Online Indices for Predictive Top-k Entity and Aggregate Queries on Knowledge Graphs

Yan Li
University of Massachusetts, Lowell
yan_li1@student.uml.edu

Tingjian Ge
University of Massachusetts, Lowell
ge@cs.uml.edu

Cindy Chen
University of Massachusetts, Lowell
cchen@cs.uml.edu

***Abstract*—Knowledge graphs have seen increasingly broad applications. However, they are known to be incomplete. We define the notion of a virtual knowledge graph which extends a knowledge graph with predicted edges and their probabilities. We focus on two important types of queries: top-k entity queries and aggregate queries. To improve query processing efficiency, we propose an incremental index on top of low dimensional entity vectors transformed from network embedding vectors. We also devise query processing algorithms with the index. Moreover, we provide theoretical guarantees of accuracy, and conduct a systematic experimental evaluation. The experiments show that our approach is very efficient and effective. In particular, with the same or better accuracy guarantees, it is one to two orders of magnitude faster in query processing than the closest previous work which can only handle one relationship type.**

I. Introduction

A knowledge graph is a knowledge base represented as a graph. It is a major abstraction for heterogeneous graph representation of data with broad applications including web data [1], user and product interactions and ratings [2], medical knowledge and facts [3], and recommender systems [4], [5]. Due to the enormous and constantly increasing amount of information in such knowledge bases and the limited resources in acquiring it, a knowledge graph is inherently incomplete to a great extent [5].

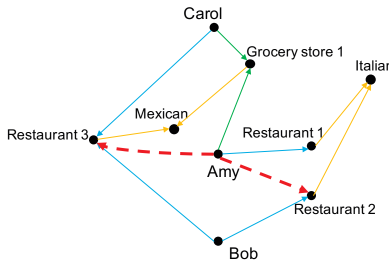

Fig. 1: Illustrating a virtual knowledge graph.

We consider it as a *virtual knowledge graph*, i.e., a graph complemented with probabilistic edges. The probabilistic edges are virtual and predicted with an estimated probability, given by an algorithm $\mathcal{A}$ associated with the graph, as formulated in Section II and discussed in Section V-B. Figure 1 illustrates a virtual knowledge graph where we have different vertex types including users (e.g., Amy, Bob), restaurants, grocery stores, and styles of food (e.g., Italian, Mexican). The graph also has different relationship types in edges (illustrated with various colors) such as "rates high" (of a restaurant), "frequents" (a grocery store), and "belongs to" (a style of food). This knowledge graph is incomplete. For example, it misses the information that Amy likes Restaurant 2 (i.e., would give a high rating) with a certain probability, and she likes Restaurant 3 with a certain probability, illustrated as red dashed edges in Figure 1.

We envision that one of the major uses of a virtual knowledge graph is to answer entity information queries given another entity and a relationship. For instance, in Figure 1, a useful query may be **(Q1)** "What are the top-5 most likely restaurants Amy would rate high but has not been to yet?". Another query, which involves aggregation, is **(Q2)** "What is the average age of all the people who would like Restaurant 2?".

A knowledge graph consists of triples (i.e., edges) in the form of (h, r, t), where h is the head entity, r is a relationship, and t is the tail entity. For example, (*Carol*, *frequents*, *Grocery store 1*) is such a triple in the graph of Figure 1. Each of the two example queries above is either, given h and r, to query information about t (Q1), or, given t and r, to query information about h (Q2).

Due to the potentially gigantic number of entities in a large knowledge graph, it would be very slow and less than desirable to process each entity on the fly and select the top ones with the highest probabilities. Moreover, adding to the difficulty, there can be thousands or more relationships in a heterogeneous knowledge graph. Our experiments in Section VI show that queries are over 3 orders of magnitude faster with our index compared to no index—the larger the knowledge graph, the greater the difference.

Then, how can one possibly index a virtual knowledge graph which has potential edges not even materialized? We propose to leverage, and build upon, the *knowledge graph embedding* techniques [5] for this purpose. Informally, knowledge graph embedding performs automatic feature extraction and dimensionality reduction and produces a different vector for each vertex and each relationship type of the graph. These embedding vectors incorporate all relevant information in the vertex types, relationship types, and the graph topology. Many knowledge graph embedding algorithms produce vectors that maintain a constraint within a triple (h, r, t) for each relationship type r that holds between entities h and t such as $\mathbf{h}+\mathbf{r} \approx \mathbf{t}$ [6], where the boldface symbols $\mathbf{h}$, $\mathbf{r}$, and $\mathbf{t}$ represent the embedding vectors of h, r, and t, respectively. Embedding has been shown to be the state-of-the-art technique for link prediction in knowledge graphs [5] .

The basic idea is that we perform spatial indexing of the embedding vectors. Then for a query (Q1) that is given the head entity h and a relationship type r and that requests information on the tail entity t (either top matches or aggregate information), we first use the embedding vectors $\mathbf{h}$ and $\mathbf{r}$ to calculate a new vector $\mathbf{h} + \mathbf{r}$, and then look up the spatial index to find entities whose embedding vectors are close to point $\mathbf{h} + \mathbf{r}$ in the multidimensional space. It is analogous for the other type of query (Q2) where we search the neighborhood of point $\mathbf{t} - \mathbf{r}$ for head entities.

One problem with the above basic approach is that common spatial indices, including the R-tree variants, do not work well for high dimensionalities such as tens and hundreds for graph embedding vectors [7]. Operating in high dimensionalities severely penalizes performance. Our solution is to first use a Johnson-Lindenstrauss (JL) type random projection [8] into a low dimension space (such as 3), and then build a spatial index in this low dimension space. However, classical JL transform and its analysis require the resulting dimensionality to be rather high, at least in the hundreds. Thus, we perform a novel analysis of accuracy guarantees for any low dimensions.

We observe that building such an index is still very time and space consuming, and the search space is very uneven given the space of queries. In other words, it is wasteful to build a full index while most nodes will never be visited. Thus, analogous to a cracking index of B+ tree [9], [10] in a relational database, we propose a novel way to build a cracking R tree index online as needed by queries. In addition to adopting a greedy approach for this purpose, we devise an algorithm that performs A* search with top-k best choices in building the index. Finally, we propose query processing algorithms using this index for both top-k and aggregate queries, and give a novel proof of the accuracy guarantees using the martingale theory and Azuma inequality [11].

To the best of our knowledge, this is the first work that proposes an incremental index technique for answering predictive queries with accuracy guarantees. The closest related work, H2-ALSH, uses locality sensitive hashing (LSH) for approximate nearest neighbor search [12], but can only handle *one* relationship type. H2-ALSH uses collaborative filtering and searches for entities of one type that are closest to an entity of another type based on the inner product distance metric—hence it can only handle relationship of a single type, but not heterogeneous knowledge graphs. Nevertheless, even for a homogeneous graph with a single relationship type, our cracking index does not incur a long, offline index-building time as H2-ALSH does, and our query processing time is one order of magnitude faster than H2-ALSH for a smaller dataset and two orders of magnitude faster for a larger dataset. That is, our method scales better due to our overall tree-structure index (unlike the flat buckets of LSH) with a cost logarithmic of the data size. In summary, our contributions are as follows:

- The notion of a virtual knowledge graph and its indexing for top-k and aggregate predictive queries.
- Graph embedding transform with provable accuracy guarantees (Section III) combined with an incremental index technique (Section IV).
- Query processing algorithms for top-k and aggregate queries with a novel proof of accuracy (Section V).
- A comprehensive experimental study using real-world knowledge graph datasets (Section VI).

II. Problem Formulation

A *knowledge graph* $G = (V, E)$ is a directed graph whose vertices in V are *entities* and edges in E are *subject-property-object* triple facts. Each edge is of the form (*head entity*, *relationship*, *tail entity*), denoted as (h, r, t), and indicates a relationship r from entity h to entity t.

Definition 1. ***(Virtual Knowledge Graph)*** *A virtual knowledge graph $\mathcal{G} = (V, \mathcal{E})$ induced by a prediction algorithm $\mathcal{A}$ over a knowledge graph $G = (V, E)$ is a probabilistic graph that has the same set of vertices (V) as G, but has edges $\mathcal{E} = E \cup E'$, where each $e \in E'$ is of the form (h, r, t, p), i.e., a triple (h, r, t) extended with a probability p determined by algorithm $\mathcal{A}$. Accordingly, edges in E have a probability 1.*

Remarks. Logically, a virtual knowledge graph $\mathcal{G}$ can be considered as a complete graph, even though some edges may have tiny or zero probabilities. However, we never materialize all the edges—only the highest probability ones are retrieved on demand. The algorithm $\mathcal{A}$ needs to return a probability for a retrieved edge not in E. In this paper, we focus on two types of queries over $\mathcal{G}$, namely the top-k queries and aggregate queries. A *top-k* query is that, given a head entity h (resp. tail entity t) and a relationship r, we return the top k entities t (resp. h) with the highest probabilities in E'. In the same context, an *aggregate* query returns the *expected* aggregate value (COUNT, SUM, AVG, MAX, or MIN as in SQL) of the attributes of entities t (resp. h) in E'. Q1 in Section I is an example of a top-k query, while Q2 is an aggregate query.

