Pay-as-you-go Approximate Join Top-k Processing for the Web of Data

Andreas Wagner[†], Veli Bicer[‡], and Thanh Tran[†]
[†] Karlsruhe Institute of Technology and [‡] IBM Research Centre Dublin
{a.wagner,thanh.tran}@kit.edu, velibice@ie.ibm.com

Abstract. For effectively searching the Web of data, ranking of results is a crucial. *Top-k processing* strategies have been proposed to allow an efficient processing of such ranked queries. Top-k strategies aim at computing k top-ranked results *without complete result materialization*. However, for many applications result computation time is much more important than result accuracy and completeness. Thus, there is a strong need for *approximated ranked results*. Unfortunately, previous work on approximate top-k processing is not well-suited for the Web of data. In this paper, we propose the *first approximate top-k join framework for Web data and queries*. Our approach is very lightweight – *necessary statistics are learned at runtime in a pay-as-you-go manner*. We conducted extensive experiments on state-of-art SPARQL benchmarks. Our results are very promising: we could achieve up to 65% time savings, while maintaining a high precision/recall.

Keywords: #swc2014Wagner

1 Introduction

With the proliferation of the Web of data, RDF has become an accepted standard for publishing data on the Web. RDF data comprises a set of *triples* $\{\langle s, p, o\rangle\}$, which forms a data graph, cf. Fig. 1-a.

User-/Query-Dependent Ranking. For web-scale data, queries often produce a large number of results (*bindings*). Given large result sets, *result ranking* becomes a key factor for an effective search. However, ranking functions often need to *incorporate query or user characteristics* [1, 4, 19]:

Example 1. *Find movies with highest ratings, featuring an actress "Audrey Hepburn", and playing close to Rome, cf. Fig. 1.*

Exp. 1 would require a ranking function to incorporate the movie rating, quality of keyword matches for "Audrey Hepburn", and distance of the movie's location to Rome. While one may assume that a higher `rating` value is preferred by any user and query, *scores for keyword and location constraint dependent on query and user characteristics.* For instance, in order to rank a binding for "Audrey Hepburn", a function may measure the edit distance between that keyword and the binding's attribute value, Fig. 1-c. Notice, given another keyword (e.g., only "Audrey"), the very same attribute value would yield a different score. Further, depending on the user's geographic knowledge of Italy, she may have different notions of "closeness" to Rome, e.g., distance ≤ 100 km, cf. Fig. 1-c.

Join Top-k Processing. *Top-k processing* aims at computing k top-ranked bindings without full result materialization [7, 8]. That is, after computing some

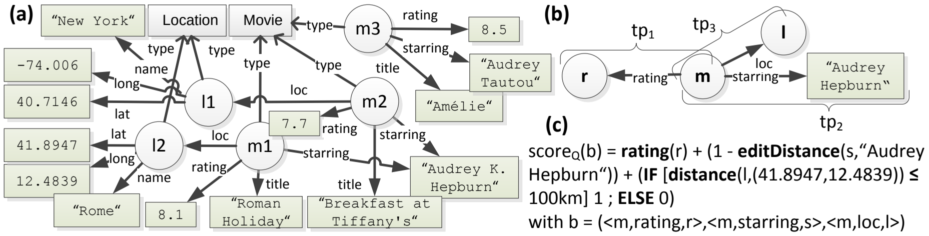

Fig. 1: (a) RDF data graph about the movies "Roman Holiday", "Breakfast at Tiffany's", and "Amélie". (b) Query graph asking for a movie `starring` "Audrey Hepburn". (c) Scoring function that aggregates scores for triple pattern bindings (bold): movie ratings, edit distance w.r.t. "Audrey Hepburn", and distance of the movie's location to Rome (lat: 41.8947, long: 12.4839) $\leq$ 100 km.

bindings, the algorithm can terminate early, because it knows that no binding with higher ranking score exists. For efficiently processing ranked queries over Web data, two recent works employed top-k processing techniques [9, 22].

However, many applications do not require a high result accuracy or completeness. In fact, result computation time is often the key factor. Thus, there is a strong need for *approximated ranked results*. That is, a system should be able to trade off result accuracy and completeness for computation time.

Approximate Join Top-k Processing. Unfortunately, existing approaches for top-k processing over RDF data compute *only exact and complete results* [9, 22]. Moreover, previous works for approximate top-k processing over relational databases [2, 3, 12, 18, 20] are not suitable for Web queries/data. This is because these works *assume complete ranking score statistics at offline time:*

(P.1) *Web Queries.* Query-/user-dependent ranking functions are employed for many important Web queries, e.g., keyword, spatial or temporal queries [1, 4, 19]. However, such *ranking scores are only known at runtime.* Consider tp_2 and tp_3 in Fig. 1-b: binding scores are decided by query (i.e., edit distance to query keyword "Audrey Hepburn") or user characteristics (i.e., the user-defined distance to Rome). So, no offline score statistics can be computed for tp_2 or tp_3.

(P.2) *Web Data.* Web data is commonly *highly distributed and frequently updated.* For instance, movie `ratings` for pattern tp_1 (Fig. 1-b) may be spread across multiple data sources – some of them even "hidden" behind SPARQL endpoints. Moreover, these sources may feature constantly updated `rating` scores. Thus, while constructing an offline statistic for `rating` scores is feasible, it comes with great costs in terms of maintenance. This problem is exacerbated by the fact that RDF allows for very *heterogeneous data.* For example, the `rating` predicate in tp_1 could be used to specify the rating of movies as well as products, restaurants etc. Thus, score statistics may grow quickly and become complex.

Contributions. (1) This is the first work towards approximate top-k join processing for the Web of data. That is, we propose a lightweight approach, which addresses problem P.1 and P.2: (P.1) We learn score distributions in a pay-as-you-go manner at runtime. (P.2) Our score statistics have a constant space complexity and a computation complexity bounded by the result size. (2) We conducted experiments on two SPARQL benchmarks: we could achieve time savings of up to 65%, while still allowing for a high precision/recall.

Outline. We outline preliminaries in Sect. 2 and present the approximate top-k join in Sect. 3. In Sect. 4, we discuss evaluation results. Last, we give an overview over related works in Sect. 5 and conclude with Sect. 6.

2 Preliminaries

Data and Query Model. We use RDF as data model:

Definition 1 (RDF Graph). *Given a set of edge labels ℓ, a RDF graph is a directed labeled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \ell)$, where $\mathcal{V} = \mathcal{V}_E \uplus \mathcal{V}_A$ with entity nodes as $\mathcal{V}_E$ and attribute nodes as $\mathcal{V}_A$. Edges $\mathcal{E} = \{\langle s, p, o \rangle\}$ are called triples, with $s \in \mathcal{V}_E$ as subject, $p \in \ell$ as predicate, and $o \in \mathcal{V}_E \uplus \mathcal{V}_A$ as object.*

An example is depicted in Fig. 1-a. Further, we employ basic graph patterns (BGPs) as query model:

Definition 2 (BGP Query). *A BGP query $\mathcal{Q}$ is a directed labeled graph $\mathcal{Q} = (\mathcal{V}^{\mathcal{Q}}, \mathcal{E}^{\mathcal{Q}})$, with $\mathcal{V}^{\mathcal{Q}} = \mathcal{V}^{\mathcal{Q}}_V \uplus \mathcal{V}^{\mathcal{Q}}_C$ as union of variables $\mathcal{V}^{\mathcal{Q}}_V$ and constants $\mathcal{V}^{\mathcal{Q}}_C$. Edges $\mathcal{E}^{\mathcal{Q}}$ are called triple patterns. Triple pattern $tp = \langle s, p, o \rangle$ with $s \in \mathcal{V}^{\mathcal{Q}}_V \uplus \mathcal{V}^{\mathcal{Q}}_C$, $p \in \ell \uplus \mathcal{V}^{\mathcal{Q}}_V$, and $o \in \mathcal{V}^{\mathcal{Q}}_V \uplus \mathcal{V}^{\mathcal{Q}}_C$. We write $\mathcal{Q}$ as set of its triple patterns: $\mathcal{Q} = \{tp_i\}$.*

Example 2. In Fig. 1-b, pattern $\langle m$, `starring`, "Audrey Hepburn"$\rangle$ has m as variable, constant "Audrey Hepburn" as object, and `starring` as predicate.

