Distributed Subgraph Enumeration via Backtracking-based Framework

Zhaokang Wang, Weiwei Hu, Chunfeng Yuan, Rong Gu*, Yihua Huang*

Abstract—Finding or monitoring subgraph instances that are isomorphic to a given pattern graph in a data graph is a fundamental query operation in many graph analytic applications, such as network motif mining and fraud detection. The state-of-the-art distributed methods are inefficient in communication. They have to shuffle partial matching results during the distributed multiway join. The partial matching results may be much larger than the data graph itself. To overcome the drawback, we develop the Batch-BENU framework (B-BENU) for distributed subgraph enumeration. B-BENU executes a group of local search tasks in parallel. Each task enumerates subgraphs around a vertex in the data graph, guided by a backtracking-based execution plan. B-BENU does not shuffle any partial matching result. Instead, it stores the data graph in a distributed database. Each task queries adjacency sets of the data graph on demand. To support dynamic data graphs, we propose the concept of incremental pattern graphs and turn continuous subgraph enumeration into enumerating incremental pattern graphs at each time step. We develop the Streaming-BENU framework (S-BENU) to enumerate their matches efficiently. We implement B-BENU and S-BENU with the local database cache and the task splitting techniques. The extensive experiments show that B-BENU and S-BENU can scale to big data graphs and complex pattern graphs. They outperform the state-of-the-art methods by up to one and two orders of magnitude, respectively.

Index Terms—backtracking-based framework, continuous subgraph matching, distributed graph querying, subgraph isomorphism, subgraph matching.

1 INTRODUCTION

Given a big data graph \( G \) and a small pattern graph \( P \), subgraph enumeration is to find all the subgraph instances of \( G \) that are isomorphic to \( P \). The subgraph instances are the matching results of \( P \) in \( G \). Subgraph enumeration is a fundamental query operation in many graph analytic applications, including network motif mining [1], graphlet-based network comparison [2], network evolution analysis [3], and social network recommendation [4].

When the data graph is dynamic, the subgraph enumeration problem becomes the continuous subgraph enumeration problem. The edge set of a dynamic data graph evolves over time. The matching results of a pattern graph also change consequently. The continuous subgraph enumeration focus on monitoring the changes in the matching results as the data graph evolves. Detecting appearing subgraph instances of suspicious pattern graphs timely is essential in real-world applications like fraud detection [5] [6] and cybersecurity [7].

1.1 Motivation

Enumerating instances of a pattern graph in a big data graph is challenging due to two difficulties. First, the core operation of subgraph enumeration is subgraph isomorphism. It is an NP-complete problem and has high computational complexity. Second, the sizes of (partial) matching results can be much larger than the data graph itself [8] [9]. Table 1 shows the numbers of matches of some typical pattern graphs in real-world data graphs. The numbers of matching results can be 10 to 100 times larger than the numbers of edges in data graphs. Just scanning matching results takes considerable computational costs.

Some serial in-memory subgraph enumeration algorithms like [10] [11] and out-of-core algorithm [12] are proposed, but the computing power of a single machine limits their performance. The emerging need to process big data graphs inspires researchers to design efficient distributed subgraph enumeration methods. Based on whether a distributed algorithm shuffles intermediate results, we divide the existing distributed algorithms into two groups: DFS-style and BFS-style.

The DFS-style algorithms do not shuffle intermediate results. Instead, they shuffle the data graph. QFrag [13] broadcasts the data graph to each machine and enumerates subgraphs in memory on each machine concurrently. However, it cannot scale to data graphs bigger than the memory capacity. Afrati et al. [14] use the one-round multiway join to enumerate subgraphs with MapReduce. However, it cannot scale to complex pattern graphs due to large replication of edges, empirically performing worse than the BFS-style algorithm [15] [16].

The BFS-style algorithms decompose the pattern graph recursively into a series of join units. A join unit is a simple partial pattern graph whose matching results can be enumerated easily from the data graph or a pre-computed index. The BFS-style algorithms enumerate matching results of join units first and assemble them via one or more rounds of joining to get the matching results for the whole pattern graph.
graph. The algorithms shuffle the partial matching results (intermediate results) during the join. Researchers propose varieties of join units (Edge [19], Star [8] [20], TwinTwig [15] [16], Clique [8], Crystal [9]) and join frameworks (Left-deep join [15], [16], Bushy join [8], Hash-assembly [9], Generic join [19]) to reduce intermediate results.

However, BFS-style algorithms are still costly. First, shuffling partial matching results is inevitable in the join-based framework, causing high communication costs. The typical pattern graphs in Table 1 are the core structures of many complex pattern graphs in Fig.8. Just shuffling matching results of the core structures will cause high communication costs. Second, some cutting-edge algorithms like SEED [8] and CBF [9] build extra index structures like SCP index (in SEED) or clique index (in CBF) for each data graph to achieve high performance. The index requires non-trivial computation costs to construct and store. It also requires extra costs to maintain if the data graph is dynamic, which is common in the industry.

The drawbacks of the existing methods inspire us designing a new distributed (continuous) subgraph enumeration framework that 1) avoids shuffling partial matching results, 2) does not rely on any extra index, and 3) scales to large data graphs and complex pattern graphs.

### 1.2 Contributions

In this work, we present two new distributed Backtracking-based subgraph ENUMeration (BENU) frameworks: the Batch-BENU framework for static data graphs and the Streaming-BENU framework for dynamic data graphs. An earlier version of this work [21] was presented at the 35th IEEE International Conference on Data Engineering (ICDE 2019). In that version, we proposed the Batch-BENU framework for distributed subgraph enumeration in static undirected data graphs and implemented it with MapReduce. Batch-BENU does not shuffle intermediate results or use indices. Instead, it stores the data graph in a distributed database and queries the adjacency sets of the data graph on demand, driven by backtracking-based execution plans.

However, simply extending Batch-BENU to process dynamic data graphs is inefficient. Batch-BENU has to enumerate subgraphs in the latest data graph snapshot at every time step and compare the matching results with the previous time step to detect appearing/disappearing subgraphs (i.e., incremental matches). Enumerating subgraphs from scratch repeatedly contains lots of redundant computation. The long execution time of subgraph enumeration can hardly meet the near real-time performance requirement of online applications. Supporting dynamic data graphs is not a trivial extension. The challenge is reducing redundant computation as much as possible and enumerating incremental matches directly from the update of each time step.

To achieve the target, the cutting-edge methods for dynamic data graphs either maintain the latest matching results in memory [22] [23] [24] or have to eliminate contradictory results with extra shuffling [19]. They are inefficient in storage and communication, respectively.

In this work, we propose a novel concept—incremental pattern graphs—for continuous subgraph enumeration in dynamic graphs. We prove that finding incremental matches at every time step is equivalent to enumerating subgraph instances of incremental pattern graphs. We propose the Streaming-BENU framework to enumerate them from the updated edges directly, guided by backtracking-based incremental execution plans. Streaming-BENU outputs valid and duplication-free results without maintaining any matching result in memory.

Overall, we make the following contributions.

First, we propose a distributed subgraph enumeration framework Batch-BENU. Batch-BENU generates local search tasks for every data vertex and executes the tasks in parallel on a distributed computing platform. A local search task enumerates matches of the pattern graph in the local neighborhood for a data vertex, following a backtracking-based execution plan. Batch-BENU does not shuffle any partial matching result or use any index. Instead, it queries the data graph stored in a distributed database on demand.

Second, we propose a search-based method to generate the best execution plan. The method includes three execution plan optimization techniques (common subexpression elimination, instruction reordering, and triangle caching), a cost estimation model, and two pruning techniques.

Third, we propose the concept of incremental pattern graphs to support continuous subgraph enumeration in dynamic graphs. Based on the concept, we solve continuous subgraph enumeration by enumerating matches of incremental pattern graphs in the data graph snapshots at each time step. We develop the Streaming-BENU framework to enumerate their matches efficiently.

Forth, we propose efficient implementations of Batch-BENU and Streaming-BENU. We propose the local database technique to reduce communication costs and the task splitting technique to balance workloads. The experimental results validate the efficiency and scalability of the two frameworks. They outperform the state-of-the-art methods on complex pattern graphs by up to one and two orders of magnitude, respectively.

We organize the rest of the paper as follows. Section 2 defines the problem and introduces related techniques. Section 3 describes the Batch-BENU framework. Section 4 presents the method to generate the best execution plan for Batch-BENU. Section 5 elaborates on the Streaming-BENU framework for dynamic graphs. Section 6 discusses the implementations. Section 7 experimentally evaluates their performance. Section 8 briefly surveys the related work. Section 9 concludes the work.

---

1. The two frameworks are open sourced at https://github.com/PasaLab/BENU.
2 Preliminaries

We first define the problem of subgraph enumeration and its continuous variant. Then, we introduce the backtracking-based framework for subgraph matching.

2.1 Problem Definition

In this work, we focus on processing simple unlabeled graphs. We define a static graph $g$ as $g = (V(g), E(g))$, where $V(g)$/$E(g)$ is the vertex/edge set of $g$. If $g$ is undirected, we denote the adjacency set of a vertex $v$ as $\Gamma_g(v) = \{w | (w, v) \in E(g)\}$. The degree of $v$ is $d_g(v) = |\Gamma_g(v)|$. If $g$ is directed, we denote the incoming/outgoing adjacency set of a vertex $v$ as $\Gamma_g^+(v) = \{w | (w, v) \in E(g)\}$/$\Gamma_g^-(v) = \{w | (v, w) \in E(g)\}$. 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 $g(V') = (V' \cap V(g), \{(u, w) | (u, w) \in E(g), u \in V', w \in V'\})$.

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. We use $v_i$/$u_i$ to denote a vertex from the data/pattern graph. 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 $V(P) = \{u_1, u_2, \ldots, u_n\}$. A match of $P$ in $G$ is defined in Definition 1. An isomorphic subgraph of $P$ in $G$ is defined in Definition 2. Taking Fig. 1 as the example, the subgraph shown with bold lines in $G$ is isomorphic to $P$ with a match $f' = (v_1, v_2, v_3, v_4, v_5, v_8)$.

**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 $\forall x,y : (x, y) \in E(P) \Rightarrow (f(x), f(y)) \in E(G)$. A match $f$ is denoted as $f = (f_1, f_2, \ldots, f_n)$, 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 $|E(P)| = |E(g)|$.

We follow [8] to define the subgraph enumeration problem in Definition 3. We denote the set of the isomorphic subgraphs of $P$ in $G$ as $R_G(P)$. The task of subgraph enumeration is to calculate $R_G(P)$. Subgraph enumeration focuses on undirected $P$ and $G$. We extend to directed ones in continuous subgraph enumeration.

**Definition 3**. Given a static undirected pattern graph $P$ and a static undirected data graph $G$, the task of subgraph enumeration is to enumerate all subgraphs of $G$ that are isomorphic to $P$.

When data graphs are dynamic, the subgraph enumeration becomes the continuous subgraph enumeration. In a dynamic graph, vertices and edges are inserted/removed from the graph in a streaming manner. Since a vertex insertion/removal operation can be decomposed into multiple edge insertion/removal operations, we focus on handling dynamic graphs with edge updates.

A dynamic data graph is defined as $G' = (V(G_0), E(G_0), \Delta G')$, where $V(G_0)$/$E(G_0)$/$\Delta G'$ is the initial vertex/edge set of $G'$. $\Delta G'$ is the update stream of $G'$. $\Delta G' = \{\Delta o_1, \Delta o_2, \ldots\}$ consists of a sequence of batch updates $\Delta o_t, \Delta o_t = \{(op_1, v_{i_1}, v_{i_2}), (op_2, v_{j_1}, v_{j_2}), \ldots\}$ consists of inserting and deleting edges between time step $t$ and $t - 1$ ($t \geq 1$). $op_t$ can be + or −, indicating inserting or deleting the edge $(v_{i_j}, v_{j_k})$ to or from $G'$. We assume that an edge appears at most once in $\Delta o_t$, either inserted or deleted. By applying $\Delta o_1, \Delta o_2, \ldots, \Delta o_t$ to the initial graph $G_0$ of $G'$, we can get the snapshots of $G'$ $G'_1, G'_2, \ldots, G'_t$ in turn. We use $E(G'_t)$ to denote the edge set of $G'_t$ at time step $t$. Fig. 5 shows a dynamic data graph $G'$ and its snapshots, $G'$ is directed. The inserting/deleting edges in $\Delta o_t$ are listed below the arrow. The solid blue edges in $G'_t$ are inserting edges while the faded dotted edges are deleting edges.

