Title	Evaluating multi-way joins over discounted hitting time
Author(s)	Zhang, W; Cheng, R; Kao, B
Citation	The 30th IEEE International Conference on Data Engineering (ICDE 2014), Chicago, IL., 31 March-4 April 2014. In International Conference on Data Engineering Proceedings, 2014, p. 724-735
Issued Date	2014
URL	http://hdl.handle.net/10722/203650
Rights	International Conference on Data Engineering Proceedings. Copyright © IEEE Computer Society.

Evaluating Multi-Way Joins over Discounted Hitting Time

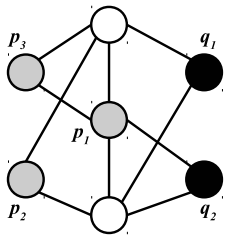
Wangda Zhang, Reynold Cheng, Ben Kao
Department of Computer Science, The University of Hong Kong
{wdzhang2, ckcheng, kao}@cs.hku.hk

***Abstract*—The discounted hitting time (DHT), which is a random-walk similarity measure for graph node pairs, is useful in various applications, including link prediction, collaborative recommendation, and reputation ranking. We examine a novel query, called the *multi-way join* (or n-way join), on DHT scores. Given a graph and n sets of nodes, the n-way join retrieves a set of n-tuples with the k highest scores, according to some aggregation function of DHT values. This query enables analysis and prediction of complex relationship among n sets of nodes. Since an n-way join is expensive to compute, we develop the *Partial Join* algorithm (or PJ). This solution decomposes an n-way join into a number of top-m 2-way joins, and combines their results to construct the answer of the n-way join. Since PJ may necessitate the computation of top-$(m+1)$ 2-way joins, we study an incremental solution, which allows the top-$(m+1)$ 2-way join to be derived quickly from the top-m 2-way join results earlier computed. We further examine fast processing and pruning algorithms for 2-way joins. An extensive evaluation on three real datasets shows that PJ accurately evaluates n-way joins, and is four orders of magnitude faster than basic solutions.**

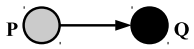
I. Introduction

Inter-related data is often found in social networks, bibliographic databases, and bioinformatics [1]. For instance, Facebook users establish friendship links with each other, and create interest groups. Large social sharing websites (e.g., YouTube and Flickr) generate relationship information among videos and photos, in order to recommend resources to users. As another example, DBLP and CiteSeer provide a rich source of collaboration and citation information. A *graph* naturally models this kind of information. Figure 1(a) illustrates the relationship among people in a social network, where an edge between two persons (nodes) indicates that they are friends.

A lot of research effort has been invested on the retrieval of interesting information from large graphs. Solutions such as link prediction [2], collaborative recommendation [3], and query suggestion [4], have been studied. These methods make use of the *hitting time*, a similarity measure that estimates the expected length of a path between two nodes [5]. The hitting time considers the structure properties of a graph [6], and is resilient to noises [7]. An enhanced version of the hitting time, known as *discounted hitting time* (or *DHT*), has been recently proposed [8], [9]. In this paper, we present a query called the *multi-way join* (or n-way join). Given n groups of nodes, this query returns k lists of n nodes retrieved from these groups, which are ranked the highest according to some aggregation function of DHT. This query can be used in a wide range of applications, as illustrated below.



(a) An example graph.



(b) A query graph.

Rank	(p_i, q_j)
1	(p_1, q_2)
2	(p_2, q_2)
3	(p_1, q_1)

(c) Top-3 results.

Fig. 1: Illustrating a 2-way join.

• **Example 1: 2-way join.** Here, two sets of nodes (called *node sets*) are involved. Figure 1(a) shows two node sets $P = \{p_1, p_2, p_3\}$ (in grey) and $Q = \{q_1, q_2\}$ (in black). A "query graph", shown in Figure 1(b), connects P and Q. If $k = 3$, this query retrieves three pairs of nodes in P and Q with the highest DHT scores (Figure 1(c)). Conceptually, these three pairs of nodes are the closest to each other, in terms of the expected lengths of all the paths between them. [1] This query facilitates link prediction, which is useful in friend suggestion [10] and bioinformatics [11]. Let us suppose that the graph in Figure 1(a) is a social network, where an edge between two nodes represent friendship between two persons. Also assume that P and Q are two interest groups (e.g., soccer and basketball). The 2-way join predicts that p_2 and q_2, who have some common friends, can become friends later.

• **Example 2: 3-way join (triangle).** A bibliographic network can be modeled as a graph, where a node represents an author, and a directed edge from node u to v means that author u cites the paper written by another author v. We can use DHT to capture the closeness of u and v. Suppose that this graph contains experts from three domains: Database (DB), Artificial Intelligence (AI), and System (SYS). A researcher plans to set up a lab that involves cross-disciplinary research work in DB, AI, and SYS. He is looking for experts from each of these three areas, who may be interested in joining this lab. To identify these experts, we can execute a *3-way join* on DB, AI, and SYS, on a "triangular" query graph (Figure 2(a)) [2]. This query returns k lists of authors' names retrieved from DB, AI, and SYS. Since these authors are close to each other in terms of DHT, they can be good candidates for the lab.

[1] The DHT between nodes u and v, or $h(u, v)$, is asymmetric, i.e., $h(u, v)$ is different from $h(v, u)$. Hence, an edge in a query graph is also directional.

[2] For ease of presentation, here we use a single line between two query graph nodes to denote two edges with opposite directions.

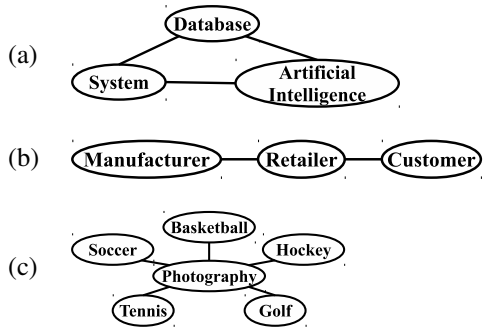


Fig. 2: n-way join: (a) traingle, (b) chain, and (c) star.

• **Example 3: 3-way join (chain).** In e-commerce, a retailer may be looking for new customers and manufacturers. A 3-way join on a social network, containing groups of Manufacturer (M), Retailer (R), and Customer (C), can be useful (Figure 2(b)). This query returns k lists of people obtained from these three groups; for each group, a manufacturer is near to a retailer, who is close to a customer, in terms of DHT scores. An e-commerce system can use this result to suggest potential manufacturers and customers to the retailer.

• **Example 4: 6-way join (star).** Suppose that in a social network, there are six groups of members, who are interested in Soccer (S), Basketball (B), Hockey (H), Golf (G), Tennis (T), and Photography (P). A member of P, Mary, is a professional photographer in sports. She wants to form a "multi-interest" group that contains hobbyists in different sports. To identify these people, we can execute a *6-way join* on node sets S, B, H, G, and T, with P at the centre of the query graph (Figure 2(c)). This query returns k lists of sports lovers who are close to members in P. Mary can use this list to ask her friends in P to suggest the people they know from other sports groups.

Challenges. Consider a top-k n-way join associated with a query graph $\mathcal{Q}$. We name the n node sets involved in this query as $R_1, R_2, \ldots, R_n$. A straightforward way of evaluating this query (called *Nested-Loop Join*, or `NL`) is to first enumerate all the candidate query answers, in the form of n-tuples, where each attribute of the n-tuple is some node r_i selected from R_i. We then compute the "score" of each candidate answer $\mathcal{A}$. This is done by: 1) computing the DHT score of every pair of nodes in $\mathcal{A}$ whose corresponding node sets have an edge in $\mathcal{Q}$; 2) evaluating an aggregate function f of DHT values for these query edges. Finally, `NL` returns the candidate answers with the k highest f values. To illustrate, consider a top-1 3-way join with query graph $\mathcal{Q}$ shown in Figure 2(b). Suppose that the three node sets involved (i.e., M, R, and C) contain nodes "Tom", "Gary", and "Alice" respectively. Thus, a candidate answer $\mathcal{A}$ is (Tom, Gary, Alice). We next compute the f value of $\mathcal{A}$. Let f be the sum of the DHT values for node pairs retrieved from (M, R) and (R, C). The DHT scores of the node pairs relevant to $\mathcal{A}$ are shown in Table I. We see that the f value of $\mathcal{A}$ is 0.5 + 0.3, or 0.8. If this is the highest among the f values of all the candidate answers, then $\mathcal{A}$ is returned by the query.

TABLE I: Computing a 3-way join for Figure 2(b).

(R_i, R_j)	(r_i, r_j)	DHT score
(M, R)	(Tom, Gary)	0.5
(R, C)	(Gary, Alice)	0.3

The `NL` algorithm is extremely expensive. It has to perform numerous DHT computations, each of which is very costly. Specifically, given a node pair (r_i, r_j) that appears in $\mathcal{A}$, its DHT is calculated by performing random walks over all the paths connecting r_i and r_j. In a large graph, many long paths may exist between r_i and r_j; this makes the DHT calculation of (r_i, r_j) extremely inefficient. A more efficient algorithm for executing a n-way join is thus desirable.

Efficient n-way join. Notice that each edge in $\mathcal{Q}$ can be viewed as a 2-way join between two node sets specified in $\mathcal{Q}$. Based on this fact, we develop a fast n-way join algorithm, called the *Partial Join*, or `PJ`. This solution evaluates a top-m 2-way join query for every edge in $\mathcal{Q}$, where m is a tunable parameter. It then uses these results to construct the top-k answers of the n-way join. Since the efficiency of `PJ` depends on the performance of top-m 2-way join queries, we study efficient algorithms for them. We first develop a general formula of DHT, which can be customized to the two common variants of DHT proposed in [8], [9]. Based on this formula, we examine an adaptation of existing 2-way join algorithms for DHT. We further propose a class of solutions, called *backward processing* algorithms, for joining node sets R_i and R_j. These solutions simulate "backward random walks" from every node in $q \in R_j$ to R_i, in order to derive DHT scores between every node in R_i and q. We study two ways of bounding a DHT score and use them to prune nodes from q. These new techniques allow the performance of `PJ` to attain up to four orders of magnitude of improvement over `NL` experimentally. Moreover, `PJ` can be used on the two existing variants of DHT.

