Efficient Batch Processing for Multiple Keyword Queries on Graph Data

Lu Chen[†], Chengfei Liu[†], Xiaochun Yang[§], Bin Wang[§], Jianxin Li[¶], and Rui Zhou[‡]
[†]Swinburne University of Technology, Australia
[§]Northeastern University, China
[¶]University of Western Australia, Australia
[‡]Victoria University, Australia
[†]{luchen, cliu}@swin.edu.au [§]{yangxc,binwang}@mail.neu.edu.cn
[¶]jianxin.li@uwa.edu.au [‡]rui.zhou@vu.edu.au

ABSTRACT

Recently, answering keyword queries on graph data has drawn a great deal of attention from database communities. However, most graph keyword search solutions proposed so far primarily focus on a single query setting. We observe that for a popular keyword query system, the number of keyword queries received could be substantially large even in a short time interval, and the chance that these queries share common keywords is quite high. Therefore, answering keyword queries in batches would significantly enhance the performance of the system. Motivated by this, this paper studies efficient batch processing for multiple keyword queries on graph data. Realized that finding both the optimal query plan for multiple queries and the optimal query plan for a single keyword query on graph data are computationally hard, we first propose two heuristic approaches which target maximizing keyword overlap and give preferences for processing keywords with short sizes. Then we devise a cardinality based cost estimation model that takes both graph data statistics and search semantics into account. Based on the model, we design an A* based algorithm to find the global optimal execution plan for multiple queries. We evaluate the proposed model and algorithms on two real datasets and the experimental results demonstrate their efficacy.

1. INTRODUCTION

Over the last decade, web search technology has developed into a multi-billion-dollar industry that serves answers to hundreds of millions of users each day. Users, in turn, have come to rely on search engines for an ever-increasing share of their information and other needs, in the process supplanting many other printed, electronic, and human information resources. Similarly, there is a clear trend that database service providers are also eager to provide their data consumers a keyword search interface for exploring many different types of information, including web sites, books, news, videos, product information and technical documents. Keyword search has been extensively studied in the field of database, e.g., relational database [1, 9], XML database [16, 22], graph database [2, 8, 9, 11, 12, 14, 15, 17] as well as spatial database [4, 6]. However, all the above existing work focused on single query processing in keyword search. They design their algorithms and indices based on the performance of answering single keyword queries. Such single query based techniques are not enough to support real query processing systems due to several reasons. Normally, a query processing system should support multiple types of users. For example, beyond general users, a third-party company as an important data consumer may perform significant analysis and mining of the underlying data in order to optimize their business by issuing a group of queries as a batch query. Here, the third-party company may be an industry sector collecting their interested data from online databases, a researcher comparing the scientific results from scientific databases. In all the cases, the batch of queries issued from the third-party company are used to mine information from the databases and optimize their business or targeting benefit. It is also important that such a query processing system is designed with the goal of returning results in fractions of a second for a large number of queries to be received in a very short time. Recently, domain-specific search engines have been widely used as they provide specific and profound information that well satisfies users' search intentions. Usually, the underlying data is highly structured, and in most cases, is represented as graphs. We observe that for a popular domain-specific search engine, the number of keyword queries received could be substantially large even in a short time interval, and the chance that these queries share common keywords is quite high. Therefore, answering keyword queries in batches would significantly enhance the performance of domain-specific search engines. However, most graph keyword search solutions proposed so far primarily focus on a single query setting.

In this paper, we study the problem of batch processing of keyword queries on graph data. Our goal is to process a set of keyword queries as a batch while minimizing the total time cost of answering the set of queries. Batch query processing (also known as multiple-query optimization) is a classical problem in database communities. In relational database, Sellis et. al. in [20] studied multiple SQL query optimization. The key idea is to decompose SQL queries into subqueries and guarantee each SQL query in the batch can

CIKM'16 , October 24 - 28, 2016, Indianapolis, IN, USA

ISBN 978-1-4503-4073-1/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2983323.2983806

be answered by combining subset of subqueries. However, it may incur a challenging issue to maintain the intermediate results of all the possible subqueries, which leads to expensive space cost and extra I/O cost. To do this, Roy et. al. in [18] evaluated the tradeoff between reuse and recomputation of intermediate results for subqueries by comparing pipeline cost and reusing cost. In addition, Jacob and Ives in [10] addressed the problem of interactive keyword queries as a batch query in relational database. In their work, the keyword search semantics is defined by the candidate networks [9], which requires to know the relational data schema in advance. Batch query processing was also studied in other contexts, e.g., spatial-textual queries [5], RDFSPARQL [13], and XQueries [3].

After investigating batch query processing in different contexts and single keyword query processing in graph databases, we observe that all the existing techniques cannot be applied to our problem - batch keyword query processing on graph data. The main reasons come from the following significant aspects. (1) *Meaningful Result Semantics*: r-clique can well define the semantics of keyword search on graph data as r-clique can be used to discover the tightest relations among all the given keywords in a query [12], but there is no existing work that studies batch query processing with this meaningful result semantics; (2) *Complexity of the Optimal Batch Processing*: it is an NP-complete problem to optimally process multiple keyword queries in a batch. This is because each single query corresponds to several query plans, and obviously we cannot enumerate all the possible combinations of single query plans to get the optimal query plan for multiple queries; (3) *Not-available Query Historic Information*: unlike the batch query processing [23], we do not have the assumption that we know the result sizes of any subqueries before we actually run these queries because this kind of historic information is not always available.

Although we can simply evaluate the batch queries in a pre-defined order and re-use the intermediate results in the following rounds as much as we can. But there is no guarantee the batch queries can be run optimally. To address this, we firstly develop two heuristic approaches which give preferences for processing keywords with short sizes and maximize keyword overlaps. Then we devise a cardinality estimation cost model by considering graph connectivity and result semantics of r-clique. Based on the cost model, we can develop an optimal batch query plan by extending A* search algorithm. Since the A* search in the worst case could be exhaustive search, which enumerates all possible global plans, we propose pruning methods, which can efficiently prune the search space to get the model based optimal query plan.

We make the following contributions in this paper:

- In this work, we propose and study a new problem of batch keyword query processing on native graph data, which is popular to be used in modern data analytics and management systems.
- We formalize the proposed problem, which is NP-complete. To address it, we develop two heuristic solutions by considering the features of batch keyword query processing.
- To optimally run the batched queries, we devise an estimation-based cost model to assess the computational cost of possible sub-queries, which is then used to identify the optimal plan of the batch query evaluation.
- We conduct extensive experiments on DBLP and IMDB dataset to evaluate the efficiency of proposed algorithms and verify the precision of the cost model.

The rest of this paper is organized as follows. In Section 2, we introduce preliminaries and formally define the problem. In Section 3, we present two heuristic rule based approaches, a shortest list eager one and a maximal overlap driven one. In Section 4, we propose a cost estimation model for estimating the cardinalities of r-join operations used for evaluating r-cliques. Based on this estimation model, we then discuss how to find the cost-based optimal query plan for multiple queries efficiently in Section 5. We show the experimental results in Section 6, and introduce the related work in Section 7. Finally, we conclude our paper in Section 8.

2. PRELIMINARIES AND PROBLEM DEFINITIONS

In this section, we introduce preliminaries and define the problem that is to be addressed in this paper.