We use $R_{G'_t}(P)$ to denote the set of isomorphic subgraphs of $P$ in $G'_t$. The target of continuous subgraph enumeration is to detect changes in $R_{G'_t}(P)$ and report incremental matches $\Delta R^+_t$ and $\Delta R^-_t$ as defined in Definition 4. We assume that the batch size $|\Delta o_t|$ is much smaller than $|E(G'_t)|$. Taking the demo case in Fig. 5 as the example, the output of each time step is shown in two rows.

Table 2 summarizes the frequently used notations in this work.

**Definition 4**. Given a static pattern graph $P$ and a dynamic data graph $G'$, the task of continuous subgraph enumeration is to report appearing matches $\Delta R^+_t = R_{G'_t}(P) \setminus R_{G'_{t-1}}(P)$ and disappearing matches $\Delta R^-_t = R_{G'_{t-1}}(P) \setminus R_{G'_t}(P)$ for every time step $t$ ($t \geq 1$), where $\setminus$ is the set difference operator.

2.2 Symmetry Breaking

A match $f$ of the pattern graph $P$ in the data graph $G$ (snapshot $G_1$) corresponds to a subgraph $g$ isomorphic

| Notation | Description |
|----------|-------------|
| $G, G_1$ | The data graph $G$. If $G$ is dynamic, $G_1$ is the snapshot at time step $t$. |
| $P, n, m$ | The pattern graph $P$. $n = |V(P)|$, $m = |E(P)|$. |
| $\Delta P_i$ | The $i$-th incremental pattern graph of $P$. |
| $\Gamma_g(x), \Gamma_g^+(x), \Gamma_g^-(x)$ | The (incoming/outgoing) adjacency set of the vertex $x$ in the graph $g$. |
| $f = (f_1, \ldots, f_n)$ | A match $f$ of $P$ in $G$. $f_i = f(u_i)$. |
| $R_G(P), R_{G_1}(P)$ | The set of matches of the pattern graph $P$ in the data graph $G$ (snapshot $G_1$). |
| $\Delta R^+_t, \Delta R^-_t$ | The appearing/disappearing matches of the pattern graph at time step $t$. |
to $P$ in $G (G_1)$. However, multiple matches may correspond to the same subgraph due to the automorphism in $P$. In Fig. 1, the match $f' = (v_1, v_2, v_3, v_4, v_5, v_6)$ and $f'' = (v_1, v_4, v_5, v_4, v_3, v_2)$ both correspond to the subgraph $g$ shown with bold lines in $G$. Enumerating all matches of $P$ in $G$ may report duplicate subgraphs.

We adopt the symmetry breaking technique [25] to avoid such duplication. The technique requires a total order $\prec$ defined on $V(G)$. It also imposes a partial order $\prec'$ on $V(P)$. 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 $f(u_i) \prec f(u_j)$ in $V(G)$. Under the new definition, if a subgraph $g$ is isomorphic to $P$, there is one and only one match $f$ of $P$ in $g$ [25]. It establishes a bijective mapping between matches of $P$ in $G$ and isomorphic subgraphs of $P$ in $G$. In Fig. 1, the partial order imposed on $P$ is $v_2 < v_5$. Assuming $v_3 \prec v_5$ in the total order, the subgraph $g$ shown with bold lines in $G$ is isomorphic to $P$ with only one match $f' = (v_1, v_2, v_3, v_4, v_5, v_6)$.

We take advantage of the technique to convert the problem of enumerating subgraphs into enumerating matches. In the following sections, we use matches to represent isomorphic subgraphs interchangeably. For static data graphs $G$, we use the degree-based total order $\prec$ defined in [8]. For dynamic data graphs, we use the natural order of vertex IDs as the total order.

2.3 Backtracking-based Framework

The backtracking-based framework is popular among serial subgraph isomorphism algorithms. It incrementally maps each pattern vertex to data vertices in the match $f$ according to a given matching order $O$. Algorithm 1 shows a simplified version of the original framework [26].

The SUBGRAPHSEARCH procedure finds all the matches of $P$ in $G$ recursively. The NEXTPATTERNVERTEXTOMATCH function returns the next unmapped pattern vertex $u_i$ in $f$ according to the matching order $O$. The REFINECANDIDATES function calculates a candidate set $C_i$ of the data vertices 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 vertices are mapped in $f$. Different algorithms have different implementations for FIRSTPATTERNVERTEXTOMATCH, NEXTPATTERNVERTEXTOMATCH, and REFINECANDIDATES.

3 Batch-BENU Framework

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 [13] that load the whole data graph in memory are not feasible here.

3.1 Framework Overview

The DFS-style distributed subgraph enumeration method [14] 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. Consider 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 try enumerating triangles first and then 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 in Algorithm 1 to enumerate matches. Only when the framework needs to access the data graph in the REFINECANDIDATES function, it queries the database. Once a partial match $f$ fails in the search that generates an empty candidate set for a pattern vertex, the framework skips $f$ and backtracks, not wasting any effort on mapping other pattern vertices in $f$. By this way, the technique avoids querying useless edges. It also avoids shuffling any partial matching result.

Around the on-demand shuffle technique, we develop the Batch-BENU framework (B-BENU, for short) for distributed subgraph enumeration. Algorithm 2 shows its workflow. B-BENU stores the data graph $G$ in a distributed database $DB$ in parallel (Line 1). Given a pattern graph $P$, B-BENU computes its best execution plan $E$ to enumerate the pattern graph $P$ on the master node (Line 2) and broadcasts $E$ and $P$ to worker nodes (Line 3). The execution plan is a core concept in B-BENU. An execution plan follows the backtracking-based framework to enumerate matches of $P$. It gives out the matching order and detailed steps to calculate the candidate set for every pattern vertex. We elaborate on it later. B-BENU generates a local search task for each data vertex $v$ in $V(G)$ (Line 4). $v$ is the starting vertex of the local search task. B-BENU executes all tasks in parallel with a distributed computing platform. A local search task enumerates matches of $P$ in the neighborhood around the starting vertex $start$ (Line 5 to Line 8). It initializes an empty mapping $f$ and maps the first pattern vertex $u_j$ in the matching order to $start$ (Line 6). A local search task then follows the execution plan $E$ to match the remaining pattern vertices in $f$. During the task execution, the execution plan queries the database on demand.

3.2 Execution Plan

The execution plan gives out the detailed steps to enumerate matches of $P$ in $G$. It is the core of the B-BENU framework.

### Algorithm 1 Backtracking-based Framework

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

1. $f \leftarrow$ an empty mapping from $V(P)$ to $V(G)$;
2. $u_i \leftarrow$ FIRSTPATTERNVERTEXTOMATCH($O$);
3. for all $v_j \in V(G)$ do
   4. $f_i \leftarrow v_j$;
   5. SUBGRAPHSEARCH($P, G, O, f$);
4. procedure SUBGRAPHSEARCH($P, G, O, f$);
   6. if all pattern vertices are mapped in $f$ then output $f$;
   7. else
      8. $u_i \leftarrow$ NEXTPATTERNVERTEXTOMATCH($O$);
      9. $C_i \leftarrow$ REFINECANDIDATES($P, G, f, u_i$);
      10. for all $v_k \in C_i$ do
          11. $f_i \leftarrow v_k$;
      12. SUBGRAPHSEARCH($P, G, O, f$);
      13. $f_i \leftarrow$ NULL; \> Make $u_i$ unmapped in $f$;
Since a database querying operation is expensive due to its high latency, the execution plan queries the database on the level of adjacency sets instead of edges, to reduce the number of database operations. The execution plan implements the three core functions in Algorithm 1 as:

1. **FirstPatternVertexToMatch** and NextPatternVertexToMatch: Each execution plan is bound with a static matching order \( O \). The functions return the first pattern vertex in \( O \) that is *unmapped* in the partial match \( f \) as the first/next vertex to match.

2. **RefineCandidates**: The execution plan intersects adjacency sets to calculate the candidate set \( C_j \) for an unmatched pattern vertex \( u_j \). \( \mathcal{N}(u_j) \) is the set of \( u_j \)'s neighbors in \( P \) that are before \( u_j \) in \( O \). \( \mathcal{N}(u_j) = \{ u_x | u_x \in \Gamma_P(u_j), u_x \text{ is before } u_j \text{ in } O \} \). The pattern vertices in \( \mathcal{N}(u_j) \) are already mapped in \( f \) when we calculate \( C_j \). If \( \mathcal{N}(u_j) = \emptyset \), \( C_j = V(G) \). Otherwise, for any vertex \( u_i \in \mathcal{N}(u_j) \), if we map \( u_j \) to \( v_x \) in \( f \), \( v_x \) should be adjacent to \( f_i \) in \( G \). In other words, \( C_j = \bigcap_{u \in \mathcal{N}(u_j)} \Gamma_G(f_i) \). Mapped \( u_j \) to any vertex outside \( C_j \) will violate the match condition \((u_j, u_i) \in E(P) \Rightarrow (f_j, f_i) \in E(G)\). \( C_j \) is further filtered to ensure that the data vertices in it do not violate the injective condition and the partial order constraints.

Example 1. In Fig. 1, assume the matching order is \( O : u_1, u_2, \ldots, u_6 \). Suppose \( u_1 \) and \( u_2 \) are mapped in the partial match \( f = (v_1, v_2, ?, ?, ?, ?) \). \( u_3 \) is the next pattern vertex to match. \( \mathcal{N}(u_3) = \{ u_1, u_2 \} \). The candidate set \( C_3 = \{ v \in \Gamma_G(v_1) \cap \Gamma_G(v_2), v \neq v_1, v \neq v_2 \} = \{ v_3, v_7 \} \).

Integrating the core functions, we can get an abstract execution plan. A demo execution plan for \( P \) in Fig. 1a is shown in Fig. 2a. The Filter operation filters out data vertices not satisfying either the injective condition or the partial order constraints. The demo execution plan is bound with the matching order \( O : u_1, u_3, u_5, u_2, u_6, u_4 \), expressed by the order of loop variables. Each loop corresponds to a recursive search level (Line 11 to 14) in Algorithm 1. For ease of presentation, the recursion is expanded.

### 4 Execution Plan Generation

In this section, we present the method to generate a concrete B-BENU execution plan for a given pattern graph \( P \). For a clear illustration, we use the same running example through the whole section. The pattern graph is Fig. 1a and the matching order is \( O : u_1, u_3, u_5, u_2, u_6, u_4 \). We first introduce how to generate a raw execution plan from a given matching order \( O \).

#### 4.1 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.

##### 4.1.1 Execution Instruction

A B-BENU execution instruction is denoted as

\[
X := \text{Operation}(\text{Operands})[[\text{FCs}]]
\]

It contains three parts: (1) a target variable \( X \) that stores the result of the instruction, (2) an operation \( \text{Operation}(\text{Operands}) \) describing the conducted operation and its operands, and (3) optional filtering conditions \( \text{FCs} \).

There are 6 kinds of execution instructions in B-BENU as listed in Table 3. B-BENU uses two kinds of filtering conditions: (1) a symmetry breaking condition, denoted as \( > f_i \) or \( < f_i \), means that vertices in \( X \) should be bigger or smaller than \( f_i \) under the total order \( \prec \); (2) an injective condition, denoted as \( \neq f_i \), means that \( f_i \) should be excluded from \( X \).

##### 4.1.2 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 \( O \): \( f_{k_1} := \text{Init}(\text{start}) \) and \( A_{k_1} := \text{GetAdj}(f_{k_1}) \). 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:

1. \( T_{k_i} := \text{Intersect}(\ldots) \). 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. \( C_{k_i} := \text{Intersect}(T_{k_i})[[\text{FCs}]] \). 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, the corresponding 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. If \( u_j \) and \( u_{k_i} \) are adjacent, the injective condition can be omitted, since \( T_{k_i} \subseteq A_j, f_j \notin A_j \) and thus \( f_j \notin T_{k_i} \).