Given a query $\mathcal{Q}$, a *binding* b is a vector $(t_1, \ldots, t_n)$ of triples such that: each triple t_i *matches* exactly one pattern $\mathrm{tp_i}$ in $\mathcal{Q}$ and triples in b form a subgraph of the data graph, $\mathcal{G}$. We say b *binds* variables to nodes in the data via the matching of patterns in $\mathcal{Q}$. Formally, for binding b there is a function $\mu_b : \mathcal{V}^{\mathcal{Q}}_V \mapsto \mathcal{V}$ that maps every variable in $\mathcal{Q}$ to an entity/attribute node in the data.

Partial bindings (featuring some patterns with no matching triple) occur during query processing. For a partial binding b, we refer to a pattern $\mathrm{tp_i}$ with no matching triple as *unevaluated* and write $*$ in b's i-th position: $(t_1, \ldots, t_{i-1}, *, t_{i+1}, \ldots, t_n)$. We denote the set of unevaluated patterns for partial binding b as $\mathcal{Q}^u(b) \subseteq \mathcal{Q}$. A binding b *comprises* a binding b', if all triples in b' are also contained in b. If b comprises b', we say binding b' *contributes* to b.

Example 3. Given Fig. 1-b, a partial binding $b_{31} = (*, *, t_{31} = \langle m_1$, `loc`, $l_2 \rangle)$ in Fig. 2-a matches pattern tp_3, while $\mathcal{Q}^u(b_{31}) = \{tp_1,\ tp_2\}$ are unevaluated. b_{31} binds variable m and l to entity m_1 and l_1. Further, the complete binding $b = (t_{12}, t_{21}, t_{31})$ comprises partial binding $b_{31} = (*, *, t_{31})$. b_{31} contributes to b.

Ranking Function. To quantify the relevance of a binding b w.r.t. a query/user, we employ a *ranking function*: $score_{\mathcal{Q}} : \mathcal{B}^{\mathcal{Q}} \mapsto \mathbb{R}$, with $\mathcal{B}^{\mathcal{Q}}$ as set of all partial/complete bindings for $\mathcal{Q}$. That is, $score_{\mathcal{Q}}(b)$ is defined as aggregation over b's triples: $score_{\mathcal{Q}}(b) = \bigoplus_{t \in b} score_{\mathcal{Q}}(t)$, with $\oplus$ as monotonic aggregation function. A ranking function for our example is in Fig. 1-c. Note, $score_{\mathcal{Q}}$ could be defined as part of the query, e.g., by means of the `ORDER BY` clause in SPARQL.

Sorted Access. For every pattern $\mathrm{tp_i}$ in query $\mathcal{Q}$, a *sorted access sa_i* retrieves *matching triples in descending score order.* Previous works on join top-k processing over Web data introduced efficient sorted access implementations for RDF stores [9, 22]. Let us present simple approaches for our example (Fig. 2-a):

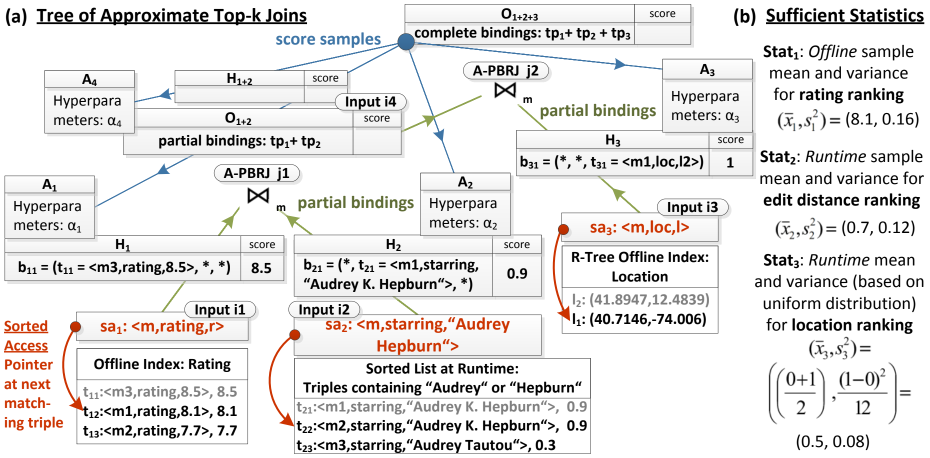
(a) Tree of Approximate Top-k Joins
score samples
O_{1+2+3}
score
complete bindings: tp_1+ tp_2 + tp_3
A_4
Hyperparameters: α_4
H_{1+2}
score
Input i4
O_{1+2}
score
partial bindings: tp_1+ tp_2
A-PBRJ j2
$\bowtie_m$
partial bindings
A_3
Hyperparameters: α_3
H_3
score
b_{31} = (*, *, t_{31} = <m1,loc,l2>)
1
A_1
Hyperparameters: α_1
A-PBRJ j1
$\bowtie_m$
partial bindings
A_2
Hyperparameters: α_2
H_1
score
b_{11} = (t_{11} = <m3,rating,8.5>, *, *)
8.5
H_2
score
b_{21} = (*, t_{21} = <m1,starring, “Audrey K. Hepburn“>, *)
0.9
Input i3
sa_3: <m,loc,l>
R-Tree Offline Index: Location
l_2: (41.8947,12.4839)
l_1: (40.7146,-74.006)
Sorted Access Pointer at next matching triple
Input i1
sa_1: <m,rating,r>
Offline Index: Rating
t_{11}:<m3,rating,8.5>, 8.5
t_{12}:<m1,rating,8.1>, 8.1
t_{13}:<m2,rating,7.7>, 7.7
Input i2
sa_2: <m,starring,“Audrey Hepburn“>
Sorted List at Runtime: Triples containing “Audrey“ or “Hepburn“
t_{21}:<m1,starring,“Audrey K. Hepburn“>, 0.9
t_{22}:<m2,starring,“Audrey K. Hepburn“>, 0.9
t_{23}:<m3,starring,“Audrey Tautou“>, 0.3
(b) Sufficient Statistics
Stat_1: Offline sample mean and variance for rating ranking
$(\bar{x}_1, s_1^2) = (8.1, 0.16)$
Stat_2: Runtime sample mean and variance for edit distance ranking
$(\bar{x}_2, s_2^2) = (0.7, 0.12)$
Stat_3: Runtime mean and variance (based on uniform distribution) for location ranking
$(\bar{x}_3, s_3^2) = \left(\left(\frac{0+1}{2}\right), \frac{(1-0)^2}{12}\right) =$
(0.5, 0.08)

Fig. 2: (a) A-PBRJ tree for Fig. 1-b. Two information flows occur in the tree: partial bindings (green) and score samples (blue). (b) Sufficient statistics based on scores observed at indexing time (stat_1) and runtime (stat_2 and stat_3).

