Efficient Subgraph Skyline Search Over Large Graphs

Weiguo Zheng
Peking University
Beijing, China
zhengweiguo@pku.edu.cn

Lei Zou*
Peking University
Beijing, China
zoulei@pku.edu.cn

Xiang Lian
University of Texas - Pan American
Edinburg, TX 78539, USA
lianx@utpa.edu

Liang Hong
Wuhan University
Wuhan, China
hong@whu.edu.cn

Dongyan Zhao
Peking University
Beijing, China
zhaody@pku.edu.cn

ABSTRACT

Subgraph search is very useful in many real-world applications. However, users may be overwhelmed by the masses of matches. In this paper, we propose subgraph skyline search problem, denoted as S^3, to support more complicated analysis over graph data. Specifically, given a large graph G and a query graph q, we want to find all the subgraphs g in G, such that g is graph isomorphic to q and not dominated by any other subgraphs. In order to improve the efficiency, we devise a hybrid feature encoding incorporating both structural and numeric features. Moreover, we present some optimizations based on partitioning strategy. We also propose a skylayer index to facilitate the dynamic subgraph skyline computation. Extensive experiments over real dataset confirm the effectiveness and efficiency of our algorithm.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications

Keywords

Subgraph Skyline; Feature Encoding; Skylayer

1. INTRODUCTION

Due to the schema-relaxable nature [12], graph has attracted increasing attention these years. A lot of real-world data (e.g., social network [17], knowledge graph [5], heterogenous information network [22], and semantic web [26]) can be represented by graph model. As we know, various types of researches over graphs have been investigated, such as shortest path query [2], subgraph search [24], and reachability query [6]. However, effectively and efficiently conducting advanced analysis on graphs, particularly subgraph skyline analysis as will be addressed in this paper, remains an open problem.

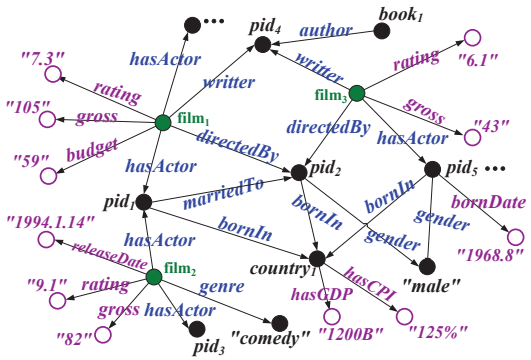


Figure 1: An example of knowledge graph.

As a well-known research problem, subgraph search is meaningful and useful in many applications. For example, answering SPARQL queries in Semantic Web is actually equivalent to conducting subgraph isomorphism match over graphs [26]. However, users may be overwhelmed by the enormous matching results of some queries. Owing to the different requirements in varieties of applications, it is nontrivial to design a generic function to measure the "goodness" of these matches. In this paper, we propose subgraph skyline (Def. 5) on large graphs. To the best of our knowledge, there is no existing study to address this problem.

A knowledge graph is a heterogeneous open-domain resource that integrates lots of information in various fields, such as person, sport, movie, music and so on. Fig. 1 shows a small fraction of a knowledge graph, where the hollow vertices and their adjacent attributes are numeric vertices and numeric attributes, respectively. Using a knowledge graph, we can conduct many interesting subgraph skyline analyses as follows.

Motivating Example 1. For example, we may find excellent NBA player partners over the knowledge graph. Specifically, one player is a "guard" with excellent techniques

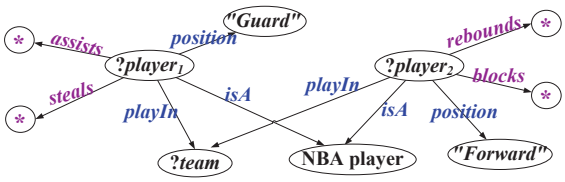


Figure 2: Excellent basketball partner analysis.

*corresponding author: Lei Zou, zoulei@pku.edu.cn

CIKM'14, November 03 - 07 2014, Shanghai, China.

http://dx.doi.org/10.1145/2661829.2662037.

in "assists" and "steals". The other one is a "forward" with excellent techniques in "rebounds" and "blocks". What is more, these two players are expected to serve in the same team. This query can be represented by the graph in Fig. 2, where $player_1$ is a "guard" and $player_2$ is a "forward". The vertices labeled with ' * ' are their respective technical statistics, i.e., the numeric attributes. That is, we want to find all the partners who are no worse than other partners in terms of these attributes. Answering this query over the knowledge graph can find meaningful results. More analyses using real NBA game records will be reported in Section 6.

Motivating Example 2. As another example, with the knowledge graph we can also explore the American excellent actors/actresses who are singers as well. Specifically, the gross of the film that the actor/actress starred in and the copies of his/her album are expected to be large. Fig. 3 illustrates this query graph. In general, subgraph search may return lots of matches without considering the numeric values. Hence, subgraph skyline analysis is useful and interesting over knowledge graph.

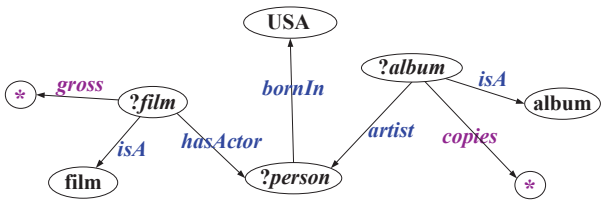


Figure 3: Versatile artist analysis.

Motivated by the examples above, we propose the problem of **S**ubgraph **S**kyline **S**earch over large graphs (denoted as S^3). Specifically, given a large graph G and a user specified query graph q which has several numeric attributes, an S^3 query returns all the subgraphs in G that are isomorphic to q and not dominated by any other isomorphic subgraphs in terms of numeric attributes in q (formally defined in Def. 5).

There are two possible methods to solve the S^3 problem. The first method is to utilize the traditional skyline techniques designed for relational data. However, the input of these techniques are relational tables, which are non-trivial to update, i.e., data addition, deletion, and refreshing. In comparison, the numeric attributes of S^3 skyline are online extracted from subgraphs. More importantly, due to subgraph isomorphism constraint of S^3 problem, the existing pruning techniques for traditional skyline may not be employed directly. For example, entity v_1 dominates entity v_2 over the numeric attributes specified in the query q. But no subgraph containing v_1 is isomorphic to q. Whereas, entity v_2 is in the skyline without considering v_1 and there exists a subgraph g containing v_2 is isomorphic to q. Thus, the subgraph g should be an answer. Furthermore, the previous methods for skyline join do not consider any structure constraint or the corresponding structural optimization. Hence, these techniques dedicated to the relational skyline computation are oblivious to support S^3 queries.

The other method resorts to the graph-based strategy. Although there have been many studies on subgraph queries [20, 4, 12, 24, 26], none of these works considers the skyline constraint. A naive idea is to enumerate all subgraphs in G that are isomorphic to the query graph q, i.e., obtaining all the candidate subgraphs with the existing techniques. Then, we can check the dominating relationship and return true answers. Obviously, this method is inefficient in terms of response time, because it may generate a large number of intermediate results, that is, many redundant isomorphic subgraphs not belonging to the skyline will be exhausted at the cost of expensive subgraph isomorphism checking.

