2019 IEEE 35th International Conference on Data Engineering (ICDE)

BENU: Distributed Subgraph Enumeration with Backtracking-based Framework

Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, Yihua Huang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
wangzhaokang@smail.nju.edu.cn, gurong@nju.edu.cn, weiweihu@smail.nju.edu.cn, {cfyuan,yhuang}@nju.edu.cn

***Abstract*—Given a small pattern graph and a large data graph, the task of subgraph enumeration is to find all the subgraphs of the data graph that are isomorphic to the pattern graph. The state-of-the-art distributed algorithms like SEED and CBF turn subgraph enumeration into a distributed multi-way join problem. They are inefficient in communication as they have to shuffle partial matching results that are much larger than the data graph itself during the join. They also spend non-trivial costs on constructing indexes for data graphs. Different from those join-based algorithms, we develop a new backtracking-based framework *BENU* for distributed subgraph enumeration. BENU divides a subgraph enumeration task into a group of local search tasks that can be executed in parallel. Each local search task follows a backtracking-based execution plan to enumerate subgraphs. The data graph is stored in a distributed database and is queried as needed. BENU only queries the necessary edges of the data graph and avoids shuffling partial matching results. We also develop an efficient implementation for BENU. We set up an in-memory database cache on each machine. Taking advantage of the inter-task and intra-task locality, the cache significantly reduces the communication cost with controllable memory usage. We conduct extensive experiments to evaluate the performance of BENU. The results show that BENU is scalable and outperforms the state-of-the-art methods by up to an order of magnitude.**

***Keywords*-subgraph isomorphism; subgraph matching; task parallel; backtracking;**

I. Introduction

Given an undirected and unlabeled data graph G and a pattern graph P, subgraph enumeration is to find all the subgraph instances of G that are isomorphic to P, i.e. the matching results. Subgraph enumeration is a fundamental problem in graph analysis and is widely used in many applications, including network motif mining [1] , graphlet-based network comparison [2] , network evolution analysis [3], and social network recommendation [4].

A. Motivation

Enumerating instances of a pattern graph in a large data graph is challenging due to two difficulties: First, the core operation of subgraph enumeration, i.e. subgraph isomorphism, is NP-complete which brings high computational complexity to the problem. Second, the sizes of the partial and final matching results can be much larger than the data graph itself [5] [6]. The performance of the single-machine in-memory algorithms [7] [8] and out-of-core algorithm [9] is limited by the computing power of a single machine. The emerging need for processing large data graphs inspires researchers designing efficient distributed subgraph enumeration methods. Based on whether the intermediate results are shuffled, the existing distributed algorithms can be divided into two groups: DFS-style and BFS-style.

The DFS-style algorithms do not shuffle intermediate results. They shuffle the data graph. QFrag [10] broadcasts the data graph and enumerates subgraphs in memory in parallel. The memory space of a machine limits its scalability. Afrati et al. [11] adopt the one-round multi-way join on MapReduce to enumerate, but it cannot scale to complex pattern graphs due to large replication of edges, empirically performing worse than the BFS-style algorithm [12].

The BFS-style algorithms decompose the pattern graph recursively into a series of join units. They enumerate the matching results of the join units first and assemble them via one or more rounds of join to get the matching results for the whole pattern graph. The partial matching results (i.e. intermediate results) are shuffled during the join. Varieties of join units (Edge [13], Star [5] [14], TwinTwig [12], Clique [5] and Crystal [6]) and join frameworks (Left-deep join [12], Bushy join [5], Hash-assembly [6] and Generic join [13]) are proposed to reduce sizes of intermediate results.

However, the BFS-style algorithms are still costly. *First*, shuffling partial matching results is inevitable in the join-based framework, causing high communication costs. Table I shows the numbers of matches of some typical pattern graphs in real-world data graphs. Those pattern graphs are core structures of many complex pattern graphs as shown in Fig. 6. The matching results of the core structures are already 10 to 100× larger than the data graphs. Shuffling them will cause a lot of communication. *Second*, the cutting-edge algorithms SEED [5] and CBF [6] use extra index structures like SCP index (in SEED) or clique index (in CBF) for each data graph to achieve high performance. The index requires not only non-trivial computation costs to construct but also disk space to store. It requires extra costs to maintain if the data graph is updated, which is common in industry.

Can we overcome those drawbacks and design a distributed subgraph enumeration algorithm that avoids shuffling partial matching results, does not use extra index, and can handle large data graphs and complex pattern graphs?

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00021

Table I
NUMBERS OF MATCHES OF TYPICAL PATTERN GRAPHS IN REAL-WORLD DATA GRAPHS

Data Graph	$\|\mathbf{V}\|$	$\|\mathbf{E}\|$	△	⊠	⧅
as-Skitter (as)	1.7E6	1.1E7	2.9E7	1.5E8	2.0E9
LiveJournal (lj)	4.8E6	4.3E7	2.9E8	9.9E9	7.6E10
Orkut (ok)	3.1E6	1.2E8	6.3E8	3.2E9	6.7E10
uk-2002 (uk)	1.8E7	2.6E8	4.4E9	1.6E11	2.7E12
FriendSter (fs)	6.5E7	1.8E9	4.2E9	9.0E9	1.8E11

B. Contributions

Our answer is a new distributed **B**acktracking-based subgraph **ENU**meration framework *BENU*. BENU is a DFS-style framework that can be implemented with a distributed computing platform that supports task-parallel paradigm (like Hadoop) and a distributed key-value database (like HBase). We make the following contributions in this paper.

First, we propose a distributed subgraph enumeration framework BENU that is built upon the *on-demand "shuffle"* technique. BENU divides a subgraph enumeration task into a group of local search tasks. The search tasks enumerate matches of the pattern graph in parallel, following a backtracking-based execution plan. Unlike the existing DFS-style algorithms that shuffle the data graph before enumeration in a one-round manner, BENU stores adjacency sets of the data graph in a distributed key-value database and queries the database *as needed* during enumeration. It helps BENU only query (i.e. "shuffle") necessary parts of the data graph. Except for the data graph, BENU does not shuffle any partial matching result or rely on any index.

Second, we propose a search-based method to generate the best execution plan with the least cost for BENU. The method includes a series of execution plan optimization techniques, a cost estimation model and two pruning techniques.

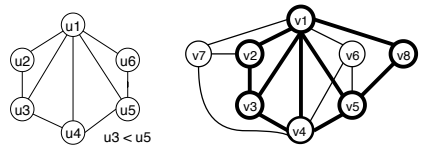
Third, we propose two efficient implementation techniques for BENU. We set up an in-memory database cache on each machine to store the adjacency sets fetched from the distributed database. Straightforward but effective, the cache can make use of the inter-task and intra-task locality hidden in the execution plan to significantly reduce the communication cost. We also propose the task splitting technique to handle the skewed workloads of local search tasks brought by the power-law degree distribution in real-world data graphs.

Fourth, we conduct extensive experiments to evaluate the performance of BENU. The experimental results validate the efficiency and scalability of BENU. BENU outperforms the state-of-the-art methods by up to an order of magnitude, particular on complex pattern graphs.

II. PRELIMINARIES

A. Problem Definition

In this paper, we focus on undirected and unlabeled simple graphs. We define a graph g as $g = (V(g), E(g))$ where $V(g)$ and $E(g)$ are the vertex set and edge set of g respectively. For a vertex $v \in V(g)$, we denote its adjacency set as $\Gamma_g(v) = \{w|(w, v) \in E(g)\}$. The degree of v is $d_g(v) = |\Gamma_g(v)|$. A subgraph g' of g is a graph such that $V(g') \subseteq V(g)$ and $E(g') \subseteq E(g)$. An *induced subgraph* $g(V')$ of a graph g on a vertex set V' is defined as $g(V') = (V' \cap V(g), \{(u, v) \in E(g)|u, v \in V'\})$.



(a) Pattern graph P (b) Data graph G

Figure 1. Demo graphs.

The subgraph enumeration involves two graphs: a data graph G and a pattern graph P. Let $N = |V(G)|$, $M = |E(G)|$, $n = |V(P)|$ and $m = |E(P)|$. The pattern graph P is usually much smaller than G, i.e. $n \ll N, m \ll M$. We assume P is connected as we can decompose a disconnected pattern graph into connected components and enumerate subgraphs for each component separately. We use v_i/u_i to denote a vertex from the data/pattern graph respectively. Without loss of generality, we assume that vertices in G and P are consecutively numbered, i.e. $V(G) = \{v_1, v_2, \ldots, v_N\}$ and $V(P) = \{u_1, u_2, \ldots, u_n\}$. We follow the definitions in [5] to define the problem.