2.1 Keyword Query on Graph Data

Native graph data. The native graph data $G(V, E)$ consists of a vertex set $V(G)$ and an edge set $E(G)$. A vertex $v \in V(G)$ may contain some texts, which are denoted as $v.KS = \{v.key_1, \ldots, v.key_z\}$. We call the vertex that contains texts *content vertex*. An edge $e \in E(G)$ is a pair of vertices (v_i, v_j) $(v_i, v_j \in V)$. The shortest distance between any two vertices v_i and v_j is the number of edges in the shortest path between v_i and v_j, denoted as $dist(v_i, v_j)$.

Query processing for Single keyword query. Given a keyword query $q = \{k_1, \ldots, k_m\}$ on a graph G, the answer to q is a set of sub-graphs of G, each subgraph is generated using an r-clique [12] of G, which is a set of vertices that contains texts that match all keywords in q and the distance between any two vertices in the r-clique is less than or equals to r. Given a query q on G, we can get a set of r-cliques, denoted $\mathbb{RC}(q, G)$. For example, Figure 1 shows a subgraph G' of native graph data G. Given a query $q_1 = \{k_1, k_2, k_3, k_4\}$, let $r = 1$. An answer to q_1 is the thick vertex set in Figure 1, in which the vertex set $\{v_7, v_8, v_{10}\}$ is a 1-clique.

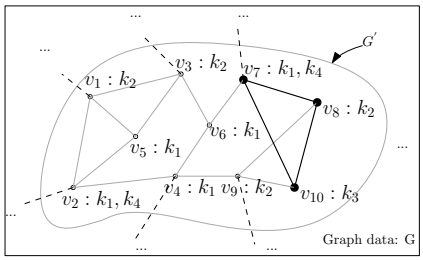
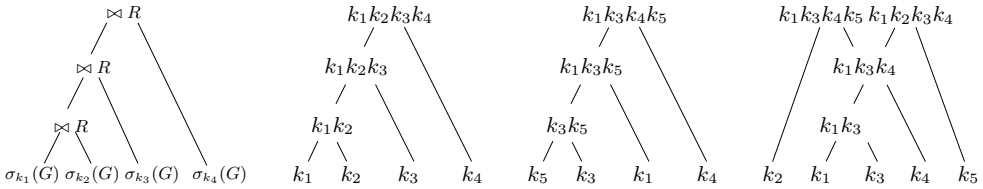


Figure 1: An example graph G and the answer subsgraphs to q_1 in the subgraph G'

Figure 2(a) shows a query plan for q_1. A query plan is an operation tree that contains two types of operations, one is a selection operation $\sigma_{k_i}(G)$ that selects vertices on graph G whose texts match keyword k_i, and the other is an r-join operation $\bowtie_{\mathcal{R}}$ that join two r-cliques of G. There could be many query plans to generate the final r-clique set based on the different processing order of r-joins. For simplifying the presentation, we use Figure 2(b) to express the query plan shown in Figure 2(a).



(a) $p_1(q_1)$ **(b)** $p_1(q_1)$ **(c)** $p_2(q_2)$ **(d)** $p_3(\{q_1, q_2\})$

Figure 2: Query plans for single queries q_1, q_2, and batch multiple queries $\{q_1, q_2\}$

In order to make the selection $\sigma_{k_i}(G)$ efficiently, we can build up an inverted list of vertices for each keyword contained in graph G. Then the cost of a selection is $O(1)$. Therefore the main cost of a query plan depends on the costs of its r-join operations.

2.2 Batched Multiple-Keyword Queries

Consider a batch of keyword queries $Q = \{q_1, \ldots, q_n\}$ on a native graph G, it returns answers to each query $q_i \in Q$.

A naive way to answer batched keyword queries is to run those queries one by one. For example we can run the query plans $p(q_1)$ and $p(q_2)$ shown in Figure 2(b) and Figure 2(c) one by one. Obviously it is inefficient. Ideally we hope to share some (intermediate) results of processed queries to avoid duplicate computation. For example, Figure 2(d) shows a query plan for a batch query $\{q_1, q_2\}$, where $q_2 = \{k_1, k_3, k_4, k_5\}$ and the intermediate results of r-joins $\sigma_{k_1}(G) \bowtie_{\mathcal{R}} \sigma_{k_3}(G)$ and $(\sigma_{k_1}(G) \bowtie_{\mathcal{R}} \sigma_{k_3}(G)) \bowtie_{\mathcal{R}} \sigma_{k_4}(G)$ can be shared by their upper-level r-join operations. Then the cost of this query plan $p_3(\{q_1, q_2\})$ is the summation of the cost of every r-join operations in the plan.

Problem definition. Given a batch of keyword queries $Q = \{q_1, \ldots, q_n\}$ on a native graph G, our aim is to construct a query plan $p(Q)^{opt}$ for all queries in Q such that $p(Q)^{opt}$ requires minimum cost. It is a typical NP-complete problem [19].

Finding the optimal query plan is non-trivial due to the following reasons.

- A single query corresponds to several query plans, and obviously we do not want to enumerate combinations of query plans for multiple queries to get the optimal one; and
- Let $\mathbb{RC}(K_1 \cup K_2, G) = \mathbb{RC}(K_1, G) \bowtie_{\mathcal{R}} \mathbb{RC}(K_2, G)$. The size of $\mathbb{RC}(K_1 \cup K_2, G)$ is not proportional to the sizes of $\mathbb{RC}(K_1, G)$ and $\mathbb{RC}(K_2, G)$. Therefore, it is not easy to predict the size of $\mathbb{RC}(K_1 \cup K_2, G)$.

3. HEURISTIC-BASED APPROACHES

We propose two heuristic approaches to target a "good" query plan and answer queries in the batch Q.

3.1 A Shortest List Eager Approach

We first propose an approach Basic whose main idea is to process every query in turn in the batch $Q = \{q_1, \ldots, q_n\}$, and for each query $q_i \in Q$ it starts from the shortest list to eagerly join with existing intermediate results if they exist.

Rule 1. *Given two inverted lists of keywords k_i and k_j, respectively. $\mathbb{RC}(\{k_i\}, G)$ takes precedence to r-join with the the existing intermediate results, if the list of k_i is shorter than that of k_j.*

Algorithm 1: Basic

```
Data: A graph G, queries Q={q_1, ..., q_n}
Result: R={RC(q_1, G), ..., RC(q_n, G)}
1  Load index H of inverted lists of vertices for keywords;
2  R ← ∅;
3  for i from 1 to n do
4      RC(q_i, G) ← ∅;
5      Processed keywords K_p ← ∅;
6      foreach keyword k in q_i do
7          if k is processed in previous queries then
8              K_p ← K_p ∪ {k};
9          else
10             RC({k}, G) ← Hash vertices in the inverted
               lists of keyword k;
11     Key ← K_p;
       // compute processed keyworkds
12     repeat
13         Find maximal set of processed keywords K_max;
14         RC(q_i, G) ← RC(q_i, G) ⋈_R RC(K_max, G);
15         K_p ← K_p − K_max;
16     until K_p is empty;
       // compute unprocessed keywords
17     Rank all remaining RC({k}, G) by their sizes in
       ascending order (k'_1, ..., k'_m);
18     foreach remaining keywork k in q_i do
19         RC(Key, G) ← RC(Key, G) ⋈_R RC({k}, G) ;
20         Key ← Key ∪ {k};
21     RC(q_i, G) ← RC(Key, G);
22 return R;
```

Algorithm 1 shows the detail of the algorithm Basic, which avoids processing keywords that have been processed. Therefore, for each iteration, it checks if the keywords of the current query q_i have been processed. For those processed keywords, it uses the intermediate results of the maximal set of processed keywords, and for those unprocessed keywords, it starts an r-join $\bowtie_{\mathcal{R}}$ between the processed intermediate results and the $\mathbb{RC}(\{k\}, G)$ with the smallest size.