Considering the observations above, we address three challenges to answer S^3 queries efficiently, and carefully design the corresponding solutions.

Challenge 1: Dynamic skyline computation. To answer S^3 queries, a crucial task is to compute skyline according to the numeric attributes specified in query graphs. In the worst case, we need to exhaust all the numeric entities to find the skyline if there is no any optimization measures. More importantly, when we cannot find any answer with the current skyline entity v (i.e., v does not satisfy the structural constraint), we need to dynamically compute skyline entities, which is computationally expensive.

Challenge 2: Efficient querying on graphs. As discussed above, we need to check the structural constraint before reporting the true answers. In order to improve the time efficiency, we should reduce the search space and avoid the costly subgraph isomorphism checking as much as possible. Thus, it is better to consider the structural feature as well as the numeric feature.

Challenge 3: Reducing expensive storage cost. Since there may be a mass of numeric features (e.g., the numeric attributes) and structural features (e.g., path, tree, and subgraph) especially when the knowledge graph G is very large, we should carefully select these features to enhance the pruning power and organize them in an efficient way so as to reduce the overall storage cost.

In order to tackle these challenges, we propose to partition the data space into grids so that we can compute skyline grid by grid instead of entity by entity. We also carefully devise a hybrid encoding incorporating both structural and numeric features at low storage cost. To achieve better pruning effectiveness, we discuss optimizations on how to adaptively find a good partitioning strategy. Furthermore, we maintain the grids using skylayer that facilitates the dynamic computation of skyline entities. More importantly, exploiting the encoding and partitioning strategies we prune the unpromising grids that cannot generate the true results. Beyond that, we also utilize skylayer index to guide the S^3 query processing. Thus, to a large extent the search space is reduced.

In summary, we make the following contributions.

- We propose the problem of **s**ubgraph **s**kyline **s**earch (denoted by S^3) over large graphs, and gives an efficient method to answer S^3 queries.
- Partitioning the data space into grids, we compute skyline grid by grid instead of entity by entity. More importantly, we discuss optimizations on how to adaptively find a good partitioning strategy and prove that finding an optimal partition is NP-hard.
- We propose a hybrid feature encoding incorporating both the structural and numeric features to enhance the pruning ability. We also maintain grids using skylayer in order to facilitate the dynamic computation of subgraph skyline.
- Extensive experiments over real dataset have demonstrated the effectiveness and efficiency of our method.

2. RELATED WORK

There are two related researches to be reviewed, i.e., skyline computation and subgraph search.

Skyline Computation. Most existing skyline literature focus on multi-dimensional relational data. Their inputs are relational tables. *BNL* [1] is the first proposed to compute the skyline. Instead of going over the entire dataset, Bitmap [16] represents a data point p using an m-bit vector, and compute the skyline points progressively. In order to answer subspace skyline queries, *SUBSKY* [18] converts each multi-dimensional point to 1D values, and index these converted values by a single B-tree. Jin et al. [8] group pairs that share the same subspaces into maximal space index. In practice, this method still suffers from expensive storage cost.

To compute the join-based skyline efficiently, several techniques have been proposed [7, 21, 11]. Jin et al. [7] integrate state-of-the-art join methods, such as sort-merge join and nested loop join, with single-relation skyline algorithm. SFSJ (sort-first-skyline-join) [21] computes the skylines by accessing only a subset of the input tuples. Instead of performing tuple-to-tuple dominance checks, S^2J (skyline-sensitive join) [11] employs a layer/region pruning strategy. There are some other works aiming to compute the join-based skyline, such as FlexPref [10], SKIN [13], and Prefjoin [9].

Notice that an important pruning principle of the existing algorithms is: tuples that do not belong to group skylines [21] cannot contribute to the join skyline. However, the group skyline is very hard to compute in the graph scenario. Furthermore, not being devised for S^3 problem, they do not consider any structural feature to facilitate the query process. In contrast, we integrate numeric pruning with structural pruning based on the grid-based partition of data space.

Subgraph Search. Subgraph search problem has been extensively studied in the past decades [20, 4]. Ullmann [20] and VF2 [4] are the two early efforts to verify the subgraph isomorphism between two graphs. In order to improve the efficiency in subgraph search, most of the proposed algorithms follow filtering-and-verification framework. In the filtering phase, some structural features, including frequent paths [19], trees [23], and subgraphs [3], are chosen as basic index units. Also, some non-feature-based methods are proposed, such as GCodeing [25] and SPath [24]. Most of them employ the neighborhood structures of vertices. Based on the indexes, we can first prune some data graphs that are impossible to be results. Then, we verify each candidate data graph by employing the subgraph isomorphism algorithm, such as Ulluman [20], VF2 [4], and QuickSI [14].

3. SUBGRAPH SKYLINE

In this section, we first formally define the subgraph skyline, and then give a naive method to solve S^3 problem.

3.1 Subgraph Skyline

S^3 runs queries over knowledge data graphs, which is formally defined as follows.

Definition 1. (**Knowledge Graph**). A knowledge graph is defined as $G = (V, E, L)$, where each vertex $v \in V$ represents an entity or a numeric value, each $e = (v_i, v_j) \in E$ represents a directed edge from vertex v_i to vertex v_j, and $L(v)$ (resp. $L(e)$) is the label of vertex v (resp. edge e).

Fig. 1 shows an example of knowledge graph. Note that, if entity v has some numeric attributes, v is called *numeric entity*. Let $v.d$ denote the value on numeric attribute d for entity v.

Definition 2. (**Graph Isomorphism**). Given two subgraphs g_1 and g_2 in graph G, g_1 is graph isomorphic to g_2 *iff* there exists a bijective function $f(.)$ such that (1) for each vertex $v \in g_1$ (excluding the numeric values), $f(v) \in g_2 \wedge L(v) = L(f(v))$; (2) for each $e = (v_i, v_j) \in g_1$, we have $f(e) = (f(v_i), f(v_j)) \in g_2$, and $L(e) = L(f(e))$.

Definition 3. (**Dominant/Equivalent Entity**). Given two numeric entities v_1 and v_2 in a knowledge graph G and their numeric attribute set D, v_1 *dominates* v_2, denoted by $v_1 \prec v_2$, if (1) for each attribute d_i, $v_1.d_i \leq v_2.d_i$, and (2) there exists at least one attribute d_j such that $v_1.d_j < v_2.d_j$. We say v_1 is *equivalent* to v_2, denoted by $v_1 = v_2$, if $v_1.d_i = v_2.d_i$ on each numeric attribute $d_i \in D$.

To facilitate the presentation, let $v_1 \preceq v_2$ denote that entity v_1 dominates or is equivalent to entity v_2.

Definition 4. (**Subgraph Dominating Relationship**). Given two subgraphs g_1 and g_2 in G, g_1 *dominates* g_2 if

- g_1 is graph isomorphic to g_2 without considering numeric values;
- It holds that $v_i \preceq f(v_i)$ for each numeric entity $v_i \in g_1$;
- There exists at least a numeric entity $v_j \in g_1$ such that $v_j \prec f(v_j)$.

where $f(.)$ is the mapping function defined in Def. 2.