Example 4. Given the keyword pattern $tp_2 = \langle m,$ `starring`, “Audrey Hepburn”$\rangle$, a sorted access must materialize all triples, which have a value that contains “Audrey” or “Hepburn”. After materialization, these triples are sorted with descending similarity w.r.t. that keyword (e.g., measured via edit distance). On the other hand, for pattern $\langle m,$ `loc`, $l\rangle$, an R-tree on the attribute pair (`lat`, `long`) may be used. This offline computed index yields two hits: l_1 and l_2. While l_2 is an exact match (thus, triple t_{31} has max. score 1), l_1 is more distant from Rome. Last, an index for attribute `rating` can be constructed offline: triples are stored with descending rating value. Then, sorted access sa_1 can iterate over this list.

Partial bindings retrieved from sorted accesses are combined via joins. That is, an equi-join combines two (or more) inputs. This way, multiple joins form a tree. For instance, three sorted accesses are combined via two joins in Fig. 2-a.

Problem. Our goal is to compute k high-ranked query bindings that may differ from the true top-k results in terms of false positives/negatives. These approximations aim at saving computation time. For this, we use a top-k test: *given a partial binding, we estimate its probability for contributing to the final top-k results and discard such bindings that have only a small a probability.*

We exploit *conjugate priors* for learning necessary probability distributions.

Bayesian Inference. Let Θ be a set of parameters. One may model *prior beliefs* about these parameters in the form of probabilities: $\Theta \sim P(\Theta \mid \alpha)$ with $\Theta \in \mathbf{\Theta}$ [6]. Here, α is a vector of *hyperparameters* allowing to parametrize the prior distribution. Suppose we observe relevant data $\mathbf{x} = \{x_1, \ldots, x_n\}$ w.r.t. Θ, where each $x_i \sim P(x_i \mid \Theta)$. Then, the dependency between observations $\mathbf{x}$ and prior parameters Θ can be written as $P(\mathbf{x} \mid \Theta)$. Using the Bayes theorem we can estimate a *posterior* probability, which captures parameters Θ conditioned on observed events $\mathbf{x}$. In simple terms, *a posterior distribution models how likely parameters Θ are, in light of the seen data $\mathbf{x}$ and the prior beliefs* [6]:

$$P(\Theta \mid \mathbf{x}, \alpha) \propto P(\mathbf{x} \mid \Theta) \cdot P(\Theta \mid \alpha) = \frac{P(\mathbf{x} \mid \Theta) \cdot P(\Theta \mid \alpha)}{\sum_{\mathbf{\Theta}} P(\mathbf{x} \mid \Theta) P(\Theta)} \tag{1}$$

Example 5. *For pattern tp_1 in Fig-2-a, scores are based on rating values. So, we can compute sufficient statistics (mean $\bar{x}_1 = 8.1$ and variance $s_1^2 = 0.16$) for these scores at offline time, cf. $stat_1$ in Fig-2-b. Such statistics represent prior beliefs about the "true" distribution, which is capturing only those scores for bindings of tp_1 that are part of a complete binding. Only triple t_{12} and t_{13} contribute to complete bindings. Thus, only their scores should be modeled via a distribution. We update the prior beliefs using scores samples $\mathbf{x}$ observed during query processing, thereby learning the true (posterior) distribution as we go.*

As we are interested in *unobserved* events x^*, we need the *posterior predictive distribution*, i.e., the distribution of new events given observed data $\mathbf{x}$:

$$P(x^* \mid \mathbf{x}, \alpha) = \sum_{\mathbf{\Theta}} P(x^* \mid \Theta) P(\Theta \mid \mathbf{x}, \alpha) \tag{2}$$

An important kind of Bayesian priors are the *conjugate priors*. Intuitively, conjugate priors require the posterior and prior distribution to belong to the same distribution family. In other words, these priors provide a "computational convenience", because they give a closed-form of the posterior distribution [6]. Thus, posterior computation is easy and efficient for conjugate priors.

3 Approximate Top-k Join

We now present an *approximate* top-k processing for the Web of data. In contrast to existing works [2, 3, 12, 18, 20], we follow a *lightweight* approach: (1) We *learn all necessary score statistics at runtime*, cf. Algo. 2 (P.1, Sect. 1). (2) We show our score distribution learning to have a *constant space complexity* and a *runtime complexity bounded by the result size*, cf. Thm. 1 (P.2, Sect. 1).

3.1 Approximate Rank Join Framework

We follow [17] and define an approximate Pull/Bound Rank Join (A-PBRJ) framework that comprises three parts: a pulling strategy $\mathcal{PS}$, a bounding strategy $\mathcal{BS}$, and a probabilistic component $\mathcal{PC}$. $\mathcal{PS}$ determines the next join input to pull from [17]. The bounding strategy $\mathcal{BS}$ gives an upper bound, β, for the maximal possible score of unseen join results [17]. Last, we use $\mathcal{PC}$ to estimate a probability for a partial binding to contribute to the final top-k result.

Approximate Pull/Bound Rank Join. The A-PBRJ is depicted in Algo. 1. Following [17], on line 4 we check whether output buffer $\mathbf{O}$ comprises k complete bindings and if there are unseen bindings with higher scores (measured via bound β). If both conditions hold, the A-PBRJ terminates and reports $\mathbf{O}$. Otherwise, $\mathcal{PS}$ selects an input i to pull from (line 5) and produces a new partial binding b from the sorted access on input i, line 6. After materialization, we update β using bounding strategy $\mathcal{BS}$.

Example 6. *In Fig. 2-a, join j_2 decides (via strategy $\mathcal{PS}$) to first pull on sa_3 and load partial binding t_{31}. Then, join j_2 pulls on input i_4 (join j_1), which in turn pulls on its input i_1 (sa_1) loading binding t_{11} and afterwards on input i_2 (sa_2) loading t_{21}. The join attempt $t_{11} \bowtie t_{21}$ in join j_1 fails, because entity $m_3 \neq m_1$.*