Note that alternatively we could have the query semantics for all the edges in $E \cup E'$; however, we focus on E' in this paper as it contains previously unknown edges in the graph and is of practical significance to applications such as recommender systems. As part of the graph, for each entity point, we know its set of neighbors as in E. Thus, for our default semantics that only considers E', if index access in Section V retrieves an entity point that is a neighbor (in E) of the query entity, we simply skip it and continue to the next entity (this is typically insignificant as real knowledge graphs' node-degrees follow the power law [13], and most nodes have relatively low degrees in E).

Due to the large number of entities, a brute force way of iterating over every entity would be slow and undesirable. Furthermore, materializing all edges in E' would be too costly, since knowledge graphs are known to be very incomplete [5], and there can be many relationships (e.g., in

Freebase [14]). In this paper, we propose a novel and efficient indexing method to solve this problem, based on a knowledge graph embedding algorithm $\mathcal{A}$, which has been shown to be the state-of-the-art technique for link prediction [5].

III. Transform of Embedding Vectors

A. Overall Approach

We first apply an existing knowledge graph embedding scheme, such as TransE [6] or TransA [15], to get the embedding vectors of each node and relationship type. This embedding is the algorithm $\mathcal{A}$ that induces the virtual knowledge graph. We aim to index these embedding vectors. However, typically they are of tens or hundreds of dimensions, which is too inefficient for commonly used spatial indices. Hence, in this section, we apply the Johnson-Lindenstrauss (JL) transform [8] to convert the embedding vectors into a low dimension (such as 3) before indexing them.

B. Transforming to Space $\mathbb{S}_2$

A knowledge graph embedding scheme results in vectors of dimensionality d in the vicinity of tens or hundreds, in an embedding space $\mathbb{S}_1$. There is one vector for each vertex (entity) and for each relationship type. We then perform JL transform on these vectors. However, a major technical challenge is how to provide accuracy guarantees, since classical JL transform and its proof of distance preservation require the resulting dimensionality to be typically rather high, at least in the hundreds—while we need it to be a small number α (e.g., 3). Let this α-dimensional space be $\mathbb{S}_2$. Specifically, the mapping of this transform is:

$$\mathbf{x} \mapsto \frac{1}{\sqrt{\alpha}}\mathbf{A}\mathbf{x}$$

where the $\alpha \times d$ matrix $\mathbf{A}$ has each of its entries chosen i.i.d. from a standard Gaussian distribution $N(0, 1)$, and d is the dimensionality of $\mathbb{S}_1$. Intuitively, each of the α dimensions in $\mathbb{S}_2$ is a random linear combination of the original d dimensions in $\mathbb{S}_1$, with a scale factor $\frac{1}{\sqrt{\alpha}}$, so that the L_2-norm of $\mathbf{x}$ is preserved.

Note that our proof of the following result, Theorem 1, is inspired by, but differs significantly from that in [8]. In particular, the analysis and proof in [8] only apply to the case when the ε below is between 0 and 1 for the upper bound; moreover, we obtain a tighter bound for a small dimensionality α and a relaxed ε range. For succinctness and clarity, the proofs of all the theorems/lemmas in the paper are in our technical report [16].

Theorem 1. *For two points $\boldsymbol{u}$ and $\boldsymbol{v}$ in the embedding space $\mathbb{S}_1$ that are of Euclidean distance l_1, their Euclidean distance l_2 in space $\mathbb{S}_2$ after the transform has the following probabilistic upper bound:*

$$Pr[l_2 \geq \sqrt{1+\varepsilon} \cdot l_1] \leq \Delta_u(\varepsilon) \doteq \left(\frac{\sqrt{1+\varepsilon}}{e^{\varepsilon/2}}\right)^{\alpha} \quad (1)$$

for any $\varepsilon > 0$, where α is the dimensionality of $\mathbb{S}_2$. Similarly, the probabilistic lower bound is

$$Pr[l_2 \leq \sqrt{1-\varepsilon} \cdot l_1] \leq \Delta_l(\varepsilon) \doteq \left(\sqrt{1-\varepsilon} \cdot e^{\varepsilon/2}\right)^{\alpha} \quad (2)$$

for any $0 < \varepsilon < 1$.

To see an example of the upper bound, we set $\varepsilon = 3$, and suppose the JL transform has dimensionality $\alpha = 3$, then with confidence 91.2%, $l_2 < 2l_1$. For an example of lower bound, by setting $\varepsilon = \frac{15}{16}$ (again $\alpha = 3$), we have that, with confidence at least 94%, $l_2 > \frac{l_1}{4}$.

IV. Cracking and Uneven Indices for Virtual Knowledge Graphs

Once we transform all entity points into the low-dimensional space $\mathbb{S}_2$, we perform indexing in order to answer queries on the virtual knowledge graph. A simple approach is to just use an off-the-shelf spatial index, such as an R-tree [7]. However, we observe that this can be quite wasteful, given that the number of entities is often huge, and that the query regions (e.g., using $\mathbf{h} + \mathbf{r}$ to search for top-k tail entities with embedding $\mathbf{t}$) are typically skewed and only cover a small fraction of the whole space of entity points.

This motivates us to build a *cracking* and *uneven* R-tree index (similar ideas work on other variants of index too). The basic idea is that we bulk-load/build an R-tree index in a top-down manner, and continue the "branching" on demand—only as needed for queries. As a result, regions in the embedding space $\mathbb{S}_2$ that are more relevant to queries (e.g., $\mathbf{h} + \mathbf{r}$) are indexed in finer granularities, while the irrelevant regions stay at high levels of the tree. Thus, unlike the traditional R-trees that are balanced, the index that we build is imbalanced (uneven).

A. Preliminary: R-tree Bulk-Loading Algorithm

We first give some background on the top-down R-tree bulk-loading [17], as shown in the algorithm BulkLoadChunk. Later in this section, we will devise our cracking index algorithms on top of BulkLoadChunk. We are indexing a set of rectangular objects D (for our problem, they are actually just a set of points—a special case of rectangles—in space $\mathbb{S}_2$). The basic idea of BulkLoadChunk is to first sort D into a few *sort orders* $D^{(1)}, D^{(2)}, ..., D^{(S)}$, for example, based on the 2α coordinates of the α-dimensional rectangles (e.g., $S = 2\alpha$). Note that each $D^{(i)}$ is a sorted list of rectangles.

Then BulkLoadChunk performs a greedy top-down construction of the R-tree. Due to dense packing, it is known in advance how many data objects every node *covers*. Each time, we perform a binary split of an existing *minimum bounding region* (MBR) at a node based on one of the *sort orders* and a *cost model*. A cost model penalizes a potential split candidate that would cause a significant overlap between the two MBRs after the split. The cost model (which we omit in the pseudocode) is invoked in the *cost* function in line 4 of the BestBinarySplit function. This binary split is along

Algorithm 1: BulkLoadChunk $(\mathbf{D}, h)$

```
Input: D: rectangles in a few sort orders $D^{(1)}, ..., D^{(S)}$
       ($D^{(i)}$ is a list of rectangles in a particular sort order)
  $h$: the height of the R-tree to build
Output: the root of the R-tree built
1 if $h = 0$ then
2     return BuildLeafNode($D^{(1)}$)
3 $m \leftarrow \lceil \frac{D^{(1)}}{M} \rceil$ //$M$ is capacity of a nonleaf node;
      $m$ is # rectangles per child node's subtree
4 $\{\mathbf{D}_1, ..., \mathbf{D}_k\} \leftarrow$Partition $(\mathbf{D}, m)$ //$k = M$ unless in
      the end
5 for $i \leftarrow 1$ to $k$ do
6     $n_i \leftarrow$ BulkLoadChunk($\mathbf{D}_i, h-1$) //Recursively
          bulk load lower levels of the R-tree.
7 return BuildNonLeafNode($n_1, ..., n_k$)

Function Partition $(\mathbf{D}, m)$ //partition data into
    $k$ parts of size $m$
1 if $|D^{(1)}| \leq m$ then
2     return D //one partition
3 L, H $\leftarrow$ BestBinarySplit $(\mathbf{D}, m)$
4 return concatenation of Partition($\boldsymbol{L}, m$) and
      Partition($\boldsymbol{H}, m$)

Function BestBinarySplit $(\mathbf{D}, m)$ //find best
    binary split of D
1 for $s \leftarrow 1$ to $S$ do
2     $F, B \leftarrow$ ComputeBoundingBoxes($D^{(s)}, m$)
3     for $i \leftarrow 1$ to $M-1$ do
4         $i^*, s^* \leftarrow i$ and $s$ with the best $cost(F_i, B_i)$
5 $key \leftarrow$ SortKey($D^{s^*}_{i^* \cdot m}, s^*$) //sort key of split
      position
6 for $s \leftarrow 1$ to $S$ do
      //split each sorted list based on key of $s^*$
7     $L^{(s)}, H^{(s)} \leftarrow$ SplitOnKey($D^{(s)}, s^*, key$)
8 return $\boldsymbol{L}, \boldsymbol{H}$
```

one of the $M-1$ boundaries if we are partitioning a node into M child nodes (thus it requires $M-1$ binary splits). This choice is greedy.