The main problem of `PJ` is that the top-m 2-way join results produced from each query graph edge may not be sufficient for devising the answer of the n-way join. If this happens, we have to compute the top-$(m+1)$ 2-way join for some edges of $\mathcal{Q}$ from scratch. Since a top-$(m+1)$ query can be expensive to evaluate, the efficiency of `PJ` can be affected. We develop a variant of `PJ`, called the *Incremental Partial Join*, or `PJ-i`. This solutions allows a top-$(m+1)$ 2-way join query for R_i and R_j to be efficiently derived from the result of the top-m join between these two node sets. In our experiments, `PJ-i` is up to 50 times faster than `PJ`.

Our contributions. We propose a n-way join operator over DHT scores. To evaluate this query efficiently, we develop the `PJ` and `PJ-i` algorithms. They can handle two recently-proposed DHT scores in [8], [9]. They also support any monotonic aggregate DHT function f (e.g., sum, or minimum of DHT scores on query graph edges). We have performed a detailed evaluation of these algorithms on a bibliographic dataset (*DBLP*), a social-sharing website (*YouTube*), and a protein-interaction dataset (*Yeast*). Our solutions are highly effective and efficient on these databases.

The rest of our paper is organized as follows. Section II discusses the related work. In Section III, we present the formal definition of the multi-way join query. We present the solution framework of PJ in Section IV. In Section V, we describe two existing variants of DHT, and examine how existing 2-way join algorithms that can be applied to them. Section VI presents the backward processing algorithms, as well as PJ−i. Our experimental results are reported in Section VII. We conclude in Section VIII.

II. Related Work

We now discuss previous studies about similarity joins.

2-way join. The problem of joining two sets of objects, based on a similarity or distance measure, is an important topic in the database literature. In high dimensional databases, [12] studied efficient 2-way join algorithms based on Euclidean distances. In text databases, [13] and [14] examined set and string similarity joins for data cleaning applications by using Hamming and edit distances. In spatial databases, fast 2-way join algorithms based on Euclidean distances were examined [15], [16]. Recently, the issues of supporting a similarity join in a database system have been addressed [17].

In graph databases, [16] and [18] investigated 2-way joins for road networks and graph pattern matching, respectively. They use the shortest-path distance. In [19], we examined an efficient 2-way join algorithm, IDJ, which can be applied to common random-walk measures (e.g., Personalized PageRank [20], SimRank [21], and DHT [8], [9]). However, it does not perform a detailed study on DHT. Here we address DHT in more detail. We examine how to generalize two recently proposed DHT measures [8], [9] in a common form. We then adapt IDJ to run a 2-way join for DHT. We propose an algorithm, called B−IDJ, and study two pruning techniques for it. We show that B−IDJ is theoretically and experimentally faster than IDJ. We also use B−IDJ to support a n-way join query, which has not been studied before.

***n*-way join**.This query generally involves joining n sets of entities (with $n \geq 3$), according to a query graph $\mathcal{Q}$. The authors in [22], [23] examined fast n-way join algorithms for relational databases. The efficiency of of n-way join has also been addressed in interval databases (e.g., [24], [25]) and spatial databases (e.g., [16], [26], [27]).

However, little work has been done on n-way join for graph databases. In [18], the authors study this query based on the shortest path distance. Given node sets $\{R_1, R_2, \ldots, R_n\}$, and nodes $r_i \in R_i$, the query returns all the n-tuples $(r_1, r_2, \ldots, r_n)$, such that the shortest path distance between (r_i, r_j) does not exceed a global threshold δ, and there exists an edge between R_i and R_j in $\mathcal{Q}$. As discussed in [8], [9], [20], [21], the shortest path measure is often inferior to random walk metrics in terms of accuracy in prediction and recommendation tasks. Also, setting an appropriate value of δ can be difficult. However, the query in this paper returns k n-tuples with the highest value of f (i.e., the aggregate function of DHT on $\mathcal{Q}$). It may be easier for a user to specify the value of k rather than δ. Our solution also allows f to be any monotonic aggregate function of DHT.

III. Preliminaries

We now describe the data and query models in Section III-A. We then discuss two basic solutions for answering a multi-way join, in Section III-B.

A. Data and Query Models

Let G be a directed and weighted graph. Also, let V_G and E_G be the sets of nodes and edges of G respectively. Given two nodes u and v in G, w_{uv} denotes the weight of edge (u, v). We use O_u (I_u) to denote the set of out-neighbor (in-neighbor) nodes of u. We assume that G is stored in an adjacency list, so that the neighbors of a node can be found quickly.

Let $h(u, v)$ be the DHT score of nodes u and v. As discussed before, $h(u, v)$ measures the closeness between u and v. Notice that a DHT score is asymmetric, i.e., $h(u, v) \neq h(v, u)$. We will detail the properties of $h(u, v)$ in Section V. A *node set* of G is a subset of V_G. Let $R_1, R_2, \ldots, R_n$ be n node sets of G, with $n \geq 2$. They constitute a *query graph*.

Definition 1. A Query Graph $\mathcal{Q}$ is an unweighted and directed graph, whose sets of nodes and edges are respectively denoted by $V_\mathcal{Q}$ and $E_\mathcal{Q}$. Specifically, $V_\mathcal{Q} = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n\}$, where $\mathcal{R}_i \in V_\mathcal{Q}$ corresponds to a node set $R_i \subseteq V_G$.

Definition 2. The Aggregate Score f of a query graph $\mathcal{Q}$ is a monotonic function of $|E_\mathcal{Q}|$ real-valued inputs.

Definition 3. Given a query graph $\mathcal{Q}$, a Candidate Answer $\mathcal{A}$ is an n-tuple within domain $R_1 \times R_2 \times \ldots \times R_n$.

To compute the aggregate score of $\mathcal{A}$ (denoted by $\mathcal{A}.f$), we can evaluate the DHT score $h(r_i, r_j)$ for all pairs of nodes that appear in $\mathcal{A}$, such that (1) $r_i \in R_i$ and $r_j \in R_j$, and (2) an edge is incident on nodes $\mathcal{R}_i$ and $\mathcal{R}_j$. Then, f is applied to these $|E_\mathcal{Q}|$ DHT scores. Notice that if $\mathcal{Q}$ has a single edge only, $\mathcal{A}.f$ is the DHT of the node pair specified in $\mathcal{A}$.

An example f is the SUM function, which sums up the DHT scores of the $|E_\mathcal{Q}|$ node pairs. Given a candidate answer $\mathcal{A}$, $\mathcal{A}.f$ computes the "overall closeness" of the $|E_\mathcal{Q}|$ node pairs appearing in $\mathcal{A}$. Another example f is MIN, which returns the minimum DHT value among all the $|E_\mathcal{Q}|$ scores. The value of $\mathcal{A}.f$ is the lowest similarity score among the node pairs of $\mathcal{A}$.

We use the f function to rank the candidate answers in a multi-way join, as defined below.

Definition 4. Multi-way Join (or n-way join). Given a graph G, a query graph $\mathcal{Q}$, n node sets ($\{R_1, R_2, \ldots, R_n\}$), an aggregate function f, and a natural number k, the ***n*-way join** returns a sorted list of k candidate answers of $\mathcal{Q}$, i.e., $\{A_1, A_2, \ldots, A_k\}$, such that (1) the $A_i.f$ values are the highest among all the candidate answers of $\mathcal{Q}$; and (2) $A_i.f \geq A_j.f$ for every $i, j \in [1, k]$ and $i < j$.

Essentially, an n-way join returns a sorted list of candidate answers with the highest f values.

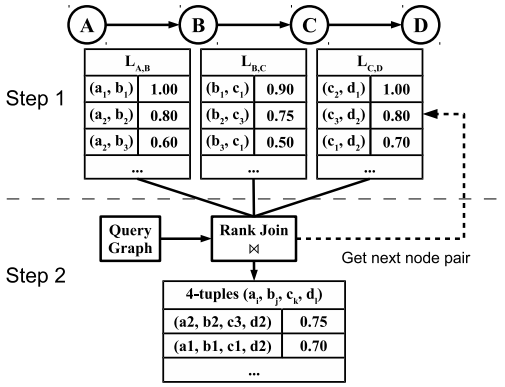


Fig. 3: Solution Framework of PJ.

B. Basic Solutions

We now discuss two simple n-way join algorithms.

(1) Nested Loop (NL). As discussed in Section I, NL enumerates all the candidate answers. This is done by using n nested loops on the n node sets (i.e., $R_1, R_2, \ldots, R_n$). For each candidate answer, we compute its aggregate score f. These answers are then sorted according to their f values, and those with the k highest scores are returned. This solution is slow, because (1) it has to generate a huge number ($\Pi_{i=1}^{n}|R_i|$) of candidate answers; and (2) for each candidate answer, a DHT computation is performed for every edge in $\mathcal{Q}$. As we will see in Section V, computing a DHT score is expensive.

(2) All Pairs (AP). This algorithm decomposes an n-way join into $|E_\mathcal{Q}|$ joins of node set pairs specified in $\mathcal{Q}$. These results are combined to construct the final solution. Let us illustrate AP by Figure 2(b). This query can be processed by first generating the DHT scores for all pairs of nodes in (M, R) and (R, C). The $|M| \times |R|$ node pairs from (M, R) are then joined with the $|R| \times |C|$ node pairs from (R, C). The aggregate scores of the 3-tuples from (M, R, C) are computed, and the k 3-tuples with the highest scores are returned. The number of DHT computations required by this solution is: $\sum_{(\mathcal{R}_i,\mathcal{R}_j)\in E_\mathcal{Q}}(|R_i| \times |R_j|)$, which can be less than that of NL. To further improve the performance of AP, a *rank join* algorithm ([28]–[30]) for finding the top k candidate answers can be employed. In our experiments, we use the PBRJ, which is a common rank join algorithm [28]. The PBRJ is also used in our better n-way join algorithm, PJ, as discussed next.