Definition 1 (Match)**.** *Given a pattern graph P and a data graph G, a mapping $f : V(P) \to V(G)$ is a* match *of P in G if f is injective and $\forall x \forall y((x, y) \in E(P) \to (f(x), f(y)) \in E(G))$. A match f is denoted as $f = (f_1, f_2, ..., f_n)$ where $f_i = f(u_i)$ for $1 \leq i \leq n$.*

Definition 2. *Given a pattern graph P and a data graph G, a subgraph g of G is isomorphic to P if and only if there exists a match f of P in g, $|V(P)| = |V(g)|$, and $|E(P)| = |E(g)|$. The task of* subgraph enumeration *is to enumerate all subgraphs of G that are isomorphic to P.*

A match f of P in G corresponds to a subgraph g isomorphic to P in G. However, multiple matches may correspond to the same subgraph due to the automorphisms in P. Taking the demo case in Fig. 1 as the example, the match $f' = (v_1, v_2, v_3, v_4, v_5, v_8)$ and $f'' = (v_1, v_8, v_5, v_4, v_3, v_2)$ both correspond to the subgraph g shown with the bold lines in Fig. 1b. Enumerating all matches of P in G may report duplicate subgraphs.

We incorporate the *symmetry breaking* technique [15] to avoid the duplication. The symmetry breaking technique requires a total order $\prec$ predefined on $V(G)$. It calculates and imposes a partial order $<$ on $V(P)$ to break the automorphisms in P. The technique redefines a match f of P in G as a mapping satisfying both Definition 1 and the partial order constraints: if $u_i < u_j$ in $V(P)$, then

Table II
NOTATIONS.

Symbol	Description
G, N, M	The data graph G. $N = \|V(G)\|$, $M = \|E(G)\|$.
P, n, m	The pattern graph P. $n = \|V(P)\|$, $m = \|E(P)\|$.
u, u_i	An arbitrary/The i-th vertex in P.
v, v_i	An arbitrary/The i-th vertex in G.
$\Gamma_g(x)$	The adjacency set of the vertex x in the graph g.
$d_g(x)$	The degree of the vertex x. $d_g(x) = \|\Gamma_g(x)\|$.
$f = (f_1, ..., f_n)$	A match f of P in G. $f_i = f(u_i)$.
T_i	A temporary set in the execution plan.
C_i	The candidate set for the pattern vertex u_i.
A_i	The adjacency set of f_i in G.
P_i	The partial pattern graph induced on the set of the first i vertices in the matching order O.

Algorithm 1 Backtracking-based Framework

Input: Pattern graph P, Matching order O, Data graph G.

```
1: f ← an empty mapping from V(P) to V(G);
2: u_i ← FirstPatternVertexToMatch(O);
3: for all v_j ∈ V(G) do
4:     f_i ← v_j;
5:     SUBGRAPHSEARCH(P, G, O, f);
6: procedure SUBGRAPHSEARCH(P, G, O, f)
7:     if all the pattern vertices are mapped in f then output f;
8:     else
9:         u_i ← NextPatternVertexToMatch(O, f);
10:        C_i ← RefineCandidates(P, G, f, u_i);
11:        for all v_k ∈ C_i do
12:            f_i ← v_k;
13:            SUBGRAPHSEARCH(P, G, O, f);
14:            f_i ← NULL;          ▷ Make u_i unmapped in f
```

$f(u_i) \prec f(u_j)$ in $V(G)$. Under this new definition, if a subgraph g is isomorphic to P, there is one and only one match f of P in g [15]. In Fig. 1, the partial order imposed by the symmetry breaking technique on P is $u_3 < u_5$. Assuming $v_3 \prec v_5$ in the total order, then the subgraph g shown with bold lines in G is isomorphic to P with only one match $f' = (v_1, v_2, v_3, v_4, v_5, v_8)$.

The symmetry breaking technique establishes a bijective mapping between matches f of P in G and subgraphs g isomorphic to P in G. Therefore, enumerating all subgraphs is equivalent to enumerating all matches with the partial order constraints. In this paper, we use the technique to convert the problem of subgraph enumeration into match enumeration. We use the same total order $\prec$ defined in [5].

The frequently used symbols are summarized in Table II.

B. Backtracking-based Framework

The backtracking-based framework is a popular framework among serial subgraph isomorphism algorithms. The framework incrementally maps each pattern vertex to data vertices in the match f according to a given matching order.

Algorithm 1 shows a simplified version of the original framework [16]. The `SubgraphSearch` procedure finds all the matches of P in G. The *NextPatternVertexToMatch* function finds the next unmapped pattern vertex u_i according to the matching order. The *RefineCandidates* function calculates a candidate data vertex set C_i that we can map u_i to. Mapping u_i to any data vertex in C_i should not break the match conditions in Definition 1 and the partial order constraints. The framework recursively calls `SubgraphSearch` until all the vertices are mapped in f.

Different algorithms have different implementations for the three core functions: *FirstPatternVertexToMatch*, *NextPatternVertexToMatch* and *RefineCandidates*.

III. BENU FRAMEWORK

In this paper, we consider the shared-nothing cluster as the target distributed environment. Each machine in the cluster has a limited memory that may be smaller than the data graph. The approaches like [10] that load the whole data graph in memory are not feasible here.

A. Framework Overview

The DFS-style method [11] is not efficient because of its one-round shuffle design. It blindly shuffles edges before enumeration and cannot exploit the information of partial matching results. Considering a special case where the data graph has no triangle but the pattern graph has, a more efficient way than one-round shuffle is to enumerate triangles first and stop immediately after finding there is no triangle.

It inspires us to propose the *on-demand shuffle* technique. The main idea is to store the edges of the data graph in a distributed database and query ("shuffle") the edges as needed during enumeration. The technique follows the backtracking-based framework to enumerate matches. Only when the framework needs to access the data graph in the *RefineCandidates* function, does it query the database. Once a partial match f fails in the search (i.e. generating an empty candidate set for a pattern vertex), the framework skips f and backtracks, not wasting any effort on mapping other pattern vertices in f. Thus, the technique avoids querying useless edges. It also avoids shuffling partial matching results.

Based on the on-demand shuffle technique, we develop the BENU framework for distributed subgraph enumeration. Algorithm 2 shows the workflow of BENU and Fig. 2 presents its architecture. The data graph is stored in a distributed database. Given a pattern graph P, the best execution plan E to enumerate P is computed on the master node (line 2) and broadcasted to worker machines (line 3). Execution plan is a core concept in BENU. It gives out the detailed steps to enumerate matches. We elaborate on it in the next subsection. BENU parallelizes the outer loop (line 4) and generates a local search task for each data vertex in $V(G)$. BENU executes the tasks in parallel on a distributed computing platform. The local search task enumerates the matches of P in G guided by the execution plan E. During the enumeration, the database is queried as needed.

B. Execution Plan

The execution plan gives out the detailed steps to enumerate matches of P in G. In this paper, we design a new

Algorithm 2 BENU Framework

Input: Pattern graph P, Data graph G, Database DB

1: Store G in DB; ▷ Preprocessing independent of P
2: $E \leftarrow$ GenerateBestExecutionPlan(P);
3: Broadcast P and E to worker machines;
4: **for** $start \in V(G)$ **do** *in parallel*
5: $f \leftarrow$ an empty mapping from $V(P)$ to $V(G)$;
6: $u_j \leftarrow$ FirstPatternVertexToMatch(E);
7: $f_j \leftarrow start$;
8: Match the remaining pattern vertices in f guided by E;

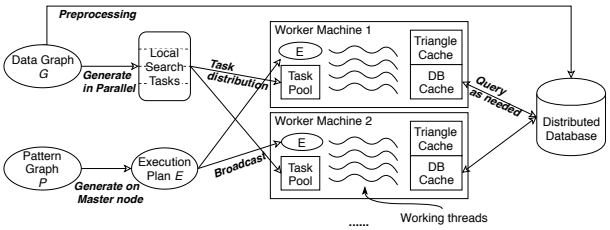


Figure 2. Architecture of BENU.

kind of execution plan, i.e. BENU execution plan, specially optimized for the architecture of BENU.