Definition 5. (**Subgraph Skyline**). A subgraph $g \in G$ is in the *subgraph skyline,* if g is graph isomorphic to the query graph q and not dominated by any other subgraphs $g' \in G$, on those specified numeric attributes in q.

Subgraph Skyline Search Problem (denoted as S^3). Given a large graph G and a query graph q containing numeric attributes, the S^3 problem is to compute the subgraph skyline on G.

In real applications, the query graphs are given by users, and the numeric attributes can be specified according to the ad-hoc requirements.

3.2 A Naive Method

Before giving the naive method, we briefly review the bitmap [16] method. Its main idea is representing an object o using m-bit vector. Then we can progressively determine whether o is in the skyline by performing bitwise operations over the corresponding bitmaps.

High-level idea: In the offline phase, we store all the bitmaps of numeric entities. In the online phase, we first compute the candidates for the numeric entities in q, and then find skyline entities from these numeric entities. Finally, we verify the structure constraint to obtain S^3 answers employing bitmaps.

Obviously, this naive method is inefficient, since there is no any guidance to find the skyline vertex. Furthermore, its storage cost is $O(n^2 \cdot |D|)$, where n and $|D|$ are the number of numeric entities and numeric attributes, respectively. Hence, we propose an efficient method exploiting hybrid feature encoding in the following section.

4. HYBRID FEATURE ENCODING

4.1 Adaptive Space Partitioning

The rationale of partitioning the data space (i.e., the space consisting of numeric entities) is that: if we can compute the skyline entities grid by grid instead of exhausting them one by one, the time efficiency will improve a lot. Moreover, we can only maintain the numeric encoding of grids rather than all numeric entities, which reduces the storage cost considerably. Thus, we propose to partition the data space into grids (Def. 6) in this paper.

Definition 6. (Grid). Given a data space D consisting of the numeric entities, grids are obtained by partitioning each dimension $d_i \in D$ using hyperplanes.

After partitioning the data space into grids, each grid can be represented by its minimal corner (given in Def. 7).

Definition 7. (**Minimal Corner**). Given a grid B in the multi-dimensional data space $D = \{d_1, \ldots, d_{|D|}\}$, its minimal corner is the point whose value on each dimension $d_i \in D$ is the minimum.

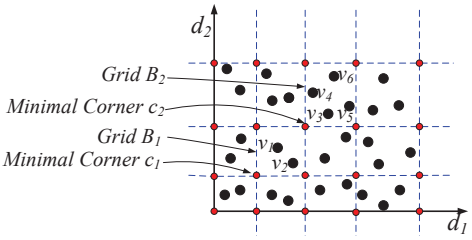


Figure 4: A partition of the data space.

Example 1. *As shown in Fig. 4, the 2-dimensional space is partitioned into 12 grids. Each grid can be represented by its minimal corner (i.e., the bottom left point of the grid). For instance, the grid B_2 consisting of entities v_3, v_4, v_5, and v_6 can be represented by point c_2.*

Utilizing the minimal corners, we can compute the skyline entities grid by grid instead of point by point. Before presenting the technical details, we introduce the definition of "strict dominance" relationship.

Definition 8. (**Strict Dominance**). Given two corners c_1 and c_2 and a specified numeric attribute set D, we say c_1 strictly dominates c_2, denoted as $c_1 \prec c_2$, if on each numerical attribute $d_i \in D$, $c_1.d_i < c_2.d_i$ holds.

The strict dominance imposes more stringent restrictions upon two objects. Clearly, if $c_1 < c_2$, it holds that $c_1 \prec c_2$. Moreover, we can derive the following properties.

Lemma 1. *Given a grid B and its minimal corner c, it holds that c dominates each numeric entity $v \in B$.*

PROOF. It is straightforward according to the definition of minimal corner (Def. 7). □

Lemma 2. *Given two minimal corners, c_1 of grid B_1 and c_2 of grid B_2, if $c_1 < c_2$, then all the entities in B_1 dominate the minimal corner c_2.*

PROOF. This lemma can be proved by using the contradiction. Assume that $v_1 \not\prec c_2$, where $v_1 \in B_1$. Thus, there must exist a dimension d_i such that $v_1.d_i > c_2.d_i$, which contradicts with the prescriptive regular partition (i.e., partition each dimension respectively). □

With the two lemmas above, we can obtain a useful theorem, which is the critical principle of pruning.

THEOREM 1. *Given two minimal corners, c_1 of grid B_1 and c_2 of grid B_2, if $c_1 < c_2$, then all the entities in the grid B_1 dominate that in B_2.*

PROOF. Assume v_1 and v_2 are two entities in grids B_1 and B_2, respectively. According to Lemmas 1 and 2, we have $v_1 \prec c_2$ and $c_2 \prec v_2$. Thus, it holds that $v_1 \prec v_2$. □

It is obvious that given the number of grids, there are many different partitions which may result in different effects. We will discuss optimizations on how to adaptively find a good partitioning strategy later in Section 4.3.

4.2 Space Partition Based Feature Encoding

Based on the space partition, we present the hybrid feature encoding which consists of structural features (Section 4.2.1) and numeric features (Section 4.2.2).

4.2.1 Structural Feature Encoding

Structural encoding for entities. Provided that the entities in q and G are encoded in the same method, we can check the match according to their encodings.

Local Structural Encoding. Since bit operation (e.g., AND, OR, and NOT) is easy and time efficient, we hash the local structure of an entity v to a bitstring, denoted by $lbStr(v)$, which is similar to but different from the previous work [26]. The differences are listed as follows.

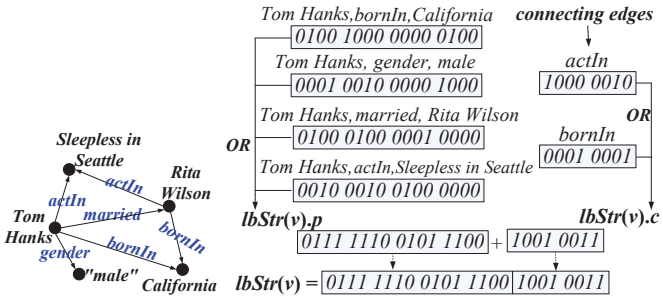
- We integrate the adjacent edge and the corresponding neighbor vertex together (denoted by *1-hop* path) instead of considering them separately. In this way, it reduces false drops compared with the previous method.
- We utilize more structural information, i.e., connecting edge (Def. 9), to improve the pruning ability.

Definition 9. (Connecting Edge). Given a vertex v_1 and its two neighbor vertices v_2 and v_3, the edge $e = (v_2, v_3)$ between v_2 and v_3 is v_1's connecting edge.

The bitstring of v's local structure $lbStr(v)$ has two parts: $lbStr(v).p$ and $lbStr(v).c$, where the first part $lbStr(v).p$ denotes the *1-hop* path labels (v's adjacent edge label combining the corresponding neighbor vertex label), and the second part $lbStr(v).c$ denotes the connecting edge labels.