IV. The Partial Join Algorithm

Recall that AP decomposes an n-way join to $|E_\mathcal{Q}|$ 2-way joins. Each of these operations is expensive to evaluate, since the DHT scores of all the pairs in the two node sets involved have to be computed. However, not many of these results are useful. In our experiments (Section VII), under a wide range of values of k, less than 1% of the 2-way join results are used to construct the n-way join answers. Based on this observation, we develop the **Partial Join** algorithm (or PJ), which can generate much fewer 2-way join results. We present the framework of this solution here, and explain its details in Sections V and VI.

Algorithm 1: n-way Join (PJ)

Input: graph G; query graph $\mathcal{Q}$; node sets $R_1, \ldots, R_n$; aggregate function f; integers k, m
Output: top-k n-tuples with the highest f values
Data: list $\mathcal{L}_{R_i,R_j}$; candidate buffer $\mathcal{C}_{R_i,R_j}$; output $\mathcal{O}$

1 $\mathcal{O} \leftarrow \emptyset$
2 **foreach** *edge* $(\mathcal{R}_i, \mathcal{R}_j)$ *in* $\mathcal{Q}$ **do**
3 $\mathcal{C}_{R_i,R_j} \leftarrow \emptyset$
4 $\mathcal{L}_{R_i,R_j} \leftarrow$ twoWayJoin(G, R_i, R_j, m)
5 $\tau \leftarrow \infty$
6 **while** $|\mathcal{O}| < k$ *or* $\min_{\mathcal{A}\in\mathcal{O}} \mathcal{A}.f < \tau$ **do**
7 $\mathcal{L}_{R_i,R_j} \leftarrow$ roundRobin$(\mathcal{L}_{R_i,R_j}|(\mathcal{R}_i, \mathcal{R}_j) \in E_\mathcal{Q})$
8 $(r_i, r_j), h(r_i, r_j) \leftarrow$ next entry in $\mathcal{L}_{R_i,R_j}$
9 **if** $(r_i, r_j) = null$ **then**
10 $\mathcal{L}_{R_i,R_j}$.append(getNextNodePair(G, R_i, R_j))
11 add $((r_i, r_j), h(r_i, r_j))$ to $\mathcal{C}_{R_i,R_j}$
12 $\mathcal{I} \leftarrow$ getCandidate$\big((r_i, r_j), h(r_i, r_j), \{C_{R'_i,R'_j}|(\mathcal{R}'_i, \mathcal{R}'_j) \neq (\mathcal{R}_i, \mathcal{R}_j)\}\big)$
13 insert every $\mathcal{A} \in \mathcal{I}$ to $\mathcal{O}$, retaining only the k candidate answers with the highest f values
14 $\tau \leftarrow$ cornerBound$(h(r_i, r_j))$
15 return $\mathcal{O}$

Let us first illustrate PJ with an example. Figure 3 shows a query graph that contains four nodes (A, B, C and D). To evaluate this 4-way join, PJ first performs a top-m 2-way join for the three node set pairs: (A, B), (B, C), and (C, D). Notice that m is a parameter of PJ, and $m = 3$ here. These results, sorted in descending order of DHT scores, are stored in three lists ($\mathcal{L}_{A,B}$, $\mathcal{L}_{B,C}$, and $\mathcal{L}_{C,D}$). Next, PJ performs a rank join on these lists, and produces top-k 4-way join results, in the form of 4-tuples, and their aggregate scores. If the information in the $\mathcal{L}$ lists is not enough to generate all the n-way join results, PJ fetches more answers from the 2-way joins involved. For example, PJ can request the fourth node pair in $\mathcal{L}_{C,D}$, by issuing a top-4 2-way join on C and D.

Algorithm 1 shows the details of PJ. We use $\mathcal{O}$ to denote a priority queue of size k, which stores the candidate answers with the k highest f scores. Initially, $\mathcal{O}$ is empty (Step 1). Then, for each edge $(\mathcal{R}_i, \mathcal{R}_j)$ that appears in $\mathcal{Q}$, in Step 3 we initialize its "candidate buffer" $\mathcal{C}_{R_i,R_j}$, which stores node pairs retrieved from R_i and R_j. (We will discuss more about this buffer later.) We also issue a top-m 2-way join for R_i and R_j (Step 4). This is done by calling twoWayJoin, which performs a top-m join on node sets R_i and R_j. The result, containing m node pairs with DHT scores, is stored in $\mathcal{L}_{R_i,R_j}$. The value of m ranges from 0 to $\min_{(\mathcal{R}_i,\mathcal{R}_j)\in E_\mathcal{Q}}(|R_i| \times |R_j|)$. If $m = 0$, twoWayJoin just returns an empty list. Section V studies this function in more detail.

Steps 5–14 employs the rank join on the lists obtained in Step 4, for producing n-way join answers. We choose the commonly-used *Pull/Bound Rank Join* (PBRJ) [28] as the framework of the rank join. The PBRJ has a *stopping*

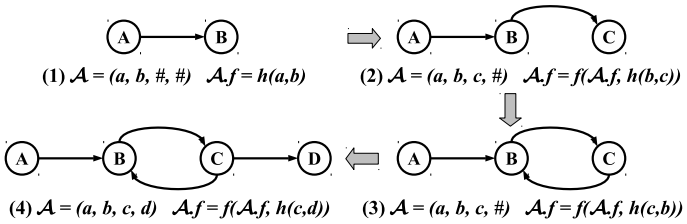


Fig. 4: Generating a candidate answer.

threshold τ, which denotes the lower bound of the k-th highest f scores of the candidate tuples. Initially, τ is set to ∞ (Step 5). Steps 7–14 are repeated as long as one of the conditions in Step 6 are satisfied: (1) $|\mathcal{O}|$ is k or less; or (2) the minimum f value of the candidate answers in $\mathcal{O}$ is less than τ. In Step 7, we select $\mathcal{L}_{R_i,R_j}$ among all the 2-way join result lists in a round-robin fashion, according to the well-known `HRJN` scheme [29]. We then retrieve the next unread entry (r_i, r_j) with its DHT score from $\mathcal{L}_{R_i,R_j}$ (Step 8). If this is not found (since the end of the top-m list has been reached), we execute `getNextNodePair` to find the next node pair of the 2-way join of R_i and R_j, and append it to $\mathcal{L}_{R_i,R_j}$ (Steps 9–10). This procedure can be implemented by simply running a top-$(m+1)$ join (i.e., `twoWayJoin`$(G, R_i, R_j, m+1)$), and obtain the $(m+1)$-th node pair. We develop a faster solution that uses the results obtained from the top-m join query. Section VI gives details on this.

Next, we construct candidate answers from the retrieved node pair (r_i, r_j). In Step 11, we insert (r_i, r_j) and its DHT score to $\mathcal{C}_{R_i,R_j}$. This is a 2D array of dimensions $|R_i| \times |R_j|$, where $\mathcal{C}_{R_i,R_j}[r_i][r_j]$ stores the value of $h(r_i, r_j)$. In Step 12, `getCandidate` is executed. This algorithm uses (r_i, r_j), and the node pairs stored in other candidate buffers, to generate a set $\mathcal{I}$ of candidate answers. We will examine this function shortly. In Step 14, we use $h(r_i, r_j)$ to update the value of τ, based on the *corner-bound* strategy in `HRJN` [29]. After the loop of Steps 5-14 ends, $\mathcal{O}$ contains the candidate answers with the k highest values of f, and is returned in Step 15.

Figure 4 illustrates `getCandidate` for a query graph with four node sets. Initially, a node pair (a, b) is retrieved from the list $\mathcal{L}_{A,B}$. A "partial answer" $\mathcal{A}$ is created for (a, b), namely $(a, b, \#, \#)$, where $\#$ stands for unknown elements. Following the edges in $\mathcal{Q}$, we found the edge (B, C). We then obtain the node pair (b, c) by searching $\mathcal{C}_{B,C}$ with b. Then, $\mathcal{A}$ becomes $(a, b, c, \#)$. The f score of $\mathcal{A}$ is updated to $f(h(a, b), h(b, c))$. We repeat this process by visiting edge (C, B) in $\mathcal{Q}$, which contains the edge (c, b). Finally, we process (C, D), and $\mathcal{A}$ becomes (a, b, c, d), which is then inserted to $\mathcal{I}$. Notice that two or more edges may be found from a candidate table. For instance, based on node b, we obtain two node pairs (b, c_1) and (b, c_2) from $\mathcal{C}_{B,C}$. If this happens, we create two partial answers, namely $(a, b, c_1, \#)$ and $(a, b, c_2, \#)$. Finally, `getCandidate` returns all the partial answers in $\mathcal{I}$ that do not carry any unknown elements.

We next discuss the details of Algorithm 1. Section V describes existing algorithms for supporting `twoWayJoin` in Step 4. The same procedure can be executed faster by using *backward processing* techniques (Section VI). In Section VI-D, we present an efficient solution for `getNextNodePair`, which is invoked in Step 10.

V. DHT and 2-Way Joins

Recall that in our n-way join algorithm (Algorithm 1), `twoWayJoin` is frequently invoked (e.g., Step 2 and Step 7). Hence, it is crucial for this function to be efficient. However, this objective is complicated by the facts that (1) computing the DHT score of a node pair is expensive; and (2) there are two variants of DHT measures. To enable efficient 2-way join, we study how DHT variants can be generalized to a common form, in Section V-A. We then examine how existing techniques can be used to evaluate 2-way join over this common form, in Section V-B.

A. A General Form of DHT

Let us first explain the notion of *random walk*. Given a graph G, a random walker starts from a node in V_G, and performs a move by following the outgoing edges in E_G. In every step, the walker moves from u (the node where he is currently located), to an out-neighbor v of u with a *transition probability* p_{uv}. For a weighted graph, $p_{uv} = w_{uv}/(\sum_{v' \in O_u} w_{uv'})$.