Since the database querying operation is expensive due to its high latency, the BENU execution plan queries the database on the level of adjacency sets instead of edges to reduce the number of the operations. The BENU execution plan implements the three core functions in Algorithm 1 as:

First(Next)PatternVertexToMatch: Each BENU execution plan is bound with a static matching order O. The function returns the first pattern vertex in O that is unmapped in the partial match f as the first/next vertex to match.

RefineCandidates: The BENU execution plan uses the set intersection operation to calculate the candidate set C_j for an unmapped pattern vertex u_j in f. If none of u_j's neighbors in P is mapped in f, $C_j = V(G)$. Otherwise, suppose $\{u_{i_1}, \ldots, u_{i_k}\}$ are adjacent to u_j in P and already mapped in f. The data vertices that u_j can map to should be adjacent to $\{f_{i_1}, \ldots, f_{i_k}\}$ in G, i.e. $C_j = \bigcap_{u_i \in \Gamma_P(u_j), f_i \text{ is mapped in } f} \Gamma_G(f_i)$. Mapping u_j to any vertex outside C_j will violate the match condition $(u_j, u_{i_x}) \in E(P) \rightarrow (f_j, f_{i_x}) \in E(G)$ for $1 \leq x \leq k$. C_j is further filtered to ensure that the data vertices in it do not violate the injective condition and partial order constraints. Taking the case in Fig. 1 as the example, if the partial match $f = (v_1, v_2, ?, ?, ?, ?)$, the candidate set C_3 for u_3 is $C_3 = \{v | v \in \Gamma_G(v_1) \cap \Gamma_G(v_2), v \neq v_1, v \neq v2\} = \{v_3, v_7\}$.

Integrating the core functions, we can get a full BENU execution plan. A demo execution plan for the pattern graph in Fig. 1a is shown in Fig. 3a. The *Filter* operation filters out the data vertices not satisfying either the injective condition or the partial order constraints. The execution plan is bound with the matching order $u_1, u_3, u_5, u_2, u_6, u_4$, expressed by the order of loop variables. Each loop corresponds to a recursive search level in line 11 to 14 in Algorithm 1. For ease of presentation, the recursion is expanded.

A BENU execution plan guarantees to find all valid matches of P in G without missing or duplication. On the one hand, the BENU execution plan guarantees that a found match f satisfies the match condition and the partial order constraints. And It outputs f only once. On the other hand, the execution plan does not miss any valid match f. If f is missed, there must be a data vertex v that is mapped to f_i in f not included in the candidate set C_i in the search. According to the method to generate C_i, mapping u_i to any vertex outside C_i will make f violate some matching conditions or partial order constraints. It is contrary to the hypothesis that f is valid. Hence, a BENU execution plan can enumerate all valid matches of P in G, i.e. solving the subgraph enumeration problem.

With different matching orders, a pattern graph may have multiple legal execution plans. We introduce how to get the best concrete execution plan in the next section.

IV. Execution Plan Generation

In this section, we present the method to generate a concrete BENU execution plan for a given pattern graph P. We first introduce how to generate an execution plan from a given matching order O in Section IV-A and optimize it in Section IV-B. We define the cost of an execution plan in Section IV-C. Finally, we present how to find the best matching order for P that generates the execution plan with the least cost in Section IV-D. We use the same running example through the whole section: The pattern graph is Fig. 1a and the matching order is $O : u_1, u_3, u_5, u_2, u_6, u_4$.

A. Raw Execution Plan Generation

Given a matching order O: $u_{k_1}, u_{k_2}, \ldots, u_{k_n}$, the raw execution plan consists of a series of execution instructions.

Execution Instruction. An execution instruction is denoted as $X := Operation(Operands)[|FCs]$. It contains three parts: (1) a target variable X that stores the result of the instruction, (2) an operation $Operation(Operands)$ describing the conducted operation and its operands, and (3) filtering conditions FCs that are optional. There are 6 kinds of operations in the BENU execution plan (Table III).

BENU uses two kinds of filtering conditions to guarantee the correctness: (1) A symmetry breaking condition denoted as $\succ f_i$ or $\prec f_i$ means that the vertices in the result set should be bigger or smaller than f_i under the total order $\prec$ on $V(G)$; (2) An injective condition denoted as $\neq f_i$ means that f_i should be excluded from the result set.

Instruction Generation. We generate instructions for each pattern vertex successively according to O . We first generate two instructions for the first vertex u_{k_1} in O: $f_{k_1} := Init(start)$ and $A_{k_1} := GetAdj(f_{k_1})$. The two instructions prepare related variables for u_{k_1}. For each of the remaining vertices u_{k_i} in O ($2 \leq i \leq n$), we generate the following instructions in sequence:

Table III
TYPES OF EXECUTION INSTRUCTIONS

Type	Instruction Operation	Meaning
Initialization instruction (**INI**)	$f_i := Init(start)$	Map u_i to the start vertex in the partial match f.
Database querying instruction (**DBQ**)	$A_i := GetAdj(f_i)$	Get the adjacency set of the data vertex f_i from the database.
Set intersection instruction (**INT**)	$X := Intersect(\dots)$	Intersect the operands and store the result set in X.
Enumeration instruction (**ENU**)	$f_i := Foreach(X)$	Map u_i to the vertices in X one by one in the partial match f and enter the next level in the backtracking search.
Triangle caching instruction (**TRC**)	$X := TCache(f_i, f_j, A_i, A_j)$	Triangle enumeration calculation with triangle cache.
Result reporting instruction (**RES**)	$f := ReportMatch(f_1, f_2, ...)$	Successfully find a match f of P in G that maps u_i to f_i.

1) $T_{k_i} := Intersect(\dots)$. This INT instruction calculates the raw candidate set for u_{k_i} by intersecting related adjacency sets. For any u_j that is before u_{k_i} in O and adjacent to u_{k_i} in P, we add f_j's adjacency set A_j as an operand of the instruction. If u_{k_i} is not adjacent to any vertex before it in O, we add $V(G)$ as the operand.

2) $C_{k_i} := Intersect(T_{k_i})[|FCs]$. This INT instruction calculates the refined candidate set for u_{k_i} by applying the filtering conditions. For any u_j before u_{k_i} in O, if u_j and u_{k_i} have a partial order constraint, a symmetry breaking condition is added. If u_j and u_{k_i} are not adjacent in P, an injective condition $\neq f_j$ is added. Otherwise, the injective condition is omitted as $T_{k_i} \subseteq A_j \wedge f_j \notin A_j \rightarrow f_j \notin T_{k_i}$.

3) $f_{k_i} := Foreach(C_{k_i})$. This ENU instruction maps u_{k_i} to the vertices in C_{k_i} one by one in the partial match f, and enters the next level in the backtracking search.

4) $A_{k_i} := GetAdj(f_{k_i})$. If there is any vertex u_j that is adjacent to u_{k_i} in P and is after u_{k_i} in O, A_{k_i} will be used by a subsequent INT instruction to calculate the raw candidate set for u_j. In this case, we add a DBQ instruction to fetch A_{k_i}. Otherwise, we skip the instruction as A_{k_i} will not be used by any subsequent INT instruction.

Finally, we add the RES instruction to the execution plan.

After generating instructions, we conduct the *uni-operand elimination*: If an INT instruction has one operand and no filtering condition like $T_i := Intersect(X)$, we remove the instruction and replace T_i with X in the other instructions.

Example. Fig. 3b shows the raw execution plan generated for the running example. Taking u_4 as the example, the generated instructions for u_4 are the 15th to 17th instruction.

The generated execution plan is well-defined, i.e. all the variables are defined before used. It materializes the abstract BENU execution plan as shown in Fig. 3a and can be converted to the actual code easily.

The time complexity of instruction generation is $\mathcal{O}(n^2)$ dominated by the operation of adding injective conditions.

BENU supports integrating other filtering techniques like degree filter by adding corresponding filtering conditions. In practice, adding filtering conditions to the instructions nested by many ENU instructions should be very careful, since they may bring considerable overhead.

B. Execution Plan Optimization

Though the generated raw execution is functional, it contains redundant computation. We propose three optimization techniques to reduce it.

Optimization 1 (Common subexpression elimination). We borrow the concept of common subexpression from programming analysis. Some combinations of the adjacency sets appear as operands in more than one INT instructions. They are common subexpressions in the execution plan. For example, in the raw execution plan in Fig. 3b, $\{A_1, A_3\}$ is a common subexpression. The common subexpressions should be eliminated as they bring redundant computation.