It is clear that the algorithm Basic is better than the naive approach which simply processes the queries one after another while does not consider reusing processed intermediate results.

3.2 A Maximal Overlapping Driven Approach

The Algorithm Basic does not make full use of the shared (overlapping) keywords among all the queries in the batch Q. Therefore, in this section we propose a new approach based on the observation that more keywords often imply more

Algorithm 2: OVERLAP

```
   Data: Q={q_1, ..., q_n}
   Result: R={RC(q_1,G), ..., RC(q_n,G)}
1  Algorithm Overlap()
2      Calculate sharing factors in Q;
3      repeat
4          Calculate frequencies of unprocessed sharing
           factors;
5          Choose the precedent sharing factor sf in Q
           with maximal |sf|·freq(sf) according to Rule 2;
6          RC(sf,G) ← Cal(sf);
7          Remove the subtree rooted at sf;
8          Insert sf to a heap H;
9          while H is not empty do
10             Pop the first factor from H to sf;
11             foreach factor s ⊃ sf and |s|−|sf|= 1 do
12                 RC(s,G) ← RC(sf,G)⋈_R RC(s\sf,G);
13                 if s is a query q in Q then
14                     Q ← Q\{q};
15                 else
16                     Insert s to H;
17             Remove sf;
18     until Q is empty;
19     return R;
20 Procedure Cal(sf)
21     RC(sf,G) ← ∅;
22     if sf contains sub-sharing factors SF_c then
23         Choose the precedent sharing factor sf_c among
           SF_c with maximal sf_c · freq(sf_c);
24         RC(sf,G) ←Cal(sf_c);
25     Let k'_1, ..., k'_v be keywords sf − sf_c whose inverted
       lists are ranked in ascending order;
26     foreach keyword k'_i do
27         RC(sf,G) ← RC(sf,G)⋈_R RC(k'_i,G);
28     return RC(sf,G);
```

processing cost, and as a result, processing more frequently shared keywords first will benefit more queries. Before continuing, we define Sharing Factor first.

DEFINITION 1. *Sharing factor. Given a batch query* $Q=\{q_1,\ldots,q_n\}$*, for any two queries* $q_i, q_j \in Q (i \neq j)$*, we use the intersection of* q_i *and* q_j *to express their overlapped keywords, which is called* sharing factor *of* q_i *and* q_j*, denoted* $SF(q_i,q_j)$*.* $SF(q_i,q_j) = q_i \cap q_j$*.*

RULE 2. *Given a batch* Q *and let* $\mathcal{S}$ *be the set of sharing factors w.r.t.* Q*. For any two sharing factors* SF_i *and* SF_j *in* $\mathcal{S}$*,* $\mathbb{RC}(SF_i,G)$ *takes precedence over* $\mathbb{RC}(SF_j,G)$ *if* $|SF_i|\cdot freq(SF_i) > |SF_j|\cdot freq(SF_j)$*, where* $freq(SF)$ *is the frequency of* SF *in* Q*.*

Algorithm 2 shows the algorithm OVERLAP based on Rule 2. Given a batch of queries Q, the algorithm OVERLAP first calculates all sharing factors among queries in Q. It chooses a sharing factor sf with maximal $|sf|\cdot freq(sf)$ according to Rule 2 (line 5). Then in line 6 it calculates the intermediate result $\mathbb{RC}(sf,G)$ by invoking CAL(sf), which recursively processes sharing factors whose keywords are subsets of the keywords in sf (lines 20–28). This intermediate result $\mathbb{RC}(sf,G)$ can benefit all factors whose keywords are supersets of the keywords of sf. Such benefit can be propagated to queries in Q. Therefore, the algorithm OVERLAP pushes all such sharing factors that can be benefited from computing sf into a heap (line 8). Finally, it calculates r-cliques of the benefited queries (lines 9–17) and removes these processed queries (line 14). The algorithm repeats the above process until all queries have been processed.

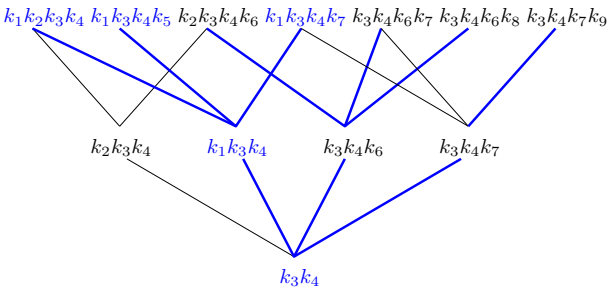
Figure 3 shows an example illustrating the algorithm OVERLAP. Given a keyword query batch Q containing queries: $q_1=\{k_1,k_2,k_3,k_4\}$, $q_2=\{k_1,k_3,k_4,k_5\}$, $q_3=\{k_2,k_3,k_4,k_6\}$, $q_4=\{k_1,k_3,k_4,k_7\}$, $q_5=\{k_3,k_4,k_6,k_7\}$, $q_6=\{k_3,k_4,k_6,k_8\}$, and $q_7=\{k_3,k_4,k_7,k_9\}$, the Algorithm OVERLAP firstly calculates sharing factors, which are shown in Figure 3(b).

In the first iteration, all the sharing factors $\{k_1,k_3,k_4\}$, $\{k_3,k_4,k_6\}$, and $\{k_3,k_4,k_7\}$ have the largest size-frequency product as 9 (see the first row in Figure 3(b)). Without loss of generality, suppose $\{k_1,k_3,k_4\}$ is selected to evaluate first. It invokes CAL to calculate $\mathbb{RC}(\{k_1,k_3,k_4\},G)$ as follows: since calculating it may require intermediate results of other sharing factors whose keywords are a subset of $\{k_1,k_3,k_4\}$, CAL recursively finds the most promising sub-sharing factors of $\{k_1,k_3,k_4\}$, i.e. $\{k_3,k_4\}$ in this case (it is also the only sub-sharing factor). After $\{k_3,k_4\}$ is processed, it goes back to the previous recursion state where $\{k_1,k_3,k_4\}$ is then processed by using the result of $\mathbb{RC}(\{k_3,k_4\},G)$. Finally CAL returns $\mathbb{RC}(\{k_1,k_3,k_4\},G)$. Then the algorithm needs to process all queries that can benefit from $\mathbb{RC}(\{k_1,k_3,k_4\},G)$ by pushing the sharing factors that are superset of $\{k_1,k_3,k_4\}$ into a heap, because these superset sharing factors can benefit from computing $\mathbb{RC}(\{k_1,k_3,k_4\},G)$, i.e., $\{k_1,k_2,k_3,k_4\}$, $\{k_1,k_3,k_4,k_5\}$, and $\{k_1,k_3,k_4,k_7\}$ are pushed into a heap. Then it processes each sharing factor s in the heap to calculate $\mathbb{RC}(s,G)$ and pushes its superset into the heap until its superset is an original query of the batch. After the first iteration, queries q_1, q_2, and q_4 have been processed. In the second iteration, there are only four queries left, which are q_3, q_5, q_6, and q_7. Their sharing factors and corresponding frequencies are shown in the second row in Figure 3(b). The algorithm chooses $\{k_3,k_4,k_6\}$ and q_3, q_4, and q_5 can be answered based on $\{k_3,k_4,k_6\}$. Finally, in the last iteration, since only q_7 left, the algorithm chooses $\{k_3,k_4,k_7\}$ to support query q_7. The blue lines in Figure 3(a) shows the final query plan by using the algorithm OVERLAP.