Some functions are omitted for succinctness. For instance, ComputeBoundingBoxes takes as input a sorted list of rectangles and the size m of each part to be partitioned, and returns two lists of bounding rectangles F (front) and B (back), where F_i and B_i are the two resulting MBRs if the binary split is at the ith (equally spaced) position ($1 \leq i \leq M-1$). Lines 3-4 of BestBinarySplit get the optimal split position with the least cost—position i^* of sort order s^*, and line 5 retrieves the split key from that sort order list. Based on this binary split, we maintain (i.e., split) all the S sorted lists in lines 6-7, and return them.

B. Insights and Node-Splitting Cost Model

Instead of doing offline BulkLoadChunk, we build a cracking and partial index online upon the arrival of a sequence of queries. We start with some insights and informal description of performing on-demand top-down bulk-load of R-tree upon a query's rectangle region $\mathcal{Q}$. The BulkLoadChunk algorithm in general indexes rectangle objects. In our case, we only index *data points* (i.e., entities). In the complete R-tree BulkLoadChunk, all nodes are fully partitioned top-down until the leaves at the bottom, resulting in a balanced R-tree. In our online incremental build, however, we only grow the partitions of nodes that contain the data points in query region $\mathcal{Q}$; moreover, we do not need to break a partition if it contains data points all in $\mathcal{Q}$. Thus, we end up seeing an R-tree that is imbalanced with some partitions unsplit yet. As also demonstrated in our experiments (Section VI), the saving is significant since the full balanced index is quite wide.

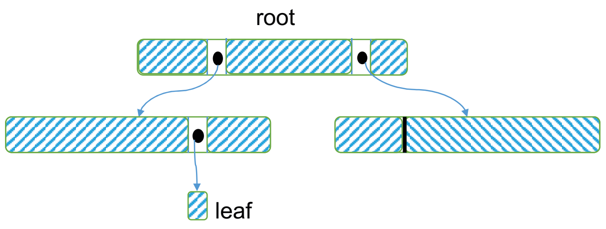

Fig. 2: Illustration of a contour in a cracking R-tree.

Definition 2. ***(Contour)***. *The contour C of a cracking R-tree is the set of current partitions (inside nodes) that do not have a corresponding child node, together with any terminal leaf nodes. We say that each such partition or leaf node is an* element *e of the contour.*

Figure 2 illustrates a contour that is shaded and has eight elements, one of which is a leaf node (the number of data points that it covers is small enough). We first have a basic observation as stated below.

Lemma 1. *Consider a contour of the R-tree at any time instant. Each element of the contour contains a mutually exclusive set of data points, and together they contain all the data points.*

Definition 3. ***(Leaf Distance).*** *At time t during the lifetime of a cracking R-tree index, if two data points d_1 and d_2 are in the same element of the contour (i.e., the same leaf node or the same partition), then we say that their* leaf distance *at time t, denoted as $l_t(d_1, d_2)$, is 0. Otherwise, their leaf distance is 1.*

Thus, leaf distance is a time-variant binary random variable that depends on the sequence of incoming queries. We next have the following lemma (the proof follows directly from the algorithm).

Lemma 2. *Consider two data points d_1 and d_2 in a partition. After a binary split at time τ, if d_1 and d_2 are still in the same partition, then in every sort order s, the positions of d_1 and d_2 can only be closer or stay the same due to the split. If d_1 and d_2 are separated into two partitions due to the split, then $l_t(d_1, d_2) = 1$ for any $t > \tau$.*

From Lemma 1, we know that all the required data points in the query region $\mathcal{Q}$ must be in the current contour $\mathcal{C}$ of the index. Thus, our incremental index building algorithm will locate each element $e \in \mathcal{C}$ that overlaps with $\mathcal{Q}$ and determine if we need to further split e for the query. At this point, we need to revise the cost model to optimize the access cost for the current query region $\mathcal{Q}$, in addition to the previous cost function that penalizes the MBR overlap due to the split. Intuitively, after the split, the data points in $\mathcal{Q}$ should be close to each other to fit in a minimum number of pages (i.e., their leaf distance should be small). Based on the *principle of locality* in database queries [18], this optimization has a lasting benefit.

1) Node-Splitting Cost Model: The key idea is that we extend the cost model into a two-component cost $(c_{\mathcal{Q}}, c_{\mathcal{O}})$, where $c_{\mathcal{Q}}$ is the cost estimate for accessing query region $\mathcal{Q}$, while $c_{\mathcal{O}}$ is the cost incurred by overlaps between partitions. At a contour $\mathcal{C}$ of the index at any time instant, we define $c_{\mathcal{Q}}$ to be the minimum number of leaf pages to accommodate all the data points in $\mathcal{Q}$, given the current configuration of elements in $\mathcal{C}$ and any possible future node splits. Lemma 2 implies that after every split, the leaf distance between any $d_1, d_2 \in \mathcal{Q}$ still in the same partition is expected to either get smaller or stay the same, while two separated points will be in different leaf nodes. This leads to the following result.

Lemma 3. *At a contour $\mathcal{C}$ of an index, a lower bound of the number of leaf nodes that we need to access and process is $\sum_{e \in \mathcal{C}} \lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil$, where $\mathcal{Q} \cap e$ is the set of data points in the element e (of $\mathcal{C}$) that are also in the query region $\mathcal{Q}$, and N is maximum number of data point entries that can fit in a leaf node.*

Note that for each leaf node accessed for $\mathcal{Q}$, we will need to convert each data point in it to the original embedding space $\mathbb{S}_1$ and calculate the distance to the query center point (e.g., **h+r**, to be detailed in Section V). Hence, the number of leaf nodes accessed and processed is a reasonable measure of the first cost component $c_{\mathcal{Q}}$. The second component of cost is $c_{\mathcal{O}}$, the cost for overlaps between partitions. We increment $c_{\mathcal{O}}$ by $\beta^h \cdot \frac{\|O\|}{min(\|L\|, \|H\|)} (\beta \geq 1)$ at each binary split during the runs of the algorithm, where O is the overlap region between two resulting partitions L and H of the binary split, $\| \cdot \|$ denotes the volume of a region, h is the height of the R-tree where the split happens, and $\beta \geq 1$ is a constant indicating that an overlap higher in the R-tree has more impact and is more costly (as an R-tree search is top-down).

A remaining issue is that $c_{\mathcal{Q}}$ and $c_{\mathcal{O}}$ are two types of cost measured differently—making the whole node-splitting cost a composite one. However, it is important to be able to compare two node-splitting costs, as required by our index building algorithms in Section IV-C. We observe that, for our problem, the query region $\mathcal{Q}$ is derived from a ball around the center point (e.g., **h+r**), and is hence continuous in space and should not be too large (otherwise the links are too weak). Thus, it is reasonable to attempt to achieve optimal $c_{\mathcal{Q}}$ as a higher priority. As a query-workload optimized approach, we treat $c_{\mathcal{Q}}$ as the *major order* and $c_{\mathcal{O}}$ as the *secondary order* when comparing two composite costs.

C. Incremental Partial Index Algorithms

Having developed the cost model, we are now ready to present our online cracking index algorithms. The algorithms incrementally build an index and use it to search at the same time. The idea is that for the initial queries more building of the index is done, while it is mainly used for search (and little is changed to the index) for subsequent queries. Overall, the cracking index only performs a very small fraction of the binary splits performed by the full BulkLoadChunk, as verified in our experiments in Section VI.

1) Main Algorithm: With the insights given in Section IV-B, we describe our main cracking index algorithm, IncrementalIndexBuild, based on the key functions given under BulkLoadChunk in Section IV-A. IncrementalIndexBuild takes as input a query region $\mathcal{Q}$ and the current index $\mathcal{I}$ (with contour $\mathcal{C}$). Initially $\mathcal{I}$ (and $\mathcal{C}$) is just a root node containing all the data points.