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

4. \( A_{k_i} := \text{GetAdj}(f_{k_i}) \). 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.

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

After generating instructions, we conduct the uni-operand elimination. If an INT instruction has only one operand and no filtering condition like \( T_i := \text{Intersect}(X) \), we remove the instruction and replace \( T_i \) with \( X \) in the other instructions. If an INT instruction generates a candidate set \( C_x \)
and \( C_x \) will be output by the VCBC compression technique (introduced later in Section 4.2.4), we do not eliminate the instruction. After eliminating all uni-operand instructions, we get the raw execution plan.

The raw execution plan is well-defined. All the variables are defined before used. It materializes the abstract execution plan as shown in Fig. 2a. It can be converted to the actual code or be executed by an interpreter easily.

**Example 2.** Fig. 2b shows the raw execution plan generated for the running example. The instructions generated for \( u_4 \) are the 14th to 16th instruction.

B-BENU supports integrating other filtering techniques like the 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 overheads.

### 4.2 Execution Plan Optimization

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

**4.2.1 Opt1: Common Subexpression Elimination**

We borrow the concept of common subexpression from the programming analysis. Some combinations of adjacency sets appear as operands in more than one INT instruction. For example, the common subexpression \( \{A_1, A_3\} \) appears twice in the raw execution plan in Fig. 2b. We should eliminate it as it brings 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 adjacent sets. We pick the subexpression with the most adjacent sets.

**Example 3.** In Fig. 2b, \( \{A_1, A_3\} \) and \( \{A_1, A_5\} \) are both common subexpressions. According to the order, we pick \( \{A_1, A_3\} \) to eliminate. After replacing it with \( T_7 \) in Fig. 2c, there is no other
common subexpression and the optimization stops.

4.2.2 Opt2: Instruction Reordering

The position of the 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. To optimize instruction positions, we reorder instructions in an execution plan with three steps.

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

Second, construct the 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 \( I_1 \rightarrow I_2 \). \( I_1 \) should always be before \( I_2 \), otherwise \( I_2 \) will use an undefined variable. We construct a dependency graph to describe such dependency relations. In the graph, instructions are vertices, and dependency relations are directed edges. For example, Fig. 3 is the dependency graph of the execution plan in Fig. 2c. In Fig. 3, we use the target variable to represent an instruction and we omit the RES instruction.

Third, reorder instructions. We reorder the instructions by conducting topological sorting on the dependency graph. The topological sort guarantees that the dependency relations between instructions are not violated. During the sorting, it is common that several instructions can all be the candidate instructions for the next instruction. For example, in Fig. 3, after sorting the first three instructions \([f_1, A_1, f_3]\), 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 wasting efforts on a doomed-to-fail partial match. The TRC instructions involve cache accessing. The DBQ instructions conduct database operations that are much more expensive than computation. The ENU instructions are the most expensive as they add a level in the backtracking search and make the following instructions executed for more time. We want to postpone them as much as possible. The relative order of DBQ and ENU instructions reflects the matching order. The ranking method also guarantees that the relative order is not changed.

Example 4. For the execution plan in Fig. 2c with its dependency graph in Fig. 3, we can get a reordered execution plan in Fig. 2d. The 14th instruction in Fig. 2c is moved forward, crossing two ENU instructions \( f_2 \) and \( f_6 \).

4.2.3 Opt3: Triangle Caching

Suppose \( u_{k_1} \) is the first vertex in the matching order. If \( u_j \) is a neighbor of \( u_{k_1} \) in the pattern graph \( P \), then \( f_{k_1} \) and \( f_j \) are neighbors in the data graph. The INT instruction \( X := \text{Intersect}(A_{k_1}, A_j) \) 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 starting vertex \( f_{k_1} \). For example, in Fig. 2d, \( T_7 := \text{Intersect}(A_1, A_8) \) and \( T_8 := \text{Intersect}(A_1, A_7) \) both enumerate triangles around the starting vertex \( f_1 \). Their computation is redundant. The existing methods [8] and [9] 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 non-trivial disk space to store.

In B-BENU, we propose the triangle caching technique to reduce such redundancy on the fly. We set up a triangle cache for each local search task to cache the locally enumerated triangles. For an INT instruction \( X := \text{Intersect}(A_i, A_j) \), if one of \( f_i \) and \( f_j \) is the starting vertex and the other one is its neighbor, we replace the INT instruction with a triangle caching instruction: \( X := \text{TCache}(f_i, f_j, A_i, A_j) \). 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 into the cache. Otherwise, it uses the pre-computed set in the cache as the result.

Example 5. In Fig. 2d, the marked instructions are replaced by the triangle caching instructions in Fig. 2c.

4.2.4 Support VCBC Compression

The VCBC compression (vertex-cover based compression) [9] is an efficient technique to compress the subgraph matching results based on a vertex cover \( V_c \) of \( P \). Given a pattern graph \( P \) and its vertex cover \( V_c \), \( \text{core}(P) \) is the induced subgraph of \( P \) on \( V_c \). In VCBC, the matches of \( \text{core}(P) \) in \( G \) are halves. For each helve, the matches of the pattern vertices not in \( V_c \) are compressed in conditional image sets. The halves and their conditional image sets form the compressed codes of the matching results of \( P \) in \( G \).

With modification, a B-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 \( O \) forms a vertex cover \( V_c \) of \( P \) while the first \( k - 1 \) vertices do not. The matches of the first \( k \) pattern vertices are the halves. 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 $C_j$ for $u_j$. $C_j$ is equal to the conditional image set of $u_j$ in the VCBC compression. We replace $f_j$ in the RES instruction with $C_j$ to directly output the compressed codes.

**Example 6.** The execution plan in Fig. 2e can be modified to Fig. 2f to support the VCBC compression. The first three vertices $[u_1, u_3, u_5]$ in $O$ form the vertex cover $V_c$. 

### 4.2.5 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 common subexpressions in that instruction is $O(2^n)$. Since there are $O(n)$ INT instructions, the complexity of enumerating common subexpressions in all instructions is $O(n2^n)$. The complexity of eliminating a subexpression is $O(n^2)$. The elimination will repeat $O(n)$ times until there is no common subexpression. The worst-case time complexity of Optimization 1 is $O(n2^n)$.

As for Optimization 2, the execution plan after flattening has $O(m)$ instructions. Each instruction has at most 2 operands and $n$ injective conditions, so the dependency graph has $O(m)$ vertices and $O(nm)$ edges. The complexity of topological sort is $O(nm)$. If we use a heap to find the next instruction with the highest rank, the maintenance cost of topological sort is $O(n^2)$. The elimination will repeat $O(n)$ times until there is no common subexpression. The worst-case time complexity of Optimization 2 is $O(nm)$.

The costs of Optimization 3 and supporting VCBC compression are both linear to the number of instructions in the execution plan, which is $O(n)$. Summarily, the computation complexity of the whole optimization is $O(n2^n)$, dominated by Optimization 1.

### 4.3 Best Execution Plan Generation

Given a pattern graph $P$, any permutation of pattern vertices is a legal matching order. Different matching orders correspond to different execution plans, having different execution costs. We propose a search-based method to generate the best execution plan for a pattern graph.

#### 4.3.1 Execution Cost Estimation

The execution cost of an execution plan $E$ is made up of the computation cost and the communication cost. We define the computation cost as the number of executed times of all INT/TRC instructions. We define the communication cost as the number of executed times of all DBQ instructions. Thus, the core problem in estimating execution costs is to estimate numbers of executed times of instructions.

For an instruction, its number of executed times is related to the ENU instructions before it. 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 first $i$ vertices in $O$ as the partial pattern graph $P_i$. The leftmost column in Fig. 4 shows the partial pattern graphs $P_i$ with the corresponding ENU instructions. The pattern graph used in Fig. 4 is Fig. 1a. The first $i$ ENU instructions actually enumerates matches of $P_i$ in $G$. Thus, the number of executed times of the $i$-th ENU instruction is equal to the number of matches of $P_i$ in $G$. The instructions between the $i$-th and $i + 1$-th ENU instructions have the same number of executed times as the $i$-th ENU instruction.

We develop the ESTIMATECOMPUTATIONCOST function in Algorithm 3 to estimate the computation cost of an execution plan $E$. The function tracks the partial pattern graph $P'$ as scanning instructions and uses the estimation model proposed in Section 5.1 of [8] to estimate the number of matches of $P'$. If $P'$ is disconnected, we decompose it into connected components and multiply the numbers of their matches together. The estimation model can be replaced if a more accurate model is proposed later.

#### 4.3.2 Best Execution Plan Search

We define the best execution plan as the execution plan with the least communication cost, since executing a DBQ instruction consumes much more time than an INT/TRC instruction. If several execution plans have the same least communication cost, we define the one with the least computation cost as the best.

We propose a search-based algorithm (Algorithm 3) to find the best execution plan $E_{best}$ for a given pattern graph $P$. The communication cost of an execution plan is determined by the relative order of DBQ and ENU instructions. As the optimizations in Section 4.2 do not affect the relative order, the communication cost is solely determined by the matching order. Thus, Algorithm 3 calls the SEARCH procedure to find the set of candidate matching orders $O_{cand}$ that have the least communication cost without actually generating execution plans. For matching orders in $O_{cand}$, the algorithm generates optimized execution plans and picks the one with the least computation cost as $E_{best}$.

The SEARCH procedure uses backtracking to iterate all permutations of pattern vertices. It maintains the unused pattern vertices in $C$ and recursively enumerates the next pattern vertex in the partial matching order $O$ from $C$ one by one. To avoid blindly exploring all permutations, 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 [27] between pattern vertices. For two vertices $u_i$ and $u_j$ in $P$, $u_i$ is SE to $u_j$ (denoted as $u_i \simeq u_j$) if and only if $\Gamma_P(u_i) - \{u_j\} = \Gamma_P(u_j) - \{u_i\}$. For example, in q4 of Fig. 8,
$u_1 \simeq u_4$ 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 execution 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 in Section 4.1.2. 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 keep the matching order that $u_j$ appears before $u_i$.

Cost-based Pruning: When Algorithm 3 searches candidate matching orders, it maintains the current partial pattern graph $P'$ and the partial communication cost $commCost'$ simultaneously in Line 13 to Line 18. 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 4.1, a DBQ instruction will be generated for $u$. The execution times of the instruction are equal to the number of matches of $P'$. In case 2, no DBQ instruction will be generated. The partial communication cost remains unchanged. If the partial communication cost is already bigger than the current best cost, $O$ and all the orders expanded from $O$ can be pruned safely.

The time complexity of the search procedure is dominated by the estimation operation in Line 15. The complexity of the operation is $O(m)$ and we denote its executed times as $\alpha$. The time complexity of Line 4 to Line 7 is dominated by the partial communication cost estimation. The complexity of the operation is $O(n^2\alpha)$ and we denote its executed times as $\beta$. Therefore, the time complexity of Algorithm 3 is $O(nm + \beta n^2\alpha)$. $\alpha$ and $\beta$ are affected by the pattern graph. The upper bound of $\alpha$ is $\sum_{i=1}^{n} P(n, i)$ ($P(n, i)$ is $i$-permutations of $n$). The upper bound of $\beta$ is $n!$. In practice, $\alpha$ and $\beta$ are much less than their upper bounds.

5 Streaming-BENU Framework

When data graphs are dynamic, the subgraph enumeration problem becomes the continuous subgraph enumeration problem. A naive approach to the problem is conducting subgraph enumeration on $G'_t$ and $G'_{t-1}$ separately at each time step $t$ and calculating differences of $R_{G'_t}(P)$ and $R_{G'_{t-1}}(P)$. However, enumerating subgraphs from scratch is expensive and contains redundant computation. To overcome the drawback, we propose the Streaming-BENU framework that enumerates subgraphs in $\Delta R^+_t$ and $\Delta R^-_t$ incrementally from the batch update $\Delta O_t$. For a clear illustration, we use the same example in Fig. 5 through the section.