We use a frequent-item mining algorithm like Apriori to find all the common subexpressions with at least two adjacency sets. We pick the subexpression with the most adjacency sets to eliminate. If two subexpressions have the same number of adjacency sets, we pick the more frequent one according to their appearances. If they further have the same frequency, we pick the one appearing first. We add an INT instruction $T_j := Intersect(Subexpression)$ before the first instruction that the subexpression appears. Here, j is an unused variable index. Then we replace the subexpression appeared in other INT instructions with T_j to eliminate the redundancy. We repeatedly eliminate the common subexpressions until there is no common subexpression. Finally, we conduct the uni-operand elimination.

Example. In the demo raw execution plan in Fig. 3b, $\{A_1, A_3\}$ and $\{A_1, A_5\}$ are both common subexpressions. According to the order, we pick $\{A_1, A_3\}$ to eliminate. After replacing $\{A_1, A_3\}$ with T_7 in Fig. 3c, there is no other common subexpression and the optimization stops.

Optimization 2 (Instruction reordering). The position of an instruction in the execution plan significantly affects the performance. If an instruction can be moved forward and nested by fewer ENU instructions, it will be executed many fewer times. As long as the correctness is not violated, we should move such instructions as forward as possible. We reorder instructions in an execution plan in three steps:

1) Flattening INT instructions. For an INT instruction having more than two operands, we sort its operands according to their definition positions. The operand defined earlier is put in the front. Then we flatten the instruction into a series of INT instructions with at most two operands. For example, $T_j := Intersect(A, B, C)$ can be flattened into two INT instructions $T_{j'} := Intersect(A, B)$ and $T_j := Intersect(T_{j'}, C)$ where j' is an unused variable index. Flattening INT instructions does not affect the correctness of the execution plan but enables us to reorder set intersection

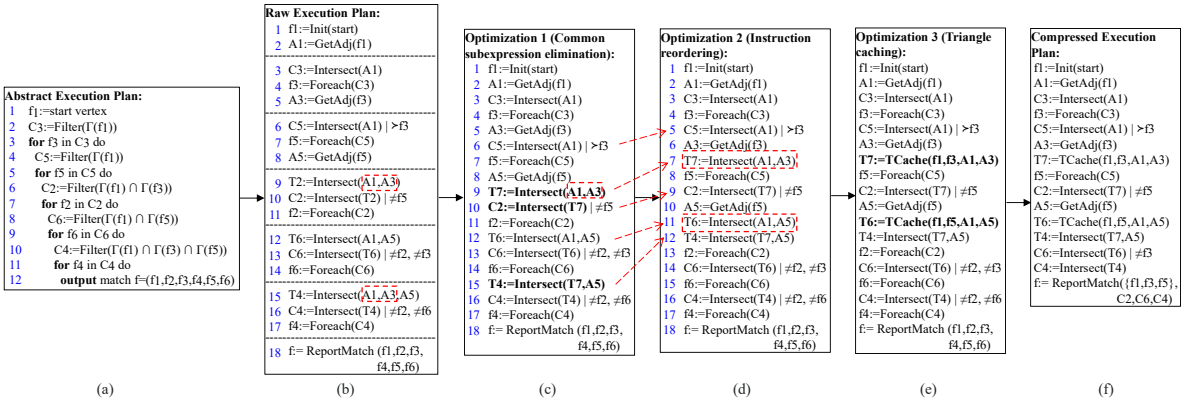


Figure 3. Demo execution plans with optimizations.

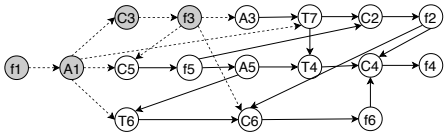


Figure 4. Dependency graph of the demo execution plan. We use the target variable to represent each instruction. The RES instruction is omitted.

operations in a finer granularity.

2) Constructing dependency graph. The instructions in an execution plan have dependency relations among them. For two instructions I_1 and I_2, if I_2 uses the target variable of I_1 in its operands or filtering conditions, then I_1 and I_2 have a dependency relation $I_1 \rightarrow I_2$. If $I_1 \rightarrow I_2$, I_1 should always be before I_2, otherwise I_2 will use an undefined variable. We construct a dependency graph to describe the dependency relations. In the dependency graph, the instructions are the vertices and the dependency relations between them are the directed edges. The dependency graph of the execution plan in Fig. 3c is illustrated in Fig. 4.

3) Reordering instructions. We reorder the instructions by conducting topological sort on the dependency graph. The topological sort can keep the dependency relations.

During the sort, it is common that several instructions can all be the candidate instructions for the next instruction. For example, in Fig. 4, after determining the first four instructions $[f_1, A_1, C_3, f_3]$, both A_3 and C_5 can be the next instruction under the topological order. At this time, we rank the candidate instructions in an ascending order based on their instruction types: INI $<$ INT $<$ TRC $<$ DBQ $<$ ENU $<$ RES. If two candidate instructions have the same type, the instruction in the front ranks higher. This order guarantees that the INI and RES instructions must be the first and last instructions. The order of the other instructions is defined based on their execution costs. The INT instructions are the cheapest as they only involve pure computation. Moreover, if we can detect failed INT instructions that generate empty result sets earlier, we can stop the framework from finishing a doomed-to-fail partial match. The TRC instructions involve cache accessing. The DBQ instructions involve expensive database operations. The ENU instructions are the most expensive as they add a level in the backtracking search and make the following instructions executed for more times. We want to postpone them as much as possible. The rank guarantees that the relative order of DBQ and ENU instructions that reflects the matching order is not changed.

Example. For the execution plan in Fig. 3c with its dependency graph in Fig. 4, we can get a reordered execution plan in Fig. 3d. The 15th instruction in Fig. 3c is moved forward crossing the ENU instructions of f_2 and f_6.

Optimization 3 (Triangle caching). Suppose u_{k_1} is the first vertex in the matching order. If u_{k_1} and u_j are neighbors in the pattern graph P, then f_{k_1} and f_j are also neighbors in the data graph. The INT instruction $X := Intersect(A_{k_1}, A_j)$ actually calculates the vertices that can form a triangle with f_{k_1} and f_j. We find that some INT instructions in the execution plan repeatedly enumerate triangles around the start vertex. For example, in Fig. 3d, $T_7 := Intersect(A_1, A_3)$ and $T_6 := Intersect(A_1, A_5)$ both enumerate triangles around the start vertex f_1. Their computation is redundant. The previous methods [5] and [6] avoid such redundancy by pre-enumerating triangles and storing them as an index. The index requires non-trivial computation costs to maintain when the data graph is updated and occupies disk space to store.

In BENU, we propose the triangle cache technique to reduce such redundancy on the fly. As shown in Fig. 2, we set up a triangle cache for each working thread. For an INT instruction $X := Intersect(A_i, A_j)$, if one of f_i and f_j is the start vertex and the other one is the neighbor of the start vertex, we replace the INT instruction with a triangle caching instruction: $X := TCache(f_i, f_j, A_i, A_j)$. The triangle caching instruction queries the triangle cache with the key $[f_i, f_j]$ first. If the cache misses, it calculates $A_i \cap A_j$ and stores the result set to the cache. Otherwise, it uses the pre-computed set in the cache as the result set. In the running example in Fig. 3d, two marked instructions are replaced by the triangle caching instructions in Fig. 3e.

The triangle cache technique could be extended to other kinds of frequent motifs, like cliques. For an INT instruction $T_i := Intersect(Y, Z)$, we can restore it by replacing its

operands Y and Z with the adjacency sets calculating them as $T_i := Intersect(A_{x_1}, \ldots, A_{x_k})$. If $u_{x_1}, \ldots, u_{x_k}$ form a k-clique in the pattern graph, then T_i actually stores the set of vertices that can form a $k+1$-clique with $f_{x_1}, \ldots, f_{x_k}$. We can store T_i to cache the $k+1$-cliques. However, other kinds of motifs usually have a lot more matches than triangles, requiring that the cache can handle a lot of entries efficiently. It is not easy and is in our future work.

Support VCBC Compression. The VCBC compression (vertex-cover based compression) [6] is an efficient technique to compress the subgraph matching results. It compresses the matching results of a pattern graph P based on a vertex cover V_c of P. Given a data graph G, the matches of $core(P)$ in G are helves in VCBC where $core(P)$ is the induced subgraph of P on V_c. For each helve, the matches of the pattern vertices not in V_c are compressed in conditional image sets. The helves and the corresponding conditional image sets form the compressed codes of the matching results of P in G.