The *Hitting Time* (or *HT*) is a random-walk-based measure that considers the ensemble of paths between nodes u and v in graph G, and computes the expected path length between u and v ([2]–[4], [6], [7], [31], [32]). Intuitively, HT measures the "time" for a random walker to reach v from u. The *Discounted Hitting Time*, or *DHT*, is an enhanced version of HT, which places more emphasis on neighboring nodes, and less so on long-range relationships. As discussed in [8], [9], DHT is a better measure than HT in graph applications. Recently, two DHT measures have been proposed:

- DHT_e [8]: a random walker, who starts at u, stops when he reaches v. The formula of DHT_e is given by:

$$DHT_e(u, v) = \sum_{i=1}^{\infty} e^{-(i-1)} \mathcal{P}_i(u, v) \quad (1)$$

where $\mathcal{P}_i(u, v)$ is the *hitting probability* that a random walker starting from node u first hits node v after i steps. Notice that when i increases, $e^{-(i-1)}$ decreases. Hence, DHT_e weighs higher for shorter paths between u and v.

- DHT_λ [9]: the formula of DHT_λ is given by:

$$DHT_\lambda(u, v) = -1 + \lambda \sum_{w \in O_u} p_{uw} \cdot DHT_\lambda(w, v) \quad (2)$$

where $DHT_\lambda(v, v) = 0$, and $\lambda \in (0, 1)$ is called the *decay factor*. Again, a random walker starting at u stops when he reaches v. In each step of the random walk, he may stop with a probability $\lambda \in (0, 1)$. Here, λ is also used to reduce the effect of long paths on the DHT score. [3]

General form. We found that the two versions of DHT above can be generalized as follows:

[3]The original definition of DHT_λ in [9] is a *distance* measure, which increases with the average path length between u and v. We negate its score, which becomes Equation 2, to measure the similarity between u and v.

TABLE II: Generalization of DHT variants.

DHT (Equation 3)	α	β	λ
DHT_e (Paper [8])	e	0	$1/e$
DHT_λ (Paper [9])	$1/(1-\lambda)$	$-1/(1-\lambda)$	λ

Definition 5. Given two distinct nodes $u, v \in V_G$, the GENERAL FORM OF DHT, denoted by $h(u, v)$, is:

$$h(u,v) = \alpha \sum_{i=1}^{\infty} \lambda^i \mathcal{P}_i(u,v) + \beta \quad (3)$$

where α and β are real-valued coefficients, with $\alpha \neq 0$. We call $\lambda \in (0, 1)$ the *decay factor*. Notice that $h(u, v)$ may be different from $h(v, u)$, since $\mathcal{P}_i(u, v)$ may be different from $\mathcal{P}_i(v, u)$ on a weighted and directed graph.

Table II shows how the three parameters of Equation (3) (i.e., α, β, and λ) are customized for the DHT_λ and DHT_e measures. The parameter values for DHT_e can be easily derived by rewriting Equation (1) into the form of Equation (3). For DHT_λ, we first obtain the recursive formula of Equation (3), by expressing $h(u, v)$ in terms of $h(w, v)$, where $w \in O_u$. We then compare this with Equation (2) and get the parameter values of DHT_λ. Please refer to our technical report [33] for the derivation details. In the sequel, the term "DHT" refers to the general form in Definition 5.

Practical DHT score evaluation. As we can see in Equation (3), evaluating DHT (i.e., $h(u, v)$) requires summing up an infinite number of terms. Let us consider a more practical way of computing $h(u, v)$:

$$h_d(u,v) = \alpha \sum_{i=1}^{d} \lambda^i \mathcal{P}_i(u,v) + \beta \quad (4)$$

where d is the number of steps in a random walk. Notice that (1) $h_d(u, v)$ monotonically increases with d; and (2) $h(u, v) = \lim_{d\to\infty} h_d(u, v)$. Hence, if d is sufficiently large, $h_d(u, v)$ can be regarded as $h(u, v)$. We thus use $h_d(u, v)$ to represent a DHT score. In fact, this way of approximating other random-walk measures with a limited number of steps has also been considered in previous works (e.g., [8] and [19]).

We next describe a lemma in [19], which approximates a random-walk measure having the same form of $h(u, v)$.

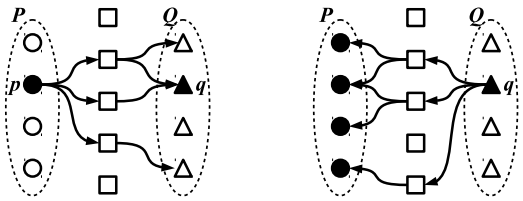
Lemma 1. *Given that $\varepsilon \in \Re$, if $|h(u,v) - h_d(u,v)| \leq \varepsilon$, then $d \geq \log_\lambda \frac{\varepsilon(1-\lambda)}{\alpha\lambda}$.*

This result allows us to obtain the minimum value of d such that the error between $h(u, v)$ and $h_d(u, v)$ is bounded by ε.

2-way joins. We can now study efficient solutions for DHT. Let P and Q be two node sets in V_G. Given the values of α, β, and λ, our goal is to retrieve k pairs of nodes from P and Q with the highest $h_d(p, q)$ values. These $h_d(p, q)$ values are also returned. Next, we study solutions collectively known as *forward processing*. In Section VI, we present another class of solutions known as *backward processing algorithms*.

B. Forward Processing

Given a node pair (p, q) (where $p \in P$ and $q \in Q$), a *forward processing algorithm* computes $h_d(p, q)$ by performing random walks along edges from p to q. Figure 5(a) illustrates this process. Here we discuss a simple solution, called F−BJ, and a faster one, called F−IDJ.



(a) Forward (b) Backward

Fig. 5: Forward and backward processing.

• The **Forward Basic Join** (or F−BJ) algorithm performs a 2-way join by computing $h_d(p, q)$ for every node pair (p, q), and then returns the pairs with the k highest values. To evaluate $h_d(p, q)$, we adopt an approach similar to that of [8]. In particular, a vector r with $|V_G|$ entries is used to store the probabilities of nodes in V_G that have been visited by the walker from p. Initially, $r[p] = 1.0$, and the rest of entries in r are set to zero. At iteration i (where $i = 1, 2, \ldots, d$), we refresh r by performing one step of random walk from every node u, where $u \in V_G - \{q\}$, to node $v \in O_u$. To do this, let r' be another vector of size $|V_G|$. We first assign $r'[v]$ by the value $\sum_{u\neq q \wedge (u,v)\in E_G} (r[u] \cdot p_{uv})$. Then, $r'[q]$ is the same as $\mathcal{P}_i(p, q)$, and we compute $h_i(p, q)$ by using Equation (4). We next overwrite r by r'. This process continues until $i = d$. The complexity of F−BJ, which is $O(|P||Q| \cdot d|E_G|)$, is high, since the DHT score of every node pair from (P, Q) needs to be evaluated. We next describe a better method that avoids scanning all the (p, q) pairs.

• The F−IDJ algorithm. In [19], the **Iterative Deepening Join** framework (or IDJ) was proposed to handle any kind of random-walk measures having the form of Equation (4). The F−IDJ algorithm was an adaptation of IDJ for evaluating a 2-way join over DHT scores. It uses the fact that in Equation (4), λ^i decreases exponentially with i. Thus, the influence of $\mathcal{P}_i(u, v)$ on $h_d(u, v)$ decreases sharply with i. If i is small, $h_i(p, q)$ can be used to approximate $h_d(p, q)$ with a high accuracy; and F−IDJ uses $h_i(p, q)$ to prune nodes from P. Another fact is that $h_i(p, q)$ can be quickly computed when i is small, since fewer random walks are involved. Hence, F−IDJ computes values of $h_i(p, q)$ for small values of i first.

In detail, F−IDJ contains $\lceil \log d \rceil$ iterations. In the j-th iteration (where $j = 1, 2, \ldots, \lceil \log d \rceil - 1$), random walks of up to $l = 2^{j-1}$ steps are carried out for each node $p \in P$. For every node pair $(p, q) \in (P, Q)$, F−IDJ use the l-step-random walk information to derive (1) the lower bound of $h(p, q)$, denoted by $h_d^-(p, q)$; and (2) the upper bound of $h(p, q)$ between p and any $q \in Q$, denoted by $h_d^+(p, Q)$. A set $\mathcal{B}$ of k node pairs with the highest $h_d^-(p, q)$ scores are maintained. Let T_k be the k-th largest $h_d^-(p, q)$ score of the pairs in $\mathcal{B}$. A node $p_0 \in P$ is pruned, if $h_d^+(p_0, Q) < T_k$, since none of the $h_d(p_0, q)$ values are higher than the scores of the node pairs in $\mathcal{B}$. This process is repeated for $\lceil \log d \rceil - 1$ iterations. In the final iteration, F−IDJ evaluates the actual DHT scores (i.e., $h_d(p, q)$) for node pairs that cannot be pruned, based on the

solution discussed in F−BJ. The worst-case running time of F−IDJ is still $O(|P||Q| \cdot d|E_G|)$.

Different from F−BJ, F−IDJ does not perform d steps of random walks for every node pair in (P, Q). As the number of iterations increases, the number of nodes in P is reduced. As pointed out in [19], many nodes in P are pruned in early iterations at a low computation cost, and very few nodes in P need to have $h_d(p, q)$ computed. This makes F−IDJ run much faster than F−BJ in our experiments. A detailed discussion of F−IDJ and its complexity can be found in [33].

The main problem of both F−BJ and F−IDJ is that for each pair (p, q), they have to perform random walks from p to q, which significantly affect their running times. We next examine another type of solution that alleviates this problem.

VI. Backward Processing for 2-Way Joins

We now study another class of 2-way join algorithms known as *backward processing*. Figure 5(b) illustrates its main intuition: given a node $q \in Q$, we perform *backward random walks* over G, in order to compute all the $h_d(p, q)$ values for every $p \in P$. As we shall see, solutions that are developed based on this idea are faster than their forward processing counterparts. We first discuss a basic solution in Section VI-A. Then, Section VI-B presents another algorithm that enables node pruning, and Section VI-C discusses two node pruning techniques. Based on these solutions, Section VI-D develops an "incremental" join algorithm, which facilitates the execution of getNextNodePair in Algorithm 1.