Algorithm 1: Approx. Pull/Bound Rank Join (A-PBRJ).

```
   Param.: Pulling strategy $\mathcal{PS}$, bounding strategy $\mathcal{BS}$, probabilistic comp. $\mathcal{PC}$.
   Index  : Sorted access $sa_i$ and $sa_j$ for input $i$ and $j$, respectively.
   Buffer : Output buffer $\mathbf{O}$. $\mathbf{H}_i$ and $\mathbf{H}_j$ for "seen" bindings from $sa_i$ and $sa_j$.
   Input  : Query $\mathcal{Q}$, result size $k$, and top-$k$ test threshold $\tau$.
   Output: Approximated top-$k$ result.
 1 begin
 2     $\beta \leftarrow \infty, \quad \kappa \leftarrow -\infty$
 3     $\mathcal{PC}$.initialize()
 4     while $\mid \mathbf{O} \mid < k$ or $\min_{b' \in \mathbf{O}} score_{\mathcal{Q}}(b') < \beta$ do
 5         $i \leftarrow \mathcal{PS}$.input() // choose next input via pulling strategy $\mathcal{PS}$
 6         $b \leftarrow$ next partial binding from sorted access $sa_i$
 7         $\beta \leftarrow \mathcal{BS}$.update($b$) // update $\beta$ via bounding strategy $\mathcal{BS}$
           // top-$k$ test, cf. Algo. 3
 8         if $\mathcal{PC}$.probabilityTopK($b$,$\kappa$) $> \tau$ then
 9             $\mathbf{O} \leftarrow \mathbf{H}_j \bowtie \{b\}$
10             $b \cup \mathbf{H}_i$ // add $b$ to buffer $\mathbf{H}_i$
11             if #new bindings $\mathbf{b}$ in $\mathbf{O} \geq$ training threshold then
                   // score distribution learning, cf. Algo. 2
12                 $\mathcal{PC}$.train($\mathbf{b}$)
13                 Retain only $k$ top-ranked bindings in $\mathbf{O}$
14             if $\mid \mathbf{O} \mid \geq k$ then $\kappa \leftarrow \min_{b' \in \mathbf{O}} score_{\mathcal{Q}}(b')$
       // return approximated top-$k$ results
15     return $\mathbf{O}$
```

In line 8, $\mathcal{PC}$ estimates the probability for partial binding b leading to a complete top-k binding: *the top-k test*. If b fails this test, it will be *pruned*. That is, we do not attempt to join it and do not insert it in $\mathbf{H}_i$. $\mathbf{H}_i$ is a buffer that holds "seen" bindings from input i. Otherwise, if the top-k test holds, b is further processed (lines 9 - 14). That is, we join b with seen bindings from the other input j and add results to $\mathbf{O}$. Further, b is inserted into buffer $\mathbf{H}_i$, line 10. For learning the necessary probability distributions, $\mathcal{PC}$ trains on seen bindings/-scores in $\mathbf{O}$, line 12. Notice, we continuously train $\mathcal{PC}$ throughout the query processing – every time "enough" new bindings are in $\mathbf{O}$, line 11. $\mathcal{PC}$ requires parameter κ for its pruning decision. κ holds the the smallest currently known top-k score (line 14). On line 2, κ is initialized as $-\infty$.

Choices for $\mathcal{BS}$ and $\mathcal{PS}$. Multiple works proposed bounding strategies, e.g., [5, 7, 10, 17] as well as pulling strategies, e.g., [7, 11]. Commonly, the *corner bound* [7] is employed as bounding strategy $\mathcal{BS}$:

Definition 3 (Corner Bound). *For a join operator, we maintain u_i and l_i for each input i. u_i is the highest score observed from i, while l_i is the lowest observed score on i. If input i is exhausted, l_i is set to $-\infty$. The bound for scores of unseen join results is $\beta := \max\{u_1 \oplus l_2, u_2 \oplus l_1\}$.*

In example Fig. 2-a, join j_1 currently has $\beta = \max\{8.5 + 0.9, 0.9 + 8.5\}$, with $u_1 = l_1 = 8.5$ and $u_2 = l_2 = 0.9$. On the other hand, the *corner-bound-adaptive strategy* [7] is frequently used as pulling strategy $\mathcal{PS}$:

Definition 4 (Corner-Bound-Adaptive Pulling). *The corner-bound-adaptive pulling strategy chooses the input i such that: $i = 1$ iff $u_1 \oplus l_2 > u_2 \oplus l_1$ and $i = 2$ otherwise. In case of a tie, the input with less unseen bindings is chosen.*

For instance, in join j_1 (Fig. 2-a) either input may be selected, because $8.5+0.9 = 0.9 + 8.5$ and both inputs have two unseen partial bindings.

3.2 Probabilistic Component $\mathcal{PC}$

Given a partial binding b, we wish to know how likely b will contribute to the final top-k results. For this, the top-k test exploits two probabilities: (1) The probability that b contributes to a complete binding (*binding probability*). (2) The probability that complete bindings comprising b have higher scores than the current top-k bindings (*score probability*).

Binding Probability. To address the former probability, we use a selectivity estimation function *sel*. Simply put, given a query $\mathcal{Q}$, $sel(\mathcal{Q})$ estimates the probability that there is at least one binding for $\mathcal{Q}$ [14, 15]. For example, selectivity of pattern $tp_3 = \langle m, \texttt{loc}, l\rangle$ is $sel(tp_3) = \frac{2}{3}$, because out of the three movie entities only two have a `loc` predicate, cf. Fig. 1-a.

Further, we define a *complete binding indicator* for a partial binding b:

$$\mathbf{1}\{\mathcal{Q}^u(b) \mid b\} := \begin{cases} 1 & \text{if } sel(\mathcal{Q}^u(b) \mid b) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Intuitively, for a partial binding b, $\mathbf{1}\{\mathcal{Q}^u(b) \mid b\}$ models *whether matching triples for b's remaining unevaluated patterns can exist, given variable assignments dictated by* b. That is, $\mathcal{Q}^u(b) \mid b$ is a set of patterns $\{\overline{\text{tp}}_i\}$, such that pattern $\text{tp}_i \in \mathcal{Q}^u(b)$ and each variable v in tp_i that is bound by b is replaced with its assignment in b, $\mu_b(v)$, which results in a new pattern $\overline{\text{tp}}_i$.

Example 7. Consider partial binding $b_{11} = (t_{11} = \langle m_3,$ `rating`$, 8.5\rangle, *, *)$ in Fig. 2-a. $\mathcal{Q}^u(b_{11}) \mid b_{11} = \{\langle m_3,$ `starring`$,$ "Audrey Hepburn"$\rangle, \langle m_3,$ `loc`$, l\rangle\}$, because variable m in pattern tp_2 and tp_3 is replaced with its assignment in b_{11}, $\mu_{b11}(m) = m_3$. $\mathbf{1}\{\mathcal{Q}^u(b_{11}) \mid b_{11}\} = 0$, as selectivity for both patterns is 0.

Notice, *any selectivity estimation implementation* may be used for the complete binding indicator. We employed [14, 15] for our experiments.

Score Probability. For a partial binding b, let *scores for bindings of b's unevaluated patterns, $\mathcal{Q}^u(b)$*, be captured via a random variable $X^s_{\mathcal{Q}^u(b)}$.

Example 8. In Fig. 2-a, partial binding b_{31} currently has a score of 1. However, scores for bindings to tp_1 and tp_2 are unknown and modeled via $X^s_{\mathcal{Q}^u(b_{31})}$.

Then, we can obtain the probability for b contributing to a complete binding that has a score $\geq x$ as:

$$P\left(X^s_{\mathcal{Q}^u(b)} \geq \delta(x, b)\right) \tag{4}$$

where $\delta(x, b) := x - score_{\mathcal{Q}}(b)$. Note, partial binding b has a current score, $score_{\mathcal{Q}}(b)$, and only the score for its unevaluated patterns is unknown. So, $\delta(x, b)$ is the "delta" between b's current score and a desired score x.