4. COST ESTIMATION FOR QUERY PLANS

The maximal overlapping driven approach tries to maximize the sharing of intermediate results. However, this does not mean the overall cost of the query plan is optimal. Therefore, we provide a cost-based solution to support multiple keyword query on graph, which mainly contains two parts: (i) estimating cost of a query plan, and (ii) generating global optimal plan based on the estimated cost.

In this section, we propose a cost model to estimate the query plan. The cost of a query plan is determined by the cost of involved r-join operations. Therefore, in Section 4.1 we analyse the cost of an r-join and in Section 4.2 we estimate the cardinality of an intermediate result of an r-join between two r-cliques. Finally we present our estimation cost model for a query plan.



(a) A quey plan

Iterations	Sharing factors and their frequencies
1	$freq(\{k_2,k_3,k_4\})$=2, $freq(\{k_1,k_3,k_4\})$=3, $freq(\{k_3,k_4,k_6\})$=3, $freq(\{k_3,k_4,k_7\})$=3, $freq(\{k_3,k_4\})$=4
2	$freq(\{k_3,k_4,k_6\})$=3, $freq(\{k_3,k_4,k_7\})$=2 $freq(\{k_3,k_4\})$=2
3	$\emptyset$

(b) Sharing factors and their frequencies

Figure 3: An example of processes in the algorithm OVERLAP

4.1 Cost of an r-Join

In order to estimate the size of an r-join between two r-cliques, we first illustrate our implementation of an r-join operation $\bowtie_{\mathcal{R}}$.

Given two keyword sets K_i and K_j, let $\mathbb{RC}(K_i,G)$ and $\mathbb{RC}(K_j,G)$ be two r-clique sets for K_i and K_j, respectively. Algorithm 3 shows the implementation of an r-join operation between two r-clique sets $\mathbb{RC}(K_i,G)$ and $\mathbb{RC}(K_j,G)$. For any r-clique pair $\langle rc', rc''\rangle$ ($rc' \in \mathbb{RC}(K_i,G)$ and $rc'' \in \mathbb{RC}(K_j,G)$), an r-join operation chooses vertex pairs $\langle v, v'\rangle$ from the pair $\langle rc', rc''\rangle$ such that $dist(v,v') \leq r$. In order to calculate $dist(v,v')$ efficiently, we pre-store all shortest paths between every two vertices in G into a shortest path set $SP(G)$. Then the cost of the r-join operation $o = \mathbb{RC}(K_i,G) \bowtie_{\mathcal{R}} \mathbb{RC}(K_j,G)$ is

$$cost(o) = \mathcal{O}(n_i \times n_j \times |K_i| \times |K_j|), \quad (1)$$

where n_i and n_j are numbers of r-cliques in $\mathbb{RC}(K_i,G)$ and $\mathbb{RC}(K_j,G)$, respectively. It shows that the cost of an r-join operation is determined by its inputs $\mathbb{RC}(K_i,G)$ and $\mathbb{RC}(K_j,G)$.

Algorithm 3: rJOIN

Data: Two r-cliques $\mathbb{RC}(K_i,G)$ and $\mathbb{RC}(K_j,G)$
Result: $\mathbb{RC}(K_i \cup K_j,G) \leftarrow \mathbb{RC}(K_i,G) \bowtie_{\mathcal{R}} \mathbb{RC}(K_j,G)$

```
1  RC(Ki ∪ Kj, G) ← ∅;
2  foreach rc' ∈ RC(Ki, G) do
3      foreach rc'' ∈ RC(Kj, G) do
4          FLAG← true;
5          foreach v ∈ rc' do
6              if dist(v, v') > r for any v' ∈ rc'' then
7                  FLAG← false; break;
8          if FLAG=true then
9              RC(Ki∪Kj, G) ← RC(Ki∪Kj, G)∪{rc'∪rc''};
10 return RC(Ki ∪ Kj, G);
```

4.2 Estimating Cardinality of an r-Join Result

We observe that given a query q, following the pipeline of r-join operations $\mathbb{RC}(q,G)$ can be derived by a recursive process as follows.

$$\mathbb{RC}(q,G) = \begin{cases} \mathbb{RC}(\{k\},G) \text{ if } q=\{k\}, \\ \mathbb{RC}(q\backslash\{k\},G) \bowtie_{\mathcal{R}} \mathbb{RC}(\{k\},G) \text{ if } |q\backslash\{k\}| \geq 1. \end{cases} \quad (2)$$

If q contains only one keyword k, the size $|\mathbb{RC}(q,G)|$ equals to the length of the inverted list $L(k)$. So we only need to estimate the size of $\mathbb{RC}(q,G) = \mathbb{RC}(q\backslash\{k\},G) \bowtie_{\mathcal{R}} \mathbb{RC}(\{k\},G)$. $\mathbb{RC}(q,G)$ merges vertices from $\mathbb{RC}(q\backslash\{k\},G)$ and $\mathbb{RC}(\{k\},G)$, so that for any vertex $v \in \mathbb{RC}(q\backslash\{k\},G)$ and $v' \in \mathbb{RC}(\{k\},G)$, $dist(v,v') \leq r$.

According to the r-join operation, we know that for a vertex $v \in L(k)$, v cannot contribute to the result of $\mathbb{RC}(q,G)$ if for each $v' \in \mathbb{RC}(q\backslash\{k\},G)$ their distance $dist(v,v') > r$. We call such v *invalid* vertex w.r.t. a parameter r, and we can construct a *valid* inverted list $L_v^r(k)$ such that each vertex in it is a valid vertex w.r.t. r. Given a graph G and the parameter r, we can easily construct valid inverted lists of keywords as follows. For each processing vertex $v \in V(G)$, we check every unprocessed vertex v' whose keywords $v'.KS$ do not overlap with the keywords of v. If $\forall v'$ belonging to unprocessed vertices, $dist(v,v') > r$, we say v is invalid and will not appear in any list of keywords. Then a *valid* shortest path set $SP_v^r(G)$ for all valid vertices in G can be loaded before receiving any queries. For each $v \in L_v^r(k)$, let $p_r(v)$ be probability of appearance of $v' \in \mathbb{RC}(q\backslash\{k\},G)$ such that $dist(v,v') \leq r$. Then $|\mathbb{RC}(q,G)|$ can be estimated as $p_r(v) \times |L_v^r(k)| \times |\mathbb{RC}(q\backslash\{k\},G)|$.