5.1 Incremental Subgraph Matching

Given a data graph $G'_t$ and a time step $t (t \geq 1)$, $G'_{t-1}$ and $G'_t$ are the snapshots related to $t$. We can classify the edges $e$ of $G'_{t-1}$ and $G'_t$ into two types: 1) $e$ is a delta edge if $e \in E(G'_t) \setminus E(G'_{t-1})$ (inserting edge) or $e \in E(G'_t) \setminus E(G'_{t-1})$ (deleting edge); 2) $e$ is an unaltered edge if $e \in E(G'_t) \cap E(G'_{t-1})$. For the toy case in Fig. 5 at time step 2, $(v_1, v_4)$ is an inserting edge in $G'_2$, $(v_1, v_3)$ is a deleting edge in $G'_1$, and $(v_4, v_1)$ is an unaltered edge.

As stated by Theorem 1, any subgraph in the appearing matches $\Delta R^+_t$ or the disappearing matches $\Delta R^-_t$ must contain a delta edge. It indicates that we only need to enumerate isomorphic subgraphs of $P$ that contain at least a delta edge. Since $|\Delta O_t| \ll |E(G'_t)|$, the number of such subgraphs are much less than $|R_{G'_t}(P)|$.

Theorem 1. For any $g \in \Delta R^+_t$, $g$ contains an inserting edge. For any $g \in \Delta R^-_t$, $g$ contains a deleting edge.

Proof. We proof the theorem by contradiction. For any $g \in \Delta R^+_t$, we assume that $g$ does not contain any inserting edge. Since $g$ is an isomorphic subgraph of $P$ in $G'_t$, $g$ does not contain any deleting edge. $g$ only contains unaltered edges. $g$ is also a subgraph of $G'_{t-1}$, $g \in R_{G'_{t-1}}(P)$. It is inconsistent with $g \in \Delta R^+_t$ (i.e. $R_{G'_t}(P) \setminus R_{G'_{t-1}}(P)$). Therefore, for any $g \in \Delta R^+_t$, $g$ contains at least one inserting edge. The proof for $g \in \Delta R^-_t$ is similar.

Algorithm 3 Best Execution Plan Generation

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

1. $E_{best} \leftarrow$ NULL; $O_{and} \leftarrow \{\}$; ▷ Global variables
2. $bCommCost \leftarrow +\infty; bCompCost \leftarrow +\infty$; ▷ Best costs
3. $SEARCH(0, V(P))$, new PartialPatternGraph(), [], 0); ▷ $O_{and}$ is updated in SEARCH
4. for all $O \in O_{and}$ do
5. $E \leftarrow GENERATEOPTIMIZEDEXECUTIONPLAN(P, O)$;
6. $cost \leftarrow ESTIMATECOMPUTATIONCOST(P, E)$;
7. if $cost < bCommCost$ then $E_{best} \leftarrow E$, $bCompCost \leftarrow cost$
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 $u \in C$ passing dual condition checking do
12. $O[i] \leftarrow u; C' \leftarrow C \setminus \{u\}$;
13. $p' \leftarrow$ Add $u$ to the partial pattern graph $p$;
14. if $\Gamma_p(u) \cap C \neq \emptyset$ then ▷ Case 1
15. $s \leftarrow$ Estimate the number of matches of $p'$;
16. else ▷ Case 2
17. $s \leftarrow 0$;
18. $commCost' \leftarrow 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 $bCommCost \leftarrow commCost; O_{and} \leftarrow \{O\}$;
23. else if $commCost = bCommCost$ then $O_{and} \leftarrow O_{and} \cup \{O\}$;
24. function $ESTIMATECOMPUTATIONCOST(P, E)$
25. $cost \leftarrow 0; curNum \leftarrow 0; p' \leftarrow$ new PartialPatternGraph();
26. for all instruction $I \in E$ do
27. if $I$ type is ENU then
28. Update $p'$ with $I$;
29. $curNum \leftarrow$ estimate the number of matches of $p'$;
30. else if $I$ type is INT or TRC then
31. $cost \leftarrow cost + curNum$;
32. return $cost$. 

$E(G'_{t-1})$ (inserting edge) or $e \in E(G'_{t-1}) \setminus E(G'_t)$ (deleting edge); $2) e$ is an unaltered edge if $e \in E(G'_t) \cap E(G'_{t-1})$. For the toy case in Fig. 5 at time step 2, $(v_1, v_4)$ is an inserting edge in $G'_2$, $(v_1, v_3)$ is a deleting edge in $G'_1$, and $(v_4, v_1)$ is an unaltered edge.

 Proof. We proof the theorem by contradiction. For any $g \in \Delta R^+_t$, we assume that $g$ does not contain any inserting edge. Since $g$ is an isomorphic subgraph of $P$ in $G'_t$, $g$ does not contain any deleting edge. $g$ only contains unaltered edges. $g$ is also a subgraph of $G'_{t-1}$, $g \in R_{G'_{t-1}}(P)$. It is inconsistent with $g \in \Delta R^+_t$ (i.e. $R_{G'_t}(P) \setminus R_{G'_{t-1}}(P)$). Therefore, for any $g \in \Delta R^+_t$, $g$ contains at least one inserting edge. The proof for $g \in \Delta R^-_t$ is similar.
5.1.1 Incremental Pattern Graph and Its Match
To find the isomorphic subgraphs with delta edges, we extend a pattern graph $P$ with $m$ edges into $m$ incremental pattern graphs $\Delta P_i$ ($1 \leq i \leq m$) as defined in Definition 5. We number edges of $P$ consecutively. Edge IDs are necessary to define the edge type mapping $\tau_i$ of $\Delta P_i$; $\tau_i$ assigns every edge of $P$ to one of three types. Fig. 5 shows the incremental pattern graphs of $P$. $\tau_i$ is illustrated with edge colors.

**Definition 5 (Incremental Pattern Graph).** Given a pattern graph $P$ with its edges numbered consecutively as $E(P) = \{e_1^P, e_2^P, \ldots, e_m^P\}$, $P$ has $m$ incremental pattern graphs. The $i$-th incremental pattern graph (denoted as $\Delta P_i$) is a graph $\Delta P_i = (V(P), E(P), \tau_i)$, where $\tau_i : E(P) \to \{\text{either, delta, unaltered}\}$ is an edge type mapping.

$$\tau_i(e_k^P) = \begin{cases} 
\text{either} & 1 \leq k < i \\
\text{delta} & k = i \\
\text{unaltered} & i < k \leq m 
\end{cases}$$

Definition 6 defines the incremental match of an incremental pattern graph in $G'_t$ and $G'_{t-1}$ for every time step $t$. An incremental match $f$ is a match of $P$ in the snapshot, but $f$ has type constraints on the data edges that a pattern edge can map to. Definition 7 defines the isomorphic subgraph of an incremental pattern graph. It is easy to see that a subgraph isomorphic to an incremental pattern graph is also isomorphic to the pattern graph.

**Definition 6 (Incremental Match).** Given a dynamic data graph $G'$, a pattern graph $P$, a time step $t$, and an incremental pattern graph $\Delta P_t$, a mapping $f : V(P) \to V(G')$ is an incremental match of $\Delta P_t$ in $G'_t (G'_{t-1})$ if and only if $f$ satisfies:

1) $f$ is a match of $P$ in $G'_t (G'_{t-1})$ satisfying the partial order constraints;
2) For every edge $e_P = (s, t) \in E(P)$ and the data edge that $e_P$ is mapped to $e_G = (f(s), f(t))$;
   - If $\tau_i(e_P) = \text{either}$, $e_G$ is an edge of $G'_t (G'_{t-1})$;
   - If $\tau_i(e_P) = \text{delta}$, $e_G$ is a delta edge of $G'_t (G'_{t-1})$;
   - If $\tau_i(e_P) = \text{unaltered}$, $e_G$ is an unaltered edge of $G'_t (G'_{t-1})$.

**Definition 7.** Given a dynamic data graph $G'$, a pattern graph $P$, a time step $t$, and an incremental pattern graph $\Delta P_t$, a subgraph $g$ of $G'_t (G'_{t-1})$ is isomorphic to $\Delta P_t$ if and only if $f$ exists an incremental match $f$ of $\Delta P_t$ in $g$, $|V(P)| = |V(g)|$ and $|E(P)| = |E(g)|$.

**Example 7.** For $t = 2$ in Fig. 5, $G'_2$ and $G'_1$ are related snapshots. $f_1/f_2/f_3$ is an incremental match of $\Delta P_1/\Delta P_2/\Delta P_3$ in $G'_2$, respectively. $f'_2$ is an incremental match of $\Delta P_2$ in $G'_1$. Their corresponding subgraphs in $G'_2$ and $G'_1$ are marked on the left. In $f_2$, $e_2^P$ is mapped to a delta edge $(v_1, v_2)$ of $G'_2$.

5.1.2 Continuous Subgraph Enumeration via Subgraph Enumeration
Given a dynamic graph $G'$, a pattern graph $P$ and a time step $t$, we denote the set of subgraphs isomorphic to $\Delta P_t$ in $G'_t/G'_{t-1}$ as $\Delta R_t^+/\Delta R_t^-$, respectively. $\Delta R_t^+$ and $\Delta R_t^-$ have a strong connection with the output of continuous subgraph enumeration at each time step $t$.

**Theorem 2.** For any $1 \leq i \leq |E(P)|$, if $g \in \Delta R_t^+$, then $g \in \Delta R_{t+1}^+$; if $g \in \Delta R_t^-$, then $g \in \Delta R_{t+1}^-$. Proof.

To prove Theorem 2, there is an incremental match $f$ of $\Delta P_t$ in $g$ according to Definition 7. For the $i$-th edge $e_P^P = (s, t) \in P$, $\tau_i(e_P^P) = \text{delta}$. According to Definition 6, $f$ maps $e_P^P$ to a delta edge $(f(s), f(t)) \in E(G'_t) \setminus E(G'_{t-1})$. For any $g \notin R_{G'_{t-1}}(P)$, since $g$ is isomorphic to $\Delta P_t$ in $G'_t$, $g$ is also isomorphic to $P$ in $G'_t$. Thus, $g \in R_{G'_t}(P)$. According to Definition 4, $g \in \Delta R_t^+$. For any $g \in \Delta R_t^-$, the proof is similar.

**Theorem 3.** For any $g \in \Delta R_t^+$, $\exists i : g \in \Delta R_{t+1}^+$. For any $g \in \Delta R_t^-$, $\exists i : g \in \Delta R_{t+1}^-$. Proof.

To prove Theorem 3, $g$ is isomorphic to $P$ in $G'_t$. According to Definition 2, $|V(P)| = |V(g)|$ and $|E(P)| = |E(g)|$. There is one and only one match $f$ of $P$ in $g$ satisfying the partial order constraint. We number the edges of $g$ according to $P$. For the $k$-th edge of $P$ $e_k^P = (s_k, t_k)$, we number the edge $e_k^g = (f(s_k), f(t_k))$. As $g$ is a subgraph of $G'_t$, the edges of $g$ are also classified as the
Proof. According to Definition 6, $\exists f \in \mathcal{P}$, we can get the matching results of the time step $t$. S-BENU outputs $\Delta R_i^{t+}$ and $\Delta R_i^{t-}$ at each time step.

Some existing continuous subgraph enumeration methods [22] [23] [28] [24] maintain the (partial) matching results of each time step in memory or on disk. They use the matching results of the time step $t$ to compute the matching results of the time step $t+1$, avoiding re-computing some intermediate results. However, maintaining matching results is only feasible when the pattern graph is highly selective [22] and the size of matching results is not big. If the data graph is big, the size of the matching results will become considerable as indicated by Table 1. Moreover, users may monitor multiple pattern graphs simultaneously, multiplying the storage cost. In S-BENU, we choose not to maintain any matching result. Instead, S-BENU only stores the data graph in a distributed file system and shares the data graph among all pattern graphs. S-BENU can store matching results in a distributed file system when it is needed.

Algorithm 4 presents the pseudo-code of the S-BENU framework. S-BENU consists of two phases.