Bitstring Generation. Given a neighbor vertex v' of v and the corresponding edge e between v and v', we combine $e.Label$ and $v'.Label$ together to get the label ($p.Label$) of v's *1-hop* path. We generate the bitstring for $p.Label$, i.e., $lbStr(v).p$ ($|lbStr(v).p| = M_1$). We utilize m different hash functions to set m out of M_1 bits in $lbStr(v).p$ to be '1'. All the other bits are set to be '0'. Similarly, we can obtain the other part $lbStr(v).c$.

Example 2. *Fig. 5(a) shows the local structure of entity v (Tom Hanks). It has 4 adjacent edges and 2 connecting edges. As shown in Fig. 5(b), $lbStr(v)$ consists of $lbStr(v).p$ and $lbStr(v).c$, which are the unions of the bitstrings for v's 1-hop paths and connecting edges, respectively.*



(a) local structure (b) local structural encoding

Figure 5: The local structure of v and its encoding

Global Structural Encoding. Given a numeric entity v in graph G, we collect the set of numeric entities $NS_h(v)$, such that $dist(v, v') \leq h$ for each $v' \in NS_h(v)$, where $dist(v, v')$ is the shortest path distance between v and v'.

We first generate the bitstring for $v_i.Id$ ($v_i \in NS_h(v)$), denoted as $bStr(v_i)$ ($|bStr(v_i)| = M_3$). Then, we utilize m different hash functions to set m out of M_3 bits in $bStr(v_i)$ to be '1'. All other bits are to be '0'. Then $gbStr(v)$ is generated by performing bitwise OR operation over the bitstrings of $v_i \in NS_h(v)$, i.e., $gbStr(v) = bStr(v_1)| \ldots |bStr(v_m)$.

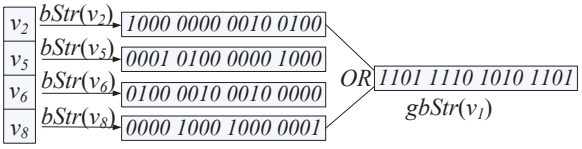


Figure 6: The global numeric encoding of entity v_1

Example 3. *Provided that $NS_3(v_1) = \{v_2, v_5, v_6, v_8\}$, as shown in Fig. 6, we generate the bitstring for each entity in $NS_3(v_1)$, and then perform the OR bitwise operation over these bitstrings to obtain $gbStr(v_1)$.*

Structural encoding for grids. The intuition for encoding grids lies in that we can check the minimal corner of a grid B before accessing the entities in B. If a minimal corner cannot match any entity in the query graph, we can safely prune the whole grid without exploring the corresponding entities.

The structural encoding for a grid B, i.e., $lbStr(B)$, is formed by performing the bitwise OR operation over the local structural bitstrings of the entities in B. Formally, $lbStr(B) = lbStr(v_1)| \ldots |lbStr(v_m)$, where $v_i \in B$ ($1 \leq i \leq m$). Then we have the following theorem.

THEOREM 2. *If $lbStr(u)$ & $lbStr(B) \neq lbStr(u)$, any numeric entity in grid B does not match u, where u is a numeric entity in query graph q.*

PROOF. It is straightforward according to definition of $lbStr(B)$. □

Based on Theorem 2, we first examine the structural encoding of a grid B before exploring the entities in B. For a query entity u in q, if $lbStr(u)\&lbStr(B) \neq lbStr(u)$ we can determine that all the entities in grid B do not match u. Thus, the whole grid B is filtered out.

4.2.2 Numeric Feature Encoding

The previous method Bitmap [16] can progressively determine whether a point is in the skyline. However, as discussed earlier, it is costly to maintain the Bitmap in terms of storage cost.

The numeric encoding $nbStr(B)$ in this paper is distinct from Bitmap [16]:

- The size of $nbStr(B)$ is smaller than Bitmap, i.e., $K \cdot |D| < n \cdot |D|$, where K, n, and $|D|$ are the number of grids, numeric entities and attributes, respectively.
- The encoding technique of $nbStr(B)$ is simpler than Bitmap.
- The usage of our numeric encoding is different from Bitmap. We propose an invalid vector to support the computation of valid skyline objects (Def. 13).

We maintain a bitstring for each grid B, i.e., $nbStr(B)$, which consists of $|D|$ parts: $nbStr(B).d_1, \ldots, nbStr(B).d_{|D|}$.

Take an example, we generate the numeric encoding for grid B on dimension d_i, i.e., $nbStr(B).d_i$ ($|nbStr(B).d_i| = K$), where K is the number of grids. For each grid B_j if B_j's value on dimension d_i is better than B's value on dimension d_i, i.e., $B_j.d_i < B.d_i$, the jth bit of $nbStr(B).d_i$ is set to 1, otherwise it is set to 0.

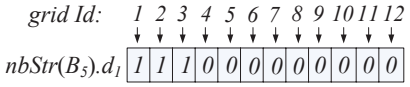


Figure 7: The numeric encoding of B_5

Example 4. *Consider the partition in Fig. 4. The numeric encoding of grid B_5 is shown in Fig. 7.*

Obtaining the numeric encoding for grid B, we can determine whether B is in the skyline on dimensions $d_1, \ldots, d_m$ ($m \leq |D|$). Let $X = nbStr(B).d_1$ & ... & $nbStr(B).d_m$, where '&' represents the bitwise AND operation. If the result of the operation, X, is a non-zero value, we can conclude that there must be a certain grid strictly dominates B.

As defined in S^3 problem, a true answer should satisfy both the skyline and structural constraints. Thus, even if the numeric entities are in the skyline, the corresponding subgraph is not a true answer on condition that the structural constraint is not satisfied. In this case, entities that are not in the skyline originally may become skyline entities without considering their dominating entities. Hence, we utilize an invalid skyline vector, ISB, to support the dynamic skyline queries. More details will be discussed in Section 5.2.1.

4.3 Optimization on Space Partitioning

As presented in Section 4.1, different partitions may result in different pruning effects. In this subsection, we discuss what a good partition is and how to partition the data space efficiently.

4.3.1 What is a Good Partition?

Obtaining the minimal corners in the skyline, we need to check whether all the entities in the corresponding grids are valid skyline entities (Def. 13). Thus, the less false positives are generated, the better the partition will be.

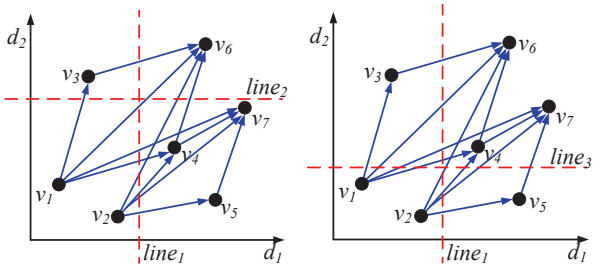
Observation 1: A good partition should generate less false positives.

To make it clear, we introduce "dominating edge" between two entities in the data space.