Estimating cardinality of an r-join between two inverted lists for keywords. We first consider the simple case where $Q = \{q_1, q_2\}$. Given a graph G, let $L_v^r(k_1)$ and $L_v^r(k_2)$ be the two valid inverted lists of keywords k_1 and k_2, respectively. Then $|\mathbb{RC}(q,G)|$ can be estimated as:

$$|\mathbb{RC}(q,G)| = p_r(v) \times |L_v^r(k_1)| \times |L_v^r(k_2)|, \quad (3)$$

where $p_r(v) = \frac{|SP_v^r(G)|}{|V(G)|^2}$, $|V(G)|^2$ is the number of shortest paths in G, and $|SP_v^r(G)|$ is the number of shortest paths in $SP_v^r(G)$. As we explained above, such statistic value $SP_v^r(G)$ can be collected offline for a given a graph G.

Estimating cardinality of an r-join between an r-clique set and an inverted list for a keyword. We use Equation 4 to iteratively estimate the number of cliques in $\mathbb{RC}(q,G)$ when q has more than two keywords.

$$|\mathbb{RC}(q,G)| = \left(\frac{|SP_v^r(G)|}{|V(G)|^2}\right)^{|q|-1} \times |\mathbb{RC}(q\backslash\{k\},G)| \times |L_v^r(k)|, \quad (4)$$

where $|q| > 2$ and $\left(\frac{|SP_v^r(G)|}{|V(G)|^2}\right)^{|q|-1}$ is the probability $p_r(v)$.

Let a query plan $p(q)$ w.r.t. a query q contains a list of r-join operations, then the final cost of $p(q)$ is $cost(P) = \sum_{o \in p(q)} cost(o)$, where $cost(o)$ is the cost of the r-join operation $o \in p(q)$ (see Equation 1).

5. ESTIMATION-BASED QUERY PLANS

Based on the estimated cost of query plans, we could find a global optimal query plan by utilizing the state-of-the-art approach A* algorithm. By using A*, we could assess generated partial plans and only expand the most promising partial plan to find the global optimal plan based on estimated cost. In Section 5.1 we show how to construct a search space for A* algorithm, and in Section 5.2 we propose pruning approaches to reduce the search space to make the search more efficient.

5.1 Finding Optimal Solution based on Estimated Cost

In this section, we adopt the solution in [21] which is based on A* algorithm to model our problem as a state space search problem.

Search space. The search space $\mathcal{S}(Q)$ for a query batch Q= $\{q_1, \ldots, q_n\}$, can be expressed as: $\mathcal{S}(Q) = \{P(q_1) \times \ldots \times P(q_n)\}$, where $P(q_i)$ $(1 \leq i \leq n)$ is a set of query plans for single keyword query q_i, each of which contains a pipeline of r-join operations. Let a global query plan for the batch query Q have the form of $\langle p_1 \in P(q_1), \ldots, p_n \in P(q_n)\rangle$, where $p_i \in P(q_i)$.

Therefore, each state s_i in the search space is an n-tuple $\langle p_{i1}, \ldots, p_{in}\rangle$, where p_{ij} is either a $\{NULL\}$ or a query plan for the i-th query $q_i \in Q$. The search space contains an initial state s_0= $\langle NULL, \ldots, NULL\rangle$ and several final states S_F where each p_{ij} in a final state $s_f \in S_F$ corresponds to a query plan for $q_i \in Q$. The value of each state $s_i = \langle p_1, \ldots, p_n\rangle$ equals to the summation of the cost of all query plans in s_i, i.e.,

$$v(s_i) = \sum_{p \in s_i, p \neq NULL} cost(p).$$

The A* algorithm starts from the initial state s_0 and finds a final state $s_f \in S_F$ such that $v(s_f)$ is minimal among all paths leading from s_0 to any final state. Obviously, $v(s_f)$ is the total cost required for processing all n queries.

In order for an A* algorithm to have fast convergence, a lower bound function $lb(s_i)$ is introduced on each state s_i. This function is used to prune down the size of the search space that will be explored. When A* starts from s_{i-1} and determines if it is worth to traverse a state s_i, it gets its lower bound $lb(s_i)$ as follows:

$$lb(s_i) = v(s_{i-1}) + pre\_cost(s_i), \tag{5}$$

where $pre\_cost(s_i)$ is the minimal optimistic approximation cost of traversing the next state s_i from s_{i-1}. That is, starting from s_{i-1}, the A* algorithm needs at least $pre\_cost(s_i)$ cost to arrive s_i, where a new query plan p' for query q_i is to be traversed. Let p' contain a set of r-joins, then $pre\_cost(s_i) = \sum_{o \in p'} \widehat{cost}(o)$, where $\widehat{cost}(o)$ is the minimal optimistic cost of the r-join. For each such an r-join, if it is shared in previous query plans in s_{i-1}, we do not need extra cost to compute it, therefore, $\widehat{cost}(o) = 0$; otherwise, suppose this r-join operation can be reused at most n_l times by remaining queries from q_i to q_n, $\widehat{cost}(o) = \frac{cost(o)}{n_l}$ (see Equation 6).

$$\widehat{cost}(o) = \begin{cases} 0 & \text{if } o \text{ is shared in } s_i, \\ \frac{cost(o)}{n_l} & \text{otherwise.} \end{cases} \tag{6}$$

The cost(o) in Equation 6 is the estimated cost of o defined in Equation 1.

According to the above analysis, we propose our algorithm EstPlan based on A* algorithm as follows. If for any state s_j $(i \neq j)$, $lb(s_i) < v(s_j)$, EstPlan continues to traverse pointed from s_i, otherwise it jumps to state s_j and continues traversing states pointed from s_j since the best global plan that is derived from s_i cannot beat that of s_j. Since the search space is tree structure and the lower bound we used is always less or equal to actual cost (assume reuse maximumly), the first global plan, generated by EstPlan that has lowest lower bound in all expanded states, is the global optimal plan based cost estimation model.

5.2 Reducing Search Space

In this section, we analyze how to reduce the search space of query plans. Recall, for a particular keyword query q in the keyword batch, we will eventually choose only one query plan to evaluate q. During the evaluation process, some plans in $P(q)$ can be found as not promising to be the chosen plan for this particular keyword query, and therefore these plans can be safely pruned. We will introduce two Theorems serving as the pruning conditions.

Theorem 1. *Let $p_i, p_j \in P(q)$ be any two query plans of the single query q. The plan p_i can be pruned, if $cost(p_i) > cost(p_j)$ and p_i does not contain a sharing factor that is not contained in p_j, i.e. $SF(p_i) \subseteq SF(p_j)$ where $SF(p)$ denotes all the sharing factors of plan p.*

Proof. Basically, Theorem 1 requires both two conditions be met to prune p_i, i.e., (a) $cost(p_i) > cost(p_j)$ and (b) $SF(p_i) \subseteq SF(p_j)$. We prove by contradiction.

Case 1: if $cost(p_i) \leq cost(p_j)$, apparently p_i is less expensive than p_j. Under the circumstance that p_j does not have shared factors with other queries in Q. Plan p_i is always better than plan p_j. As a result, p_i cannot be pruned.