(1) Instead of a top-down complete bulk-load, upon a query region $\mathcal{Q}$, we perform an incremental partial top-down build of index $\mathcal{I}$ to the elements in the current contour that overlap $\mathcal{Q}$. We store a set of data points contained in each element e of the current contour $\mathcal{C}$ in addition to its MBR. A non-leaf element e contains the S sort orders of the data points.

(2) The incremental algorithm probes $\mathcal{I}$ until it reaches an element e in $\mathcal{C}$ that has data points contained in $\mathcal{Q}$, and calls Partition over e.

(3) The Partition function simply returns its input **D** if it satisfies the *stopping condition*, which is $\mathcal{Q} \cap e = \emptyset$ *or* $\lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil = \lceil \frac{|e|}{N} \rceil$, where e is the current element of $\mathcal{C}$ that has the **D** partition.

(4) If a Partition call returns from its line 2 (i.e., already smallest partition at its level), we then call BulkLoadChunk over it (the same as line 6 of BulkLoadChunk).

(5) The *cost* function in line 4 of BestBinarySplit is revised as stated in Section IV-B.

In (3) above, the stopping condition of binary partition over an element e in the current contour is either $\mathcal{Q} \cap e = \emptyset$ indicating that e is irrelevant to $\mathcal{Q}$, or $\lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil = \lceil \frac{|e|}{N} \rceil$ indicating that almost all the data points in e are in $\mathcal{Q}$. A special case of the latter stop condition is when e is a single leaf node that has data points in $\mathcal{Q}$, although in general it may stop at an element larger than a leaf without splitting it. In (4) above, the top-down recursive algorithm proceeds to the next lower level of the tree.

As part of this top-down probing process, the qualified data points are found and returned. Note that we can start processing the first query when the index only has a root node. As more queries come, the index grows incrementally and the node splits are optimized for the query usage (via the cost functions in Section IV-B1). As shown in our experiments in Section VI, only a very small fraction of the splits are performed compared to BulkLoadChunk, since the space of queried embedding vectors is much smaller than that of all data points. Thus, the amortized cost of incremental index building is much smaller than bulk loading.

2) More Split Choices: Our main indexing algorithm makes a greedy choice to select a locally optimal cost for each split, as the original bulk loading algorithm does. However, we observe that, since we are now only incrementally build the index for each query, we may afford to explore more than one single split choice. We will iterate over a small number (e.g., $k = 2$ to 4) of split choices, with the goal of getting a good global index tree. Furthermore, we will use A^* style aggressive pruning to cut down the search space for a query.

The key idea is to use a priority queue $\mathbb{Q}$ (a heap) to keep track of "active" contours as *change candidates.* We do not adopt a change candidate until it completely finishes its splits for the current query and is determined to be the best plan based on A^* pruning. The sort order of the priority queue is the two-component cost of a contour. The minimum cost contour (i.e., change candidate) is popped out from $\mathbb{Q}$, and is expanded (with the top-k choices for the next split). The algorithm is shown in TOP-KSPLITSINDEXBUILD.

Algorithm 2: TOP-KSPLITSINDEXBUILD $(\mathcal{I}, \mathcal{Q})$

```
Input: $\mathcal{I}$: current index; $\mathcal{Q}$: query region
Output: revised index
1  if $\mathbb{Q}$ does not exist yet then
2      create a priority queue $\mathbb{Q}$
3      add into $\mathbb{Q}$ the initial contour $\mathcal{C}$ with only the root node of $\mathcal{I}$ and cost $(\sum_{e\in\mathcal{C}}\lceil\frac{\mathcal{Q}\cap e}{N}\rceil, 0)$ as weight
4  while $\mathbb{Q}$ is not empty do
5      $\mathcal{C} \leftarrow$ pop head of $\mathbb{Q}$
6      $e \leftarrow$ first element of $\mathcal{C}$ whose MBR overlaps $\mathbb{Q}$
7      while $\mathcal{Q} \cap e = \emptyset$ or $\lceil\frac{|\mathcal{Q}\cap e|}{N}\rceil = \lceil\frac{|e|}{N}\rceil$ do
           //stopping condition
8          $e \leftarrow$ next element of $\mathcal{C}$ whose MBR overlaps $\mathbb{Q}$
9          if $e = null$ then
10             break
11     if $e = null$ then
12         return the index with $\mathcal{C}$ //all $e \in \mathcal{C}$ are exhausted
13     process $e$ as INCREMENTALINDEXBUILD does, except:
14     in BESTBINARYSPLIT, we get top-$k$ best splits
15     for each of the $k$ splits do
16         $\mathcal{C}' \leftarrow \mathcal{C}+$ the split
17         $c_{\mathcal{Q}} \leftarrow \mathcal{C}.c_{\mathcal{Q}} - \lceil\frac{|\mathcal{Q}\cap e|}{N}\rceil + \lceil\frac{|\mathcal{Q}\cap e'|}{N}\rceil + \lceil\frac{|\mathcal{Q}\cap e''|}{N}\rceil$ //$e$ is split into $e'$ and $e''$
18         $c_{\mathcal{O}} \leftarrow \mathcal{C}.c_{\mathcal{O}} + \beta^h \cdot \frac{\|O\|}{min(\|L\|,\|H\|)}$
19         add $\mathcal{C}'$ into $\mathbb{Q}$ with cost $(c_{\mathcal{Q}}, c_{\mathcal{O}})$ as weight
```

In lines 1-3, we create the priority queue $\mathbb{Q}$ for the very first query. Initially, the index only has the root node, and this contour is added into $\mathbb{Q}$ with the two-component cost as discussed in Section IV-B (the overlap cost is 0). Line 5 pops out the head of the queue which has the least cost. The stopping condition of processing an element e in line 7 is the same as that in INCREMENTALINDEXBUILD.

Lines 6 and 8 are based on a certain traversal order such as depth-first-search. If all elements of the top change candidate (contour) satisfy the stopping condition, and hence the next element is null (lines 9-12), this candidate must be the best among all. Otherwise, in lines 13-19, we will continue to process (split) this change candidate and expand it with the top-k splits. We add these k new candidates into $\mathbb{Q}$ with updated costs as their weights.

V. ALGORITHMS TO ANSWER QUERIES

Let us now proceed to discussing the algorithms to answer queries on a virtual knowledge graph, given the indexing algorithms in Section IV. The accuracy guarantees are shown in Theorems 2 and 4 below. The intuition is that the original space $\mathbb{S}_1$ typically has 50-100 dimensions but is very sparse. The JL transform provably preserves pairwise point-distances (Theorem 1), while top-k and statistical queries are based on such point-distances.

A. Top-k Queries

We first consider queries that ask for top-k tail entities, given a head entity h and a relationship r (note that answering queries for top-k head entities given t and r is analogous and is omitted). We use boldface letters **h** and **r** to represent the corresponding vectors in $\mathbb{S}_2$. The basic idea of our algorithm is to iteratively refine (reduce) the query rectangle region, until the k data points nearest to **h+r**'s corresponding vector in the original space $\mathbb{S}_1$ are identified, which correspond to the matches with top-k probabilities based on the embedding algorithm. The algorithm is shown in FINDTOP-KENTITIES.

Algorithm 3: FINDTOP-KENTITIES $(\mathcal{I}, h, r)$

```
Input: $\mathcal{I}$: current index; $h$: head entity; $r$: relationship
Output: top-$k$ entities most likely to have relationship $r$ with $h$
1  $\mathbf{q} \leftarrow \mathbf{h+r}$
2  probe $\mathcal{I}$ for the smallest node that contains $\mathbf{q}$, and get $k$ data points $N_q$
3  $r_q \leftarrow r_k^*(N_q) \cdot (1+\varepsilon)$
4  $\mathcal{Q} \leftarrow$ region of $B(\mathbf{q}, r_q)$
5  while data points in $\mathcal{Q}$ have not been all examined do
6      $N_q \leftarrow$top-$k$ data points so far closest to $\mathbf{q}$ in $\mathbb{S}_1$
7      $r_q \leftarrow r_k^*(N_q) \cdot (1+\varepsilon)$
8      $\mathcal{Q} \leftarrow$ region of $B(\mathbf{q}, r_q)$
9  TOP-KSPLITSINDEXBUILD $(\mathcal{I}, \mathcal{Q})$
10 return $N_q$
```

In line 2, we probe the current index and locate the element e (either a partition or a leaf) in its contour that contains **q**. Recall that the data points in e are sorted in several orders; we arbitrarily pick one sort order s, and traverse the data points of e in increasing distance from $\mathbf{q}$ based on the linear order in s, and get the first k data points. This is the set N_q in line 2. In line 3, $r_k^*(N_q)$ denotes the kth smallest distance to $\mathbf{q}$'s corresponding vector in $\mathbb{S}_1$ after mapping the data points in N_q to $\mathbb{S}_1$. Recall that our ultimate goal is to find closest entities in $\mathbb{S}_1$. The ε in line 3 is based on the accuracy guarantees in Theorem 1, and will be discussed in our next two theorems. In line 4, we get the minimum bounding box region of the ball in $\mathbb{S}_2$ centered at $\mathbf{q}$ with radius r_q. Then the traversal order of data points in line 5 is exactly the same as that described above for line 2.