Definition 10. (Dominating Edge). Given two entities v_1 and v_2 in the data space, if v_1 dominates v_2, a directed edge starting from v_1 to v_2 is added. This directed edge is called a dominating edge, denoted as $e(v_1, v_2)$.

Example 5. *As shown in Fig. 8(a),entity v_1 dominates entity v_3, there is a directed edge between v_1 and v_3. Similarly, we can obtain other dominating edges.*

Fig. 8 shows two different partitions for a data space. Although both these two partitions destroy 10 dominating relations, the partition in Fig. 8(b) is better than that in Fig. 8(a), because the grid consisting of v_1 and v_2 only prunes one entity (i.e., v_6) in Fig. 8(a). In contrast, the grid consisting of v_1 and v_2 in Fig. 8(b) can prune three entities, i.e., v_4, v_6, and v_7.



(a) less common destructions (b) more common destructions

Figure 8: Two partitions for a data space.

Therefore, we find that in order to prune an entity v, all the incoming dominating edges of v should be destroyed on every attribute. Thus, we have the following observation.

Observation 2: An effective partition should destroy as many common dominating edges as possible.

For instance, the partition lines in Fig. 8(a) destroy 2 dominating edges in common. The partition lines in Fig. 8(b) destroy 6 dominating edges in common. Therefore the latter partition has stronger pruning power than the former one.

In real applications, we can determine the number of grids based on the storage cost. For simplicity, we assume that the number of partition lines on each dimension is given in this discussion. According to the aforementioned observations, we define the "maximum common partition" below.

Definition 11. (**Maximum Common Partition**) (denoted as **MCP**). Given the number of partition lines on each dimension and a partition P, the number of dominating edges destroyed on all dimensions is denoted as $|P|$. If there exist no other partition P' such that $|P| < |P'|$, partition P is the maximum common partition.

Given a set of data, it is better to find the maximum common partition to obtain the strongest pruning power. However, we have proven that it is an NP-hard problem.

THEOREM 3. *Given a set of data, computing the maximum common partition is NP-hard.*

PROOF. (Proof sketch). We can reduce the minimum set cover problem to an instance of MCP. Provided that $D = \{d_1, d_2\}$, the number of partition lines on d_2 dimension is 1. We construct a dataset in which all dominating edges are destroyed on dimension d_2. Then any destroyed dominating edges on dimension d_1 are the common destroyed dominating edges on both dimensions d_1 and d_2.

Thus, we only focus on d_1. Assume that there are K partition lines on dimension d_1. It is obvious that any given minimum set cover instance can be reduced to an instance of MCP. The theorem can be reached. □

4.3.2 How to Partition

Since computing the maximum common partition is NP-hard, we design an efficient greedy algorithm with the time complexity $O(n \cdot k \cdot |D|)$, where k is the average number of partition lines on each dimension.

Assume that there are n numeric entities (v_1, ... v_n), and $v_1.d_i \leq v_2.d_i \leq \ldots \leq v_n.d_i$. The projections of all dominating edges on dimension d_i form a universe set E. A partition line located between $v_j.d_i$ and $v_{j+1}.d_i$ may destroy a number of dominating edges. These destroyed dominating edges form a subset of E. Thus, computing the MCP on dimension d_i corresponds to selecting a predefined number of subsets that cover as many elements as possible.

Example 6. *Considering Fig. 8(a), there are 6 optional partition lines ($k_1, \ldots, k_6$). The partition line k_1 located between $v_1.d_1$ and $v_3.d_1$ destroys 4 dominating edges, i.e., $e(v_1, v_3)$, $e(v_1, v_6)$, $e(v_1, v_7)$, and $e(v_1, v_4)$. Hence, the corresponding subset is $E_1 = \{e(v_1, v_3), e(v_1, v_6), e(v_1, v_7)$, and $e(v_1, v_4)\}$. Similarly, we can also obtain the other 5 subsets.*

The main idea is that we greedily select k_i sets on dimension d_i, and store the union of these selected sets, denoted as U. When considering the next dimension d_j, we greedily select k_j sets whose intersections with U is the largest, and update set U. The intuition is that: Since it requires destroying more common edges (edges that are destroyed on all dimensions), we should intersect the selection union on dimension d with the current selected sets U. More details are presented in Alg. 1.

Specifically, the algorithm consists of three steps.

- We obtain the family of dominating edge sets on each dimension d_i, denoted by F_i (lines 1-2).
- We select a dimension d_i to start. The current largest set $E \in F_i$ is selected. Then remove all elements $e \in E$ from the remaining sets, and select the new largest set from the updated sets in next loop. The selection process terminates until k_1 sets have been selected. The union of these selected sets is denoted as U (lines 3-9).
- Consider the next dimension d_j. Select the set $E \in F_j$ such that the intersection of E and U (i.e., $|E \cap U|$) is the largest (line 13). Then remove all elements $e \in E$ from the remaining sets (lines 14-15). The selection process terminates until k_i sets have been selected. Use the intersection of selected sets R and U, i.e., $U \cap R$ to update U. The algorithm stops when all dimensions have been considered (lines 10-18).

Time complexity. Assume there are n distinct numeric entities. The number of different partition lines on a dimension is $(n-1)$ at most. For each partition line, it is trivial to obtain the corresponding destroyed dominating edges. Hence, the time complexity of the first step is $O(n \cdot |D|)$. In the second step, the time complexity of selecting the largest sets is $O(n)$. Thus, the time complexity of selecting k sets is $O(n \cdot k)$. Similarly, the time complexity of computing on each dimension is $O(n \cdot k)$. Therefore, the time complexity of Alg. 1 is $O(n \cdot k \cdot |D|)$.

Algorithm 1 Greedy Partition

Input: A set of numeric entities; Dimensions $D = \{d_1, \ldots d_{|D|}\}$; The number of partition lines on each dimension $k_1, \ldots, k_{|D|}$;
Output: A partition of the data space.

```
1: for Each dimension d_i ∈ D do
2:    Compute the family of dominating edge sets F_i
3: U ← ∅ , s ← 0
4: while s < k_1 do
5:    Select the largest set E ∈ F_1
6:    for Each E_j ∈ F_1 do
7:       Remove all the elements e ∈ E from E_j
8:    U ← U ∪ E_j , s ← s + 1
9:    Remove E_j from F_1
10: for Each dimension d_i ∈ D ∧ i ≠ 1 do
11:   s ← 0, R ← ∅
12:   while s < k_i do
13:      Select the set E ∈ F_i ∧E ∩ U is the largest
14:      for Each E_j ∈ F_i do
15:         Remove all the elements e ∈ E from E_j
16:      R ← R ∪ E_j , s ← s + 1
17:      Remove E_j from F_i
18:   U ← U ∩ R
```

5. GRID INDEX AND SUBGRAPH SKYLINE QUERY

In this section, we present the index designed for grids first, and then give the query processing of subgraph skyline based on the feature encoding and grid index.

5.1 Grid Index - Skylayer

To answer S^3 queries efficiently, a critical task is to obtain the skyline entities dynamically before checking the expensive subgraph isomorphism.