A. The Backward Basic Join Algorithm (B−BJ)

Similar to F−BJ, the **Backward Basic Join** (or B−BJ) computes $h_d(p, q)$ for every node (p, q) in (P, Q). However, these values are obtained in a different manner. Specifically, for every node $q \in Q$, B−BJ invokes the backWalk procedure to obtain all the values of $h_d(p, q)$ for every $p \in P$. After all the $h_d(p, q)$ scores are obtained, the node pairs with the k highest scores are returned.

The backWalk algorithm takes the node set P, node q, and step d, as its input. It uses a size-$|V_G|$ vector, $backProb$, to store the hitting probabilities from every node in V_G to q. Initially, $backProb[q] = 1$ and $backProb[v] = 0$ for any $v \in V_G - \{q\}$. A "backward random walk" is then initiated from q. Particularly, in each iteration i ($i = 1, 2, \ldots, d$), we obtain the new hitting probability $\mathcal{P}_i(u, q)$ for every $u \in V_G$:

$$\mathcal{P}_i(u,q) = \begin{cases} \sum_{(u,v)\in E_G \wedge v \neq q} p_{uv} \cdot backProb[v] & i > 1 \\ \sum_{(u,v)\in E_G} p_{uv} \cdot backProb[v] & i = 1 \end{cases} \quad (5)$$

The values of $\mathcal{P}_i(u, q)$ are then written back to $backProb[u]$. By repeating Equation (5) for d times, we obtain the values of $\mathcal{P}_i(p, q)$ ($i = 1, 2, \ldots, d$) for every node $p \in P$. Their DHT scores can then be computed by using Equation (4). Figure 5(b) illustrates this.

The backWalk algorithm produces $h_d(p, q)$ for every $p \in P$ with a complexity of $O(d|E_G|)$. Consequently, the running time of B−BJ is $O(|Q| \cdot d|E_G|)$, which is $O(|P|)$ times faster than F−BJ. The space overhead of B−BJ is $O(|V_G|)$. The detail of backWalk and its complexity is in [33].

B. The Backward IDJ Solution Framework

The B−BJ solution can still be expensive, since it has to perform a d-step backward random walk for every node q in Q. On the other hand, the **Backward IDJ** algorithm (or B−IDJ) allows some nodes in Q to be pruned during the random walk. Similar to B−BJ, B−IDJ initiates a backward random walk at q. Let $h_d^-(p, q)$ be the lower bound of $h_d(p, q)$, and $h_d^+(P, q)$ be the upper bound of $h_d(p, q)$ from any node $p \in P$ to q. At the l-th step, B−IDJ collects the following information:

- For each pair $(p, q) \in (P, Q)$, compute $h_l(p, q)$, which cannot be larger than $h_d(p, q)$ due to Equation 4. We thus let $h_d^-(p, q) = h_l(p, q)$.
- For each node $q \in Q$, evaluate

$$h_d^+(P, q) = \max_{p \in P}\{h_l(p, q)\} + \mathcal{U}_l^+ \quad (6)$$

Here, $\mathcal{U}_l^+$ is called the *upper bound function*. In this paper, we study two kinds of upper bound function, leading to two variants of B−IDJ. Their details will be addressed in Section VI-C. Now, let T_k be the k-th largest $h_d^-(p, q)$ value. Then, a node q can be *pruned* if the following is true:

- For any $q \in Q$, $h_d^+(P, q) < T_k$

Algorithm 2: Backward IDJ (B−IDJ)

Input: graph G; k; node sets P, Q
Output: top-k node pairs in $\mathcal{B}$

1 $j \leftarrow 1$
2 **while** $2^{j-1} < d$ **do**
3 $l \leftarrow 2^{j-1}$, $\mathcal{B} \leftarrow \emptyset$
4 **foreach** $q \in Q$ **do**
5 $score \leftarrow$ backWalk(G, P, q, l)
6 **foreach** $p \in P$ **do**
7 **if** $score[p] > \beta$ **then**
8 insert $((p, q), score[p])$ to $\mathcal{B}$
9 $pMaxScore \leftarrow \max(pMaxScore, score[p])$
10 $qUpper[q] \leftarrow pMaxScore + \mathcal{U}_l^+$
11 $T_k \leftarrow \mathcal{B}.getMin()$
12 **foreach** $q \in Q$ **do**
13 **if** $qUpper[q] < T_k$ **then**
14 remove q from Q
15 $j \leftarrow j + 1$
16 $score \leftarrow$ backWalk(G, P, q, d) for $q \in Q$
17 update $\mathcal{B}$ with $score$
18 return $\mathcal{B}$

The above idea is implemented in Algorithm 2. Here, we use a priority queue $\mathcal{B}$ that stores up to k node pairs with the highest scores. The algorithm iterates itself $\lfloor \log d \rfloor$ times (Steps 2-15). In the j-th iteration, a backward random walk of up to $l = 2^{j-1}$ steps is performed. The rationale is similar to that of F−IDJ: we exploit the fact that when l is small, random walks can be quickly computed, and they provide good bounds of DHT scores. In Steps 4-9, we do the following for all nodes

q in Q: Step 5 performs a l-step backward random walk by invoking `backWalk` and obtains $h_l(p,q)$ for every node p in P. These values, which can also be considered as $h_d^-(p,q)$, are stored in array $score$. In Step 8, we insert the node pairs (p,q) and their $h_l(p,q)$ scores to $\mathcal{B}$. We evaluate $h_d^+(P,q)$ in Steps 9 and 10. We compute T_k in Step 11, which is used to prune a node q from Q (Steps 12-14). After finishing all the iterations, we perform a d-step backward random walk for nodes that cannot be pruned (Step 16). Step 17 updates $\mathcal{B}$ with $score$, and Step 18 returns the top-k node pairs.

In B−IDJ, $O\big(\log d\big)$ iterations are needed. Each iteration requires $O\big(|Q| \cdot (l|E_G| + t(\mathcal{U}_l^+)\big)$ time, where $t(\mathcal{U}_l^+)$ denotes the time of computing $\mathcal{U}_l^+$. We next examine two ways of implementing $\mathcal{U}_l^+$.

C. The B−IDJ−X and B−IDJ−Y Algorithms

1. B−IDJ−X (by setting $\mathcal{U}_l^+$ as X_l^+). We first claim that:

Lemma 2. *$h(p,q) \le h_l(p,q) + X_l^+$, where*

$$X_l^+ = \alpha \sum_{i=l+1}^{\infty} \lambda^i = \frac{\alpha\lambda^{l+1}}{1-\lambda} \tag{7}$$

which can be derived from the definitions of $h(p,q)$ (Equation 3) and $h_d(p,q)$ (Equation 4). Since $h_d(p,q) \le h(p,q)$, $h_d(p,q)$ is upper-bounded by $h_l(p,q) + X_l^+$. By setting $\mathcal{U}_l^+$ as X_l^+, we obtain $h_d^+(P,q)$ (Equation 6). We call the variant of B−IDJ where $\mathcal{U}_l^+ = X_l^+$ as B−IDJ−X. The time for computing X_l^+ is $O(l)$, and the complexity of B−IDJ−X is $O(|Q| \cdot d|E_G|)$. Its space overhead is $O(|V_G|)$.

2. B−IDJ−Y (by setting $\mathcal{U}_l^+$ as Y_l^+). First, we let V_l be $\alpha \sum_{i=l+1}^{d} \lambda^i \mathcal{P}_i(p,q)$. We can rewrite $h_d(p,q)$ (Equation 4) as:

$$h_d(p,q) = h_l(p,q) + V_l \tag{8}$$

Let Y_l^+ be an upper bound of V_l. Since $h_d(p,q) \le h_l(p,q) + Y_l^+$, we can assign $\mathcal{U}_l^+$ to be Y_l^+. We name the version of B−IDJ where $\mathcal{U}_l^+ = Y_l^+$ as B−IDJ−Y.

To maximize the pruning effect, Y_l^+ should be small. This requires finding a good upper bound of $\mathcal{P}_i(p,q)$, which appears in V_l. We next study how this can be done.

First, we state the following facts. Their proofs are simple and can be found in our technical report [33].

Lemma 3. *Let $\mathcal{S}_i(p,q)$ be the probability that a random walker starting from p reaches q (not necessarily for the first time) at the i-th step. For any $p \in P$ and $q \in Q$,*

$$\mathcal{P}_i(p,q) \le \mathcal{S}_i(p,q)$$

Lemma 4. *Let $\mathcal{S}_i(P,q)$ be the probability that a random walker starting from any node $p \in P$ reaches q at the i-th step. For any $p \in P$ and $q \in Q$,*

$$\mathcal{S}_i(p,q) \le \mathcal{S}_i(P,q)$$

Based on Lemmas 3 and 4, we may use $\mathcal{S}_i(P,q)$ as an upper bound of $\mathcal{P}_i(p,q)$. However, computing $\mathcal{S}_i(P,q)$ is costly, since it involves a disjunction of all the events "at the i-th step, a walker at a node $p \in P$ reaches q, and any walker at node $p' \in P - \{p\}$ does not reach q", each of which consists of many random walks. Notice that these events are not mutually exclusive. Hence, a summation over all $\mathcal{S}_i(p,q)$ values for $p \in P$, which is not smaller than $\mathcal{S}_i(P,q)$, can be used to bound $\mathcal{S}_i(P,q)$. We thus obtain Theorem 1 below.

Theorem 1. *For any $p \in P, q \in Q$, and $0 \le l \le d$,*

$$h_d(p,q) \le h_l(p,q) + Y_l^+(P,q)$$

where

$$Y_l^+(P,q) = \alpha \sum_{i=l+1}^{d} \big[\lambda^i \cdot \min(\sum_{p \in P} \mathcal{S}_i(p,q), 1)\big] \tag{9}$$

Proof: As discussed before, $\mathcal{P}_i(p,q) \le \mathcal{S}_i(P,q) \le \sum_{p \in P} \mathcal{S}_i(p,q)$. Also, $\mathcal{P}_i(p,q) \le 1$. The theorem holds. ∎

In B−IDJ−Y, $\mathcal{U}_l^+$ is computed according to Equation (9).