Case 2: if $SF(p_i) \not\subseteq SF(p_j)$, it implies there exists one sharing factor SF ($SF \in SF(p_i)$ and $SF \notin SF(p_j)$) such that SF is shared with another query $q' \in Q, q \neq q'$. Since the computation of SF is shared between q and q', the actual $cost(p_i)$ must be less than expected and even may be smaller than $cost(p_j)$. Consequently, p_i cannot be pruned. □

To go further from the proof of Case 2 in Theorem 1, let SF be a sharing factor $SF \in SF(p_i)$ and $SF \notin SF(p_j)$, when p_i is chosen as the query plan of q and p_i is evaluated, the best case is that the intermediate result of $\mathbb{RC}(SF, G)$ has been computed in another query plan of q', and plan p_i just simply reuses the result. In such case, the actual cost of p_i is $cost(p_i) - cost(SF)$. Accordingly, we have Theorem 2 as follows:

Theorem 2. *Let $p_i, p_j \in P(q)$ be any two query plans of a single query q. Let $SF(p_i)$, $SF(p_j)$ denote the sharing factors of plan p_i, p_j respectively. The plan p_i can be pruned, if $cost(p_i) - \sum_{\forall SF \in SF(p_i) \setminus SF(p_j)} cost(SF) > cost(p_j)$.*

Proof. If plan p_i is less preferred than plan p_j, it means that $cost(p_i)$ is absolutely larger than $cost(p_j)$. This implies that even in the best case where p_i reuses as much shared computation of sharing factors as possible, p_i is still more expensive than its counterpart plan p_j. The largest possible reusable cost is $\sum_{\forall SF \in SF(p_i) \setminus SF(p_j)} cost(SF)$, which is the

maximum cost that p_i can save on the condition that the computation of the sharing factors in $SF(p_i) \backslash SF(p_j)$ has been done in other queries of the batch Q. The intuition is that, if p_i's minimal possible cost is already larger than p_j's maximal cost, p_i can be pruned safely. □

6. EXPERIMENT

In this section, we implemented *the Shortest List Eager Approach* in Algorithm 1, *the Maximal Overlapping Driven Approach* in Algorithm 2, and A* based algorithm proposed in this paper. They are briefly denoted as BASIC, OVERLAP and ESTPLAN respectively. Their performances are evaluated and compared by running them for different multiple query batches over two real datasets. All the tests were conducted on a 2.5GHz CPU and 16 GB memory PC running Ubuntu 14.04.3 LTS. All algorithms were implemented in GNU C++.

6.1 Datasets and Tested Queries

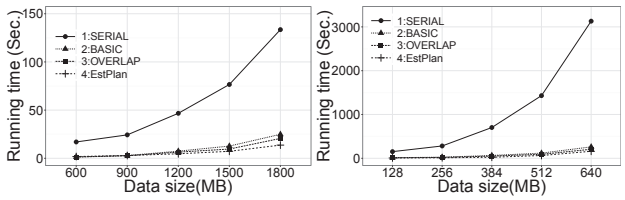
Datasets. We evaluated our algorithms on two real datasets.

- DBLP dataset[1]. We generated graphs from DBLP dataset. The generated graph contains 37,375,895 vertices and 132,563,689 edges where each vertices represents a publication and each edge represents the citation relationships over papers.
- IMDB dataset[2]. We generated graphs from a processed IMDB dataset [7]. The vertices in the generated graph represent users or movies. There are 247,753 users and 34,208 movies. The edges in the graph represent relations between the users and the movies: users may rate movies or comments on movies. In the generated graph, the total edges consists of 22,884,377 rating relations and 586,994 commenting relations.

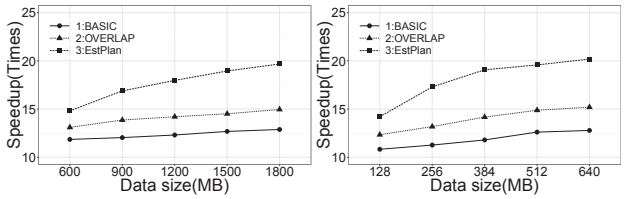
Tested Queries. For each dataset, we randomly selected 100 keywords as a keyword set used for producing tested batch queries. Table 1 shows the frequency range of the keywords in each keyword set. We created batches with different ratios of shared keywords as follows. We randomly produced 5 subsets of keywords that are picked from the 100 keywords for DBLP dataset. Each subset contains a certain number of distinct keywords, i.e., 10 distinct keywords in the 1st subset, 15 distinct keywords in the 2nd subset, 20 distinct keywords in the 3rd subset, 25 distinct keywords in the 4th subset, and 30 distinct keywords in the 5th subset. For each subset of keywords, we randomly picked 3 to 7 keywords for individual keyword queries. By repeatedly working on each subset of keywords until we generated 50 keyword queries as a query batch. We iterated the above process and generated the tested batch queries for DBLP and IMDB dataset. We use the generated query batches as experiments input and report average results. In experimental studies, we fixed the size of multiple keyword queries in a batch, say 50. Therefore, varying the number of distinct keywords of the keywords subset from which the batch is generated varies the shared computations that the query batch contains. If the batch of queries are generated from a small subset of keywords (e.g., 10 distinct keyword in the subset), the shared computation contained in the batch is high; otherwise, shared computation of the batch is low.

[1]http://dblp.uni-trier.de/xml/

[2]http://grouplens.org/datasets/movielens/



(a) DBLP (Number of dist. keywords = 20, r=3) **(b)** IMDB (Number of dist. keywords = 20, r=3)



(c) DBLP (Number of dist. keywords = 20, r=3) **(d)** IMDB (Number of dist. keywords = 20, r=3)

Figure 4: Scalability and speedup studies

6.2 Evaluation of the Efficiency

We show the computational cost with some configurations in terms of the total running time for query batches.

Parameters. Parameters that may affect the batch processing efficiency include: size of the dataset, r, and number of distinct keywords. In the following experiment configurations, the default configuration of dataset size is the full size of DBLP and IMDB, and the default configuration of r is 3, and the default number of distinct keyword in keywords subsets that generates query batches is 20.

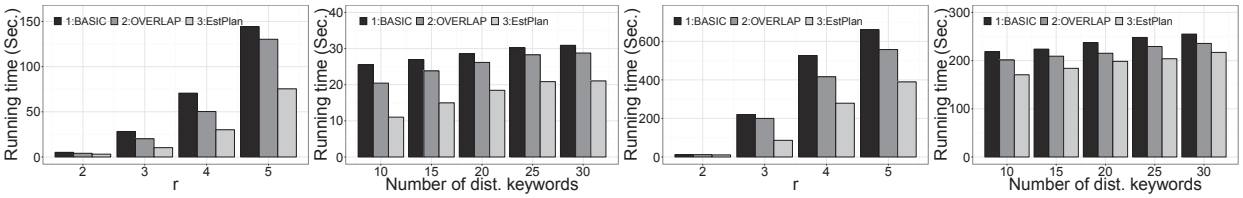
Scalability. We report the computational cost of batch processing algorithms: BASIC, OVERLAP, and ESTPLAN, when we vary the dataset size of DBLP and IMDB. To demonstrate the benefit of batch processing algorithms, we also implemented a serial query processing algorithm, denoted as SERIAL. The size of DBLP data is from 600(MB) to 1800(MB) with interval of 300(MB). The data size of IMDB is from 128(MB) to 640(MB) with interval of 128(MB). We keep the other parameters as the default values (e.g., r=3 and the number of distinct keywords is 20). Figure 4(a) and Figure 4(b) show that, the batch processing algorithms are one order of magnitude faster than SERIAL when we evaluated the batch queries on both DBLP and IMDB datasets.