The initialization phase is conducted once for $G'$. S-BENU stores the initial graph of $G'$ into a distributed key-value database $DB$ in parallel. For a vertex $v$, the key is its ID and the value is a tuple of its incoming and outgoing adjacency sets. We will elaborate on the structure of the tuple later in Section 6.2. The edges in the pattern graph $P$ are numbered. S-BENU generates the best incremental execution plans $\mathcal{E} = \{E_1, E_2, \ldots, E_n\}$ for every incremental pattern graph $\Delta P_i$ ($1 \leq i \leq |E(P)|$). $P$ and $\mathcal{E}$ are then broadcasted.

The continuous enumeration phase is conducted repeatedly for every time step $t$. It consists of three sub-phases: pre-processing (lines 7-9), enumeration (lines 10-20) and post-processing (line 21).
Algorithm 4 Streaming-BENU Framework

Input: Pattern graph $P$, Data graph $G'$, Distributed database $DB$.
Output: $\Delta R_i^+, \Delta R_i^-$ for each time step $i$.

1: Store the initial graph of $G'$ into $DB$; \textbf{Phase I: Initialization}
2: $E \leftarrow$ GENERATEBESTEXECUTIONPLANS($P$);
3: Broadcast $P$ and $E$ to all worker nodes;
4: $t \leftarrow 0$;
5: \textbf{loop} \hspace{1em} \textbf{Phase II: Continuous enumeration}
6: \hspace{1em} $t \leftarrow t + 1$; \hspace{1em} $t$ is the current time step
7: \hspace{1em} Get the batch update $\Delta_0$ of $G'$
8: \hspace{1em} Convert $\Delta_0$ into delta adjacency sets;
9: \hspace{1em} Store delta adjacency sets into $DB$;
10: \hspace{1em} \textbf{for all} start $\in \{v|\Delta \Gamma_i^ \text{in}(v) \neq \emptyset\} $ \textbf{do} in parallel
11: \hspace{2em} \textbf{for all} $E_i \in E$ \textbf{do} \textbf{execute} $E_i$
12: \hspace{3em} $f \leftarrow$ an empty mapping from $V(P)$ to $V(G')$;
13: \hspace{3em} /\ Denote the i-th edge of $P$ as $e_i^P = (u_{i_1}, u_{i_2})$
14: \hspace{3em} $f(u_{i_1}) \leftarrow$ start;
15: \hspace{3em} \textbf{for all} $(op, v_p) \in \text{Filter}(\Delta \Gamma_i^ \text{out}(f(u_{i_1})))$ \textbf{do}
16: \hspace{4em} $f(u_{i_2}) \leftarrow v_p$;
17: \hspace{4em} \textbf{if} $op = +$ \textbf{then} \hspace{2em} \textbf{Enumerate matches in} $\Delta R_i^+$
18: \hspace{3em} Match remaining vertices in $f$ in $G'_i$
19: \hspace{3em} \textbf{else} \hspace{2em} \textbf{Enumerate matches in} $\Delta R_i^-$
20: \hspace{3em} Match remaining vertices in $f$ in $G'_{i-1}$;
21: \hspace{1em} \textbf{Merge} adjacency sets in $DB$ with delta adjacency sets;

In pre-processing, S-BENU gets $\Delta_0$ from an external data source like a message queue or a file. S-BENU converts $\Delta_0$ into delta adjacency sets in parallel. For a vertex $v$, its delta adjacency sets are $\Delta \Gamma_i^ \text{in}(v) = \{(op, w)| (op, w, v) \in \Delta_0\}$ and $\Delta \Gamma_i^ \text{out}(v) = \{(op, w)| (op, v, w) \in \Delta_0\}$. S-BENU only generates delta adjacency sets for the vertices appearing in $\Delta_0$.

In enumeration, S-BENU generates a local search task for each vertex $v$ of $\Delta \Gamma_i^ \text{in}(v)$ that has a non-empty delta outgoing adjacency set. The $\text{start}$ is the starting vertex of the local search task. S-BENU executes local search tasks in parallel in a distributed computing platform (line 10). In every task, S-BENU executes incremental execution plans $E_i$ one by one (line 11). $E_i$ searches incremental matches of $\Delta P_i$ in both $G'_i$ and $G'_{i-1}$ (lines 12 - 20). We elaborate on it later in Section 5.3.

In post-processing, S-BENU fetches adjacency sets $\Gamma_i^ \text{in/out}(v)$ of the vertices appearing in $\Delta_0$ from $DB$, merges them with $\Delta \Gamma_i^ \text{out}(v)$, and stores $\Gamma_i^ \text{in/out}(v)$ back to $DB$ in parallel (line 21).

5.3 Incremental Execution Plan

The incremental execution plan is the core of S-BENU. The $i$-th incremental execution plan $E_i$ gives out the detailed steps to enumerate incremental matches of $\Delta P_i$ in both $G'_i$ and $G'_{i-1}$.

5.3.1 Abstract Plan

$E_i$ implements the three core functions in Algorithm 1 as the following.

\textbf{First (Next) Pattern Vertex To Match}: $E_i$ is bound with a static matching order $O_i$. Similar to B-BENU, the function returns the first pattern vertex in $O_i$ that is unmapped in the partial incremental match $f$. S-BENU has an extra constraint on $O_i$. Suppose the $i$-th edge of $P$ is $e_i^P = (u_{i_1}, u_{i_2})$. The first two pattern vertices in $O_i$ must be $u_{i_1}$ and $u_{i_2}$. S-BENU relies on them to determine which snapshot $f$ belongs to. As $\tau_i(e_i^P)$ = delta in $\Delta P_i$, $f$ must map $e_i^P$ to a delta edge $e_i^{G'}$. If $e_i^{G'}$ is an inserting edge (op = +), $e_i^{G'} \in E(G'_i)$ and $f$ should be an incremental match of $\Delta P_i$ in $G'_i$. Otherwise, $e_i^{G'}$ is a deleting edge (op = -) that belongs to $G'_{i-1}$. In this case, $f$ is an incremental match in $G'_{i-1}$.

\textbf{Refine Candidates}: Similar to B-BENU, S-BENU also calculates candidate set $C_j$ for $u_j$ by intersecting adjacency sets of already mapped vertices. Since edges of $\Delta P_i$ have three types (either, delta, unaltered) and two directions(in, out), there are six kinds of adjacency sets. In the snapshot $G'_i$ (can be $t$ or $t - 1$), the incoming adjacency sets of $v$ are $\Gamma_i^{\text{in/ou}(v)} = \{v_x|(v_x, v) \in E(G'_i)|, \Gamma_i^{\text{un/alt}(v)} = \{v_x|(v_x, v) \in E(G'_i), \Gamma_i^{\text{alt}(v)} = \{v_x|(v_x, v) \in E(G'_i), (v_x, v)$ is an unaltered edge). The definitions of outgoing adjacency sets are similar. Is determined by $op$. If $op = +$, $f$ is in $G'_i$ and $is t$. Otherwise, $f$ is in $G'_{i-1}$.

We use $N_i^{\text{in/out}(u_j)}$ to denote the set of $u_j$'s incoming/outgoing neighbors in $P$ that are before $u_j$ in $O_i$, i.e. $N_i^{\text{in/out}(u_j)} = \{v_x|v_x \in \Gamma_i^{\text{in/out}(u_j)}, v_x$ is before $u_j$ in $O_i\}$. The pattern vertices in $N_i^{\text{in/out}(u_j)}$ are already mapped in $f$ when we calculate $C_j$. If we want to map $u_j$ to $v$ in $f$, $v$ should satisfy two conditions: 1) for any mapped incoming neighbor $u_x \in N_i^{\text{in/out}(u_j)}$, $v \in \Gamma_i^{\text{in/out}(u_x, u_j)}$. 2) for any mapped outgoing neighbor $u_x \in N_i^{\text{in/out}(u_j)}$, $v \in \Gamma_i^{\text{out}(u_x, u_j)}$. Based on the conditions, the candidate set $C_j$ is

$$C_j = \{v_x \in N_i^{\text{in/out}(u_j)} | \Gamma_i^{\text{in/out}(u_x, u_j)} \in \{f_x\} \cap \{v_x \in N_i^{\text{in/out}(u_j)} | \Gamma_i^{\text{out}(u_x, u_j)} \in \{f_x\} \} \}.$$ 

Mapping $u_j$ to any vertex outside $C_j$ will violate Definition 6. $C_j$ is further filtered to ensure that data vertices in it do not violate injective conditions and partial order constraints.

Example 8. Take the toy case in Fig. 5 with $\Delta P_2$ and $t = 2$ as the example. Since $e_2^P = (v_1, u_3)$, $O_2$ must be $u_1, u_2, u_3$. Suppose $u_2$ is unmapped in the partial match $f = (v_1, ?, v_4)$. $f$ maps $e_2^P$ to an inserting edge $(v_1, v_4)$ with $op = +$. Thus, we use adjacency sets from $G'_2$ to calculate $C_2$. $\Gamma_i^{\text{in/out}(u_2)} = \{u_3\}$ and $\Gamma_i^{\text{out}(u_2)} = \{u_3\}$. $C_2 = \{v_x | \Gamma_i^{\text{in/out}(u_2)} \in \{f_x\} \cap \{v_x \in N_i^{\text{out}(u_2)} | \Gamma_i^{\text{out}(u_2)} \in \{f_x\} \} \} = \{v_3\}, v_3 \in \{v_3\}$, $f = (v_1, v_6, v_4)$ is an incremental match of $\Delta P_2$ in $G_2$.

Integrating the core functions, we can get an abstract incremental execution plan. Fig. 6a shows the abstract plan of $\Delta P_2$ in Fig. 5. Line 1 to line 3 in Fig. 6a correspond to line 12 to line 16 in Algorithm 4. op of the delta edge is retrieved simultaneously during the mapping of the second pattern vertex $v_3$ in $O_2$. Line 4 to line 6 in the abstract plan correspond to line 17 to line 20 in Algorithm 4.

5.3.2 Concrete Plan

We materialize the abstract incremental execution plan into a concrete one with seven kinds of execution instructions
listed in Table 3. S-BENU inherits the INI, INT, ENU, and RES instructions from B-BENU, but it modifies the DBQ instruction and adds the Delta-ENU and INS instructions.

The modified DBQ instruction adds three extra parameters as $X := \text{GetAdj}(f_1, \text{type}, \text{direction}, \text{op})$. It fetches the adjacency set $I'_{\text{type, direction}}(f_1)$ from the distributed database, where $\text{type}$ can be either $\text{delta}/\text{unaltered}$, and $\text{direction}$ can be $\text{in}/\text{out}$. If $\text{op} = +$ (or $-$), the instruction gets the adjacency set $I'_{\text{type, direction}}(f_1)$. If $\text{type} = \text{delta}$ and $\text{op} = \ast$, the instruction gets the delta adjacency set $\Delta G_{t}^{\text{direction}}(f_i)$ of the current time step $t$. Vertices in the delta adjacency set are attached with flags $(+ \text{ or } -)$. To reference adjacency sets consistently in the plan, the target variable $X$ conforms to a special naming convention. The name of $X$ consists of three letters and a subscript. The first letter is always $A$, representing adjacency sets. The second letter can be $E/D/U$, depending on $\text{type}$ (Either/Delta/Unaltered). The third letter can be $I/O$, depending on direction (In/Out).

The subscript is the index of the operand $f_i$. For example, $\text{ADO}_k := \text{GetAdj}(f_3, \text{delta, out}, \ast)$ fetches $\Delta G_{t}^{\text{out}}(f_3)$ from the database. Taking the case in Fig. 5 with $t = 2$ and $f_3 = v_1$ as the example, $\Delta G_{t}^{\text{out}}(f_3) = \{(−, v_2), (−, v_3), (+, v_4)\}$.

S-BENU extends the ENU instruction to the Delta-ENU instruction $\text{op, } f_1 \Rightarrow \text{Foreach}(X)$. The instruction requires $X$ to be a (filtered) delta adjacency set. The instruction retrieves $\text{op}$ and $f_1$ simultaneously while it iterates $X$.