Since we partition the numeric dataset to grids and use grids to represent the entities, the computation of skyline entities is conducted over these grids. In general, we need to exhaust these grids to obtain the skyline. Next, we propose skylayer to avoid traversing all the grids at the time of computing skyline.

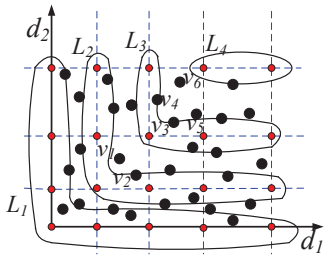


Figure 9: An example of skylayer.

Definition 12. (Skylayer). Given a set of minimal corners, we organize the corresponding grids in the several layers such that every minimal corner c_i does not strictly dominate any other minimal corner c_j in the same layer.

Example 7. *Fig. 9 shows a skylayer example of the dataset in Fig. 4. There are four layers: $L_1 \sim L_4$, where L_i maintains the grids that do not strictly dominate each other.*

Given the set of minimal corners, C, its skylayer is easy to be built by recursively employing any existing skyline algorithms [1, 18, 15]. Specifically, we compute the skyline grids over C to obtain the first layer L_1. Then we remove these grids (in L_1) from C to obtain a new set C', i.e., $C' = C - L_1$. Recursively, we can compute the new skyline grids over C' to get L_i $(i > 1)$ until $C' = \emptyset$.

Lemma 3. *Each grid B_1 in the ith $(i > 1)$ layer must has at least one grid B_2 in the $(i-1)$th layer such that B_2 strictly dominates B_1, i.e., $B_2 < B_1$*

Proof. It can be proved by contradiction. Assume that one grid B in ith layer has no dominating grids in the $(i-1)$th layer. Thus, grid B should have been discovered in the $(i-1)$th layer, which contradicts the assumption. □

Given a numeric entity v and the grid B that v belongs to, if v is in the skyline, the minimal corner c of B must be in the first skylayer. Similarly, we have the following lemma.

Lemma 4. *Any entity v in the layer L_j does not dominate any entity v' in the layer L_i, where $i < j$.*

Proof. It is straightforward according to Def. 12 and Lemma 3. □

Lemma 4 guarantees that accessing the skylayers one by one will not miss any skyline entities.

Obtaining a skyline grid B, we need to compute the skyline entities in B. Notice that, some numeric entities in the query may only involve only a part of the dimensions, i.e., it is a subspace query. Here, we employ the bitmap technique [16] to determine whether entity v $(v \in B)$ is in the skyline.

Different from the work in [16], we generate the bitmaps online instead of maintaining all the bitmaps with expensive storage cost. Moreover, it is probable that not all the entities need to be examined, that is, it may only involve a subset of the entities. Hence, it is unnecessary to generate bitmaps for all the entities. In the offline phase, entities are sorted on each dimension, based on which the bitmap generation is very simple (the generation is similar to the numeric encoding for grids in Section 4.2.2).

5.2 Subgraph Skyline Query

In this section, we give the query algorithm based on feature encoding and grid index aforementioned. Since a query graph may contain one or multiple numeric entities, we deal with these two cases in the following discussion.

5.2.1 Single Numeric Entity Query

Since we have partitioned the numeric dataset to grids, and utilize skylayers to maintain these grids, the query process starts from the skylayers.

Intuitively, it is the simplest case that there is only one numeric entity in query graph q. Alg. 2 presents the details, which has three steps: high-level pruning, skyline computation, and structure verification.

High-level Pruning. Given a query graph q which contains one numeric entity u, we generate the local structural encoding for each vertex in q. Then the high-level pruning (structural pruning and numeric pruning) is performed.

Structural Pruning. Specifically, we first generate the local structural encoding of u is $lbStr(u)$. For each grid B in skylayer L, we check whether u can match B. Based on Thm. 2, if $lbStr(u)\&lbStr(B) \neq lbStr(u)$, we can conclude that any numeric entity in grid B can not match u.

Algorithm 2 Single Numeric Entity Query

Input: A knowledge graph G, the feature encoding for grids and entities of G, and a query graph q containing one numeric entity;
Output: The subgraph skyline in G.

```
1: for Each skylayer L do
2:   for Each grid B in L do
3:     perform structural pruning over B
4:     perform numeric pruning over B
5:     if B is a valid skyline grid then
6:       for Each entity v in B do
7:         perform structural pruning over v
8:         perform numeric pruning over v
9:         if v is a valid skyline entity then
10:          perform structure verification
11:          if a graph g containing v is isomorphic to q
             then
12:            report g as a result
13:            pruning entities and grids
```

Numeric Pruning. If grid B matches u in terms of the structure constraint, we need to check whether grid B is in the *valid skyline* (Def. 13) over the query space.

Definition 13. (Valid Skyline.) An entity v (or grid B) is in the valid skyline *iff* v (or B) satisfies the structural constraint specified in query graph q, and all the entities dominating v (or B) do not satisfy the structural constraint.

For instance, assume that grid B_1 dominates B_2, whereas B_1 does not satisfy the structure constraint, and there exist no other grids dominating B_2. Thus, B_2 is in the valid skyline. Conversely, grid B_1 is an invalid skyline grid.

In order to determine whether grid B is in the valid skyline, we utilize a vector, ISB ($|ISB| = K$), to record the invalid skyline grids. At the beginning, each bit of ISB is set to be 1. When we find an invalid grid, the corresponding bit is set to be 0. If $ISB \,\&\, X = 0$, grid B is a candidate in the valid skyline at the moment, where $X = nbStr(B).d_1 \& \ldots \& nbStr(B).d_m$. Otherwise, B is not in the valid skyline, which indicates that grid B is not need to be explored further.

Skyline Computation. For a valid skyline grid B, we need to compute the valid skyline entities in B. Similar to the computation of grids, we conduct the structural pruning and numeric pruning to filter out the unpromising entities as early as possible.

Structural Pruning. Given a numeric entity v in grid B and the numeric entity u in query graph q, according to the discussion in Section 4.2.1, if $lbStr(u) \& lbStr(v) \neq lbStr(u)$, we can conclude that v does not match u.

Numeric Pruning. If entity v passes the structural pruning, we should determine whether v is a valid skyline entity. As discussed in Section 5.1, we only maintain the sorted entities on each dimension, based on which the bitmaps, $nbStr(v)$, can be generated easily.

Analogous to that of grids, we maintain a vector whose all bits are 1, ISV ($|ISV| = n$), to record the invalid skyline entities. Once we find that a skyline entity is invalid due to the structure constraint, the corresponding bit is set to be 0. If $ISV \& Y = 0$, entity v is in a valid skyline entity candidate at present, where $Y = nbStr(v).d_1 \& \ldots \& nbStr(v).d_m$. Otherwise, v is not in the valid skyline.

Structure Verification. Passing the first two pruning techniques, it requires to verify whether there exists a subgraph containing v that is graph isomorphic to query graph q. The state-of-the-art algorithms such as Ullmann [20] and VF2 [4] can be employed to achieve this verification.

Supposing that we find a subgraph containing v that satisfies both the skyline and structural constraints, we can prune all the entities that are dominated by v. Specifically, the entities in B that are dominated by v and the grids that are strictly dominated by B can be filtered out safely, where entity v belongs to grid B.