With modification, a BENU execution plan can directly output the VCBC-compressed matching results. For an execution plan E and a matching order O, assume the first k pattern vertices in O can form a vertex cover V_c of P while the first $k-1$ vertices cannot. The matches of the first k pattern vertices are the helves. For a pattern vertex u_j not in V_c, we delete the ENU instruction of f_j in E and remove f_j from the filtering conditions of other instructions. We reserve the INT instruction that calculates the candidate set for u_j. The candidate set C_j is equal to the conditional image set for u_j in the VCBC compression. Finally, we replace f_j in the RES instruction with C_j to directly output the compressed codes. Due to the space limitation, we omit the proof here.

Example. We modify the execution plan in Fig. 3e to Fig. 3f to support the VCBC compression. The first three vertices $\{u_1, u_3, u_5\}$ in the matching order form the vertex cover V_c.

Complexity Analysis. The cost of optimizing a raw execution plan depends on the pattern graph P. If the number of pattern vertices n is fixed, the most expensive pattern graph to optimize is the n-clique, because it has the most edges and its raw execution plan has the most common subexpressions. By inspecting the case of n-clique, we can get the worst-case computation complexity.

As for Optimization 1, an INT instruction in the raw execution plan has at most $n-1$ operands. Any combination of the operands is a common subexpression. The complexity of enumerating all the common subexpressions in that instruction is $\mathcal{O}(2^n)$. Since there are $\mathcal{O}(n)$ INT instructions, the cost of enumerating common subexpressions in all the instructions is $\mathcal{O}(n2^n)$. The cost of eliminating a subexpression is $O(n^2)$. Optimization 1 will repeat the elimination $\mathcal{O}(n)$ times until there is no common subexpression. The worst-case time complexity of Optimization 1 is $\mathcal{O}(n^2 2^n)$.

As for Optimization 2, the execution plan after flattening has $\mathcal{O}(m)$ instructions. Each instruction has at most 2 operands and n injective conditions, so the dependency graph has $\mathcal{O}(m)$ vertices and $\mathcal{O}(nm)$ edges. The cost of topological sort is $\mathcal{O}(nm)$. If we use a heap to find the next instruction with the highest rank, the maintaince cost of the heap during the sort is $\mathcal{O}(m \log m)$. Therefore, the worst-case time complexity of Optimization 2 is $\mathcal{O}(nm)$.

The costs of Optimization 3 and supporting VCBC compression are both linear to the number of instructions in the execution plan, which is $\mathcal{O}(m)$.

Summarily, the computation complexity of the whole optimization is $\mathcal{O}(n^2 2^n)$, dominated by Optimization 1.

C. Execution Plan Cost Estimation

The execution cost of an execution plan E is made up of the computation cost and communication cost. The computation cost is defined as the total execution times of all the INT/TRC instructions in the plan. The communication cost is defined as the total execution times of all the DBQ instructions. To estimate the execution cost of E, the core problem is to estimate the execution times of instructions.

The execution times of an instruction are related to the ENU instructions before the instruction. Assume the matching order of E is $O : u_{k_1}, u_{k_2}, \ldots, u_{k_n}$. We denote the induced subgraph of P on the vertex set of the first i vertices $\{u_{k_1}, \ldots, u_{k_i}\}$ as the partial pattern graph P_i. Matching u_{k_i} with the ENU instructions following the matching order is actually enumerating matches of P_i in G. The leftmost column in Fig. 5 shows the partial pattern graph P_i with the corresponding ENU instructions. The execution times of an ENU instruction $f_{k_i} := Foreach(C_{k_i})$ are equal to the number of matches of P_i in G. For the instructions between two adjacent ENU instructions, their execution times are equal to the execution times of the former ENU instruction.

We develop the `EstimateComputationCost` function in Algorithm 3 to estimate the computation cost of an execution plan E. The method to estimate the communication cost is similar. In the function, we use the estimation model proposed in Section 5.1 of [5] to estimate the number of matches of p' if p' is connected. If p' is disconnected, we decompose it into connected components and multiply the numbers of their matches together as the number of matches of p'. The estimation model can be replaced if a more accurate model is proposed later, but it is not our main concern in this paper.

D. Best Execution Plan Generation

The best execution plan is defined as the execution plan with the least execution cost among the execution plans generated from all the possible matching orders. For two execution plans E_a and E_b, we define $cost(E_a) < cost(E_b)$ if the communication cost of E_a is less than E_b or their communication costs are same but E_a has a less computation cost, since executing a DBQ instruction consumes much more time than an INT/TRC instruction.

We propose a search-based algorithm (Algorithm 3) to find the best execution plan E_{best} for a given pattern graph P. The algorithm calls the `Search` procedure to find the set of candidate matching orders O_{cand} that have the least communication cost. The algorithm generates optimized execution plans for O_{cand} and picks the best one.

The `Search` procedure maintains the unused pattern vertices in $\mathcal{C}$ and recursively enumerates the next pattern vertex in O from $\mathcal{C}$ one by one until O is complete. To avoid exploring all the permutations of pattern vertices, we propose two pruning strategies.

Dual Pruning. In line 11, we use the dual condition to filter out redundant matching orders. The dual condition is based on the syntactic equivalence (SE) relations [17] between pattern vertices. For two vertices u_i and u_j in P, u_i is SE to u_j (denoted as $u_i \simeq u_j$) iff $\Gamma_P(u_i) - \{u_j\} = \Gamma_P(u_j) - \{u_i\}$. For example, in q4 of Fig. 6, $u_1 \simeq u_4$ and $u_2 \simeq u_3$. Given two SE vertices $u_i \simeq u_j$ and a matching order O, we define the matching order got by swapping u_i and u_j in O as its dual matching order O'.

The execution plans generated from O and O' have the same cost. For an execution plan E generated from O, if we swap A_i/A_j, C_i/C_j and f_i/f_j in every instruction and adjust the symmetry breaking conditions correspondingly in E , we can get a dual execution plan E' with the matching order O'. E' is correct because the candidate set calculation in E' still follows the principle introduced in Section IV-A. The partial pattern graphs P_i and P'_i induced by the first i vertices in E and E' are isomorphic for any $1 \leq i \leq n$. The execution times of the i-th ENU instructions in E and E' are same. Therefore, the communication and computation costs of E and E' are same.

If $u_i \simeq u_j$ and $i < j$, we only need to explore the matching order that u_i appears before u_j, because the dual matching order cannot generate a better execution plan. The dual condition in line 11 rejects the dual matching order.

Cost-based Pruning. Since the execution plan optimization does not affect the relative order of DBQ and ENU instructions, the communication cost of an execution plan only depends on its matching order. Algorithm 3 generates the matching order and maintains its communication cost simultaneously in line 14 to 18. Depending on the neighborhood of u, the cost is updated with two cases. In case 1, at least one of u's neighbors will appear after u in O. According to Section IV-A, a DBQ instruction will be generated for u in the execution plan. The execution times of this instruction are equal to the number of matches of p'. In case 2, all of u's neighbors are used in O. No DBQ instruction will be generated, thus the communication cost remains unchanged. If the communication cost of the partial order O is already bigger than the current best cost, O and all the orders *expanded* from O can be pruned.

Complexity Analysis. The time complexity of the `Search` procedure is dominated by the estimation operations in line 15. The complexity of the operation is $\mathcal{O}(m)$ and we denote its execution times as α. The time complexity of line 4 to 7 is dominated by the optimized execution plan generation operation. The complexity of the operation is $\mathcal{O}(n^2 2^n)$ and we denote its execution times as β. Therefore, the time complexity of Algorithm 3 is $\mathcal{O}(\alpha m + \beta n^2 2^n)$. α and β are affected by the pattern graph. The upper bound of α is $\sum_{i=1}^{n} \mathcal{P}(n,i)$ ($\mathcal{P}(n,i)$ is i-permutations of n). The upper bound of β is $n!$. In practice, α and β are much less than their upper bounds.

