部分由奥地利科学基金（FWF）项目 P34962 资助。为实现开放获取，作者已对本次投稿所产生的任何作者接受稿版本采用 CC BY 公共版权许可协议。本研究还部分获得美国国家科学基金会项目 2107050 的资助。

参考文献

[1] M. Hadjieleftheriou、X. Yu、N. Koudas 和 D. Srivastava，“哈希样本：集合相似度选择查询的选择率估计器”，PVLDB，第 1 卷，第 1 期，第 201–212 页，2008 年。[2] P. Wang 和 Y. He，“Uni-detect：表格自动错误检测的统一方法”，载于 SIGMOD。ACM，2019 年，第 811–828 页。[3] M. Yu、G. Li、D. Deng 和 J. Feng，“字符串相似度搜索与连接：综述”，《计算机科学前沿》，第 10 卷，第 3 期，第 399–417 页，2016 年。[4] C. Ge、Y. Li、E. Eilebrecht、B. Chandramouli 和 D. Kossmann，“面向大数据分析的推测式分布式 CSV 数据解析”，载于 SIGMOD。ACM，2019 年，第 883–899 页。[5] G. Papadakis、M. Fisichella、F. Schoger、G. Mandilaras、N. Augsten 和 W. Nejdl，“实体解析过滤技术基准测试”，载于 ICDE，2023 年，即将发表。[6] Y.-L. Chen、J.-J. Wei、S.-Y. Wu 和 Y.-H. Hu，“一种从 sci/ssci 数据库检索文档的相似度方法”，《信息科学杂志》，第 32 卷，第 5 期，第 449–464 页，2006 年。[7] E. Zhu、F. Nargesian、K. Q. Pu 和 R. J. Miller，“LSH 集成：互联网规模域搜索”，PVLDB，第 9 卷，1196，2016 12，1185 no. pp. – 。[8] E. Zhu、D. Deng、F. Nargesian 和 R. J. Miller，“JOSIE：用于在数据湖中查找可连接表的重叠集合相似度搜索”，载于 SIGMOD，2019 年，第 847–864 页。[9] W. Mann、N. Augsten 和 P. Bouros，“集合相似度连接技术的实证评估”，PVLDB，第 9 卷，第 9 期，第 636–647 页，2016 年。[10] R. C. Fernandez、J. Min、D. Nava 和 S. Madden，“Lazo：一种基于基数的 Jaccard 相似度与包含度联合估计方法”，载于 ICDE，2019 年，第 1190–1201 页。[11] J. Wang、G. Li 和 J. Fe，“Fast-Join：一种高效的、基于模糊令牌匹配的字符串相似度连接方法”，载于 ICDE，2011 年，第 458–469 页。[12] J. Wang、G. Li 和 J. Feng，“将字符串相似度连接扩展至容错模糊令牌匹配”，《ACM 数据库系统汇刊》，第 39 卷，第 1 期，第 7:1–7:45 页，2014 年。[13] D. Deng、A. Kim、S. Madden 和 M. Stonebraker，“SILKMOTH：一种在最大匹配约束下查找相关集合的高效方法”，PVLDB，第 10 卷，第 10 期，第 1082–1093 页，2017 年。[14] Y. He、K. Ganjam 和 X. Chu，“SEMA-JOIN：利用大型表语料库连接语义相关表”，PVLDB，第 8 卷，第 12 期，第 1358–1369 页，2015 年。[15] J. R. Edmonds 和 R. M. Karp，“网络流问题算法效率的理论改进”，《ACM 杂志》，第 19 卷，第 2 期，第 248–264 页，1972 年。[16] G. Navarro，“近似字符串匹配导览”，《ACM 计算调查》（CSUR），第 33 卷，第 1 期，第 31–88 页，2001 年。[17] I. F. Ilyas、G. Beskales 和 M. A. Soliman，“关系数据库系统中 Top-k 查询处理技术综述”，《ACM 计算调查》，第 40 卷，第 4 期，第 11:1–11:58 页，2008 年。[18] V. V. Vazirani，《近似算法》。施普林格，2001 年。[19] J. Johnson、M. Douze 和 H. J´egou，“利用 GPU 进行十亿级相似度搜索”，《IEEE 大数据汇刊》，第 7 卷，第 3 期，第 535–547 页，2019 年。[20] A. Z. Broder，“文档的相似度与包含度”，载于 SEQUENCES，B. Carpentieri、A. D. Santis、U. Vaccaro 和 J. A. Storer 编，1997 年，第 21–29 页。[21] P. Agrawal、A. Arasu 和 R. Kaushik，“容错集合包含的索引编制”，载于 SIGMOD，A. K. Elmagarmid 和 D. Agrawal 编。ACM，2010 年，第 927–938 页。[22] D. Deng、G. Li、J. Feng 和 W. Li，“带编辑距离约束的 Top-k 字符串相似度搜索”，载于 ICDE，2013 年，第 925–936 页。[23] J. Wang、G. Li、D. Deng、Y. Zhang 和 J. Feng，“一石二鸟：面向 Top-k 和基于阈值的字符串相似度搜索的高效分层框架”，载于 ICDE，2015 年，第 519–530 页。[24] J. Munkres，“指派问题与运输问题的算法”，1957 年。[25] Z. Galil，“求解图中最大匹配的高效算法”，第 18 卷，第 1 期，第 23–38 页，1986 年。[26] M. L. Fredman 和 R. E. Tarjan，“斐波那契堆及其在改进网络优化算法中的应用”，《ACM 杂志》，第 34 卷，第 3 期，第 596–615 页，1987 年。[27] P. K. Agarwal 和 R. Sharathkumar，“具有度量代价和几何代价的二分图匹配近似算法”，载于 STOC，D. B. Shmoys 编。ACM，2014 年，第 555–564 页。