Top-k Test. Finally, we use (1) the complete binding indicator to determine whether b might contribute to any complete binding. Further, (2) the score

	(a) Predictive Dist.	**(b) Priors**
Input i_1	$P(X^s_{i_1})$ $\mathcal{Q}^u = \{tp_2, tp_3\}$	stat$_2\oplus$ stat$_3$: $(0.7 + 0.5, 0.12 + 0.08)$
Input i_2	$P(X^s_{i_2})$ $\mathcal{Q}^u = \{tp_1, tp_3\}$	stat$_1\oplus$ stat$_3$: $(8.1 + 0.5, 0.16 + 0.08)$
Input i_3	$P(X^s_{i_3})$ $\mathcal{Q}^u = \{tp_1, tp_2\}$	stat$_1\oplus$ stat$_2$: $(8.1 + 0.7, 0.16 + 0.12)$
Input i_4	$P(X^s_{i_4})$ $\mathcal{Q}^u = \{tp_3\}$	stat$_3$: $(0.5, 0.08)$

Fig. 3: (a) Given joins in Fig. 2-a, we train four predictive score distributions (one for each input). For instance, $X^s_{i_1}$ models scores for bindings of tp$_2$ $\bowtie$ tp$_3$. (b) Priors are based on sufficient statistics in Fig. 2-b. The aggregation function $\oplus$ is a summation in Fig. 1-c. Thus, e.g., stat$_1$ $\oplus$ stat$_3$ $= (8.1 + 0.5, 0.16 + 0.08) = (8.6, 0.24)$.

probability to estimate how likely a complete binding comprising b has a score that is larger than the smallest known top-k score, κ (cf. Algo. 1 line 14):

$$\underbrace{\mathbb{1}(\mathcal{Q}^u(b) \mid b)}_{(1)} \cdot \underbrace{P(X^s_{\mathcal{Q}^u(b)} \geq \delta(\kappa, b))}_{(2)} > \tau \tag{5}$$

with $\tau \in [0, 1]$ as top-k test threshold.

3.3 Score Distribution Learning

Distributions for random variables $X^s_{\mathcal{Q}^u(b)}$ may be obtained by learning a score distribution $P(X^s_i)$ for each join input i. Note, partial bindings, which come from the same input, have the same set of unevaluated triple patterns. Thus, X^s_i *captures scores of the unevaluated patterns from its partial bindings.*

Example 9. *In Fig. 2-a, all partial bindings from input i_1 have $\mathcal{Q}^u = \{tp_2,\ tp_3\}$ as unevaluated patterns. Thus, $P(X^s_{\mathcal{Q}^u(b_{11})}) = P(X^s_{i_1})$, as binding b_{11} is produced by input i_1. In fact, all bindings from i_1 follow the same distribution, $P(X^s_{i_1})$, which captures scores of $tp_2 \bowtie tp_3$. Overall, we learn four distributions, cf. Fig. 3.*

We do not know the true distribution for X^s_i. In such a case, a common assumption is to use a *Gaussian distribution* for X^s_i, cf. Eq. 6a. We employ a conjugate prior to train its unknown mean and variance, respectively.

As shown in [6], the mean of X^s_i follows a Gaussian distribution (Eq. 6b) and the variance of X^s_i follows an inverse-Gamma distribution (Eq. 6c). Hyperparameters $\alpha_0 = (\mu_0, \eta_0, \sigma_0^2, \nu_0)$ parameterize both distributions, where μ_0 is prior mean with quality η_0, and σ_0^2 is prior variance with quality ν_0 [6]:

$$X^s_i \sim \text{normal}\left(\mu, \sigma^2\right) \tag{6a}$$

$$\mu \mid \sigma^2 \sim \text{normal}\left(\mu_0, \frac{\sigma^2}{\eta_0}\right) \tag{6b}$$

$$\sigma^2 \sim \text{inverse-gamma}\left(0.5 \cdot \nu_0, 0.5 \cdot \nu_0 \sigma_0^2\right) \tag{6c}$$

Prior Distribution. Prior initialization is called on line 3 in Algo. 1. For each input i we specify a prior distribution for X^s_i via prior hyperparameters α_0. For α_0 we require sufficient score statistics in the form of a sample mean, $\bar{x} = \frac{1}{n} \sum_{x_i \in \mathbf{x}} x_i$, and a sample variance $s^2 = \frac{1}{(n-1)} \sum_{x_i \in \mathbf{x}} (x_i - \bar{x})^2$, with $\mathbf{x}$ as sample. There are multiple ways to obtain the necessary score samples:

Example 10. *Fig. 2-b depicts three sufficient statistics based on information from the sorted accesses: (1) Offline information in the case of sa_1. That is, scores*

Algorithm 2: $\mathcal{PC}$.train()

```
   Params: Weight $w \geq 1$ for score sample $\mathbf{x}$.
   Buffer : Buffer $\mathbf{A}$ storing hyperparameters $\alpha$.
   Input  : Complete bindings $\mathbf{B} \subseteq \mathbf{O}$ and join $j$.
1  begin
2      foreach input $i$ in join $j$ do
           // load prior hyperparameters for input $i$
3          $\alpha_n = (\mu_n, \eta_n, \sigma_n^2, \nu_n) \leftarrow \mathbf{A}_i$
           // get scores of bindings for input $i$'s unevaluated patterns
4          foreach complete binding $b \in \mathbf{B}$ do
5              get binding $b'$ comprised in $b$, which matches unevaluated patterns
6              add $score_{\mathcal{Q}}(b')$ to score sample $\mathbf{x}$
           // compute sample mean and variance
7          $\bar{x} \leftarrow \mathrm{mean}(\mathbf{x}) = \frac{1}{n}\sum x_i$
8          $s^2 \leftarrow \mathrm{var}(\mathbf{x}) = \frac{1}{(n-1)}\sum(x_i - \bar{x})^2$
           // compute posterior hyperparameters
9          $\nu_{n+1} \leftarrow \nu_n + w, \quad \eta_{n+1} \leftarrow \eta_n + w$
10         $\mu_{n+1} \leftarrow \frac{1}{\eta_{n+1}} \cdot (\eta_n \mu_n + w\bar{x})$
11         $\sigma_{n+1}^2 \leftarrow \frac{1}{\nu_{n+1}} \cdot \left(\nu_n \sigma_n^2 + (w-1)s^2 + \frac{\eta_n w}{\eta_{n+1}} \cdot (\bar{x} - \mu_n)^2\right)$
           // store new (posterior) hyperparameters for input $i$
12         $\mathbf{A}_i \leftarrow \alpha_{n+1} = (\mu_{n+1}, \eta_{n+1}, \sigma_{n+1}^2, \nu_{n+1})$
```

are known before runtime, thus, $\bar{x}_1 = 8.1$ *and* $s_1^2 = 0.16$ *can be computed offline. (2) Online information for access* sa_2*. Recall, the list of matching triples for keywords "Audrey" and "Hepburn" must be fully materialized. So,* $\bar{x}_2 = 0.7$ *and* $s_2^2 = 0.12$ *may be computed from runtime score samples. (3) Last, given access* sa_3*, we have neither offline scores, nor a fully materialized list of triples (*sa_3 *loads a triple solely upon a pull request). In lack of more information, we assume each score to be equal likely, i.e., a uniform distribution. With min. score as* 0 *and max. score as* 1*:* $\bar{x}_3 = 0.5$ *and* $s_3^2 = 0.08$*.*

We initialize hyperparameters α_0 with μ_0 as sample mean, σ_0^2 as sample variance, and $\eta_0 = \nu_0$ as sample quality. For every input, we aggregate necessary sample means/variances for μ_0/σ_0^2. For example, given input i_1 with unevaluated pattern {tp$_2$, tp$_3$}, we sum up (aggregate) statistics stat$_2$ and stat$_3$: $\bar{x}_2 + \bar{x}_3$ for μ_0 and $s_2^2 + s_3^2$ for σ_0^2, cf. Fig. 3. Note, η_0 and ν_0 are used to quantify the prior quality. For instance, stat$_1$ and stat$_2$ are exact statistics, while stat$_3$ relies on a uniform distribution. So, weighting reflects the prior's trustworthiness.