Algorithm 3 Best Execution Plan Generation

Input: Pattern graph P. **Output:** Best execution plan E_{best}.

```
1:  E_best ← NULL; O_cand ← {};                          ▷ Global variables
2:  bCommCost ← +∞; bCompCost ← +∞;                       ▷ Best costs
3:  SEARCH(i=0, C=V(P), p=new PartialPatternGraph(),
    O=new MatchingOrder(), commCost=0);                  ▷ O_cand is updated
4:  for all O ∈ O_cand do
5:      E ← GenerateOptimizedExecutionPlan(P, O);
6:      cost ← ESTIMATECOMPUTATIONCOST(P,E);
7:      if cost < bCompCost then E_best ← E, bCompCost ← cost;
8:  return E_best.
9:  procedure SEARCH(i, C, p, O, commCost)
10:     if i < |V(P)| then                               ▷ O is not complete
11:         for all u ∈ C that u passes dual condition checking do
12:             O[i] ← u; C' ← C − {u};
13:             p' ← Add u to the partial pattern graph p;
14:             if Γ_P(u) ∩ C ≠ ∅ then                   ▷ Case 1
15:                 s ← Estimate the number of matches of p';
16:             else                                     ▷ Case 2
17:                 s ← 0;
18:             commCost' ← commCost + s;
19:             if commCost' > bCommCost then continue;
20:             SEARCH(i + 1, C', p', O, commCost');
21:     else                                             ▷ O is complete
22:         if commCost < bCommCost then
23:             bCommCost ← commCost; O_cand ← {O};
24:         else if commCost = bCommCost then
25:             O_cand ← O_cand ∪ {O}.
26: function ESTIMATECOMPUTATIONCOST(P, E)
27:     cost ← 0; curNum ← 0; p' ← new PartialPatternGraph();
28:     for all instruction I ∈ E do
29:         if I.type is ENU then
30:             Update p' with I;
31:             curNum ← estimate the number of matches of p';
32:         else if I.type is INT or TRC then
33:             cost ← cost + curNum;
34:     return cost.
```

V. Efficient Implementation

In this section, we propose two implementation techniques to improve the efficiency of BENU.

A. Local Database Cache

Intra-task Locality. Inside a local search task, a queried adjacency set tends to be queried again soon in the same task. For example, in the backtracking search trees illustrated in Fig. 5, the adjacency set of v_4 is queried repeatedly in different search branches in the local search task 1. This

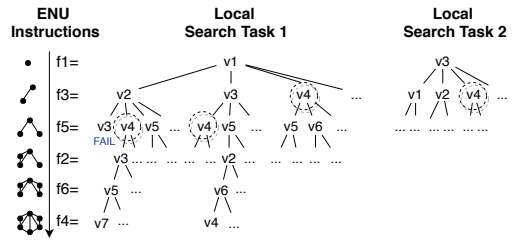


Figure 5. Backtracking search trees of local search tasks. The search is guided by the execution plan in Fig. 3e and runs on G in Fig. 1b.

kind of locality comes from the backtracking nature of the execution plan. All the vertices that a local search task visits are in a local neighborhood around the start vertex. The size of the local neighborhood is bounded by the radius of the pattern graph that is usually small. When a local search task queries many adjacency sets during the backtracking search, there are some repeated queries, bringing intra-task locality.

Inter-task Locality. Some adjacency sets are queried by many different local search tasks. For example in Fig. 5, the adjacency set of v_4 is queried in both the task 1 and 2. This kind of locality comes from the overlaps between local neighborhoods visited by different tasks. Some data vertices, especially high-degree vertices, are close to many start vertices of different tasks. Their adjacency sets are *hot* and are frequently queried by different tasks.

Database Cache. To take advantage of both kinds of locality, we set up an in-memory *database cache* (DB cache) with configurable capacity in each worker machine as shown in Fig. 2. The cache stores the adjacency sets fetched from the distributed database. The cache can capture the intra-task locality via replacement policies like LRU. To capture the inter-task locality, the cache is shared among all the working threads. The cache provides a flexible mechanism to *trade memory for reduction in communication*.

Complexity Analysis. With the cache technique, the communication cost of an execution plan E (i.e. the number of conducted database queries) is also related to the cache capacity $\mathcal{C}$. To analyze its upper bound, we first define several concepts. The r-hop neighborhood ($r \geq 0$) of a vertex v in a graph g is defined as $\gamma_g^r(v)$=$\{w \in V(g)|$ w is at most r hops away from $v\}$. The size of $\gamma_g^r(v)$ is $\mathcal{S}_g^r(v) = \sum_{w \in \gamma_g^r(v)} d_g(w)$. For a data graph G, $\mathcal{H}_G^r = \max_{v \in V(G)} \mathcal{S}_G^r(v)$ is the size of the largest r-hop neighborhood in G. As for the cache, we assume there is w working threads per machine and there exists R that $\mathcal{C} \geq w\mathcal{H}_G^R$, i.e. the cache can store the R-hop neighborhood of any data vertex for every working thread. As for the execution plan, we assume its matching order is $O : u_{k_1}, u_{k_2}, \ldots, u_{k_n}$. The first α vertices in O can cover every edge in P. Among them, there must exist r' ($0 \leq r' \leq R$) and β ($1 \leq \beta \leq \alpha$) that the r'-hop neighborhood of u_{k_β} $\gamma_P^{r'}(u_{k_\beta})$ contains $\{u_{k_\beta}, u_{k_{\beta+1}}, \ldots, u_{k_\alpha}\}$. Then, we can split O into three sections: $O : u_{k_1}, \ldots, u_{k_\beta}, \ldots, u_{k_\alpha}, \ldots, u_{k_n}$.

The total communication cost of matching f_{k_1} to f_{k_β} is $\mathcal{O}(\sum_{i=1}^{\beta} |\mathcal{R}_G(P_i)|)$ where $|\mathcal{R}_G(P_i)|$ is the number of matches of the partial pattern graph P_i in G. If f_{k_1} to f_{k_β} is fixed in f, the number of conducted database queries during matching $f_{k_{\beta+1}}$ to f_{k_α} is at most $\max_{v \in V(G)} |\gamma_G^{r'}(v)|$, because the cache can store all the adjacency sets in $\gamma_G^{r'}(f_{k_\beta})$. The total communication cost of matching $f_{k_{\beta+1}}$ to f_{k_α} for all the partial matches is $\mathcal{O}(|\mathcal{R}_G(P_\beta)| \max_{v \in V(G)} |\gamma_G^{r'}(v)|)$. Matching the remaining vertices $f_{k_{\alpha+1}}$ to f_{k_n} does not query any adjacency set. Therefore, the communication upper bound is $\mathcal{O}(\sum_{i=1}^{\beta} |\mathcal{R}_G(P_i)| + |\mathcal{R}_G(P_\beta)| \max_{v \in V(G)} |\gamma_G^{r'}(v)|)$.

If $\mathcal{C}$ is bigger than the data graph, a tighter upper bound is $\mathcal{O}(p|V(G)|)$ where p is the number of worker machines, independent of the pattern graph.

B. Task Splitting

The computation and communication costs of a local search task in BENU are positively correlated with the degree of the start vertex. Unfortunately, real-world graphs often follow the power-law degree distribution, causing the workloads of local search tasks skewed. Few heavy tasks may make several workers become stragglers in execution.

We propose the *task splitting* technique to split heavy tasks into smaller subtasks to balance the workloads. Suppose u_{k_2} is the second pattern vertex in the matching order and C_{k_2} is its candidate set in the execution plan. If the degree of the start vertex of a local search task is bigger than a given threshold τ, we split C_{k_2} into $\lceil \frac{|C_{k_2}|}{\tau} \rceil$ non-overlapping equal-sized subsets and use different subsets as C_{k_2} in different subtasks.

We conduct task splitting when we generate local search tasks for the data vertices v with $d_G(v) \geq \tau$. If u_{k_1} and u_{k_2} are adjacent in P, then C_{k_2} is calculated from A_{k_1} and we generate $\lceil \frac{d_G(v)}{\tau} \rceil$ subtasks. Otherwise, C_{k_2} is calculated from $V(G)$ and we generate $\lceil \frac{|V(G)|}{\tau} \rceil$ subtasks.

VI. Related Work

Subgraph Matching on Single Machine. Most of the serial subgraph matching approaches work with labeled graphs and follow the backtracking-based framework [16]. They differ in how to determine the matching order and the candidate sets of pattern vertices. GraphQL [18] and SPath [19] match pattern vertices with infrequent labels and paths first. Turbo$_{\text{iso}}$ [7] uses the candidate region to dynamically determine the matching order and candidate sets. CFL-Match [8] proposes the core-forest-leaf decomposition to get matching orders with Cartesian operations postponed. Those in-memory algorithms cannot work with data graphs larger than the memory. DUALSIM [9] adopts the external memory method to handle large data graphs and uses the dual approach to reduce the number of disk reads. However, the computing power of a single machine limits its performance.