Given $\Delta P_i$, to materialize its abstract plan, S-BENU generates instructions for each pattern vertex following a given matching order $O_i : u_{k_1}, u_{k_2}, \ldots, u_{k_n}$. For the first two pattern vertices in $O_i$, S-BENU generates four instructions consequently to implement lines 12 to 16 of Algorithm 4:

1) $f_{k_1} := \text{Init}(\text{start})$.
2) $\text{ADO}_{k_1} := \text{GetAdj}(f_{k_1}, \text{delta, out}, \ast)$.
3) $C_{k_2} := \text{Intersect}(\text{ADO}_{k_1})[\text{FCs}]$.
4) $\text{op, } f_{k_2} := \text{Foreach}(C_{k_2})$.

S-BENU then generates DBQ instructions to fetch the $\{\text{either, unaltered} \times \{\text{in, out}\}$ adjacency sets of $u_{k_1}$ and $u_{k_2}$, in case some INT instructions of other vertices may use them. S-BENU does not need to consider adjacency sets of the type delta, since there is only one delta edge in $\Delta P_i$.

S-BENU only checks the existence of $(f_{k_1}, f_{k_2})$ when it generates $C_{k_2}$. If there is an edge $(u_{k_2}, u_{k_1}) \in E(P)$, S-BENU checks the existence of $(f_{k_2}, f_{k_1})$ in $G_t$ with an extra INS instruction $\text{InSetTest}(f_{k_2}, A'O_{k_2})$, where $\text{set}$ depends on $\tau_i(u_{k_2}, u_{k_1})$. If $f_{k_2}$ is not in the outgoing adjacency set of $f_{k_2}$, S-BENU backtracks and tries to map $u_{k_2}$ to another data vertex in $C_{k_2}$.

For each of the remaining vertices $u_{k_i}$ in $O_i$, S-BENU generates instructions for it in a way similar to B-BENU. $T_{k_i} := \text{Intersect}(\cdots)$ and $C_{k_i} := \text{Intersect}(T_{k_i})[\text{FCs}]$ calculate the candidate set $C_{k_i}$ with related adjacency sets. $f_{k_i} := \text{Foreach}(C_{k_i})$ maps $u_{k_i}$ to the candidate data vertices in $C_{k_i}$, one by one and enters the next level in the backtracking search. S-BENU generates DBQ instructions to fetch the $\{\text{either, unaltered} \times \{\text{in, out}\}$ adjacency sets of $u_{k_i}$.

Finally, S-BENU adds a RES instruction to the execution plan.

After generating all instructions, S-BENU first removes useless DBQ instructions whose target variables are not used by any other INT/INS instruction. Then S-BENU conducts the uni-operand elimination. It removes useless INT instructions without any filtering condition like $T_x := \text{Intersect}(X)$ and replaces $T_x$ with $X$ in other instructions. After uni-operand elimination, S-BENU gets the raw incremental execution plan. The raw plan is well-defined. All variables are defined before used.

Example 9. Fig. 6b materializes the abstract plan of $\Delta P_2$ in Fig. 6a. In Fig. 6b, the instruction 1 to 5 are generated for $u_1$ and $u_3$. Instructions 6 to 7 are generated for $u_2$. Some DBQ/INT instructions are useless, like the ones related to $\text{AE1}/\text{AO2}/\text{C3}/C_2$. They are removed from the raw plan.

5.4 Best Execution Plan Generation

S-BENU optimizes the raw execution plan with the common subexpression elimination and the instruction reordering optimizations as proposed in Section 4.2. We do not adopt the triangle caching optimization in S-BENU because edges are typed and directed in incremental pattern graphs, making it hard to re-use the enumerated triangles.

S-BENU modifies Algorithm 3 to generate the best execution plan for each incremental pattern graph $\Delta P_i$.

Suppose the $i$-th edge of $P$ is $e_i = (u_{s_i}, u_{t_i})$. The first two vertices in candidate matching orders are fixed as $u_{s_i}$ and $u_{t_i}$.

The dual condition in the dual pruning technique is stricter. In $\Delta P_i$, the neighborhood of $u_{z_i}$ is contained by the neighborhood of $u_{y_i}$ if

1) For every $e = (u_{z_i}, u_{x_i}) \in E(P)$ with $u_{z_i} \neq u_{y_i}, e' = (u_{y_i}, u_{y_i}) \in E(P)$ and $\tau_i(e) = \tau_i(e')$;
2) For every $e = (u_{z_i}, u_{x_i}) \in E(P)$ with $u_{z_i} \neq u_{y_i}, e' = (u_{y_i}, u_{z_i}) \in E(P)$ and $\tau_i(e) = \tau_i(e').$

$u_{x_i}$ and $u_{y_i}$ is syntactic equivalent if and only if the neighborhood of $u_{z_i}$ is contained by the neighborhood of $u_{y_i}$ and vice versa.

When S-BENU estimates the number of matching results of partial pattern graphs, S-BENU treats them as undirected graphs and uses the model in [8] to estimate. Though the model is targeted for undirected graphs, we find it good enough in practice to distinguish good matching orders from bad ones on directed graphs. Proposing a more accurate estimation model for incremental pattern graphs is one of our future work.
6 EFFICIENT IMPLEMENTATION

Fig. 7 shows the implementation architecture of B-BENU and S-BENU. During the initialization phase, the data graph \( G \) is stored in a distributed key-value database like HBase. The update stream \( \Delta G \) in S-BENU is got from a file or a message queue. B-BENU/S-BENU generates local search tasks from \( G/\Delta G \) in parallel, respectively. The tasks are executed in a distributed computing platform like Hadoop and Spark. Building upon a distributed computing platform and a distributed key-value database, B-BENU and S-BENU naturally support fault tolerance. We further propose several implementation techniques to increase efficiency.

6.1 Local Database Cache

Inside a local search task, a queried adjacency set tends to be queried again soon by the same task. For example, in the backtracking search trees illustrated in Fig. 4, the adjacency set of \( v_4 \) is queried repeatedly in different search branches in the local search task 1. This kind of locality comes from the backtracking nature of the execution plan. All vertices that a local search task visits are in a local neighborhood around the starting vertex of the task. 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.

Some adjacency sets are queried by many different local search tasks. For example, in Fig. 4, the adjacency set of \( v_4 \) is queried in both Task 1 and Task 2. This kind of inter-task locality comes from the overlaps between local neighborhoods visited by different tasks. Some data vertices, especially high-degree ones, are included in many local neighborhoods. Their adjacency sets are frequently queried by different tasks.

To take advantage of both kinds of locality, we set up an in-memory database cache (DB cache) with configurable capacity in each worker node as shown in Fig. 7. The DB cache stores adjacency sets fetched from the distributed database. The cache captures the intra-task locality by using advanced replacement policies like LRU. It captures the inter-task locality by being shared among all working threads. The cache provides a flexible mechanism to trade memory for the 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 \( C \). 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 \( \gamma^r_G(v) = \{w \in V(G) | \) \( w \) at most \( r \) hops away from \( v \}. \) The size of \( \gamma^r_G(v) \) is \( S^r_G(v) = \sum_{w \in \gamma^r_G(v)} d_G(w) \). For a data graph \( G \), \( H^r_G = \max_{v \in V(G)} S^r_G(v) \) 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 \( C \geq wH^r_G \), 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 \( O : u_{k_1}, u_{k_2}, \ldots, u_{k_m} \). The first \( \alpha \) vertices in \( O \) can cover every edge in \( P \). Thus, matching \( u_{k_{\alpha+1}}, \ldots, u_{k_m} \) does not need to query any adjacency set. Among the first \( \alpha \) vertices, there must exist \( r' (0 \leq r' \leq R) \) and \( \beta \) (\( 1 \leq \beta \leq \alpha \)) that the \( r' \)-hop neighborhood of \( u_{k_\beta} \gamma_{r'}(u_{k_\beta}) \) contains \( \{u_{k_\beta}, u_{k_\beta+1}, \ldots, u_{k_m}\} \). Then, we can split \( O \) into three sections: \( O : u_{k_1}, \ldots, u_{k_\alpha}, u_{k_{\alpha+1}}, \ldots, u_{k_m} \).

The total communication cost of matching \( f_{k_1} \) to \( f_{k_\beta} \) is \( O(\sum_{r=1}^{\alpha} |R_G(P_r)|) \) where \( |R_G(P_r)| \) is the number of matches of the partial pattern graph \( P_r \) in \( G \). If \( f_{k_\beta} \) to \( f_{k_\beta} \) is fixed in \( f_1 \), the number of conducted database queries during matching \( f_{k_\beta+1} \) to \( f_{k_m} \) is at most \( \max_{v \in V(G)} |\gamma^r_G(v)\) because the cache can store all the adjacency sets in \( \gamma^r_G(f_{k_\beta}) \). The total communication cost of matching \( f_{k_{\beta+1}} \) to \( f_{k_m} \) for all the partial matches is \( O(|R_G(P)\max_{v \in V(G)} |\gamma^r_G(v)|) \). Matching the remaining vertices \( f_{k_{\beta+1}} \) to \( f_{k_m} \) does not query any adjacency set. Therefore, the communication upper bound is \( O(\sum_{r=1}^{\alpha} |R_G(P_r)| + |R_G(P)| \max_{v \in V(G)} |\gamma^r_G(v)|) \).

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

In this case, the complexity is independent of the pattern graph.

6.2 Data Graph Storage

B-BENU and S-BENU store adjacency sets of the data graph with key-value pairs. For B-BENU, keys are vertex IDs and values are their adjacency sets. For S-BENU, we notice that it only uses the latest \( \Delta G \) and \( \Delta \) in/out for the current snapshot \( G' \). The second form is \( (\Gamma^\text{in}_{G'}(v), \Gamma^\text{out}_{G'}(v), \emptyset, \emptyset) \). It is used in Line 1 and Line 21 of Algorithm 4 to store the current snapshot \( G' \). The second form is \( (\Gamma^\text{in}_{G'}(v), \Gamma^\text{out}_{G'}(v), \Gamma^\text{in/out}_{G'}(v), \emptyset) \). It is used in Line 9 of Algorithm 4 to store the delta adjacency sets along with the previous snapshot \( G_{t-1} \). With this form, we can retrieve the adjacency sets of both \( G_{t-1} \) and \( G_t \). The two-form design guarantees that we only need to update the vertices appearing in \( \Delta G \) in Line 9 and Line 21. For a vertex \( v \) not appearing in \( \Delta G \), its value is \( (\Gamma^\text{in}_{G_{t-1}}(v), \Gamma^\text{out}_{G_{t-1}}(v), \emptyset, \emptyset) \) before Line 9. Since \( \Delta G^{\text{in/out}}(v) = \emptyset \) and \( \Gamma^\text{in/out}_{G_{t-1}}(v) = \Gamma^\text{in/out}_{G_{t}}(v) \), we do not need to modify its value in either Line 9 or Line 21. As \( |\Delta G| \ll |E(G_t)| \), only a fraction of vertices appear in \( \Delta G \). The two-form design cuts much costs of updating the database.

In the local database cache, B-BENU uses the same key-value format as in the database, but S-BENU uses a different format. For S-BENU, keys are still vertex IDs, but
values are \((T, \Gamma_{G_{t-1}}^{\text{in}}(v), \Gamma_{G_{t-1}}^{\text{out}}(v), \Gamma_{G_t}^{\text{in}}(v), \Gamma_{G_t}^{\text{out}}(v))\). \(T\) is the time step of the key-value pair. Vertices in all adjacency sets are attached with flags, indicating whether the corresponding edge is a delta edge. When a DBQ instruction \(X := \text{GetAdj}(f_i, \text{type}, \text{direction}, \text{op})\) is conducted, the cache hits if the key \(f_i\) is contained in the cache and \(T\) is equivalent to the current time step \(t\). If the cache hits, S-BENU retrieves the corresponding adjacency set based on \(op\) and \(direction\), and S-BENU filters it with \(type\). If the cache misses, S-BENU queries the distributed database for the quad \((\Gamma_{G_{t-1}}^{\text{in}}(v), \Gamma_{G_{t-1}}^{\text{out}}(v), \Delta \Gamma_t^{\text{in}}(v), \Delta \Gamma_t^{\text{out}}(v))\) and constructs the value part from it. The format in the cache trades space for time, because the cache is much more frequently accessed than the database. If we use the same format as in the database, we have to merge \(\Delta \Gamma_t^{\text{in/out}}(v)\) with \(\Gamma_{G_{t-1}}^{\text{in/out}}(v)\) to get \(\Gamma_{G_t}^{\text{in/out}}(v)\). Merging two adjacency sets is more expensive than filtering an adjacency flag with a delta flag.