[28] A. Joulin、E. Grave、P. Bojanowski 和 T. Mikolov，《高效文本分类的技巧集》，载于 EACL，2017 年，第 427–431 页。[29] H. Jelodar 等，《基于语义学者挖掘和会议出版物主题建模的推荐系统》，《Soft Computing》，第 24 卷，第 1–30 页，2020 年。[30] F. Nargesian、E. Zhu、K. Q. Pu 和 R. J. Miller，《开放数据上的表联合搜索》，PVLDB，第 11 卷，第 7 期，第 813–825 页，2018 年。[31] J. M. Banda、R. Tekumalla、G. Wang、J. Yu、T. Liu、Y. Ding 和 G. Chowell，《用于开放科学研究的大规模 COVID-19 Twitter 讨论数据集——一项国际合作》，CoRR，第 abs/2004.03688 卷，2020 年。 [在线]。可访问：https://arxiv.org/abs/2004.03688[32] O. Lehmberg、D. Ritze、R. Meusel 和 C. Bizer，《包含时间和上下文元数据的大型公开网页表格语料库》，载于 WWW，2016 年，第 75–76 页。[33] Mcximing，《mcximing/hungarian-algorithm-cpp：匈牙利算法实现的 C 封装》。 [在线]。可访问：https: //github.com/mcximing/hungarian-algorithm-cpp [34] B. Shoshany，《用于高性能科学计算的 C++17 线程池》，CoRR，第 abs/2105.00613 卷，2021 年。[35] L. Jia、L. Zhang、G. Yu、J. You、J. Ding 和 M. Li，《集合相似性搜索与连接综述》，《International Journal of Performability Engineering》，第 14 卷，第 2 期，第 245 页，2018 年。[36] R. J. Bayardo、Y. Ma 和 R. Srikant，《WWW》，C. L. Williamson、M. E. Zurko、P. F. Patel-Schneider 和 P. J. Shenoy 编。ACM，2007 年，第 131 140 [ ] 10 1145 1242572 1242591 – 页。在线。[37] 可访问：https://doi.org/ 。/ 。S. Chaudhuri、V. Ganti 和 R. Kaushik，《数据清洗中用于相似连接的基本算子》，载于第 22 届数据工程国际会议论文集，ICDE 2006，2006 年 4 月 3–8 日，美国佐治亚州亚特兰大，L. Liu、A. Reuter、K. Whang 和 J. Zhang 编。IEEE 计算机学会，2006 年，第 5 页。 [在线]。可访问：https://doi.org/ 10 1109 2006 9 。/ICDE。 。C. Xiao、W. Wang、X. Lin、J. X. Yu 和 G. Wang，《用于近重复检测的高效相似连接》，《ACM Trans. Database Syst.》，第 36 卷，第 3 期，第 15:1–15:41 页，2011 年。 [在线]。可访问：https://doi.org/ 10 1145 2000824 2000825 。/ 。[38] A. Arasu、V. Ganti 和 R. Kaushik，《高效的精确集合相似连接》，载于第 32 届超大型数据库国际会议论文集，2006 年，第 918–929 页。[39] D. Deng、G. Li、H. Wen 和 J. Feng，《一种高效的基于分区的精确集合相似连接方法》，《Proceedings of the VLDB Endowment》，第 9 卷，第 4 期，第 360–371 页，2015 年。[40] D. Deng、C. Yang、S. Shang、F. Zhu、L. Liu 和 L. Shao，《Lcjoin：通过列表交叉切分实现集合包含连接》，载于 ICDE，2019 年，第 362–373 页。[41] A. Bogatu、A. A. A. Fernandes、N. W. Paton 和 N. Konstantinou，《数据湖中的数据集发现》，载于 ICDE，2020 年，第 709–720 页。[42] S. Castelo、R. Rampin、A. S. R. Santos、A. Bessa、F. Chirigati 和 J. Freire，《Auctus：用于数据发现与扩充的数据集搜索引擎》，PVLDB，第 14 卷，第 12 期，第 2791–2794 页，2021 年。[43] Y. Dong、K. Takeoka、C. Xiao 和 M. Oyamada，《数据湖中的高效可连接表发现：一种基于高维相似度的方法》，载于 ICDE，2021 年，第 456–467 页。[44]