Within the loop in lines 5-8, in line 6, we maintain the top-k data points that are closest to $\mathbf{q}$'s corresponding vector in $\mathbb{S}_1$. Consequently, the r_q in line 7 must be non-increasing over all the iterations, and so is the $\mathcal{Q}$ in line 8 (and hence the $\mathcal{Q}$ in line 5 used for the next iteration). Note that lines 7-8 are identical to lines 3-4. Based on the final query region $\mathcal{Q}$, line 9 updates the incremental index. Now we analyze our top-k entity query processing algorithm. Theorem 2 below provides data-dependent accuracy guarantees, while Theorem 3 addresses the performance of the algorithm.

Theorem 2. *With probability at least* $\prod_{i=1}^{k}\left[1-\frac{m_i^{\alpha}}{e^{\alpha(m_i^2-1)/2}}\right]$, *FindTop-kEntities does not miss any true top-k entities, where* $m_i=\frac{r_k^*}{r_i^*}(1+\varepsilon)$, *while the expected number of missing entities compared to the ground-truth top-k entities is* $\sum_{i=1}^{k}\frac{m_i^{\alpha}}{e^{\alpha(m_i^2-1)/2}}$.

Theorem 3. *For the final query region* $\mathcal{Q}$ *in FindTop-kEntities and* $0<\varepsilon'<1$, *the probability that a data point with distance at least* $r_k^*\cdot\frac{1+\varepsilon}{1-\varepsilon'}$ *from* $\mathbf{q}$ *in* $\mathbb{S}_1$ *may get into* $\mathcal{Q}$ *is no more than* $(1-\varepsilon')^{\alpha}\cdot e^{\alpha(\varepsilon'-\frac{\varepsilon'^2}{2})}$.

B. Aggregate and Statistical Queries

Consider the following queries: What is the total number of restaurants that Amy may like? What is the average distance of the restaurants that Amy likes? These queries involve statistical aggregation over the virtual knowledge graph.

The relevant entities are within a ball with radius r_τ around the query center point such as **h+r**. The ball corresponds to a probability threshold of p_τ (e.g., a small value 0.05). To decide the probabilities, we let the entity closest to the query center point have probability 1 for the relationship, and other entities' probabilities are inversely proportional to their distances to the query center point. In general, for each data point in the ball, we may need to access its record with attribute information for aggregation and/or for evaluating the predicates. When there are too many such data points, we may use a sample of them to estimate the query result.

Note that in this ball, each data point only has a certain probability to be relevant (in the relationship with the head entity h), with probabilities in decreasing order from the center to the sphere of the ball. Without knowledge of the distribution of a relevant attribute in these data points, our accessed sample is the a points closest to the center (i.e., with top-a probabilities) among a total of b points in the ball.

COUNT, SUM, and AVG Queries. The estimations of COUNT, SUM, and AVG query results are similar. Let us begin with a SUM query. The expectation of a SUM query result is:

$$\mathbf{E}[s]=\frac{\sum_{i=1}^{a}v_i\cdot p_i}{\sum_{i=1}^{a}p_i\big/\sum_{i=1}^{b}p_i} \tag{3}$$

where a is the number of accessed data points out of a total of b data points in the ball, $p_1\geq p_2\geq\cdots\geq p_a\geq\cdots\geq p_b$ are the data point probabilities, and v_i $(1\leq i\leq a)$ are the retrieved attribute values to sum up. Note that we know the number of entities in each element of an index contour, and hence can estimate the $b-a$ probabilities (based on the average distance of an element to a query point). The numerator in Equation (3) is the expected sum of the attribute value in the retrieved sample. This needs to be scaled up by a factor indicated in the denominator of Equation (3), the ratio between the cardinality of the sample and the cardinality of all data points.

We now give a novel analysis, based on the *martingale theory* [11] in probability, to bound the probability that the *ground truth* SUM result is at a certain distance away from the expectation in Equation (3). Note that the analysis is nontrivial since the entities/points are correlated w.r.t. their memberships in relation r with entity h in question (as the entities are connected by edges/paths in the graph).

Theorem 4. *In the algorithm for answering SUM queries, let the expectation in Equation (3) be* μ. *Then the ground truth answer* S *to the query must satisfy the following:*

$$Pr[|S-\mu|\geq\delta\mu]\leq 2e^{-2\delta^2\mu^2\big/\sum_{i=1}^{a}v_i^2+(b-a)v_m^2}$$

where v_m *is the maximum attribute value being summed among the* $b-a$ *data points not accessed.*

We can maintain minimum statistics on $|v_m|$ at R-tree nodes to facilitate accuracy estimates. Alternatively, we may estimate $|v_m|$ based on the known sample values $|v_i|$ $(i\leq a)$, without relying on any domain knowledge of the attribute. This is the same as how we estimate the expected MAX value discussed shortly.

Note that Theorem 4 is for our general algorithmic framework. If the algorithm accesses all data points in the ball, i.e., $a=b$, the result in the theorem still holds. Moreover, from Theorem 4, we can also get similar results for COUNT and AVG queries. For COUNT queries, we simply replace v_i and v_m in the theorem by 1, as COUNT can be considered as SUM(1). For AVG queries, the analysis result is essentially the same as the bound in Theorem 4, as we need to divide both μ and the increment bound in the proof by the count.

MAX and MIN Queries. Such queries select the MAX or MIN value of an attribute among the b data points in the ball centered at the query point such as **h+r**. As before, we may access a sample of a closest data points to estimate the result. We only discuss MAX queries; the treatment for MIN queries is analogous. Again, we estimate the expected value, and then bound the probability that the true MAX is far away from this expectation.

First let us estimate the expectation of MAX of the a accessed data points. Without loss of generality, we rearrange the index of the a data points so they are in non-increasing value order $\{(u_i, p_i)\}$, where $u_1 \geq \cdots \geq u_a$. Then the expectation of sample MAX is:

$$\mathbf{E}[\mathbf{M}_S] = u_1 p_1 + u_2(1-p_1)p_2 + \cdots u_a(1-p_1)...(1-p_{a-1})p_a$$

Next, given an n-value sample chosen uniformly at random from a range $[0, m]$, we can estimate the maximum value m from the sample as $(1 + \frac{1}{n})m_s$ where m_s is the sample maximum value [19], which leads to the following result for expected MAX based on the $\mathbf{E}[\mathbf{M}_S]$ above:

$$\mathbf{E}[\mathbf{M}] = \left(\mathbf{E}[\mathbf{M}_S] - \min_{1 \leq i \leq a} v_i\right)\left(1 + \frac{1}{\sum_{i=1}^{a} p_a}\right) + \min_{1 \leq i \leq a} v_i \quad (4)$$

We can then use martingale theory to bound the probability that the ground truth MAX result is far from the value in Equation (4). This is similar to Theorem 4; hence we omit the details–the main idea is that from data point $i-1$ to point i, the change to the expected MAX value $Y_i - Y_{i-1}$ should be bounded in a small range $[B_i, B_i + d_i]$, where $d_i = max(0, [v_i - \mathbf{E}[\mathbf{M}_S]] \cdot \left(1 + \frac{1}{\sum_{i=1}^{a} p_a}\right))$ for $i \leq a$ and $d_i = \frac{\mathbf{E}[\mathbf{M}_S]}{\sum_{i=1}^{a} p_a}$ for $i > a$.

VI. Experiments

A. Datasets and Setup

We use three real world knowledge graph datasets: **(1) Freebase data.** Freebase [14] is a large collaborative knowledge base, an online collection of structured data harvested from many sources, including individual, user-submitted wiki contributions. Google's Knowledge Graph was powered in part by Freebase. The dataset we use is a one-time dump through March 2013. **(2) Movie data.** This is another popular knowledge graph dataset, which describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service [20]. Entities include users, movies, genres, and tags. Ratings are made on a 5-star scale, with half-star increments (0.5 stars to 5.0 stars). We create two relationships for ratings: a user "likes" a movie if the the rating is at least 4.0 (in the range between 0 and 5.0); a user "dislikes" a movie if the rating is less than or equal to 2.0. There are also relationships "has-genres" and "has-tags". **(3) Amazon data.** This dataset [2], [21] contains product reviews and metadata from Amazon, including 142.8 million reviews spanning May 1996 to July 2014. The review rating scale ranges from 1 to 5, where 5 denotes the most positive rating. Nodes represent users and products, and edges represent individual ratings. We create relationships "likes" and "dislikes" in the same way as movie data. In addition, the data contains "also viewed" and "also bought" relationships. Some statistics of the datasets are summarized in Table I.