Implementation. To execute B−IDJ−Y, we first compute the upper bound function (i.e., Y_l^+) for every $q \in Q$ and $l \in [0,d]$. Algorithm 2 is then executed. We now briefly explain how to obtain these Y_l^+ values; the details can be found in [33]. Let $probVec$ be a size-$|V_G|$ vector, where $probVec[v]$ stores the value of $\sum_{p \in P} \mathcal{S}_i(p,v)$, for every $v \in V_G$. Initially, we set $probVec[p] = 1$ for every $p \in P$, and other entries to 0. Then we perform a d-step random walk over all nodes in V_G. In the l-th step, we compute $Y_l^+(P,q)$ for each node $q \in Q$. The running time of calculating all these Y_l^+ values is $O\big(d|E_G|\big)$. Thus, the time complexity of B−IDJ−Y is the same as that of B−IDJ−X (i.e., $O(|Q| \cdot d|E_G|)$). The space overhead of B−IDJ−Y is $O(d|V_G|)$.

Discussions. Let us compare B−IDJ−X and B−IDJ−Y by considering the following lemma:

Lemma 5. *For any $q \in Q$, $Y_l^+(P,q) \le X_l^+$.*

Proof: Based on Equation (9), $Y_l^+(P,q) \le \alpha \sum_{i=l+1}^{d} \lambda^i$. However, in Equation (7), $X_l^+ = \alpha \sum_{i=l+1}^{\infty} \lambda^i$. Hence, the lemma holds. ∎

Recall that the upper bound function, $\mathcal{U}_l^+$, is assigned to be X_l^+ and Y_l^+ for B−IDJ−X and B−IDJ−Y respectively. From Lemma 5, we see that B−IDJ−Y uses a tighter upper bound than B−IDJ−X does. Hence, the pruning power of B−IDJ−Y is better than that of B−IDJ−X. Compared with F−IDJ, the performance of a 2-way join using B−IDJ−Y is enhanced by a factor of $|P|$. Hence, among all the solutions for `twoWayJoin` used in PJ, B−IDJ−Y is the best choice. We verify these claims experimentally in Section VII.

D. The PJ−i Algorithm

In Step 10 of PJ (Algorithm 1), we execute the procedure `getNextNodePair` to obtain the next item in $\mathcal{L}_{R_i,R_j}$. As mentioned in Section IV, PJ does this by simply running a 2-way join algorithm. If this step is invoked frequently, the performance of PJ can be affected. Next, let us study a better way of implementing `getNextNodePair`.

Recall that in Step 4 of Algorithm 1, PJ evaluates a top-m 2-way join query (i.e., `twoWayJoin`) for each edge in $\mathcal{Q}$. Since the top-m and top-$(m+1)$ join results are highly similar, the information computed in Step 4 is potentially useful for the executing `getNextNodePair`, which yields the $(m+1)$-th node pair. Based on this observation, we develop PJ−i, which is a variant of PJ. The PJ−i algorithm uses B−IDJ to

evaluate 2-way joins, and reuses the information produced in `twoWayJoin` to execute `getNextNodePair`. Specifically, in `PJ-i` we make the following changes to `PJ`:

• Using a mutable priority queue, $\mathcal{F}$, for storing the information computed in a 2-way join. An entry of $\mathcal{F}$ contains four attributes: a node pair (p, q) which is also the key attribute; lower and upper bounds of $h(p, q)$, i.e., $h_d^-(p, q)$ and $h_d^+(p, q)$; and the number l of random walks used to evaluate the bounds. The entries in $\mathcal{F}$ are arranged in descending order of $h_d^+(p, q)$. We use a hash table $\mathcal{H}$ that maps (p, q) to its corresponding entry in $\mathcal{F}$, so that we can easily retrieve the entry with (p, q).

• A modified `B-IDJ`, which evaluates a top-m 2-way join in Step 4 of Algorithm 1. We change `B-IDJ` by recording its computed information in $\mathcal{F}$. Specifically, in Algorithm 2, immediately after Step 8, we consider the following tuple:

$$s =< (p, q), score[p], score[p] + \mathcal{U}_l^+, l >$$

We search the entry e with key (p, q) in $\mathcal{F}$ by using $\mathcal{H}$. If no entry is found, we insert s to $\mathcal{F}$. Otherwise, let e be the entry found in $\mathcal{F}$. We supersede e with s if the value of l in e (denoted by $e.l$) is less than $s.l$. Thus, we only store the DHT bound information of $h(p, q)$ computed with the largest number of random walk steps performed. As discussed before, the larger is the value of l, the tighter are the DHT bounds. Similar maintenance operations of $\mathcal{F}$ are performed after Step 14. We remove the k entries from $\mathcal{F}$, whose node pairs appear in $\mathcal{B}$, after Step 17.

• A new `getNextNodePair`. This procedure is used in Step 10 of Algorithm 1 for generating a node pair for a 2-way join on (R_i, R_j) with the next highest DHT score. It is invoked when the list $\mathcal{L}_{R_i,R_j}$ is exhausted. If this node pair is found in $\mathcal{F}$, then we may not have to run `twoWayJoin`. Recall that all the entries in $\mathcal{F}$ are sorted in descending order of $h_d^+(p, q)$. Let e_1 and e_2 be the first two entries of $\mathcal{F}$, with node pairs (p_1, q_1) and (p_2, q_2) respectively. If $e_1.h_d^-(p_1, q_1) > e_2.h_d^+(p_2, q_2)$, then $h_d(p_1, q_1)$ must be higher than the DHT scores of node pairs stored in other entries in $\mathcal{F}$. Hence, (p_1, q_1) must be the answer. If $e.l < d$, we compute $h_d(p_1, q_1)$ by performing a d-step backward random walk from q_1. Then, we update the entries in $\mathcal{F}$ for node pairs affected by this backward processing (i.e., $\{(p, q_1)|p \in P\}$). The answer $< (p_1, q_1), h_d(p_1, q_1) >$ is then returned by `getNextNodePair`. If it is not clear whether $h_d(p_1, q_1)$ is the highest DHT score, we "refine" its value, by performing $\min(2 \times e_1.l, d)$-step backward random walk from q_1.

Discussions. Let us now compare the performance between `PJ` and `PJ-i`. We assume that they both use `B-IDJ-Y` for performing 2-way joins. The worst-case running time of `PJ` is $O((M^2 - m) \cdot Md|E_G|)$, where $M = \max(|R_1|, \ldots, |R_n|)$. In `PJ-i`, we redesign `getNextNodePair` so that its answer could be retrieved from the $\mathcal{F}$ structure quickly. Its complexity is $O(Md|E_G|)$, which is much faster than `PJ`. Our experiments, discussed next, also reveals the superiority of `PJ-i` over `PJ`. The space overhead of `PJ-i` (for storing $\mathcal{F}$ and $\mathcal{H}$) is $O(|P||Q|)$. In practice, some nodes in Q are pruned, and the space required is much less. Please refer to our report [33] for the pseudocodes and complexity analysis of `PJ-i`.

VII. Results

In Section VII-A, we discuss the experimental setup. We then study the effectiveness and efficiency of our n-way join algorithms, in Sections VII-B and VII-C respectively. Section VII-D reports some results about the 2-way join.

A. Experimental Setup

Datasets. We examine three real graph datasets below:

• *DBLP* [34] is a graph constructed from bibliographical records in 2012. It is an undirected and weighted graph with $188k$ nodes and $1,140k$ edges. Two authors are connected if they have coauthored at least one paper. The edge weight is the number of papers that they published together. Authors who published in the same research area form a node set.

• *Yeast* [35] is a protein-protein interaction (PPI) network, where two protein nodes are connected if they may interact. The graph is undirected and unweighted, with $2.4k$ nodes and $7.2k$ edges. Nodes in this graph are partitioned into 13 (non-overlapping) sets based on their types.

• *YouTube* [36] is a large, undirected, and unweighted graph, with 1.1 million nodes and 3 million edges. It is extracted from the video-sharing social network, where users can form friendship with each other, and create interest groups. We consider such groups as node sets.

DHT and queries. We use DHT_λ as the default DHT measure, with $\lambda = 0.2$, $\alpha = 1.25$, and $\beta = -1.25$. We also examine DHT_e. We require a highly accurate DHT metric, by setting $\varepsilon = 10^{-6}$, or equivalently, $d = 8$ (Lemma 1). We have tested different top-k n-way joins on the above datasets. By default, $k = 50$, and n ranges from 2 to 7. We use the `MIN` function to calculate the aggregate score f. We will describe the query graphs used in these queries later. We have implemented all the multi-way join and 2-way join algorithms mentioned in this paper. For the `AP` algorithm, we use `F-BJ` to implement `twoWayJoin`. For `PJ`, we use `B-IDJ-Y` instead. We made these choices because `F-BJ` and `B-IDJ-Y` are respectively the best 2-way join algorithms for `AP` and `PJ` in all our experiments. [4] We set m to 50 for `PJ` and `PJ-i`. For the experiments on execution time of the algorithms, each data point is an average of running the algorithms for 10 times. Our solutions were implemented in C++ and run on a 3.40GHz Intel Core processor PC with 4GB memory and Windows Server 2012.

B. Effectiveness of n-way join

We first study the effectiveness of n-way join queries in several scenarios. Since all our n-way join algorithms produce the same answer, we only present the result for `PJ-i`.

1. Triangle and chain. We first examine the result of a top-5 3-way join on *DBLP*. The three node sets consist of respective experts from Database (DB), Artificial Intelligence (AI), and System (SYS) areas, forming a "triangle" query graph, as shown in Figure 2(a). For each node set, we select 100 authors

[4] `F-BJ` performs the best for `AP`, because `AP` computes all node pairs for each node set, and hence the pruning techniques provided by more advanced algorithms (e.g., `F-IDJ`) are not useful.

TABLE III: Top-5 3-way join on *DBLP*.