### 6.3 Task Splitting

The computation and communication costs of a local search task are positively correlated with the degree of the starting vertex. Unfortunately, real-world graphs often follow the power-law degree distribution, causing workloads of local search tasks skewed. We propose the task splitting technique to split heavy tasks into smaller subtasks to balance the workloads. Suppose \(u_{k_1}\) and \(u_{k_2}\) are the first and second pattern vertex in the matching order and \(C_{k_2}\) is the candidate set of \(u_{k_2}\).

In B-BENU, \(u_{k_1}\) is mapped to the starting vertex \(v\) of the local search task. If \(u_{k_1}\) and \(u_{k_2}\) are adjacent in \(P\), \(C_{k_2}\) is the filtered adjacency set of the starting vertex. If the degree \(d_G(v)\) is bigger than a given threshold \(\theta\), we split \(\Gamma(v)\) into \(\left\lfloor \frac{|\Gamma(v)|}{\theta} \right\rfloor\) non-overlapping equal-sized subsets. We generate a subtask for each subset and use the subset as \(C_{k_2}\) in the subtask. If \(u_{k_1}\) and \(u_{k_2}\) are not adjacent, \(C_{k_2}\) is the filtered \(V(G)\), and we generate \(\left\lfloor \frac{|V(G)|}{\theta} \right\rfloor\) subtasks in this case.

In S-BENU, \(u_{k_1}\) is also mapped to the starting vertex \(v\) of the local search task, but \(C_{k_2}\) is the filtered \(\Delta \Gamma_t^{\text{out}}(v)\). If \(|\Delta \Gamma_t^{\text{out}}(v)| \geq \theta\), we split \(\Delta \Gamma_t^{\text{out}}(v)\) into \(\left\lfloor \frac{|\Delta \Gamma_t^{\text{out}}(v)|}{\theta} \right\rfloor\) non-overlapping equal-sized subsets and generate a subtask for each subset.

### 6.4 Implementation Sketch

We implement B-BENU with a Hadoop MapReduce job. The input to the job is the data graph stored as key-value pairs in HDFS. In the map phase, B-BENU stores the data graph into HBase in parallel. B-BENU generates and emits local search (sub)tasks simultaneously. Keys are tasks and values are null. Hadoop shuffles the tasks to reducers. B-BENU runs a reducer on each worker machine. In each reducer, B-BENU uses a thread pool to execute the received tasks concurrently.

Since S-BENU needs to process batch updates \(\Delta \theta_t\) iteratively, we implement S-BENU with a long-running Spark job. S-BENU loads the initial data graph from HDFS as a RDD and stores it into HBase in parallel by conducting \texttt{foreachPartition} operator. In each time step, S-BENU loads \(\Delta \theta_t\) as a RDD from an external data source like HDFS or a message queue. S-BENU converts the \(\Delta \theta_t\) RDD into the delta adjacency set RDD with the \texttt{flatMap} and \texttt{groupByKey} operators. The delta adjacency set RDD is used to update HBase in parallel (Line 9 of Algorithm 4). It is further converted into the local search task RDD. Conducting \texttt{mapPartition} operator on it, S-BENU executes local search tasks in parallel with all executors. The delta adjacency set RDD is used again to update HBase (Line 21 of Algorithm 4).

### 7 Experiments

We introduce the experimental setup in Section 7.1 and then evaluate the effectiveness of the proposed optimization techniques in Section 7.2. The performance of B-BENU and S-BENU is compared with the state-of-the-art in Section 7.3 and Section 7.4. We finally evaluate the machine scalability of B-BENU and S-BENU in Section 7.5.

#### 7.1 Experimental Setup

**Environment.** All the experiments were conducted in a cluster with 1 master + 16 workers connected via 1Gbps Ethernet. Each machine was equipped with 12 cores, 50 Gbytes memory, and 2 Tbytes RAID0 HDD storage. All Java programs were compiled with JDK 1.8 and run under CentOS 7.0. We adopted Hadoop 2.7.2. The distributed database was HBase 1.2.6.

**B-BENU.** B-BENU was implemented with Hadoop MapReduce. B-BENU generated local search tasks in the map phase and executed the tasks in the reduce phase. We used 16 reducers (one reducer per machine). Each reducer ran the local search tasks with 24 working threads. We allocated 40 Gbytes memory to each reducer (30 Gbytes for local database cache and 10 Gbytes for task execution). The degree threshold \(\theta\) of task splitting was 500. Without otherwise mentioned, we used compressed execution plans in the experiments related to B-BENU.

**S-BENU.** S-BENU was implemented with Spark 2.2.0. All phases were implemented with RDDS. We used 16 executors (one executor per machine). Each executor used 24 cores (i.e. working threads) to run tasks. We allocated 40 Gbytes memory to each executor (30 Gbytes for local database cache and 10 Gbytes for task execution). We turned off the task split technique by default. Without otherwise mentioned, the execution time of S-BENU was the wall-clock time spent on the continuous enumeration phase of S-BENU, as the initialization phase was conducted once.

**Data Graphs.** For B-BENU, we used five real-world static data graphs in Table 1. They were also used by the previous work [8] [9]. For S-BENU, we used a real-world dynamic data graph Wikipedia (denoted as \textit{wk}) [29] with 1.9M vertices and 40.0M edges. We also used the LDBC-SNB Data Generator [30] provided by the LDBC Graphalytics Benchmark [31] to generate a synthetic dynamic social network. The scale factor of the generator was graphalytics1000. We used the person-knows-person part as the dynamic data graph (denoted as ld) with 11M vertices and 0.93B edges. We generated batch updates of dynamic graphs based on the creation time of edges.

**Pattern Graphs** (Fig. 8). For B-BENU, we used Q1 to Q9. Q1 to Q5 come from [9]. To evaluate the performance on
tough tasks, we further used Q6 to Q9. The vertex covers used in the VCB compression are illustrated with big dots. For S-BENU, we used Q1’ to Q5’ coming from [19].

7.2 Evaluation of Optimization Techniques

We evaluated the effectiveness of the proposed techniques in both B-BENU and S-BENU.

**Exp-1: Best Execution Plan Search.** We evaluated the efficiency of Algorithm 3 with random connected graphs. We generated 1000 Erdos-Renyi random pattern graphs for each number of vertices $n$. For every pattern graph, we measured the proportion (Prop.) of matching orders that pass the two pruning techniques and the wall-clock execution time of generating the best (incremental) execution plan(s). Table 4 reports the average results for every $n$. The pruning techniques were effective. The time of generating the best execution plans was very short compared to enumeration.

**Exp-2: Execution Plan Optimizations.** We evaluated the effectiveness of the execution plan optimizations proposed in Section 4.2 on B-BENU and S-BENU in Fig. 9. The X-axis represents execution plans optimized from the raw plan with more optimizations. As the compression would negate some optimization techniques, we only used the compressed execution plan for Q5. For the $awk$ dataset, we used 75% of it as the initial graph and generated 5 time steps with 1M delta edges per time step. Optimization 1 was effective for Q4 and Q5’ where it eliminated common subexpressions. Optimization 2 reduced the execution time in all cases by up to an order of magnitude. It promoted INT instructions to outer loops in all of them. Optimization 3 was effective for Q2 and Q5 where triangles were repeatedly enumerated by two INT instructions.

**Exp-3: Local Database Cache.** We evaluated the effects of the capacity of the local database cache in Fig. 10. The cache capacity is relative to the data graph (B-BENU) or the initial data graph (S-BENU). The network communication cost and the execution time are relative to the corresponding cases with the 10% relative cache capacity. We evaluated B-BENU with Q4 and Q5 on $ok$. We evaluated S-BENU with Q3’, and Q4’ on $ld$ with the 80% initial graph and 10 time steps (1M delta edges per time step). Most pattern graphs (Q4, Q5, and Q3’) were sensitive to the cache capacity. The average cache hit rates increased quickly as the cache capacity grew. Correspondingly, the communication cost and the execution time decreased quickly. The DB cache was an effective technique to improve the efficiency in most cases.

**Exp-4: Task Splitting.** To evaluate the effects of task splitting, we ran B-BENU and S-BENU with the 10% relative cache capacity. We evaluated B-BENU with Q4 and Q5 on $ok$. We evaluated S-BENU with Q3’, and Q4’ on $ld$ with the 80% initial graph and 10 time steps (1M delta edges per time step). Most pattern graphs (Q4, Q5, and Q3’) were sensitive to the cache capacity. The average cache hit rates increased quickly as the cache capacity grew. Correspondingly, the communication cost and the execution time decreased quickly. The DB cache was an effective technique to improve the efficiency in most cases.

**7.3 Comparing B-BENU with the State-of-the-art**

We compared B-BENU with the state-of-the-art MapReduce-based method CBF [9] and the worst-case-optimal-join-based BiGJoin [19]. We turned on all the compression and optimization techniques provided with the algorithms. 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: Comparison 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 5. Nearly in all the cases except Q5 on fs, B-BENU ran quicker than CBF with acceptable communication costs. In several cases like Q2 on ok/uk, Q4 on ok and Q6 on li/ok/uk, B-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. 8). The core structure had more than 2 billion matches in all data graphs (Table 1). CBF had to shuffle the clique index and the matching results of the core structure during the preparation of partition files for hash-assembly. Shuffling many key-value pairs was costly and made Hadoop throw the shuffle error exception in some cases. B-BENU ran smoothly in those cases. For the cases of Q7/Q8/Q9 on uk, the core structure had 2.7 trillion matches. Neither B-BENU nor CBF could finish in 10 hours.

**Exp-6: Comparison with BiGJoin** BiGJoin [19] is the state-of-the-art worst-case-optimal algorithm. We compared B-BENU with it on the pattern graphs that BiGJoin had specially optimized. BiGJoin (https://github.com/frankmcsherry/dataflow-join/) was written with the Timely dataflow system 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 B-BENU with both the shared-memory version (BiGJoin(S)) and the distributed version (BiGJoin(D)) of BiGJoin in Table 6. Since BiGJoin used a different communication mechanism from MapReduce, we did not report the communication costs. On ok, B-BENU ran quicker than both of BiGJoin(D) and BiGJoin(S) with complex pattern graphs. On fs, BiGJoin(S) failed due to out of memory exception, while B-BENU ran quicker than BiGJoin(D) in all cases.

### 7.4 Comparing S-BENU with the State-of-the-art

We compared the performance of S-BENU with the state-of-the-art distributed continuous subgraph enumeration algorithm Delta-BiGJoin [19]. For Delta-BiGJoin, we deployed 12 worker processes (one process per core) on each worker machine. For all datasets, we used 20% initial graph and generated 10 time steps of 20K delta edges per time step. For the ok, uk, and fs datasets that do not have timestamps attached on edges, we picked edges randomly from the remaining 80% graph to generate update operations. We measured the wall-clock execution time of S-BENU and Delta-BiGJoin spent on each time step. The execution time did not include outputing because it was independent of subgraph enumeration. Fig. 12 shows the average time of 10 time steps with error bars indicating the maximal and the minimal. S-BENU outperformed Delta-BiGJoin in all cases, by up to two orders of magnitude.

Compared to S-BENU, Delta-BiGJoin suffered from high communication costs, which is caused by shuffling intermediate matching results. Taking enumerating Q4’ on id as the example, Delta-BiGJoin shuffled the matching results of the partial pattern graphs $u_2-u_5-u_4$ and $u_3-u_5-u_1-u_4$. They had 35M and 197M matches respectively, causing high communication costs.

**7.5 Machine Scalability**

We tested the machine scalability of B-BENU and S-BENU by varying numbers of worker nodes in the cluster. For S-BENU, we used id with 80% initial graph and 1 time step.