Speedup. Figure 4(c) and Figure 4(d) report speedups of batch processing algorithms w.r.t. algorithm SERIAL. Overall speaking, ESTPLAN outperforms all the other algorithms on both tested datasets, having the highest average speedup. When data size is small, the speedup of ESTPLAN is relatively close to BASIC and OVERLAP. That is because the batch queries evaluation on small pieces of data is fast and the optimization overhead of ESTPLAN is non-trivial in this case. As the increase of the data size, the speedups of BASIC, OVERLAP, and ESTPLAN grow, which demonstrates the advantages of batch processing algorithms. Due to the significant speedup of batch processing algorithms, for the rest of experiments, we only focus on reporting and discussing the results of batch processing algorithms.

Varying r. We show how the running time will be changed when we vary the value of r while keeping the default set-

Table 1: Keyword sets for DBLP and IMDB

	Number of distinct keyword	Keyword frequency range
DBLP	100	0.015-0.075
IMDB	100	0.011-0.045



(a) DBLP (Number of dist. keywords = 20) **(b)** DBLP (r=3) **(c)** IMDB (Number of dist. keywords = 20) **(d)** IMDB (r=3)

Figure 5: Efficiency of multiple queries

tings for the data sizes and the number of distinct keywords. Figure 5(a) and Figure 5(c) show that, when r is no less than 3, EstPlan is much faster than all the other algorithms. When r is small, the r-clique computational cost is small, the optimization overhead of EstPlan dominates the overall running time. Because of that, we can see that EstPlan is close to than all the other batch processing algorithms when $r = 2$ in Figure 5(a). With the value of r is increased, the running time of all algorithms sharply increase. This is because as r increases, there are more results to answer the keyword queries in a batch. Noticed that the average running time for IMDB is almost 10 times higher than DBLP. This is because the average connectivity of the graph generated from IMDB is much higher than the graph from DBLP.

Varying the number of distinct keywords. We show how the running time varies with different numbers of distinct keywords contained in batch queries while r and the data sizes are set to be default values. Same as the above discussion, varying the number of distinct keywords is equivalent to approximately change the ratio of shared computation. Figure 5(b) and Figure 5(d) show that the time consumptions of all algorithms grow when the number of distinct keywords increases. This is because the increase of the number of distinct keywords may lead to less amount of shared computation that can be taken advantage by Basic, Overlap and EstPlan.

The experiments demonstrate that the reuse of shared computations in a batch improves the efficiency of computation. The EstPlan outperforms all other algorithms in most configurations.

6.3 Evaluation of Effectiveness

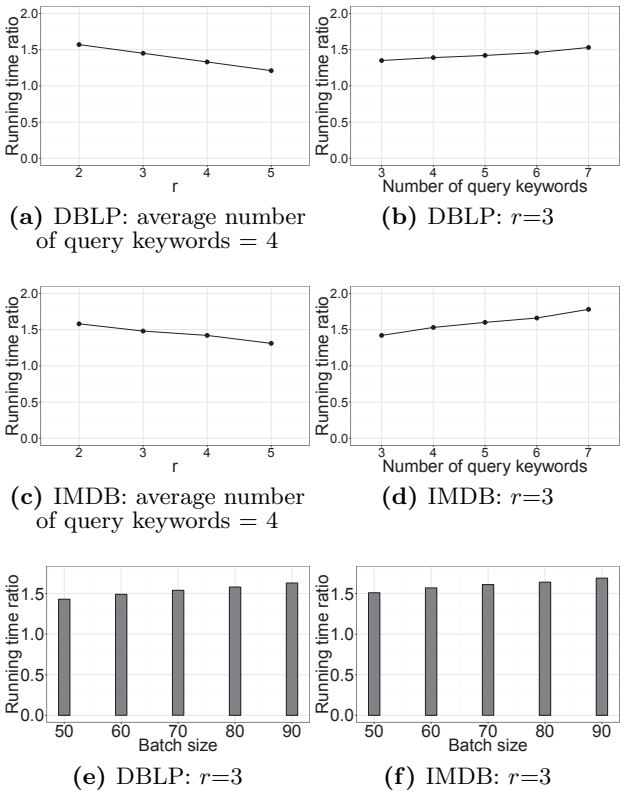
In this section, we assess the effectiveness of our proposed cardinality based computational cost estimation model and the pruning effectiveness of Theorem 1 and Theorem 2.

For a given batch of queries, EstPlan first generates a plan for the batch query based on the proposed cost estimation model and then executes the queries or sub-queries based on the generated plan. The total time is used as the exactly computational cost of the batch of queries. To measure the effectiveness of the cost model, we also need to work out the ground truth plan. We run a large number of alternative plans for the batch and select the one consuming the minimal time cost. The selected plan is treated as the ground truth plan in our experiment. As such, the effectiveness of the cost estimation can be computed by the *ratio* of the time cost of running EstPlan to the time cost of running the ground truth plan. The ratio is always no less than one. The smaller value of the ratio means the effectiveness of our cost model is higher. This is because a small ratio represents our cost model based plan to be generated can closely approach the ground truth plan. Please note that the running time has excluded the plan generation time in this section.

For an individual query, there are some factors to affect its effectiveness in terms of cost estimation: the number of keywords in a query and the value of r. This is because the cost estimation relies on cardinality estimation where the cardinality estimation depends on r and number of keywords(Section 4). In the following experimental configuration, the default number of keywords is 4 and the default value of r is 3.

Varying r. We show how the effectiveness of our cost estimation model varies with different values of r when the number of keywords in batch queries are set to be the default value. Figure 6(a) and Figure 6(c) show that, for both DBLP and IMDB, the ratio of the estimated time cost decreases when we vary the value of r from 2 to 5. As the lower ratio means higher effectiveness, the experimental study indicates that our cost estimation model based query plan can approach the ground truth plan for the query evaluation. This is because, when r is high, the results of r-cliques on the graph data generated from DBLP and IMDB tend to be Cartesian product of content vertices while the processed cardinality estimation equation follows the same tendency, which leads to more effective cost estimation.

Varying average number of query keywords in query batches. We show how the effectiveness of the cost estimation model varies when the individual queries in the batch query contain more keywords, and r is set as the default value. Figure 6(b) and Figure 6(d) show that, on both DBLP and IMDB datasets, the ratio increases with the increment of the individual query length. As we know the larger ratio means lower effectiveness of cost estimation.



(a) DBLP: average number of query keywords = 4 **(b)** DBLP: r=3

(c) IMDB: average number of query keywords = 4 **(d)** IMDB: r=3

(e) DBLP: r=3 **(f)** IMDB: r=3

Figure 6: Accuracy of cardinality estimation

The experimental study implies that the effectiveness drops as the length of individual query increases.

Varying the batch size. We also study how the effectiveness of cost estimation model varies when we change the batch size. r is set as the default value. We vary batch size from 50 to 90 with interval of 10. From the experimental study in Figure 6(e) and Figure 6(f), the ratio doesn't change a lot. The effectiveness becomes a bit low when we increase the size of query batches for both DBLP and IMDB. It says our proposed cost estimation model is stable when it is used for big batch queries or small batch queries.