Posterior Distribution. Having estimated a prior distribution, *we continuously update the distribution with scores seen during query processing.*

Intuitively, each time new complete bindings are produced, all prior distributions could be trained, cf. Algo. 1 line 11 and Algo. 2. That is, complete binding scores are used to update hyperparameters from the previous n-th training iteration, α_n, resulting in new posterior hyperparameters, α_{n+1}. For this, we use standard training on lines 10-11 (Algo. 2) [6]. In simple terms, the prior mean μ_n is updated with the new sample mean $\bar{x}$, line 10, and the prior variance σ_n^2

is updated with the sample variance s^2, line 11. Note, each input computes its "own" score sample $\mathbf{x}$ (Algo. 2, lines 5-6).

Prior hyperparameters are weighted via η_n and ν_n. Further, for each hyperparameter update, a parameter w is used as weight (indicating the quality of samples $\mathbf{x}$). Finally, new hyperparameters α_{n+1} are stored on line 12, Algo. 2.

Example 11. *Given input i_1 and $\eta_0 = \nu_0 = 1$ in Fig. 3. Then, its prior is $\alpha_0 = (1.2, 1, 0.2, 1)$. We observe scores $\mathbf{x} = \{x_1, x_2\}$ from $\mathbf{B} = \{(t_{12}, t_{21}, t_{31}), (t_{13}, t_{22}, t_{32})\}$, with $w = |\mathbf{x}| = 2$, $x_1 = 1.9 = score_{\mathcal{Q}}(t_{21}) + score_{\mathcal{Q}}(t_{31})$, and $x_2 = 0.9 = score_{\mathcal{Q}}(t_{22}) + score_{\mathcal{Q}}(t_{32})$. So, $s^2 = 0.5$, $\bar{x} = 1.4$, which leads to posterior hyperparameters: $\eta_1 = \nu_1 = 1 + 2 = 3$ and*

$$\sigma_1^2 = \frac{1}{3} \cdot \left(0.2 + (2-1) \cdot 0.5 + \frac{(1.4 - 1.2)^2}{3}\right) = 0.71$$

$$\mu_1 = \frac{(1.2 + 2 \cdot 1.4)}{3} = 1.33$$

After each such update only posterior hyperparameters are stored, thereby making the learning highly time and space efficient:

Theorem 1 (Score Distribution Learning Time/Space Complexity). *Given an A-PRBJ operator j, at any time during query processing, we require $O(1)$ of space for score distribution learning. Further, given $\mathbf{B}$ complete bindings, score learning time complexity is bounded by $O(|\mathbf{B}|)$.*

Proof. A proof can be found in our report [21].

Algorithm 3: $\mathcal{PC}$.probabilityTopK()

```
Buffer : Buffer $\mathbf{A}$ storing hyperparameters.
Input  : Partial bindings $b$, input $i$, and join $j$.
Output : Probability that $b$ will result in one (or more) final top-$k$ bindings.
1 begin
      // load hyperparameters $\alpha_n$ for input $i$
2     $\alpha_n = (\mu_n, \eta_n, \sigma_n^2, \nu_n) \leftarrow \mathbf{A}_i$
      // posterior predictive distribution based on hyperparam. $\alpha_n$
      // in closed-form as Student's $t$-distribution
3     $X_i^s \sim t_{(\nu_n)}\left(x \mid \mu_n, \frac{\sigma_n^2(\eta_n+1)}{\eta_n}\right)$
      // compute score probability
4     $p_{\mathcal{S}} \leftarrow P\left(X^s_{\mathcal{Q}^u(b)} \geq \delta(\kappa, b)\right) = P\left(X_i^s \geq \delta(\kappa, b)\right)$
      // compute binding probability
5     $p_{\mathcal{B}} \leftarrow \mathbf{1}\{\mathcal{Q}^u(b) \mid b\}$
      // probability that $b$ contributes to top-$k$ results
6     return $p_{\mathcal{S}} \cdot p_{\mathcal{B}}$
```

Predictive Distribution. In Algo. 3, we provide an implementation of the top-k test. At any point during query processing, one may need to perform this test, Algo. 1 line 8. Thus, our approach allows to always give a distribution for X_i^s based on the currently known hyperparameters α_n (Algo. 3, line 2). Since hyperparameters are continuously trained, the distributions improve over time.

More specifically, we use the posterior predictive distribution. This distribution estimates probabilities for *new* scores, based on observed scores and the prior distribution. For a Gaussian conjugate prior, this distribution can be easily obtained in a closed form as non-standardized Student's t-distribution with ν_n degrees of freedom [6], cf. Algo. 3, line 3. Then, we compute $P(X^s_{\mathcal{Q}^u(b)}) = P(X^s_i)$ by means of the posterior predictive distribution on line 4. Last, we compute the binding probability via a selectivity estimation function (Eq. 3) on line 5 and return b's top-k test probability, cf. line 6.

4 Evaluation

Benchmarks. We used two SPARQL benchmarks: (1) The SP^2 benchmark featuring synthetic DBLP data [16]. (2) The DBpedia SPARQL benchmark (DBPSB), which holds real-world DBpedia data and queries [13]. For both benchmarks we generated datasets with 10M triples. We translated the SPARQL benchmark queries to our query model (BGPs). Queries featuring no BGPs were discarded, i.e., we omitted 12 and 4 queries in DBPSB and SP^2. We generated DBPSB queries as proposed in [13]: Overall, used 8 seed queries with 15 random bindings, which led to a total of 120 DBPSB queries. For SP^2 we employed 13 queries. In total, we had a comprehensive load of 133 queries. Query statistics and a complete query listing is given in [21].

Systems. We randomly generated bushy query plans. For a given query, all systems rely on the same plan. We implemented three systems that solely differ in their join operator: (1) A system with *join-sort* operator, `JS`, which *does not employ top-k processing*, but instead produces all results and then sorts them. (2) An *exact* and complete top-k join operator, `PBRJ`, featuring the corner-bound in Def. 3 and the corner-bound-adaptive pulling strategy in Def. 4. `PBRJ` is identical to Algo. 1, however, no top-k test is applied. Note, `PBRJ` resembles previous approaches for top-k processing over RDF data [9, 22]. (3) Last, we implemented our *approximate* operator, `A-PBRJ`, see Algo 1 in Sect. 3.

Score learning and top-k test implementation for the `A-PBRJ` operator follows Algo. 2 and Algo. 3, cf. Sect. 3.3. Further, we used sufficient statistics based on a uniform distribution over $[0, 1]$, as discussed in Exp. 10 for sorted access sa_3. Prior weights ν_0 and η_0 are both 1, Algo. 2. Weight w in Algo. 2 is the sample size, $|\mathbf{x}|$. We reused the selectivity estimation implementation from [14, 15] for our binding probabilities.

Hypothesis (H.1): We expect that `JS` is outperformed by `PBRJ`, as it computes all results for a query. Further, we expect `A-PBRJ` to outperform `JS` and `PBRJ`. `A-PBRJ`'s savings come at the cost of effectiveness.