	Triangle			Chain		
	DB	AI	SYS	DB	AI	SYS
1	Franklin	Jordan	Patterson	Hellerstein	Guestrin	Stoica
2	Halevy	Weld	Gribble	Madden	Guestrin	Balakrishnan
3	Deshpande	Guestrin	Seshan	Hellerstein	Guestrin	Shenker
4	Stonebraker	Pfeffer	Seltzer	Hellerstein	Guestrin	Maniatis
5	Garofalakis	Jordan	Brewer	Deshpande	Guestrin	Seshan

who have the highest number of publications. We list the top 5 answers of this query in Table III. In the rank-1 answer, the three researchers (Franklin, Jordan, and Patterson) indeed work closely with each other. They are also from the same lab. The researchers in the rank-2 answer (Halevy, Weld, and Gribble) also come from the same school. In the rank-3 answer, the three experts (Stonebraker, Pfeffer, and Seltzer) have worked on sensor data before. Hence, the people returned by this query in this experiment are closely connected.

We have also put the same node sets in a "chain query graph": AI is linked to DB, which is connected to SYS. The structure of this query graph is the same as that of Figure 2(b). The 3-way join results, as shown in Table III, are quite different from that of the triangle query graph. In particular, we have verified that the researchers from AI and SYS did not have close collaboration relationship.

2. Link prediction. Next, we consider the usefulness of the 2-way join in *link prediction*. Particularly, we perform a 2-way join on two node sets, and check whether the node pairs returned by the query correctly predict that an edge will be generated between the nodes. For the datasets tested, we distinguish between a *test graph* $\mathcal{T}$, on which the 2-way join is applied, and a *true graph* $\mathcal{G}$, where we verify the correctness of the prediction. The three datasets described in Section VII-A are considered to be true graphs. For each dataset, the node sets (P and Q) and $\mathcal{T}$ are defined as follows:

• *DBLP*: P and Q are respectively DB and AI; $\mathcal{T}$ is the co-authorship graph by retaining only the edges before 1st January, 2010 from the DBLP website [34].

• *Yeast*: P and Q are respectively 3-U and 8-D, which are the two largest partitions of *Yeast*; $\mathcal{T}$ is obtained by randomly removing half of the edges between the node pairs in (P, Q) from *Yeast*.

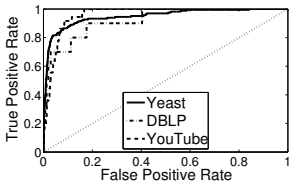
• *YouTube*: P and Q are anonymous groups with ids 1 and 5; $\mathcal{T}$ is again formed by randomly removing half of the edges between the node pairs in (P, Q) from *YouTube*.

To measure link prediction quality, we first perform a top-k 2-way join on $\mathcal{T}$, based on the query graph in Figure 1(b). For each node pair (p, q) that appears in the top-k result, but not in $\mathcal{T}$, we classify it as:

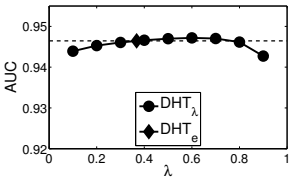
• a *true positive*, if (p, q) in $\mathcal{G}$; or

• a *false positive*, if (p, q) is not in $\mathcal{G}$.

By varying the value of k, we plot the true and false positives on *ROC curves*, and compute their *AUC* (i.e., area under the ROC curve). These two metrics are commonly used to measure accuracy, and is robust to the skewness between possible and existing edges [37]. Their values range from zero (low) to one (high). Figure 6(a) shows the ROC curves for the three datasets. Observe that at a relatively low *false positive rate* (about 0.1), the joins on the three datasets achieve high *true positive rates* at more than 0.7. Their corresponding AUC scores, as reported in Table IV, are also high (more than 0.9). Hence, the 2-way joins perform well in link prediction.



(a) ROC curves.



(b) AUC (*Yeast*).

Fig. 6: Link prediction (2-way join).

TABLE IV: AUC scores for link- and 3-clique-prediction.

Dataset	link-prediction	3-clique-prediction
Yeast	0.9453	0.9536
DBLP	0.9222	0.9998
YouTube	0.9544	0.9609

3. 3-clique prediction. We next test whether a 3-clique of a true graph $\mathcal{G}$ can be correctly predicted by running a 3-way join on the test graph $\mathcal{T}$. The three node sets (P, Q, and R) are connected as a triangle query graph (c.f. Figure 2(a)). The configuration of each dataset is as follows:

• *DBLP*: P, Q and R are respectively DB, AI, and SYS; $\mathcal{T}$ is the graph on 1st January, 2010, obtained from [34].

• *Yeast*: P, Q, and R are groups 3-U, 5-F, and 8-D; $\mathcal{T}$ is derived by randomly removing an edge from each 3-clique in *Yeast*, with each node of the 3-clique in P, Q, and R.

• *YouTube*: P, Q, and R are groups with ids 1, 5 and 88; $\mathcal{T}$ is derived by randomly removing an edge from each 3-clique in *YouTube*, with each node of the 3-clique in P, Q, and R.

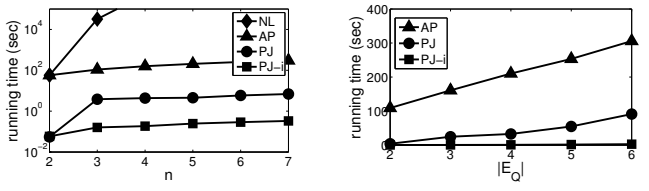
We use the same prediction quality measure described previously. Table IV shows that AUC scores for the three datasets are close to 1. This reflects that the 3-way join is also effective in predicting 3-way cliques.

4. DHT_λ and DHT_e. Figure 6(b) shows the AUC for different values of λ, using the 2-way join with DHT_λ as the similarity measure. We see that the AUC for DHT_λ is consistently higher than 0.94, and attains the highest value at $\lambda = 0.6$. The AUC for DHT_e is also high. Hence, both DHT measures are good candidates for link prediction. The effectiveness results of the two measures in other experiments are similar; they are omitted due to limited space.

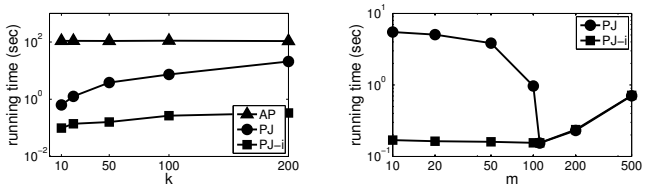
C. Efficiency of n-way join

We next examine the performance of the n-way join algorithms. Here we present the results for *Yeast* and *DBLP*. The conclusions obtained for *YouTube* are similar, and are omitted due to space constraints. The full details can be found in [33].

1) Yeast: **Effect of n** (Figure 7(a)). We first examine the performance of the n-way join algorithms under different values of n. The query graph associated with each n-way join is a chain, where the node $\mathcal{R}_1$ has a directed edge pointing to $\mathcal{R}_2$, $\mathcal{R}_2$ points to $\mathcal{R}_3$, and so on. Observe that the running times of all the four algorithms increase with n.

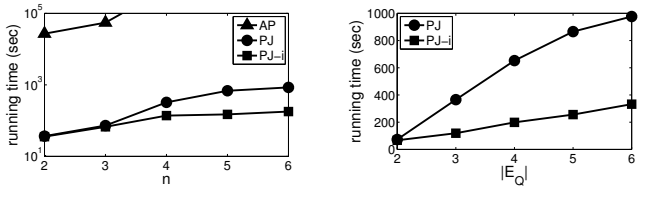


(a) Running time vs. n (b) Running time vs. $|E_{\mathcal{Q}}|$

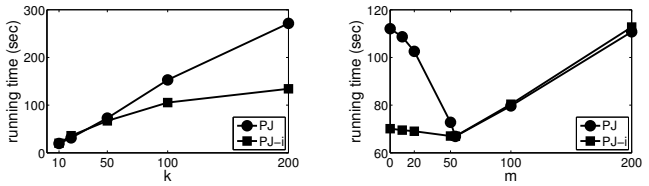


(c) Running time vs. k (d) Running time vs. m

Fig. 7: n-way join on *Yeast*



(a) Running time vs. n (b) Running time vs. $|E_{\mathcal{Q}}|$



(c) Running time vs. k (d) Running time vs. m

Fig. 8: n-way join on *DBLP*

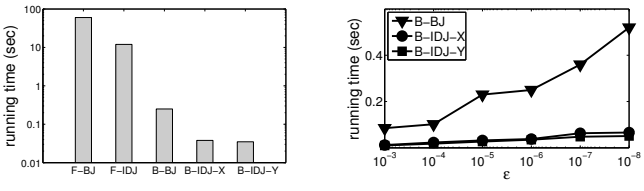
NL is the slowest, and cannot complete in a reasonable time at $n \geq 3$. The performance of AP is better than NL, since it computes less number of DHT scores and uses rank join. It is outperformed by PJ−i by an order of magnitude, since PJ−i does not generate all node pairs for every pair of node sets, and needs much less DHT computations than AP. This is supported by the fact that under a wide range of values of k, less than 1% of the 2-way join results are included in the n-way join answers. The execution time of PJ−i is about 20 times faster than PJ. Whenever getNextNodePair is invoked, PJ−i needs much less time than PJ, which runs a 2-way join from scratch. We next focus on AP, PJ, and PJ−i.

Effect of $|E_{\mathcal{Q}}|$ (Figure 7(b)). We next vary the number of edges in $\mathcal{Q}$ that contains 3 nodes. For example, when $|E_{\mathcal{Q}}| = 2$, the nodes are "chained" by two edges; when $|E_{\mathcal{Q}}| = 3$, $\mathcal{Q}$ forms a 3-clique. The detailed configurations of $\mathcal{Q}$ are described in [33]. Again, AP is the worst, and PJ−i performs the best. At $|E_{\mathcal{Q}} = 4|$, the respective execution times of AP, PJ and PJ−i are 210, 32, and 2 seconds.