TABLE I: Statistics of the datasets.

Dataset	Entities	Relationship types	Edges
Freebase	17,902,536	2,355	25,423,694
Movie	312,710	4	17,356,412
Amazon	10,356,390	4	22,507,155

We implement all the algorithms in Java. We also use the graph embedding code from the authors of [6], a high-dimensional index PH-tree from the authors of [22], and the H2-ALSH code from [12] for comparisons. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

B. Experimental Results

Queries. In order to systematically explore as much as possible the space of queried embedding vectors (e.g., $\mathbf{h} + \mathbf{r}$) in $\mathbb{S}_2$, for each query we either (1) randomly choose a head entity and a relationship and query the top-k tail entities, or (2) randomly choose a tail entity and a relationship and query the top-k head entities. We measure the execution time and accuracy of a sequence of queries—to evaluate how the response time evolves.

Freebase (Top-k). In the first set of experiments, we use the Freebase data to compare a few approaches. Two of these approaches are our main cracking index method and the top-k split-choice index build method. One baseline approach is what one would do without our work—answering the top-k entity queries without using an index by iterating over all possible entities. The second baseline uses a state-of-the-art high-dimensional index, called PH-tree [22], to index the high-dimensional (50 or 100 dimensions) embedding vectors directly, without transforming them to $\mathbb{S}_2$. Another baseline approach goes a step further by using an R-tree index by bulk-loading it, without our cracking index techniques. The results are the average of at least ten runs.

Note that for general knowledge graphs, such as the Freebase data, we cannot use the H2-ALSH scheme [12] because it can only work with one relationship type—H2-ALSH is basically a locality sensitive hashing mechanism working with collaborative filtering. Later we will use other datasets to compare against H2-ALSH.

In Figure 3, we examine the execution times of the approaches described above over the Freebase data. For the top-k split-choice index build method, there is a parameter of how many choices to take into account at each split. We show the results when this parameter is 2 or 4, respectively (i.e., the last two groups of bars). Recall that our cracking index methods do not have offline index building, but start to shape the index when queries arrive online. Hence, we measure the index building time (if any), as well as the execution times of the 1st, 6th, 11th, and 16th queries to evaluate how the response time evolves over the initial sequence of queries. Among these approaches, only PH-tree and bulk-loading have an offline index building time, which are quite

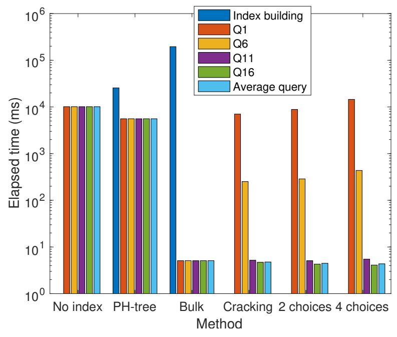

Fig 3 Method vs. elapsed time (Freebase)

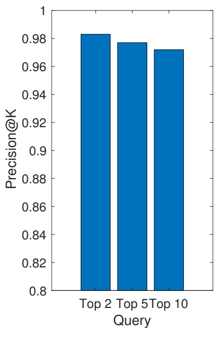

Fig 4 Accuracy (Freebase)

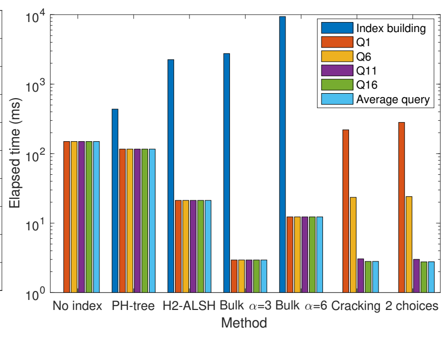

Fig 5 Method vs. elapsed time (movie)

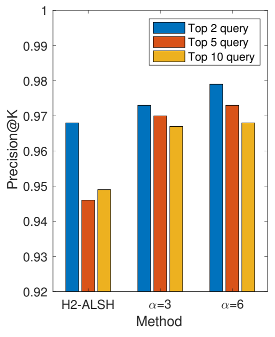

Fig 6 Accuracy (movie)

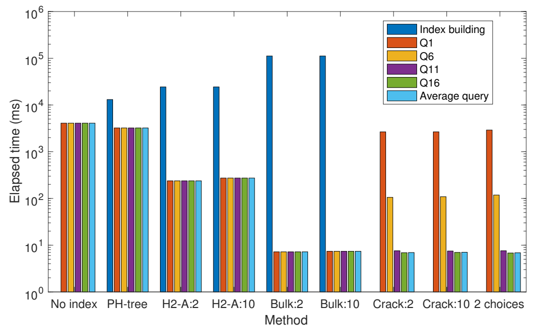

Fig 7 Method vs elapsed time (Amazon)

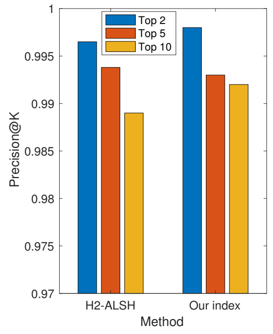

Fig 8 Accuracy (Amazon)

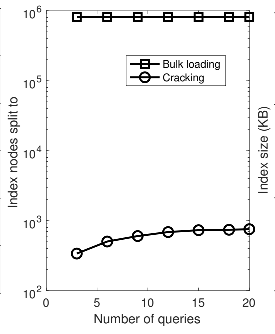

Fig 9 #Nodes (Freebase)

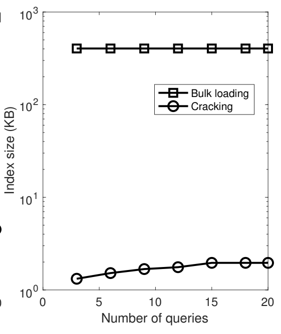

Fig 10 Index size (movie)

significant due to the large amount of data. PH-tree also has a high query cost. This is because it directly indexes the embedding vectors of dimensionality at least 50, which makes its search performance suffer significantly, almost as slow as no index (but just a linear search). The advantage of the bulk-loading approach is that the query response time is fast and even over the sequence of queries that we examine. The no-index approach has a significant overhead for each query, which is the main motivation of our work. The cracking index methods have no offline index building time, and the shape of the partial index is based on the online queries. The first query has a relatively high response time due to the initial setup for creating the index nodes (yet it is still about 30 times faster than bulk-loading in the log plot). The execution time sharply drops with more queries, and quickly flattens to a value slightly smaller than the bulk-loaded index.

Note that one may fire off the first query before the real online queries come, so that all online queries are fast. Thus, in Figure 3, we also show the average per-query execution time of 10,000 online queries after the first one issued offline. The cracking index leads to slightly better query performance than bulk-loading since it uses a different cost function in the greedy algorithm for node splitting, which optimizes the splits of data points based on the queries in the workload (Section IV-B), while a bulk-loaded offline index has no knowledge of the online queries. Although in both cases, the performance lower bound is guaranteed by Theorems 2 and 3, there can still be variations in practice (while satisfying the lower bound). In other words, the space of queried embedding vectors (e.g., $\mathbf{h}+\mathbf{r}$) in $\mathbb{S}_2$ is skewed, and is much smaller than that of all data points.

Among the 2-choice, 3-choice, and 4-choice node-split methods, they have slightly increasing costs than the main cracking index method with a single choice, but the query processing cost is eventually less when the number of choices is more. This is because a larger search space is examined with more choices. In addition, since our optimization is only with respect to one query, the exploration of extra search space with the A* aggressive pruning is still affordable when the number of choices is small. An example query with this dataset is that given a tail entity corresponding to the name “Rapper” and a relationship type “/people/person/profession”, we search for top-k head entities not in the training data (or removed before training), the result of which includes “Snoop Dogg”, “Kanye West”, and “Lil Wayne”.

Since knowledge graphs are inherently incomplete with most relations/edges absent [5], it is also the case with the datasets that we use—even the latest snapshot of the dataset will still have many edges missing. It is a challenge to evaluate the accuracy of link prediction or recommender systems [23]. One way is to mask some edges in the training data for testing purpose, while another way is to use a crowd-sourcing service like Amazon Mechanical Turk to provide an interface for everyday users where they could specify whether or not the recommendations are relevant [24].