We implemented all systems in Java 6. Experiments were run on a Linux server with two Intel Xeon 5140 CPUs at 2.33GHz, 48GB memory (16GB assigned to the JVM), and a RAID10 with IBM SAS 148GB 10K rpm disks. Before each query execution, all operating system caches were cleared. The presented values are averages collected over five runs.

Ranking Function. We chose triple pattern binding scores, $score_{\mathcal{Q}}(t)$, at random with distribution $d \in \{u, n, e\}$ (uniform, normal, and exponential dis-

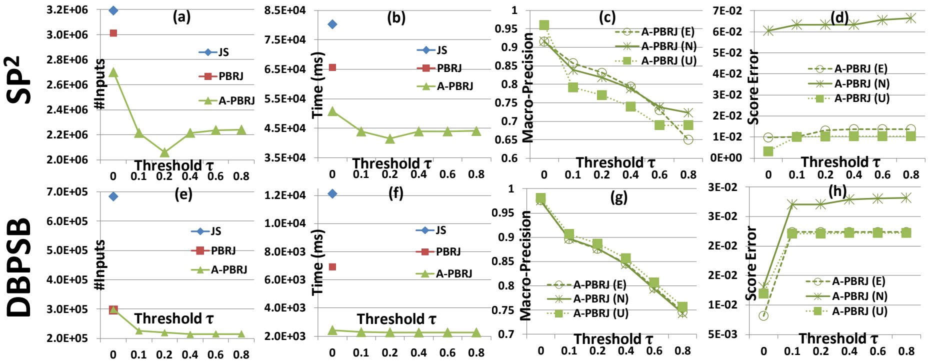

Fig. 4: Evaluation results for SP^2/DBPSB: (a)/(e) Efficiency: #inputs vs. threshold τ. (b)/(f) Efficiency: time vs. threshold τ. (c)/(g) Effectiveness: macro-precision vs. threshold τ. (d)/(h) Effectiveness: score error vs. threshold τ.

tribution). We employed a summation as aggregation function, $\oplus$. By means of varying distributions, we aim at an abstraction from a particular ranking function and examine performance for different "classes" of functions. We employed standard parameters for all distributions and normalized scores to be in $[0, 1]$. *Hypothesis (H.2): `A-PBRJ`'s efficiency and effectiveness is not influenced by the score distribution.*

Parameters. We vary the number of results $k \in \{1, 5, 10, 20\}$. *Hypothesis (H.3): We predict efficiency to decrease in parameter k for `A-PBRJ` and `PBRJ`.* Further, we used top-k test thresholds $\tau \in [0, 0.8]$ for inspecting the trade-off between efficiency and effectiveness.

Metrics. We measure efficiency via: (1) #Inputs processed. (2) Time needed for result computation. As effectiveness metrics we use: (1) Precision: fraction of approximated top-k results that are exact top-k results. (2) Recall: fraction of exact top-k results, which are reported as approximate results. Notice, precision and recall have identical values, as both share the same denominator k. We therefore discuss only precision results in the following. Further, precision is given as average over our query load (so-called macro-precision). (3) Score error: approximate vs. exact top-k score: $\frac{1}{k}\sum_{b=1,\ldots,k}|score^*_{\mathcal{Q}}(b) - score_{\mathcal{Q}}(b)|$, with $score^*_{\mathcal{Q}}(b)$ and $score_{\mathcal{Q}}(b)$ as approximated and exact score for binding b [20].

Efficiency Results. Efficiency results are depicted in Fig. 4-a/e (b/f) for SP^2 (DBPSB). *As expected in hypothesis H.1, we observed `A-PBRJ` to save #inputs and computation time.* For SP^2 (DBPSB), `A-PBRJ` needed up to 25% (23%) less inputs vs. baseline `PBRJ` and 30% (67%) vs. `JS`. We explain these gains with pruning of partial bindings via our top-k test, thereby omitting "unnecessary" joins and join attempts. In fact, we were able to prune up to 40% (90%) of the inputs, given SP^2 (DBPSB). Fewer #inputs translated to time savings of 35% (65%) vs. `PBRJ` and 47% (80%) vs. `JS`, given SP^2 (DBPSB).

Interestingly, we saw an increase in #inputs for $\tau \in [0.2, 0.4]$ in SP^2 and $\tau \in [0.4, 0.8]$ in DBPSB, cf. Fig. 4-a/e. For instance, comparing $\tau = 0.2$ and $\tau = 0.4$ in SP^2, `A-PBRJ` read 8% more inputs. DBPSB was less affected: we noticed a marginal increase of 2% for $\tau = 0.4$ vs. $\tau = 0.6$. We explain the increase in both benchmarks with a too "aggressive" pruning – too many partial bindings were pruned wrongfully. That is, many pruned bindings would have led to a larger or even a complete binding. In turn, this led to more inputs being read, in order to produce the desired k results. In fact, $\tau \in [0.6, 0.8]$ was even more aggressive. However, the ratio between pruned bindings and read inputs was high enough to compensate for the extra inputs. Overall, we saw a "sweet spot" at $\tau \approx 0.2$ for $\text{SP}^\mathbf{2}$ and DBPSB. Here, we noted pruning to be fairly accurate, i.e., only few partial bindings were wrongfully pruned. In fact, we observed high precision (recall) values for both benchmarks given $\tau \approx 0.2$: 88% (95%) in $\text{SP}^\mathbf{2}$ (DBPSB) – as discussed below. With regard to computation time for $\text{SP}^\mathbf{2}$ and DBPSB queries, we noticed similar effects as for the #inputs, cf. Fig. 4-b/f. In particular, the "sweet spot" at $\tau \approx 0.2$ is also reflected here.

As expressed by hypothesis H.3, we observed #inputs and time to increase in k for `A-PBRJ` and `PBRJ`. For instance, comparing $k = 1$ and $k = 20$, `A-PBRJ` needed a factor of 1.2 (5.7) more time, given $\text{SP}^\mathbf{2}$ (DBPSB). Similarly, 1.2 (6.8) times more inputs were consumed by `A-PBRJ` for $\text{SP}^\mathbf{2}$ (DBPSB). We explain this behavior with more inputs/join attempts being required to produce a larger result. `PBRJ` leads to a similar performance decrease. For instance given $k = 1$ vs. $k = 20$ in $\text{SP}^\mathbf{2}$, `PBRJ` needed a factor of 1.3 (1.2) more inputs (time). Note, as baseline `JS` simply computed all results, this system was not affected by k.

Furthermore, we can confirm our hypothesis H.2 with regard to system efficiency: *we cloud not find a correlation between system performance and score distributions.* In other words, score distributions (ranking functions) had no impact on `A-PBRJ`'s performance. For instance given DBPSB queries, `A-PBRJ` resulted in the following gains vs. `PBRJ` w.r.t. #inputs (time): 27% (65%) for e distribution, 23% (64%) given u distribution, and 21% (64%) for n distribution.