The cache capacity is a common hyperparameter of B-BENU and S-BENU. We measured the execution time under different capacities with 8 and 16 workers, and we reported the speedups in Fig. 13. The cache capacities were relative to the (initial) data graph. B-BENU and S-BENU achieved better scalability with a smaller cache. The phenomenon is caused by the warm-up phase of the cache. The cache in each machine had to go through a warm-up phase before it achieved a stable hit rate. In the warm-up phase, there were many cache misses. When B-BENU/S-BENU processed the same input with more workers, more database queries were conducted during the warm-up phase. It caused more cache misses and increased the total execution cost. Taking S-BENU on Q4’ with 1M batch size as an example, the total serial execution time of all tasks with 16 workers increased by 16% (1% capacity) and 33% (100% capacity) compared to 8 workers. The increased execution cost harmed the scalability.

The batch size is a hyperparameter specific to S-BENU. Fig. 13b shows that S-BENU achieved better scalability with larger batch sizes. With a larger batch, the number of conducted database queries was larger and the proportion of queries that were conducted during the warm-up phase of the cache became smaller. Taking Q5’ with 33% cache capacity as an example, the total serial execution time of all tasks with 16 workers increased by 35% (1M batch) and 18% (100M batch) compared to 8 workers.

Fig. 14 reports the machine scalability of B-BENU and S-BENU by varying the number of worker machines from 4 to 16. B-BENU shew the near-linear scalability with both big
and SPath match pattern vertices with infrequent labels and paths first. TurboISO [10] uses the candidate region to dynamically determine the matching order and candidate sets. CFL-Match [11] replicates the whole data graph in MapReduce and uses one-round multiway join on the reducer side to enumerate subgraphs. The amount of replicated edges grows quickly as the pattern graph becomes complex [16]. QFrag [13] replicates the whole data graph in the memory of every worker node and enumerates subgraphs with serial subgraph matching methods in parallel. The memory capacity limits the scale of the data graph that it can handle. RADS [36] partitions the data graph among worker nodes. It first enumerates subgraphs residing in the local partition of each node. It then exchanges and verifies undetermined edges among partitions in a region-grouped manner via MPI to enumerate subgraphs cross partitions. However, RADS relies on MPI to implement the node-to-node communication. It does not consider the fault tolerance, which is essential in distributed computing. CECl [35] constructs embedding clusters from the data graph, distributes the clusters among worker nodes, and enumerates subgraphs from the clusters in parallel. Work stealing via MPI is used to balance workloads. The memory of each node limits the size of embedding clusters that it can handle.

8 RELATED WORK

8.1 Serial Subgraph Enumeration

Most of the serial subgraph matching methods work with labeled graphs and follow the backtracking-based framework [26]. They differ in how to determine the matching order and the candidate sets of pattern vertices. GraphQL [32] and SPath [33] match pattern vertices with infrequent labels and paths first. TurboISO [10] uses the candidate region to dynamically determine the matching order and candidate sets. CFL-Match [11] replicates the whole data graph in MapReduce and uses one-round multiway join on the reducer side to enumerate subgraphs. The amount of replicated edges grows quickly as the pattern graph becomes complex [16]. QFrag [13] replicates the whole data graph in the memory of every worker node and enumerates subgraphs with serial subgraph matching methods in parallel. The memory capacity limits the scale of the data graph that it can handle. RADS [36] partitions the data graph among worker nodes. It first enumerates subgraphs residing in the local partition of each node. It then exchanges and verifies undetermined edges among partitions in a region-grouped manner via MPI to enumerate subgraphs cross partitions. However, RADS relies on MPI to implement the node-to-node communication. It does not consider the fault tolerance, which is essential in distributed computing. CECl [35] constructs embedding clusters from the data graph, distributes the clusters among worker nodes, and enumerates subgraphs from the clusters in parallel. Work stealing via MPI is used to balance workloads. The memory of each node limits the size of embedding clusters that it can handle.
The BFS-style methods follow a join-based framework. They decompose the pattern graph into join units, enumerate partial matches of join units, and join partial matches together to get matches for the whole pattern graph. They implement the join framework with a distributed dataflow engine like MapReduce and Timely that transparently support the fault tolerance. The join-based methods have to shuffle intermediate results during the join. To limit their sizes, varieties of join units (Edge [19], Star [37] [8] [20], TwinTwig [15], Clique [8] and Crystal [9]) and join frameworks (left-deep join [15], [16], two-way bushy join [8], hash-assembly [9], multiway [37] and worst-case optimal join [19]) are proposed. Lai et al. [38] experimentally survey the existing join-based methods with a unified implementation on the Timely dataflow engine. They find that no method could win all scenarios. They propose a practical guide to select a suitable method for specific inputs. Qiao et al. [9] propose the VCBC compression to compress the (partial) matching results.

8.3 Serial Continuous Subgraph Enumeration

Given a batch of edge updates, InclsoMat [39], [40] first finds out the local neighborhood affected by the batch and then conducts subgraph isomorphism in the neighborhood. InclsoMat compares the matching results before and after the update to discover incremental matches. However, InclsoMat has to conduct subgraph isomorphism from scratch for every update. It will do redundant computation if the neighborhoods affected by two updates are overlapped.

One way to avoid such redundancy is to maintain matching results of the pattern graph in memory as the data graph evolves. SJ-Tree [22] stores partial matches of the pattern graph in a binary join tree with single edges and 2-edge paths as join units. SJ-Tree stores partial matches of a tree node in a hash table. When new edges arrive, SJ-Tree joins new edges with the stored partial matches, avoiding enumerating matches from scratch. However, the hash table is a memory-consuming data structure. If there are many partial matches, the memory usage of SJ-Tree will be high. To store matching results compactly, TurboFlux [23] proposes the data-centric graph (DCG) structure. Given an edge update, TurboFlux transmits the states of the edges in DCG and detects incremental matches during the transition. However, the edge transition model is serial. It can only process update edges one by one, limiting the throughput of TurboFlux.

The other way to avoid redundancy is to compute incremental matches from update edges directly. GraphFlow [41] models the continuous subgraph enumeration as the incremental multiway-join view maintainance problem. It computes the incremental matches by joining update edges with the existing data graph together, guided by the delta rule technique [42]. It adopts a variant of GenericJoin [43] as the underlying multi-way join framework.

Though the serial methods have little execution overheads, the computing power and memory capacity of a single node prevent the serial methods from handling big graphs and large update batches.

8.4 Distributed Continuous Subgraph Enumeration

D-IDS [24] prunes the data graph with the maximal dual simulation technique and enumerates matches in subgraphs affected by the update update in parallel. It maintains the matching results in memory in a distributed way to detect disappearing matches. When the data graph is big and the pattern graph is complex, the matching results may exceed the memory capacity of a single node. Delta-BiGJoin [19] is the variant of BiGJoin for dynamic graphs. It partitions and stores the data graph in memory among all worker nodes. Delta-BiGJoin does not maintain any matching result, making the memory usage controllable. Instead, it treats the continuous subgraph enumeration as the incremental view maintenance problem in the relational database. Given a batch of edge updates, Delta-BiGJoin computes the incremental matches via a group of multi-way join queries, guided by the delta rule technique [42]. Delta-BiGJoin uses BiGJoin as its underlying multiway join implementation. Under the insertion-only workloads, Delta-BiGJoin is worst-case optimal in both computation and communication. D-IDS and Delta-BiGJoin are general-purpose methods. Some methods optimize for specific pattern graphs, like vertex-level matching [44] and cycles [6].

9 Conclusion

In this paper, we studied the distributed (continuous) subgraph enumeration problem. The state-of-the-art distributed methods are based on distributed join that has to shuffle intermediate results. When the data graph is big and the pattern graph is complex, the scale of intermediate results can be huge. To overcome the drawback, we proposed a backtracking-based framework Batch-BENU with two features: (1) shuffling data graph instead of intermediate results, (2) on-demand shuffle. Batch-BENU stores the data graph in a distributed key-value database and queried its adjacency sets on demand driven by backtracking-based execution plans. Given a pattern graph, we proposed a search-based method to generate the best execution plan for it. We also proposed three optimization techniques (common subexpression elimination, instruction reordering, and triangle cache) to reduce the execution costs of execution plans. To support dynamic data graphs, we proposed the Streaming-BENU framework. Streaming-BENU solved the continuous subgraph enumeration problem by enumerating incremental pattern graphs at each time step. We proposed efficient implementations for Batch-BENU and Streaming-BENU. Batch-BENU and Streaming-BENU outperformed the state-of-the-art distributed methods by up to one and two orders of magnitude, respectively.

In the future, we shall explore 1) extending Batch-BENU and Streaming-BENU to property graphs, 2) finding a more accurate model to estimate the scale of matching results, and 3) generalizing the triangle cache technique to cliques.
ACKNOWLEDGMENTS

This work is funded in part by National Key R&D Program of China [grant number 2019YFC1711000]; China NSF Grants [grant number U1811461]; Jiangsu Province Industry Support Program [grant number BE2017155, BK20170651]; Collaborative Innovation Center of Novel Software Technology and Industrialization; and the program B for Outstanding PhD candidate of Nanjing University.

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, no. 5594, pp. 824–827, 2002.

[2] P. Przulj, “Biological network comparison using graphlet degree distribution,” Bioinformatics, vol. 20, 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, WSDM 2012, Seattle, WA, USA, February 8-12, 2012, pp. 673–682.

[4] W. Fan, X. Wang, Y. Wu, and J. Xu, “Association rules with graph patterns,” Proc. VLDB Endow., vol. 8, no. 12, pp. 1502–1513, 2015.

[5] G. Sadowski and P. Rathle, “Fraud detection: Discovering connections with graph databases,” https://neo4j.com/whitepapers/fraud-detection-graph-databases/, Neo4j, Tech. Rep., 2017.

[6] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, “Real-time constrained cycle detection in large dynamic graphs,” Proc. VLDB Endow., vol. 11, no. 12, pp. 1876–1888, 2018.

[7] S. Choudhury, L. B. Holder, G. C. Jr., A. Ray, S. Beus, and J. Feo, “Streamworks: a system for dynamic graph search,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pp. 1101–1104.

[8] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang, “Scalable distributed subgraph enumeration,” Proc. VLDB Endow., vol. 10, no. 3, pp. 217–228, 2016.

[9] M. Qiao, H. Zhang, and H. Cheng, “Subgraph matching on compression and computation,” Proc. VLDB Endow., vol. 11, no. 2, pp. 176–188, 2017.

[10] W. Han, J. Lee, and J. Lee, “Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pp. 337–348.

[11] 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pp. 1199–1214.

[12] H. Kim, J. Lee, S. S. Bhowmick, W. Han, J. Lee, S. Ko, and M. H. A. Jarrahi, “DUALSIM: parallel subgraph enumeration in a massive graph on a single machine,” in Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pp. 1231–1245.

[13] 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, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017, 2017, pp. 214–228.

[14] F. N. Afrati, D. Fotakis, and J. D. Ullman, “Enumerating subgraph instances using map-reduce,” in 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, pp. 62–73.

[15] L. Lai, L. Qin, X. Lin, and L. Chang, “Scalable subgraph enumeration in mapreduce,” Proc. VLDB Endow., vol. 8, no. 10, pp. 974–985, 2015.

[16] ———, “Scalable subgraph enumeration in mapreduce: a cost-oriented approach,” VLDB J., vol. 26, no. 3, pp. 421–446, 2017.

[17] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, Jun. 2014.
Zhaokang Wang received the BS degree in Nanjing University, China, in 2013. He is currently working towards the Ph.D. degree in Nanjing University. His research interests include distributed graph algorithms and graph processing systems.

Weiwei Hu received the BS degree in Hunan University, China, in 2017. She is currently working towards the Master degree in Nanjing University. Her research interests include distributed subgraph matching.

Chunfeng Yuan is a professor in the computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. Her main research interests include computer architecture, parallel and distributed computing.

Rong Gu is an associate research professor at State Key Laboratory for Novel Software Technology, Nanjing University, China. Dr. Gu received the Ph.D. degree in computer science from Nanjing University in December 2016. His research interests include parallel computing, distributed systems and distributed machine learning.

Yihua Huang is a professor in State Key Laboratory for Novel Software Technology, Nanjing University, China. He received his Ph.D. degree in computer science from Nanjing University. His main research interests include parallel and distributed computing, big data parallel processing, and distributed machine learning.