While these methods may be helpful to some degree to evaluate a link prediction method, they will not be effective for the predictive *top-k* queries that we study, because it is not true that the masked k edges must be the top-k most likely edges, as there are a large number of edges (relations) missing in the dataset. We randomly mask 5 edges from our datasets, and find that they are typically in the top-10 list, but not necessarily top-5. For example, there are many movies that a user would really like to watch, but only a small fraction of them are in the dataset, as the user may not have time to watch them all, or she simply may not know all those movies (which is why a recommender system is needed).

Since previous studies show that graph embedding is the state-of-the-art method for link prediction [5], and since evaluating the effectiveness of graph embedding for link prediction is beyond the scope of this paper, we focus on evaluating our major contribution, which is the incremental R-tree indexing method to speed up predictive top-k and aggregate queries. Thus, we need to compare the accuracy loss with and without using our index. In Figure 4, we examine the accuracy of our indexing methods with respect to the no-index method. Since the no-index method is our base, we study the accuracy loss from using an approximate index (due to the transform from the embedding space $\mathbb{S}_1$ to $\mathbb{S}_2$). We use the *precision@K* metric, which is commonly used in information retrieval [25]. In our context, precision@K is the precision of the top-k result tuples using our index compared to the top-k tuples under the no-index method. From Figure 4 we can see that the precision@K of our indexing methods is high on average (at least 0.97).

Movie (Top-k). We next move on to the movie data. Here we query the top-k movies that a particular person likes or dislikes (but these facts are not in the training data). We note that, even with this dataset, the closest previous work, H2-ALSH [12], still does not fully handle it, since H2-ALSH can only take into account one relationship type (say, "like") when doing the collaborative filtering and building H2-ALSH. However, other relationship types such as "dislike" has the opposite semantic meaning and would help the prediction of "like" too. Thus, our method is a more holistic approach for virtual knowledge graphs. Nonetheless, we run H2-ALSH as well for a single relationship type to observe its performance.

We show the execution time results in Figure 5, where we also compare the two parameter choices of the dimensionality of the $\mathbb{S}_2$ (and hence the index), $\alpha = 3$ versus $\alpha = 6$. We can see that the index building time of H2-ALSH is slightly faster than bulk-loading R-trees when the dimensionality $\alpha = 3$. However, H2-ALSH's query processing time is much longer, partly due to the fact that it is not a hierarchical structure, and each bucket could still be very large. Furthermore, we see that, when the index dimensionality is higher such as $\alpha = 6$, there are significant overheads both in bulking loading (building) the index and in query processing. This is because these indices have a harder time with higher dimensionalities such as 6, as overlap regions tend to be much higher. An example of query results is that a particular person (with id "176299")'s top-k "like" movie list includes "152175, Ghosts (1997), Horror", "156903, The Waiting (2016), Thriller", and "3457, Waking the Dead (2000), Drama|Thriller", where the first field is the movie ID, the second field is the movie name (and year), and the third field is the genre(s) of the movie. It seems that this person likes the thriller/horror type of movies.

In Figure 6, we report the accuracy. The H2-ALSH numbers are based on the report from running the code of the authors [12], comparing to its no-index case. The precision@K of all these approaches are quite high (at least 0.945), and our cracking index methods are slightly more accurate. This is partly due to the way our embedding space transform preserves the distance. Moreover, when the dimensionality is higher, $\alpha = 6$, it is slightly more accurate than $\alpha = 3$, since higher dimensionality of the transform to $\mathbb{S}_2$ preserves the distance better. Of course, this is associated with much higher index operation costs, as shown in Figure 5.

Amazon (Top-k). In the next set of experiments, we use the Amazon dataset. The performance result is shown in Figure 7. Here we also examine the overhead when we vary the "k" parameter as in top-k results. In particular, we compare two cases $k = 2$ (labeled "H2-ALSH: 2" in Figure 7) and $k = 10$ (labeled "H2-ALSH: 10"). We see that increasing k from 2 to 10 has a slight impact on the performance of H2-ALSH, but has little or no impact on the performance of our index approaches. This is because this change of number of result tuples likely still has retrieved data points within the same index node. One interesting and important phenomenon when we compare Figure 7 with Figure 5 is that as we increase the dataset size (in particular, the number of entities)—as the case in going from movie dataset to Amazon dataset, the increase in query processing overhead is much higher for H2-ALSH than for our indexing approaches. Our query processing time is one order of magnitude faster than H2-ALSH for the movie dataset and two orders of magnitude faster for the Amazon dataset. Our method scales better due to its overall tree-structure index (unlike the flat buckets of LSH) with a cost logarithmic of the data size. We measure the accuracy in precision@K of these approaches in Figure 8. The comparison result is similar to those of other datasets.

Index Size. We now compare the number of index nodes split into, as well as the index size, between a bulk loaded index and our cracking index. The results are shown in Figure 9 for the index-node counts with the Freebase dataset, and in Figures 10 and 11 for the index sizes with the movie dataset and Amazon dataset, respectively (all three datasets show similar trends). We examine the node numbers and index sizes after different numbers of initial queries. For all three datasets, the cracking and uneven index has a very small fraction of node-count and index size than the full bulk-loaded index. This is because the search space is highly uneven and the queried space is only a small fraction. Furthermore, we observe that the convergence of node number and index size is very fast—typically after around 10 queries.

Aggregate Queries. We next study approximately answering

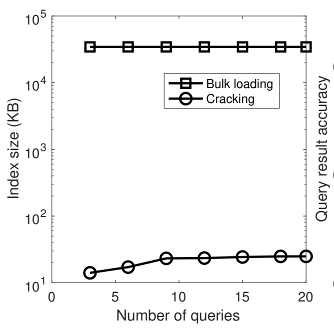

Fig 11 Index size (Amazon)

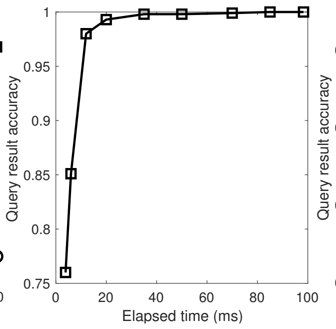

Fig 12 COUNT queries (FB)

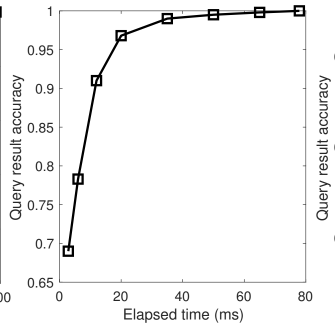

Fig 13 AVG queries (movie)

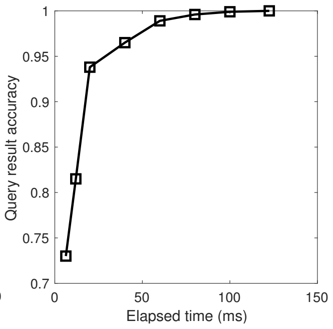

Fig 14 AVG queries (Amazon)

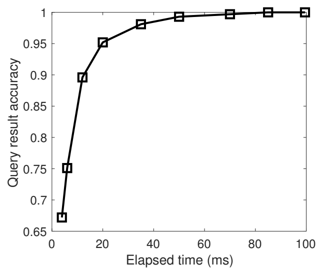

Fig 15 MAX queries (Freebase)

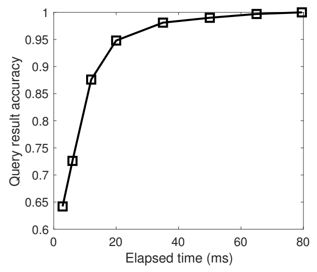

Fig 16 MIN queries (movie)

aggregate queries. Recall that there is a tradeoff between sample size (execution time) and query result accuracy. We first examine COUNT queries using the Freebase dataset, i.e., the expected count of tail entities given the head and relationship. The result is in Figure 12. The tradeoff between execution time and accuracy (compared to accessing all data points up to a probability threshold 0.01) is clearly seen here. The accuracy is measured as $1 - \frac{|v_{returned} - v_{true}|}{v_{true}}$, where $v_{returned}$ and v_{true} are query returned aggregate value and the ground truth aggregate value (from accessing all points until a probability threshold is reached), respectively. Figure 13 shows the result of AVG queries using the movie dataset. The attribute being aggregated is the year of the movie—i.e., the query returns the average year of the movies that a particular user likes. We see a similar tradeoff as in Figure 12. When the execution time (hence the number of closest data points visited) reaches a certain value, the accuracy stays at a high level. This is because the data points are in a decreasing order of probability; thus the entities that are visited later have smaller probabilities and hence have lower weight in query result. We then use the Amazon dataset and measure the AVG queries. We add an attribute to each product entity called "quality", which is the average rating this product has received (based on all existing ratings of the product). Then we query the average quality of all the products that a particular user likes. The result is shown in Figure 14. Compared to the movie dataset, the Amazon dataset takes slightly longer time to get to high accuracy due to the much larger size of the Amazon data.