Subgraph Matching in Distributed Environment. Zhao et al. [20] propose a distributed algorithm that combines

graph exploration and multiway join together for the Trinity platform. The distributed algorithms for the dataflow engine can be divided into DFS-style and BFS-style. On the DFS-style side, Afrati et al. [11] use one-round multi-way join to enumerate subgraphs. QFrag [10] broadcasts the data graph and adopts the task-parallel paradigm. However, they have scalability issues for complex pattern graphs [11] or large data graphs [10]. The BFS-style algorithms follow a join-based framework. They propose varieties of join units (Edge [13], Stars [5] [14], TwinTwig [12], Clique [5] and Crystal [6]) and join frameworks (left-deep join [12], two-way bushy join [5], hash-assembly [6] and worst-case optimal join [13]) to limit the sizes of the intermediate results in the join. Qiao et al. [6] further proposes the VCBC compression to compress the matching results. Recently, Ammar et al. propose the BiGJoin [13] algorithm that is worst-case optimal and can handle both static and dynamic data graphs. Considering both performance and novelty, we regard CBF [6] and BiGJoin [13] as the state-of-the-art methods for distributed subgraph enumeration.

VII. Experiments

In Exp-1, we evaluate the execution plan generation performance of BENU. In the other experiments, we evaluate the enumeration performance of BENU and report the wall-clock time spent on pure enumeration as the execution time, not including the time spent on join/execution plan generation and output.

Environment. All the experiments were conducted in a cluster with 17 machines (1 master + 16 workers) connected via 1 Gbps Ethernet. Each machine was equipped with 12 cores, 50 Gbytes memory and 2 Tbytes RAID0 HDD storage. All the algorithms were implemented with Hadoop 2.7.2, compiled with JDK 1.8 and run in CentOS 7.0. The map output in Hadoop was compressed with Gzip.

BENU. We stored the data graph in HBase 1.2.6 and implemented BENU in MapReduce. The local search tasks were generated in the map phase and shuffled evenly to 16 reducers (one reducer per worker machine). Each reducer ran the local search tasks in parallel with 24 working threads. We allocated 4 Gbytes memory to each mapper and 40 Gbytes to each reducer. In each reducer, 30 Gbytes memory was allocated to the local database cache. The degree threshold τ was 500 in task splitting. Without otherwise mentioned, we used the compressed version of the execution plans.

Data Graphs. We used five real-world data graphs to test the performance (Table I), which were also used by the previous work like [5] and [6]. *uk* was provided by LAW (http://law.di.unimi.it/datasets.php), and other datasets were downloaded from SNAP (http://snap.stanford.edu/data).

Pattern Graphs. We used 9 pattern graphs in Fig. 6. *q1* to *q5* came from [6]. To evaluate the performance on tough tasks, we further used *q6* to *q9* with six vertices. Usually, they have a lot more matches than the pattern graphs with

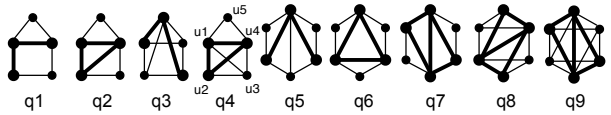
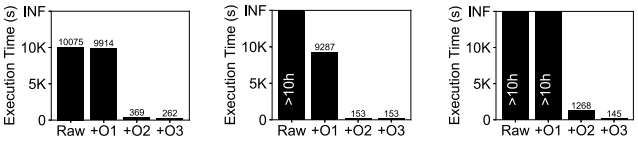


Figure 6. Query patterns.



(a) *q2* on *ok* (b) *q4* on *ok* (c) *q5* on *lj*

Figure 7. Effects of execution plan optimization techniques. The X-axis represents execution plans optimized from the raw plan with more optimization techniques. We ran the uncompressed version of *q2* and *q4*.

five vertices (*q1* to *q4*). The vertex covers of pattern graphs under the VCBC compression is illustrated with big dots.

A. Evaluation of Optimization Techniques in BENU

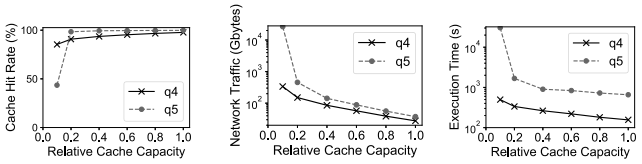
Exp-1: Best Execution Plan Generation. We evaluated the efficiency of Algorithm 3 with three kinds of pattern graphs: (1) graphs in Fig. 6, (2) cliques and (3) connected random graphs. The results were reported in Table IV. For the third kind of graphs, we generated 1000 connected random graphs for each given number of vertices n and reported the average performance. We reported the relative value of α and β compared to their upper bounds discussed in Section IV-D. $\frac{\beta}{n!}$ was less than 15% in all the cases. For the random graph cases, $\frac{\beta}{n!} < 1\%$. The pruning techniques were effective. It helped the algorithm run quickly. Compared with the enumeration time that was usually hundreds of seconds on large data graphs, the time spent on execution plan generation was negligible.

Exp-2: Execution Plan Optimization. We tested the effectiveness of the execution plan optimization techniques proposed in Section IV-B with three representative cases in Fig. 7. As the compression would negate some optimization techniques, we used the uncompressed execution plans in case (a) and (b). Optimization 1 was effective in case (b) where it eliminated the common subexpression `Intersect(A1,A4)`. Optimization 2 significantly reduced the execution time in all three cases as it promoted INT instructions to outer loops in all of them. Optimization 3 was effective in case (a) and (c) where the triangles were repeatedly enumerated by two INT instructions.

Exp-3: Local Database Cache. We tested the effects of the local database cache capacity on performance with two pattern graphs *q4* and *q5*. The results were shown in Fig. 8. When the capacity was 10% of the data graph (93 Mbytes), the average cache hit rates reached 85% and 43% on *q4* and *q5* respectively. When the relative capacity grew to 20%, the cache hit rates reached 91% and 98%. As the capacity increased, the cache hit rate increased and the communication cost and execution time decreased quickly. The cache could make use of the strong locality in BENU to significantly reduce the communication costs.

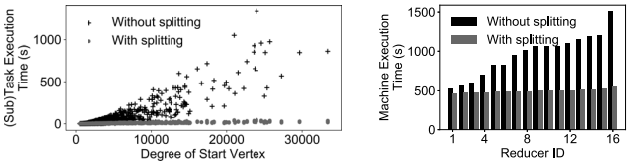
Table IV
EFFICIENCY OF BEST EXECUTION PLAN GENERATION

Measurement	Pattern Graphs in Fig. 6									Cliques (n=)			Random (n=)			
	q1	*q2*	*q3*	*q4*	*q5*	*q6*	*q7*	*q8*	*q9*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Relative α (%)	31.7	27.4	13.8	10.2	15.7	8.6	16.8	4.1	5.6	4.7	1.2	0.3	7.3	6.2	5.4	5.5
Relative β (%)	13.3	10.0	3.3	5.0	3.3	5.0	3.3	1.7	0.8	4.2	0.8	0.1	0.9	0.3	0.1	0.02
Time (s)	0.2	0.2	0.2	0.2	0.3	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.3	0.6	2.0	18.0



(a) Cache Hit Rate (b) Communication Cost (c) Execution Time

Figure 8. Effects of the local database cache capacity. The relative cache capacity is got by dividing the capacity by the size of the data graph *ok*.



(a) Task level (b) Machine level

Figure 9. Effects of the task splitting technique on different levels. Pattern graph: *q5*, data graph: *ok*.

Exp-4: Task Splitting. We measured the execution time of *q5* on *ok* with and without the task splitting technique in Fig. 9. Without task splitting, several local search tasks would take >1000s to finish. Those heavy tasks made the workloads of reducers unbalanced as shown in Fig. 9b. After task splitting with $\tau = 500$, the execution time of all the tasks was <50s with the number of tasks increasing slightly from 3.07M to 3.12M. The workloads of the subtasks distributed evenly among the reducers.