Last, with regard to parameter τ, we noted `A-PBRJ`'s efficiency to increase with $\tau \in [0, 0.2]$, given $\text{SP}^\mathbf{2}$ and DBPSB. However, as outlined above, too aggressive pruning let to "inverse" effects. An important observation is, however, that our approach was already able to achieve performance gains with a very small $\tau < 0.1$. Here, partial bindings were pruned primarily due to their low binding probability. In fact, `A-PBRJ` could even save time for $\tau = 0$: 26% (60%) with $\text{SP}^\mathbf{2}$ (DBPSB). We inspected queries leading to such saving and saw that many of their partial bindings had a binding probability ≈ 0. We argue that this a strong advantage of `A-PBRJ`: *even for low error thresholds (leading to a minor effectiveness decrease), we could achieve efficiency gains.*

Effectiveness Results. Next, we analyze `A-PBRJ` in terms of its accuracy. Baselines `PBRJ` and `JS` *always compute exact and complete results.* So, we restrict our attention to the `A-PBRJ` system and different score distributions $d \in \{u, n, e\}$.

Fig. 4-c/g (d/h) depicts the macro-precision (score error) for varying score distributions. We observed high precision values of up to 0.98 for both benchmarks, see Fig. 4-c/g. More precisely, we saw best results for a small $\tau < 0.1$ and the exponential distribution. However, differences are only marginal. That is, given $\tau < 0.1$, all distributions led to very similar precision results $\in [0.8, 0.95]$ and $[0.90, 0.98]$ for $\text{SP}^\mathbf{2}$ and DBPSB, respectively. In other words, `A-PBRJ`'s effectiveness is not affected by a particular score distribution. We explain these good approximations with accurate score/binding probabilities.

Moreover, even for large $\tau \in [0.6, 0.8]$ A-PBRJ achieved a high macro-precision in $[0.75, 0.8]$ on DBPSB queries. This is because DBPSB queries featured selective patterns and had only a small result cardinality ≤ 10. Thus, "chances" of pruning a final top-k binding were quite small – even for a large τ. Moreover, A-PBRJ let to a very effective pruning via binding probabilities, as many partial bindings had a binding probability ≈ 0 (due to the high query selectivity). This way, A-PBRJ pruned up to 97% of the total inputs for some DBPSB queries.

In order to quantify "how bad" false positive/negative results are, we employed the score error metric, see Fig. 4-d/h. For both benchmarks, we observed that score error was $\in [0.07, 0.11]$ for a small $\tau < 0.1$. We explain this with our high precision (recall). That is, A-PBRJ led to only few false positive/negative top-k results given $\tau < 0.1$. As expected, score error increased in τ, due to more false positives/negatives top-k results. Overall, however, score error results were very promising: we saw an average score error of 0.03 (0.02), given SP$^{\mathbf{2}}$ (DBPSB).

With regard to parameter k, we observed that k does not impact A-PBRJ's effectiveness. Given SP$^{\mathbf{2}}$, we saw A-PBRJ to be fairly stable in different values for parameter k. For instance, macro-precision was in $[0.8, 0.85]$ as average over all k and $\tau = 0.1$. Also for the DBPSB benchmark, we noted only minor effectiveness fluctuations: macro-precision varied around 7% with regard to different k.

We noticed A-PBRJ's effectiveness to not be influenced by varying score distributions, see Fig. 4-c/g/d/h. Given SP$^{\mathbf{2}}$, we saw a macro-precision of: 0.79 for u distribution, 0.79 for e distribution, and 0.80 for n distribution. Also for the DBPSB benchmark, we observed only minor changes in macro-precision: 0.87 for u distribution, 0.85 for e as well as n distribution.

With regard to the effectiveness of A-PBRJ versus parameter τ, we noticed that metrics over both benchmarks decreased with increasing τ. For instance, macro-precision decreased for $\tau = 0$ versus $\tau = 0.8$ with 27% (23%), given SP$^{\mathbf{2}}$ (DBPSB). Such a behavior can be expected, since chances of pruning "the wrong" bindings increase with higher τ values. *Overall, this confirms H.1: A-PBRJ trades off effectiveness for efficiency, as dictated by threshold τ.*

5 Related Work

There is a large body of work on top-k query processing for relational databases [8]. Most recently, such approaches have been adopted to RDF data and SPARQL queries [9, 22]. These works *aim at exact and complete top-k results.* However, for many applications result accuracy and completeness is not important. Instead, result computation time is the key factor.

To foster an efficient result computation, *approximate top-k* techniques have been proposed [2, 3, 12, 18, 20]. Most notably, [20] used score statistics to predict the highest possible complete score of a partial binding. Partial results are discarded, if they are not likely to contribute to a top-k result. Focusing on distributed top-k queries, [12] employed histograms to predict aggregated score values over a space of data sources. Anytime measures for top-k processing have been introduced by [2, 3]. For this, the authors used offline score information, e.g., histograms, to predict complete binding scores at runtime. In [18], approximate top-k processing under budgetary has been addressed.

Unfortunately, all such approximate top-k approaches heavily rely on *score statistics at offline time*. That is, scores must be known at indexing time for computing statistics, e.g., histograms. However, offline statistics lead to major drawbacks in a Web setting – as outlined in problem (P.1) and (P.2), cf. Sect. 1. In contrast, we propose a lightweight system: *we learn our score distributions in a pay-as-you-go manner at runtime*. In fact, our statistics cause only *minor overhead in terms of space and time*, cf. Thm. 1.

6 Conclusion

In this paper, we introduced an approximate join top-k algorithm, `A-PBRJ`, well-suited for the Web of data (P.1+P.2, Sect. 1). For this, we extended the well-known PBRJ framework [17] with a novel probabilistic component. This component allows to prune partial bindings, which are not likely to contribute to the final top-k result. We evaluated our `A-PBRJ` system by means of two SPARQL benchmarks: we could achieve times savings of up to 65%, while maintaining a high precision/recall.

References

1. R. Agrawal, R. Rantzau, and E. Terzi. Context-sensitive ranking. In *SIGMOD*, 2006.
2. B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *VLDB*, 2007.
3. B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms on exact and fuzzy data sets. *VLDB Journal*, 2009.
4. S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *TODS*, 2006.
5. J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*, 2009.
6. P. D. Hoff. *A First Course in Bayesian Statistical Methods.* Springer, 2009.
7. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 2004.
8. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 2008.
9. S. Magliacane, A. Bozzon, and E. Della Valle. Efficient execution of top-k SPARQL queries. In *ISWC*, 2012.
10. N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *TODS*, 2007.
11. D. Martinenghi and M. Tagliasacchi. Cost-Aware Rank Join with Random and Sorted Access. *TKDE*, 2012.
12. S. Michel, P. Triantafillou, and G. Weikum. KLEE: a framework for distributed top-k query algorithms. In *VLDB*, 2005.
13. M. Morsey et al. DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In *ISWC*, 2011.
14. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.
15. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

16. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.
17. K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008.
18. M. Shmueli-Scheuer, C. Li, Y. Mass, H. Roitman, R. Schenkel, and G. Weikum. Best-Effort Top-k Query Processing Under Budgetary Constraints. In *ICDE*, 2009.
19. A. Telang, C. Li, and S. Chakravarthy. One Size Does Not Fit All: Toward User- and Query-Dependent Ranking for Web Databases. *TKDE*, 2012.
20. M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, 2004.
21. A. Wagner, V. Bicer, and D. T. Tran. Pay-as-you-go Approximate Join Top-k Processing for the Web of Data, 2013. `http://www.aifb.kit.edu/web/Techreport3040`.
22. A. Wagner, T. T. Duc, G. Ladwig, A. Harth, and R. Studer. Top-k linked data query processing. In *ESWC*, 2012.