It is easy to obtain the grids that are dominated by B based on the numeric encoding of B. Regarding dimension d_i, we reverse each bit of $nbStr(B).d_i$, and set the bit corresponding to grid B_j to be 1 if $B_j.d_i = B.d_i$. Then we acquire the new encoding $nbStr(B)'.d_i$. Let $Z = nbStr(B)'.d_1 \,\&\, \ldots \,\&\, nbStr(B)'.d_m$. The grids corresponding to non-zero bits in Z are dominated by B. Thus, these numeric entities in these grids can be filtered out.

5.2.2 Multiple Numeric Entity Query

In general, there may be multiple numeric entities in query graph q. To handle this case, we propose joint pruning in this subsection. For ease of presentation, we assume that there are two numeric entities u_1 and u_2 in query graph q. The main idea is that: we select one numeric entity u_1 to match and then we compute the candidates for entity u_2 before verifying the structure starting from v_1, where numeric entity v_1 ($v_1 \in G$) is a candidate for u_1.

In order to find the candidate for u_1, we perform structural and numeric pruning which are analogous to that discussed for one single numeric entity as shown in Section 5.2.1.

Obtaining the candidate v_1 for u_1, we need to compute the candidates for u_2. Besides the high-level pruning and skyline computation, we also propose two joint pruning techniques: shortest-path-distance pruning and skyline join pruning.

Shortest-path-distance Pruning. Provided that the shortest path distance between u_1 and u_2, $dist(u_1, u_2)$, is no larger than h, where h is a predefined threshold based on which the global structural encoding is generated. Entity v_2 matches u_2 only if $v_2 \in NS_h(v_1)$. It is easy to determine whether v_2 is in $NS_h(v_1)$ using the global structural encoding of v_1, $gbStr(v_1)$.

If $gbStr(v_1) \& bStr(v_2) \neq bStr(v_2)$, we can conclude that the candidate pair (v_1, v_2) does not match (u_1, u_2), where $bStr(v_1)$ is the bitstring of $v_2.Id$. Since it avoids the costly graph isomorphism checking, the query efficiency may improve a lot.

Skyline Join Pruning. The graph isomorphism algorithm should be invoked to verify these entity pairs that pass all the pruning techniques above. If there exists a subgraph g containing entity pair v_1 and v_2 such that g is graph isomorphic to query graph q, g is in the subgraph skyline. Hence, we prune all the subgraphs that are dominated by graph g.

Actually, instead of enumerating all these pruned subgraphs, we just need to filter out the entity pairs in $dom(v_1)$ and $dom(v_2)$, where $dom(v_1)$ and $dom(v_2)$ represents the entities dominated by v_1 and v_2, respectively. Notice that, since we utilize grids to store entities, the join space is relatively small compared with joining entities directly.

In order to deal with more than two numeric entities, we can take them into consideration one by one. We omit more details due to the space limitation.

6. EXPERIMENTAL STUDY

In this section, we study our proposed method through extensive experiments. Section 6.1 introduces the experiment setting, followed by the effectiveness evaluation and efficiency evaluation in Sections 6.2 and 6.3, respectively.

6.1 Experiment Setup

We use Freebase dataset[1] which integrates *NBA*[2] and *IMDB*[3] to evaluate our method. It contains 12,130,534 vertices, 232,671,328 edges, 7,634,315 numeric entities, and 35 numeric attributes.

Regarding the queries, we generate some query graphs to study the effectiveness of our method. More examples will be present in Section 6.2. In order to study the efficiency, we randomly extract some subgraphs containing numeric entities, and vary the size of these query graphs.

All the experiments were conducted on a PC with 2.9GHz CPU and 16GB main memory running Linux operating system. For comparison, we implement the simple method presented in Section 3.2, denoted as *"Naive"*, which does not employ partition and feature encoding techniques. The partition and encoding based method is denoted as "parCode". Both the two programs were implemented in C++.

6.2 Effectiveness Evaluation

In order to verify the effectiveness of our method, we focus on the case studies of S^3 queries in this subsection, and check whether the results returned by our method are reasonable. To this end, we artificially generate some queries. Here, we present two case studies as follows.

Golden Basketball Partner Finding. As mentioned in Section 1, assume that we want to find one guard and one forward who play in the same team. The guard is expected to be excellent in techniques "assists" and "steals". The forward is expected to be excellent in techniques "rebounds" and "blocks". The query graph is shown in Fig. 2. Fig. 10 presents a subset of the results. As expected, we find several great partners, such as Gasol Pau and Bryant Kobe.

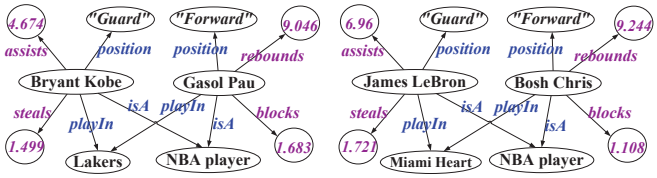


Figure 10: Golden basketball partner finding.

Excellent Versatile Artist Finding. We want to seek American outstanding artist who is a singer and an actor/actress as shown in Fig. 3. Specifically, the gross of the film that the actor/actress starred in and the copies of his/her album are expected to be large. Fig. 11 gives a fraction of the results. For example, Michael Jackson is in the skyline.

6.3 Efficiency Evaluation

In this subsection, we evaluate the performance of our proposed method and compare it with the naive method.

6.3.1 Offline Performance

Since we partition the data space into grids and utilize the feature encoding techniques in offline phase, the storage cost is acceptable. Tab. 1 shows the space cost and index building time (T.time) of the two methods. It is obvious that "Naive" consumes more space, since it encodes all the numeric entities. In contrast, instead of encoding each entity, "parCode" just encodes the minimal corner of each grid. What is more, the time consumed by "parCode" is much less than that consumed by "Naive".

[1]http://www.freebase.com/
[2]http://databasebasketball.com.
[3]http://www.imdb.com/interfaces.

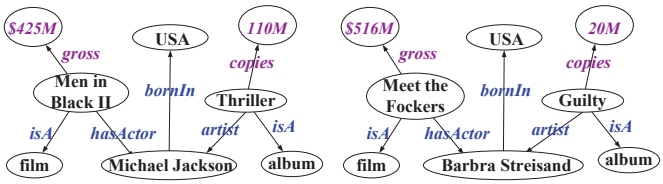


Figure 11: Excellent versatile artist finding.

Table 1: Offline Performance

Method	Space (MB)	T.time (s)	P.time (s)	E.time (s)
Naive	7,342,687	23,452	—	—
parCode	873,852	5,011	3,934	1,077

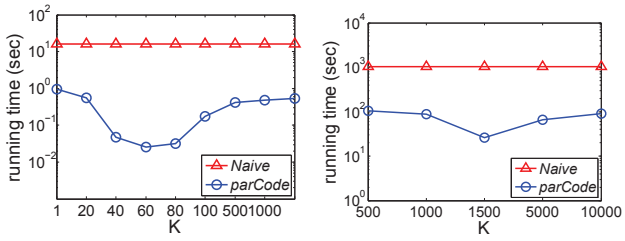
In order to study "parCode" in depth, Tab. 1 also depicts the time consumed by partition step (P.time) and encoding step (E.time), respectively. It indicates that the main time cost results from the partition process. Hence, a good and efficient partition method is pretty important.