Pruning effectiveness of theorems. Here we show the effect of proposed pruning methods discussed in Theorem 1 and Theorem 2. We adopt generation time of EstPlan that is used to get the optimal plan with inputs of: 1) pruned search space (denote as P) and 2) none-pruned search space (denote as NP). In Figure 7(a) and Figure 7(c), we compare their plan generation time with bathes having different amounts of shared computations. Figure 7(a) and Figure 7(c) show that, by increasing the distinct number of keywords in batches (means deceases amount of shared computations contained in a batch), the plan generation time of EstPlan with pruned search space decreases while that of none pruned search space is irrelevant to amount shared computations contained in batches. This is because the pruning effect is associated with shared r-join operations and fewer sharing would result in higher pruning effectiveness. On the other hand, for the none pruned global optimal plan searching space, its plan generation is independent with keyword set size but is relevant to keyword batch size and keyword query length. It is noticeable that the plan generation time of EstPlan with pruned search space is averagely 5 time less than the plan generation time of EstPlan with none pruned search space. Figure 7(b) and Figure 7(d) show that the pruning effectiveness in terms of the ratio of the average number of global plans in pruned space and the average number of global plans contained in none pruned search space. Obviously, the higher ratio will lead to the better pruning effectiveness. On both datasets, the ratio increases with the more distinct keywords in a batch (means deceases amount of shared computation contained in a batch), which represents the better pruning effectiveness. It is noticeable that the pruning effectiveness is over 0.75 on average for both DBLP and IMDB datasets.

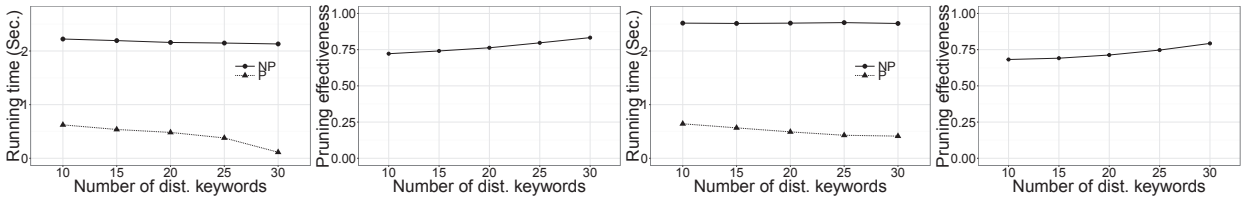
7. RELATED WORK

We classify related works into two groups: (1) single keyword query evaluation on graph data and (2) multiple query (including SQL query, XML query and keyword query) optimization in databases.

Single keyword query evaluation on graph data. The existing approaches aim at finding either steiner tree based answers or subgraph based answers. Steiner tree based answers [2, 8, 11] generates trees that cover all the search keywords and the weight of a result tree is defined as the total weight of the edges in the tree. Under this semantics, finding the result tree with the smallest weight is a well-known NP-complete problem. The graph-based methods generate subgraphs such as r-radius graph [14], r-community [17] and r-cliques [12]. In an r-radius graph, there exists a central node that can reach all the nodes containing search keywords whose distance is less than r. In an r-community, there are some center nodes. There exists at least one path between each center node and each content node such that the distance is less than r. Different from r-radius and r-community, the r-clique semantics studied in this paper is more compact and does not required the existence of a central node. It refers to a set of graph nodes which contain search keywords, and between any two nodes that contain keywords, we can find a path with a distance less than r.

Nevertheless, all of the existing works focus on single query processing not multiple query processing. Although, the proposed indices in [8, 12, 14, 17] can be used for more than one query, but evaluation techniques are exclusively designed for one query a time. In other words, none of the existing works have studied utilizing possible reusable computations over multiple keyword queries and leveraging shared computations to improve the performance for multiple keyword query evaluation at run time. In this paper, we have studied these problems.

Multiple query optimization in database. On relational data, multiple SQL query optimization have been studied in the early works [18, 20, 21], in which the main focus is to smartly handle shared operations among SQL queries. These works decompose complex SQL queries into subqueries and consider reusing common subqueries based on cost analysis. On XML data, Yao et al [23] proposed a log based query evaluation algorithm to find the optimal plan to compute multiple keyword queries under SLCA semantics [22]. Recently, multiple keyword query optimization over relational databases (rather than native graphs) has also been studied [10]. This work assumes all the keyword queries haven been transformed into candidate net-



(a) Running time comparison (DBLP) **(b)** Pruning effictivenss (DBLP) **(c)** Running time comparison (IMDB) **(d)** Pruning effictivenss (IMDB)

Figure 7: Pruning effectiveness

works (which are similar to SQL query plans), and then multiple SQL query optimization techniques are used thereafter, i.e., common SQL query operations (or subqueries) in the candidate networks are considered. Different from the above works [10, 18, 20, 21, 23], our problem focuses on native graph data where data does not have to be stored in relational tables and query results are modeled using r-clique semantics. The solution of the previous works cannot be applied to our problem, because both the data model and the query semantics are different.

8. CONCLUSION

In this paper, we have studied the new problem, batch keyword query processing on native graph data. r-clique is used as keyword query result semantics. We developed two heuristic algorithms to heuristically find good query plans, which are based on reusing shared computations between multiple queries, i.e. shortest list eager approach (reuse if possible), maximal overlapping driven approach (reuse as much as possible). To optimally run the batched queries, we devised an estimation-based cost model to assess the computational cost of possible sub-queries, which is then used to identify the optimal plan of the batch query evaluation. We have conducted extensive experiments to test the performance of the three algorithms on DBLP and IMDB datasets. Cost estimation based approach has been identified as the ideal solution.

Acknowledgments. This work is jointly supported by the grants of Australian Research Council Discovery Projects DP160102412, DP140103499 and DP160102114, the NSF of China for Outstanding Young Scholars under grant No. 61322208, the NSF of China under grant Nos. 61272178 and 61572122, and the NSF of China for Key Program under grant No. 61532021.

9. REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: a system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.

[3] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation-vs. index-based XML multi-query processing. In *ICDE*, pages 139–150, 2003.

[4] Xin Cao, Gao Cong, Christian S Jensen, and Beng Chin Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.

[5] Farhana M Choudhury, J Shane Culpepper, and Timos Sellis. Batch processing of top-k spatial-textual queries. In *2nd Intl. ACM Workshop on Managing and Mining Enriched Geo-Spatial Data*, pages 7–12, 2015.

[6] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[7] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[8] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[9] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[10] Marie Jacob and Zachary Ives. Sharing work in keyword search over databases. In *SIGMOD*, pages 577–588, 2011.

[11] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[12] Mehdi Kargar and Aijun An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[13] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for SPARQL. In *ICDE*, pages 666–677, 2012.

[14] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[15] Jianxin Li, Chengfei Liu, and Md Saiful Islam. Keyword-based correlated network computation over large social media. In *ICDE*, pages 268–279, 2014.

[16] Ziyang Liu and Yi Chen. Identifying meaningful return information for XML keyword search. In *SIGMOD*, pages 329–340, 2007.

[17] Lu Qin, J.X. Yu, Lijun Chang, and Yufei Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.

[18] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, 29(2):249–260, 2000.

[19] Timos Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Trans. Know Data Eng.*, 2(2):262–266, Jun 1990.

[20] Timos K Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[21] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12(2):197–222, 1994.

[22] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, pages 527–538, 2005.

[23] Liang Yao, Chengfei Liu, Jianxin Li, and Rui Zhou. Efficient computation of multiple XML keyword queries. In *WISE*, pages 368–381. Springer, 2013.