Finally, we examine the MAX/MIN queries. For the Freebase dataset, we add an attribute *popularity* to each entity that is the number of related entities (i.e., in-degree plus out-degree)—indicating how popular an entity is. We issue a query to return the maximum popularity of the target entity set. For the movie dataset, we issue a query which returns the minimum year (i.e., the oldest age) of a movie among all the ones that a particular user would like. The result of these two queries are displayed in Figure 15 and Figure 16, respectively. We can see that MAX and MIN queries show a similar tradeoff between performance and accuracy as we have observed for other aggregate queries. These results also verify our analysis in Section V-B.

Summary. The experiments show that our cracking index is very effective in accomplishing our goal for answering two types of important queries of virtual knowledge graphs—top-k entity queries and aggregate queries. The cracking index methods do not have offline index building time, and it takes a little longer for first query (but still 30 to 40 times faster than bulk-loading), with the query processing time slightly shorter than that of the bulk-loaded index. 2 or 3 choice node-split methods provide a tradeoff between a slight increase of initial queries' processing time and getting better performance over the long run. The previous work H2-ALSH cannot handle multiple relationships as in a knowledge graph. Nonetheless when restricted to only one relationship type, our cracking index methods have much smaller overhead for query processing while providing similar or slightly better accuracy; moreover, our methods scale better for larger datasets. The cracking indices only split a tiny fraction of nodes than a full bulk-loaded index, and is very compact and efficient. Lastly, our approximate aggregation methods can obtain high accuracy after a short processing time of the initial data points closest to the query center. Our analysis provides a theoretical guarantee.

VII. Related Work

Graph embedding. Graph embedding represents a graph in a low dimensional space that preserves as much graph property as possible—an elegant method to accomplish automatic feature extraction and dimensionality reduction. Early-days graph embedding, e.g., [26], is based on matrix factorization. It represents graph properties in a matrix and factorizes this matrix to obtain node embedding. More recently, researchers have proposed deep learning based graph embedding methods such as using random walks [27] and autoencoders [28]. Most knowledge graph embedding is based on edge reconstruction based optimization, in particular minimizing margin-based ranking loss. Such embedding methods include TransE [6], TransA [15], KGE-LDA [29], TransD [30], SE [31], MTransE [32], puTransE [33], among others. We have discussed TransE earlier, but our methods can be adapted for most of the other knowledge graph embedding methods, as they all minimize some loss function on h, r, and t for all the edge triplets in the training graph.

Cracking B+ tree and spatial indexing. A cracking B+ tree index [9], [10] aims to amortize the index costs over query processing and dynamically modifies a half-way built B+ tree during query processing. Our proposed indexing is fundamentally different. First, we work with a spatial index R-tree with completely different techniques. Second, our work has a different goal—we are not trying to reduce index maintenance costs; our R-tree index cracking is to avoid splitting R-tree nodes based on the query search space. R-tree is typically the preferred method for indexing spatial data. We use a bulk-loading algorithm of R-tree [7], which is commonly used for efficiently loading data into the index. Note that our method can be easily adapted for other variants of R-tree index (e.g., R+ tree or R* tree) as well.

Dimensionality reduction & nearest neighbors. High dimensionality is a great challenge for k-nearest neighbors (k-NN) queries. Efficient dimensionality reduction includes random projection [34] and locality sensitive hashing (LSH) [13]. We use a particular type of random projection, JL transform [8], but modify it significantly to get a low dimension and provide theoretical guarantees. Different LSH schemes are based on different similarity metrics. The one that is closest to our work is H2-ALSH [12]; we have discussed it in detail and compared against it.

VIII. Conclusions and Future Work

We propose an incremental index to answer top-k entity and aggregate queries over a virtual knowledge graph. Our scheme is based on knowledge graph embedding and transforms embedding vectors to a lower dimensional space for indexing. We prove a tight bound on the accuracy guarantees. Furthermore, we devise query processing algorithms and novel analysis of result accuracy. Experiments show that our index is very concise, and is efficient in answering queries with accuracy guarantees. As future work, we would like to consider dynamic knowledge graph updates. Intuitively, when there are local updates, the embedding changes should be local too, as most (h, r, t) soft constraints still hold. We plan to do incremental updates on our partial index.

Acknowledgments. This work is supported in part by NSF grant IIS-1633271.

References

[1] Google, "Google inside search," https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html, 2018.
[2] Amazon data. Available at http://jmcauley.ucsd.edu/data/amazon/, 2019.
[3] M. Rotmensch, Y. Halpern, A. Tlimat, S. Horng, and D. Sontag, "Learning a health knowledge graph from electronic medical records," *Scientific Reports*, vol. 7, 2017.
[4] Q. Ai, V. Azizi, X. Chen, and Y. Zhang, "Learning heterogeneous knowledge base embeddings for explainable recommendation," *Algorithms*, vol. 11, 2018.
[5] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, "A review of relational machine learning for knowledge graphs," *Proceedings of the IEEE*, vol. 104, 2016.
[6] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *NIPS*, 2013.
[7] S. Shekhar and S. Chawla, *Spatial Databases: A Tour.* Prentice Hall, 2003.
[8] S. Dasgupta and A. Gupta, "An elementary proof of a theorem of Johnson and Lindenstrauss," *Random Structures and Algorithms*, vol. 22, 2003.
[9] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007.
[10] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "The uncracked pieces in database cracking," in *VLDB*, 2013.
[11] N. Alon and J. Spencer, *The Probabilistic Method.* New York: Wiley, 1992.
[12] Q. Huang, G. Ma, J. Feng, Q. Fang, and A. K. H. Tung, "Accurate and fast asymmetric locality-sensitive hashing scheme for maximum inner product search," in *KDD*, 2018.
[13] A. Rajaraman and J. Ullman, *Mining of Massive Datasets*, 2010.
[14] Freebase data. Available at https://developers.google.com/freebase/, 2013.
[15] Y. Jia, Y. Wang, H. Lin, X. Jin, and X. Cheng, "Locally adaptive translation for knowledge graph embedding," in *AAAI*, 2016.
[16] Y. Li, T. Ge, and C. Chen. Online indices for predictive top-k entity and aggregate queries on knowledge graphs. Technical report at http://www.cs.uml.edu/~ge/paper/index_predictive_tech_report.pdf, 2019.
[17] H. Alborzi and H. Samet, "Execution time analysis of a top-down r-tree construction algorithm," *Information Processing Letters*, vol. 101, 2007.
[18] P. J. Denning, "The locality principle," *Communication Networks and Computer Systems*, 2006.
[19] A. A. Borovkov, *Mathematical Statistics.* CRC Press, 1999.
[20] Movielens data. Available at https://grouplens.org/datasets/movielens/latest/, 2017.
[21] R. He and J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering," in *WWW*, 2016.
[22] T. Zaschke, C. Zimmerli, and M. C. Norrie, "The PH-tree: a space-efficient storage structure and multi-dimensional index," in *SIGMOD*, 2014.
[23] H. Cheny, C. Chung, H. Huang, and W. Tsui, "Common pitfalls in training and evaluating recommender systems," *ACM SIGKDD Explorations Newsletter*, 2017.
[24] T. Schnabel, P. N. Bennett, S. T. Dumais, and T. Joachims, "Short-term satisfaction and long-term coverage: Understanding how users tolerate algorithmic exploration," in *WSDM*, 2018.
[25] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval.* Cambridge University Press, 2008.
[26] B. Shaw and T. Jebara, "Structure preserving embedding," in *ICML*, 2009.
[27] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *KDD*, 2014.
[28] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *ICML*, 2016.
[29] L. Yao, Y. Zhang, B. Wei, Z. Jin, R. Zhang, Y. Zhang, and Q. Chen, "Incorporating knowledge graph embeddings into topic modeling," in *AAAI*, 2017.
[30] G. Ji, S. He, L. Xu, K. Liu, and J. Zhao, "Knowledge graph embedding via dynamic mapping matrix," in *ACL*, 2015.
[31] A. Bordes, J. Weston, R. Collobert, and Y. Bengio, "Learning structured embeddings of knowledge bases," in *AAAI*, 2011.
[32] M. Y. C. Z. M. Chen and Y. Tian, "Multilingual knowledge graph embeddings for cross-lingual knowledge alignment,," in *IJCAI*, 2017.
[33] Y. Zhao, Z. Liu, and M. Sun, "Representation learning for measuring entity relatedness with rich information," in *IJCAI*, 2015.
[34] E. Bingham and H. Mannila, "Random projection in dimensionality reduction," in *KDD*, 2001.