6.3.2 Online Performance

In this subsection, we adopt two metrics, i.e., the query response time and pruning power, to evaluate the online performance, where the pruning power is the ratio of candidates that are filtered out, i.e., the number of pruned entities divided by all candidates.

Evaluate the effect of K. K is the number of grids. We fix the query size (the number of vertices) of q to be 8, and vary the number of grids. Each query may contain one or multiple numeric entities. Both the query response time and pruning power are averaged results.

Figs. 12(a) and 12(b) investigate the query response time of the two algorithms with respect to NBA and artist analyses, respectively. It shows that when K is too small or large, the response time increases. Extremely, if $K = 1$ or $K = n$, it is equivalent to the case without any partition in actual. According to this experiments, it indicates that K is better to be about $\sqrt{n}$. Since the "Naive" method is independent of the partition, its query response time is a horizontal line.

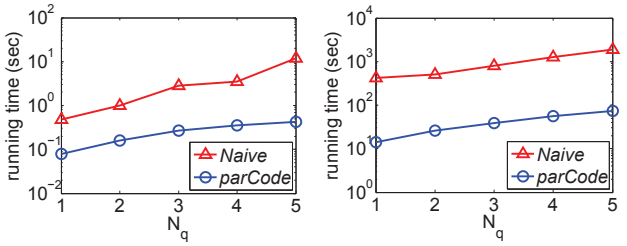


(a) Results w.r.t. NBA (b) Results w.r.t. artist

Figure 12: Query response time vs. K.

Evaluate the effect of N_q. We fix the number of grids, and vary the number of numeric entities, N_q, in q from 1 to 5. As depicted in Fig. 13, "parCode" outperforms "Naive" in terms of time efficiency. Moreover, the performance gap between "parCode" and "Naive" becomes larger when the number of numeric entities in q increases. The reason is that the "Naive" method computes skyline entities in the manner of

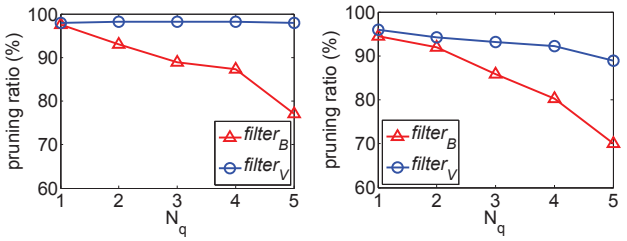
entity by entity. In comparison, "parCode" computes skyline entities in grid level and integrates numeric feature with structural feature to produce much fewer candidates.



(a) Results w.r.t. NBA (b) Results w.r.t. artist

Figure 13: Query response time vs. N_q.

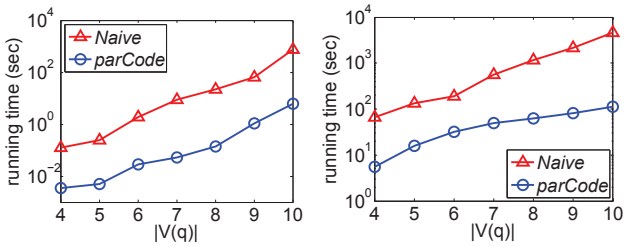
In order to study the pruning power of "parCode", we evaluate the ratio of entities (or entity pairs) that are filtered out by grid-level pruning ($filter_B$) and entity-level pruning ($filter_V$). As shown in Fig. 14, most of the candidates are pruned without invoking subgraph isomorphism verification, which contributes to the efficiency of our method.



(a) Results w.r.t. NBA (b) Results w.r.t. artist

Figure 14: Pruning power vs. $|V(q)|$.

Evaluate the effect of $|V(q)|$. Fig. 15 presents the query response time with respect to the number of vertices in query graph q. As shown in Fig. 15, both the time efficiency of "parCode" and "Naive" decrease with increasing $|V(q)|$. It is obvious that if there are more vertices in q, the time consumed by subgraph isomorphism checking will increase.



(a) Results w.r.t. NBA (b) Results w.r.t. artist

Figure 15: Query response time vs. $|V(q)|$.

7. CONCLUSIONS

In this paper, we formalize the problem of subgraph skyline search (denoted as S^3) over large graphs and propose an algorithm to answer S^3 queries. To improve the efficiency, we propose to partition the data space into grids, based on which we carefully design feature encoding to facilitate the query process. The experimental results on real datasets validate the effectiveness and efficiency of our method. As future work, there are several issues to be addressed, such as handling high dimensions, incremental updates, and the effect of data distributions.

8. ACKNOWLEDGMENTS

This work was supported by China 863 Project under Grant No. 2012AA011101, National Science Foundation of China (NSFC) under Grant No. 61370055 and 61272344, and CCF-Tencent Open Research Fund.

9. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.
[2] E. P. F. Chan and H. Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3), 2007.
[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. fg-index: Towards verification-free query processing on graph databases. In *SIGMOD*, 2007.
[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. PAMI*, 26(10), 2004.
[5] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. *CoRR*, abs/1311.2100, 2013.
[6] R. Jin, N. Ruan, S. Dey, and J. X. Yu. Scarab: scaling reachability computation on large graphs. In *SIGMOD*, 2012.
[7] W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In *ICDE*, 2007.
[8] W. Jin, A. K. H. Tung, M. Ester, and J. Han. On efficient processing of subspace skyline queries on high dimensional data. In *SSDBM*, 2007.
[9] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Prefjoin: An efficient preference-aware join operator. In *ICDE*, 2011.
[10] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. Flexpref: A framework for extensible preference evaluation in database systems. In *ICDE*, 2010.
[11] M. Nagendra and K. S. Candan. Skyline-sensitive joins with lr-pruning. In *EDBT*, 2012.
[12] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1), 2008.
[13] V. Raghavan, E. A. Rundensteiner, and S. Srivastava. Skyline and mapping aware join query evaluation. *Inf. Syst.*, 36(6), 2011.
[14] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1), 2008.
[15] C. Sheng and Y. Tao. Worst-case i/o-efficient skyline algorithms. *ACM Trans. Database Syst.*, 37(4), 2012.
[16] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.
[17] J. Tang, S. Wu, and J. Sun. Confluence: conformity influence in large social networks. In *KDD*, 2013.
[18] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, 2006.
[19] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.
[20] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.
[21] A. Vlachou, C. Doulkeridis, and N. Polyzotis. Skyline query processing over joins. In *SIGMOD*, 2011.
[22] X. Yu, Y. Sun, P. Zhao, and J. Han. Query-driven discovery of semantically similar substructures in heterogeneous networks. In *KDD*, 2012.
[23] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, 2007.
[24] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1), 2010.
[25] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, 2008.
[26] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering sparql queries via subgraph matching. *PVLDB*, 4(8), 2011.