B. Compare with Existing Approaches

We compared BENU with two state-of-the-art methods CBF [6] and BiGJoin [13]. For all the algorithms, we turned on all the compression and optimization techniques provided with the algorithms, and we reported the wall-clock time spent on pure enumeration as the execution time, not including the time spent on join/execution plan generation and output.

Exp-5: Compare with CBF. CBF is the state-of-the-art algorithm in MapReduce. We ran CBF with 12 mappers/reducers per worker machine and allocated 4 Gbytes memory to each mapper/reducer. The results were reported in Table V. Nearly in all the cases except *q5* on *fs*, BENU ran quicker than CBF with acceptable communication costs. In several cases like *q2* on *ok*/*uk*, *q4* on *ok* and *q6* on *lj*/*ok*/*uk*, BENU was up to 10× quicker than CBF. The hard test cases *q7* to *q9* shared the same core structure, i.e. the chordal square (shown with bold edges in Fig. 6). The core structure had more than 2 billion matches in all the data graphs (Table I). CBF had to shuffle the clique index and the matching results of the core structure when preparing partition files for hash-assembly. Shuffling that many key-value pairs was costly and made Hadoop throw the shuffle error exception in some cases. BENU ran smoothly in those cases. For the cases of *q7*/*q8*/*q9* on *uk*, the core structure had 2.7 trillion matches. Neither BENU nor CBF could finish in 10 hours.

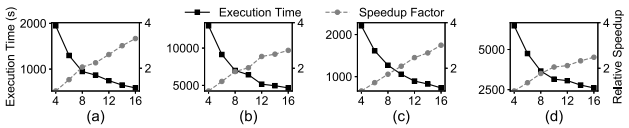


Figure 10. Scalability with varying worker machines. (a) *q5* on *ok*, (b) *q5* on *fs*, (c) *q9* on *ok*, (d) *q9* on *fs*.

Exp-6: Compare with BiGJoin. We compared the performance with BiGJoin (https://github.com/frankmcsherry/dataflow-join/) on the pattern graphs that BiGJoin had specially optimized. BiGJoin was written in Rust. In BiGJoin, the batch size was 100000, and each worker machine was deployed with 12 working processes (one process per core). We compared BENU with both the shared-memory version (BiGJoin(S)) and the distributed version (BiGJoin(D)) of BiGJoin in Table VI. Since BiGJoin was built upon Timely dataflow engine that used a different communication mechanism from MapReduce, we did not report the communication costs. On *ok*, BENU ran quicker than both of BiGJoin(D) and BiGJoin(S) on complex pattern graphs. On *fs*, BiGJoin(S) failed due to out of memory exception while BENU ran quicker than BiGJoin(D) in all cases.

C. Scalability

We tested the machine scalability of BENU by varying numbers of worker machines in the cluster. The results were reported in Fig. 10. As the number of worker machines increased, the execution time decreased and achieved a near-linear speed up. Though the relative speedup factors did not reach 4 when varying from 4 to 16 worker machines, the relative speedup factors grew near linearly.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we studied the distributed subgraph enumeration problem. We proposed a backtracking-based framework *BENU* and BENU execution plans with two features: (1) shuffling data graph instead of intermediate results, (2) on-demand shuffling. We developed execution plan optimization techniques, a best execution plan generation algorithm, the local database cache technique and the task splitting technique to improve the performance. Extensive experiments verified the efficiency of BENU. We shall explore generalizing the triangle cache technique and extending BENU to property graphs in the future.

Table V
PERFORMANCE COMPARISON WITH CBF

Dataset	q1		q2		q3		q4		q5	
	CBF	***BENU***	***CBF***	***BENU***	***CBF***	***BENU***	***CBF***	***BENU***	***CBF***	***BENU***
as	270/3G	**119/6G**	167/26G	**69/6G**	239/3G	**92/7G**	158/26G	**68/5G**	356/1G	**131/6G**
lj	396/11G	**183/16G**	662/210G	**102/16G**	348/11G	**138/17G**	656/207G	**117/15G**	190/5G	**128/14G**
ok	2942/29G	**859/30G**	1465/512G	**139/28G**	1446/28G	**425/29G**	1507/508G	**139/26G**	1024/14G	**595/29G**
uk	>7200s	**2131/90G**	>7200s	**412/81G**	>7200s	**1221/93G**	>7200s	**930/85G**	>7200s	**3549/103G**
fs	>41555s	**16622/416G**	CRASH	**5008/472G**	>10547s	**4219/391G**	>7200s	**1543/371G**	**2088/137G**	4484/392G

Dataset	q6		q7		q8		q9	
	CBF	***BENU***	***CBF***	***BENU***	***CBF***	***BENU***	***CBF***	***BENU***
as	288/4G	**68/4G**	CRASH	**1188/6G**	CRASH	**7632/7G**	CRASH	**315/5G**
lj	1000/20G	**108/12G**	CRASH	**9318/20G**	>16710s	**6684/17G**	>7200s	**2111/16G**
ok	2556/48G	**143/22G**	CRASH	**2327/31G**	>7200s	**2974/29G**	>7200s	**712/28G**
uk	16488/131G	**1090/39G**	>10h	>10h	FAIL*	>10h	FAIL*	>10h
fs	18472/691G	**1349/314G**	>11272s	**4509/424G**	>10282s	**4113/362G**	>7200s	**2464/350G**

+ In each cell, the first number is the wall-clock execution time (unit: second), and the second number is the cumulative communication cost (unit: byte).
× The quickest algorithm in each case is marked with bold font. * CBF failed when building the clique index of four vertices.

Table VI
EXECUTION TIME COMPARISON WITH BIGJOIN (UNIT: SECOND)

G	Algorithm	Triangle	Clique4	Clique5	q4	q5
ok	BiGJoin(S)	**53**	111	651	608	OOM
	BiGJoin(D)	130	OOM	OOM	>7200	OOM
	BENU	93	**99**	**129**	**139**	**595**
fs	BiGJoin(S)	OOM	OOM	OOM	OOM	OOM
	BiGJoin(D)	1749	>7200	>7200	>7200	>7200
	BENU	**1229**	**1239**	**1251**	**1543**	**4484**

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (grant numbers 61572250, U1811461, 61702254), Jiangsu Province Science and Technology Program (grant number BE2017155), National Natural Science Foundation of Jiangsu Province (grant number BK20170651), Collaborative Innovation Center of Novel Software Technology and Industrialization, and Project funded by China Postdoctoral Science Foundation.

REFERENCES

[1] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks." *Science*, vol. 298 5594, pp. 824–7, 2002.

[2] N. Przulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 23, no. 2, pp. 177–183, 2007.

[3] S. R. Kairam, D. J. Wang, and J. Leskovec, "The life and death of online groups: predicting group growth and longevity," in *Proceedings of the fifth International Conference on Web Search and Web Data Mining*, 2012, pp. 673–682.

[4] W. Fan, X. Wang, Y. Wu, and J. Xu, "Association rules with graph patterns," *PVLDB*, vol. 8, no. 12, pp. 1502–1513, 2015.

[5] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang, "Scalable distributed subgraph enumeration," *PVLDB*, vol. 10, no. 3, pp. 217–228, 2016.

[6] M. Qiao, H. Zhang, and H. Cheng, "Subgraph matching: on compression and computation," *PVLDB*, vol. 11, no. 2, pp. 176–188, 2017.

[7] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 337–348.

[8] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1199–1214.

[9] H. Kim, J. Lee, S. S. Bhowmick, W. Han, J. Lee, S. Ko, and M. H. A. Jarrah, "DUALSIM: parallel subgraph enumeration in a massive graph on a single machine," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1231–1245.

[10] M. Serafini, G. D. F. Morales, and G. Siganos, "Qfrag: distributed graph search via subgraph isomorphism," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 214–228.

[11] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 62–73.

[12] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *PVLDB*, vol. 8, no. 10, pp. 974–985, 2015.

[13] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows," *PVLDB*, vol. 11, no. 6, pp. 691–704, 2018.

[14] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 625–636.

[15] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *Research in Computational Molecular Biology*, T. Speed and H. Huang, Eds. Springer Berlin Heidelberg, 2007, pp. 92–106.

[16] J. Lee, W. Han, R. Kasperovics, and J. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *PVLDB*, vol. 6, no. 2, pp. 133–144, 2012.

[17] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *PVLDB*, vol. 8, no. 5, pp. 617–628, 2015.

[18] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2008, pp. 405–418.

[19] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1, pp. 340–351, 2010.

[20] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, pp. 788–799, 2012.