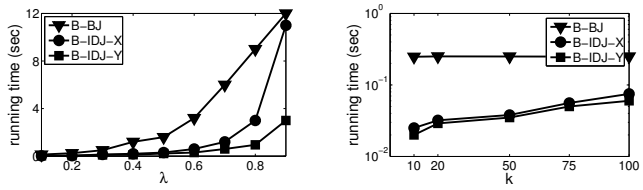
Effect of k (Figure 7(c)). We next consider a 3-way join using a chain query graph. With m fixed, when k increases, the chances of both PJ and PJ−i for running getNextNodePair increase. PJ−i obtains the next node pair faster, and consistently outperforms PJ. At $k = 200$, PJ−i is 2 orders of magnitude faster PJ. Notice that the both solutions are faster than AP.

Effect of m (Figure 7(d)). We examine the effect of m on PJ and PJ−i in a 3-way join that uses a chain query graph. Recall that both solutions run a top-m 2-way join for every query graph edge. When m is small (from 10 to 20), PJ does not acquire enough information through the top-m join for evaluating the n-way join. Hence, it has to constantly invoke getNextNodePair, which is essentially a top-s join (with $s > m$). This can seriously affect the efficiency of PJ. PJ−i is less affected, since getNextNodePair can be done much faster. When $m \geq 100$, the performance of both algorithms converge, since the top-m join results are sufficient for constructing the n-way join answers, without the need of invoking getNextNodePair. However, this may be wasteful, since some 2-way join answers may not contribute to the n-way join. We further observe that the performance of PJ−i is less sensitive to m, and hence m can be relatively easy to set compared with PJ. Our default value of m is 50, which is the same as k; for PJ−i, its running time is close to the optimal (at $m = 100$).

2) DBLP: We perform the same set of experiments for *DBLP*. As shown in Figure 8, their trends and conclusions are similar to that of *Yeast*. The only difference is that the database size of *DBLP* is much larger than *Yeast*, and so AP performs badly in most experiments. Hence, we only show some of its results in Figure 8(a).



(a) 2-way join algorithms (b) Running time vs. ε



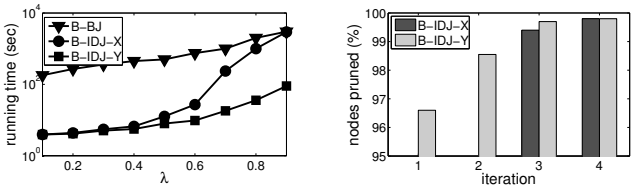
(c) Running time vs. λ (d) Running time vs. k

Fig. 9: 2-way join on *Yeast*

D. *Efficiency of* 2*-way join*

Finally, we study the performance of 2-way join algorithms. Due to space constraints, we show the results for *Yeast* and *DBLP* only. The results for *YouTube* can be found in [33]. For these experiments, we use the same node sets described in the link prediction experiment (Section VII-B).

Figure 9 presents the results on *Yeast*. The running times of all the five algorithms are shown in Figure 9(a). We can see that our backward processing algorithms, namely B−BJ, B−IDJ−X, and B−IDJ−Y, significantly outperform



(a) Running time vs. λ (b) Performance analysis

Fig. 10: 2-way join on *DBLP*

the forward processing counterparts (i.e., F−BJ and F−IDJ). Specifically, B−BJ is more than 100 times better than F−BJ, because it performs backward random walk and addresses an improvement of a factor of $O(|P|)$. For the same reason, both B−IDJ variants are two orders of magnitude faster than F−IDJ. We next focus on the backward processing algorithms.

Effect of ε (Figure 9(b)). Both B−IDJ methods enable the pruning of nodes from the Q set during their random walks. Since their pruning is effective, they are 6–8 times better than the B−BJ, especially when ε is small.

Effect of λ (Figure 9(c)). As λ, the decay factor, increases, more steps of random walks are needed. Moreover, X_l^+ increases with λ, and so the pruning is less effective. Thus, the running time of B−IDJ−X increases with λ. When λ equals to 0.9, the B−IDJ−X needs almost the same running time as B−BJ does. However, by using Y_l^+, which is much tighter and less sensitive to λ, B−IDJ−Y achieves up to 4 times better performance, at large λ values.

Effect of k (Figure 9(d)). The running time of B−BJ is not affected by the value of k, since it computes all-pair DHTs. Both B−IDJ methods consume more time as k increases, because more results need to be produced. As a result, more random walks are required, and more nodes in Q need to be examined. However, they are still better than B−BJ.

Our results for *DBLP* are shown in Figure 10. Figure 10(a) shows the running times under different λ, the trend of which is similar to Figure 9(c). The advantage of B−IDJ−Y is more profound at large λ. A detailed analysis in Figure 10(b) shows the fraction of nodes pruned in the first four iterations at $\lambda = 0.7$. B−IDJ−Y prunes more than 96.5% and 98.5% nodes from Q after the first and the second iterations respectively. However, B−IDJ−X fails to prune any node in the first two iterations, since a looser bound is used. Since early iterations are also cheaper to handle, B−IDJ−Y is highly efficient, and we use it in our PJ algorithms.

VIII. Conclusions

The prevalence of graphs in emerging applications has recently attracted a lot of attention. In this paper, we study the n-way join operator for graph databases, which can be used to discover interesting relationship among graph nodes. The best algorithm for running this query is PJ−i, which uses B−IDJ−Y, an efficient 2-way join solution. Our extensive evaluation shows that PJ−i is highly effective and efficient. We plan to extend the study of n-way join for other proximity measures on graphs, including Personalized PageRank [20], SimRank [21], and PathSim [38].

Acknowledgments

Wangda Zhang and Reynold Cheng were supported by the Research Grants Council of Hong Kong (RGC Project HKU 711309E), and the University of Hong Kong (Project 201211159083). We would like to thank the reviewers for their insightful comments.

References

[1] C. Aggarwal and H. Wang, *Managing and mining graph data*. Springer, 2010.
[2] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *JASIST*, 2007.
[3] M. Brand, "A random walks perspective on maximizing satisfaction and profit," in *SIAM*, 2005.
[4] Q. Mei, D. Zhou, and K. Church, "Query suggestion using hitting time," in *CIKM*. ACM, 2008.
[5] D. Aldous and J. Fill, "Reversible markov chains and random walks on graphs," 2002.
[6] P. Sarkar and A. Moore, "A tractable approach to finding closest truncated-commute-time neighbors in large graphs," in *UAI*, 2007.
[7] A. Joshi, R. Kumar, B. Reed, and A. Tomkins, "Anchor-based proximity measures," in *WWW*. ACM, 2007.
[8] Z. Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan, "Assessing and ranking structural correlation in graphs," in *SIGMOD*, 2011.
[9] P. Sarkar and A. Moore, "Fast nearest-neighbor search in disk-resident graphs," in *KDD*. ACM, 2010.
[10] M. Roth et al., "Suggesting friends using the implicit social graph," in *KDD*. ACM, 2010, pp. 233–242.
[11] N. Krogan et al., "Global landscape of protein complexes in the yeast saccharomyces cerevisiae," *Nature*, 2006.
[12] J. Dittrich and B. Seeger, "Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces," in *KDD*, 2001.
[13] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006.
[14] C. Xiao, W. Wang, X. Lin, J. Yu, and G. Wang, "Efficient similarity joins for near-duplicate detection," *TODS*, 2011.
[15] T. Brinkhoff, H. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," *SIGMOD*, 1993.
[16] J. Sankaranarayanan, H. Alborzi, and H. Samet, "Distance join queries on spatial networks," in *SIGSPATIAL GIS*. ACM, 2006.
[17] Y. N. Silva, W. G. Aref, and M. H. Ali, "The similarity join database operator," in *ICDE*. IEEE, 2010.
[18] L. Zou, L. Chen, and M. Özsu, "Distance-join: Pattern match query in a large graph database," *VLDB*, 2009.
[19] L. Sun, C. Cheng, X. Li, D. Cheung, and J. Han, "On link-based similarity join," *VLDB*, 2011.
[20] G. Jeh and J. Widom, "Scaling personalized web search," in *WWW*. ACM, 2003, pp. 271–279.
[21] ——, "Simrank: a measure of structural-context similarity," in *KDD*. ACM, 2002.
[22] F. N. Afrati and J. D. Ullman, "Optimizing multiway joins in a map-reduce environment," *TKDE*, 2011.
[23] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using mapreduce," *VLDB*, 2012.
[24] D. Zhang, V. J. Tsotras, and B. Seeger, "Efficient temporal join processing using indices," in *ICDE*. IEEE, 2002.
[25] J. Enderle, M. Hampel, and T. Seidl, "Joining interval data in relational databases," in *SIGMOD*. ACM, 2004.
[26] N. Mamoulis and D. Papadias, "Multiway spatial joins," *TODS*, 2001.
[27] A. Corral et al., "Multi-way distance join queries in spatial databases," *Geoinformatica*, vol. 8, no. 4, pp. 373–402, 2004.
[28] K. Schnaitter and N. Polyzotis, "Evaluating rank joins with optimal cost," in *PODS*. ACM, 2008.
[29] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases," *VLDB*, 2004.
[30] J. Finger and N. Polyzotis, "Robust and efficient algorithms for rank join evaluation," in *SIGMOD*. ACM, 2009.
[31] J. Hopcroft and D. Sheldon, "Manipulation-resistant reputations using hitting time," *Algorithms and Models for the Web-Graph*, 2007.
[32] P. Sarkar and A. Moore, "Fast dynamic reranking in large graphs," in *WWW*. ACM, 2009.
[33] W. Zhang, R. Cheng, and B. Kao, "Evaluating multi-way joins over discounted hitting time," The University of Hong Kong, http://www.cs.hku.hk/research/techreps/document/TR-2013-07.pdf, Tech. Rep., 2013.
[34] M. Ley *et al.*, "Dblp-some lessons learned," *VLDB*, 2009.
[35] D. Bu et al., "Topological structure analysis of the protein–protein interaction network in budding yeast," *Nucleic Acids Research*, vol. 31, no. 9, pp. 2443–2450, 2003.
[36] A. Mislove et al., "Measurement and analysis of online social networks," in *SIGCOMM*. ACM, 2007.
[37] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.
[38] Y. Sun et al., "Pathsim: Meta path-based top-k similarity search in heterogeneous information networks," *VLDB*, 2011.