A Hierarchical Contraction Scheme for Querying Big Graphs

Wenfei Fan[1,2,3], Yuanhao Li[1], Muyang Liu[1], Can Lu[2]
[1]University of Edinburgh [2]Shenzhen Institute of Computing Sciences [3]Beihang University
United Kingdom[1] China[2,3]
{wenfei@inf.,yuanhao.li@,muyang.liu@}ed.ac.uk,lucan@sics.ac.cn

ABSTRACT

This paper proposes a scheme for querying big graphs with a single machine. The scheme iteratively contracts regular structures into supernodes and builds a hierarchy of contracted graphs, until the one at the top fits into the memory. For each query class Q in use, supernodes carry synopses S_Q such that queries of Q are answered by using S_Q if possible, and otherwise by drilling down to the next level with decontraction of a bounded size. Moreover, we show how to adapt a variety of existing sequential (single-machine) algorithms to the hierarchy by reusing their logic and data structures. We also provide a bounded incremental algorithm to maintain the contracted graphs in response to updates, such that its cost is determined by the sizes of changes to the input and output only. Using real-life and synthetic graphs, we experimentally verify that with a single machine, the hierarchy is able to compute exact query answers when memory is as small as 7.6% of graphs, speeds up various applications by 9.8 times on average, and is even 120.1 times faster than some parallel graph systems that use 6 machines.

CCS CONCEPTS

• **Information systems → Graph-based database models**.

KEYWORDS

Graph data management; Graph contraction; Graph algorithms

ACM Reference Format:
Wenfei Fan, Yuanhao Li, Muyang Liu, Can Lu. 2022. A Hierarchical Contraction Scheme for Querying Big Graphs . In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517862

1 INTRODUCTION

Computations on big graphs are often costly and resource demanding. Consider checking the connectivity of a graph with billions of nodes and trillions of edges, *e.g.,* Hyperlink 2012 Web graph [61]. This routine application takes 341s on a 1000-node cluster with 12000 processors and requires at least 128TB memory [70]. The cost is more staggering when it comes to, *e.g.,* graph pattern matching (subgraph isomorphism), which is intractable (cf. [35]).

One might think that we could accommodate big graphs by employing parallel systems and adding more processors when needed. However, the use of more processors comes with higher communication cost. As indicated in [60], many parallel graph systems "have either a surprisingly large COST, or simply underperform one thread" due to the overhead. Moreover, many algorithms are not parallelly scalable, *i.e.,* it is not guaranteed that the more processors are used, the faster the algorithms run. Worse yet, there exist graph computation problems for which the parallel scalability is beyond reach, *i.e.,* no matter how many processors are added, the computation takes no less time regardless of what algorithms are used [30]. Even when parallel graph computations are effective, employing 12000 processors is beyond the capacity of many companies, which can typically afford only limited computing resources.

A variety of approaches have been explored to tackling this problem. One approach is to make big graphs small, *e.g.,* graph summarization [54] and compression [14]. However, these techniques target a specific query class, while in practice, multiple applications (different query classes) often run on the same graph. It is impractical to maintain a different compressed dataset for each application in use. Furthermore, there is no guarantee that the summaries or compressed graphs are small enough to fit into the memory and moreover, retain sufficient information to compute exact query answers. Another approach is proposed by GraphChi [47]. GraphChi simulates parallel computations with parallel sliding windows on a single machine, by using disk as memory extension. Mosaic [56] optimizes GraphChi by exploiting the heterogeneous devices. However, this approach requires users to recast existing graph algorithms into a vertex-centric model, and is picky on the types of computations; it is not efficient for, *e.g.,* dynamic ordering.

These give rise to several questions. Is it possible to make graphs small enough to fit into the memory of a single machine, and support multiple applications on the same graph? Is it within the reach to handle multiple query classes and compute exact answers? Can we adapt existing single-machine algorithms to the setting, instead of recasting the algorithms into a new computation model?

These questions concern whether we can query big graphs under limited resources. The capacity is also needed by mobile devices and secure computation [20], which can only handle limited amount of data. It also helps companies reduce cost and improve efficiency.

Contributions & organization. This paper proposes a new scheme for answering these questions, in the affirmative.

(1) A hierarchical scheme (Section 2). We propose a scheme that represents a big graph in a hierarchy. It iteratively contracts regular structures into supernodes, until the one at the top level fits into the memory of a single machine. For each query class Q in use, supernodes carry synopses S_Q of data needed for answering queries of Q. The depth of the hierarchy is determined by resources available. We find that typically 2-4 levels suffice to cope with real-life graphs.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517862

The scheme is *generic and lossless, i.e.,* it is able to compute exact answers to different classes of queries using *the same hierarchy*.

(2) Contracting algorithm (Section 3). We develop an algorithm to contract a graph into a hierarchy, by contracting frequent regular structures of different types of graphs, *e.g.,* knowledge bases, Web graphs, transportation networks, collaboration and social graphs. We also define common synopses of these regular structures.

(3) Exact query answers (Section 4). We show how to adapt existing sequential (single-machine) graph algorithms to the scheme by reusing their logic and data structures. Given a query $Q \in \mathcal{Q}$, we start the evaluation at the top level of the hierarchy. We answer Q with synopses S_Q if possible, and drill down to the next level by decontraction otherwise, one at a time within a bounded size.

The scheme is able to compute exact query answers. Moreover, it speeds up processing of a variety of queries. As a proof of concept, we pick six query classes: PageRank (PR), label-constrained connectivity (LCC), subgraph isomorphism (SubIso), clique decision (CD) , connected component (CC) and regular path query (RPQ), based on the dichotomies below:

- local (SubIso, PR, CD) vs. non-local (LCC, CC, RPQ);
- labeled (SubIso, LCC, RPQ) vs. non-labeled (PR, CD, CC);
- NP-hard (SubIso, CD) vs. tractable (PR, LCC, CC, RPQ).

These represent applications such as graph traversal (LCC), pattern matching (SubIso), online queries (PR), community search (CD) [63], graph partition and random walk (CC) [41], and database query language (RPQ). We show that all these queries can be exactly answered with synopses, without decontracting any supernodes of regular structures, sometimes even without any decontraction.

In principle, the scheme is able to support queries of any $\mathcal{Q}$ on big graphs, subject to I/O cost. Nonetheless, as indicated above, only limited decontraction and I/O are needed in many cases.

(4) Incremental contraction (Section 5). We develop an incremental algorithm to maintain the contracted scheme in response to updates to the original graph G, with a single machine. We show that the algorithm is *bounded* [65], *i.e.,* its cost is determined by the size of the areas affected by the updates, not by the size of the entire G.

(5) Experimental study (Section 6). Using real-life and synthetic graphs, we empirically verify the following. (a) To perform comparably with memory-based approaches, the contraction hierarchy is able to compute exact query answers when memory is as small as 7.6% of graphs. On average, (b) it is 705.7, 49.6 and 14.1 times faster than single-machine solutions GraphChi [47], Mosaic [56] and COST [60] for PR, LCC, SubIso, CC, CD and RPQ (GraphChi could not complete within 2 hours for SubIso and both GraphChi and Mosaic ran out of memory for CD), respectively. (c) It even outperforms parallel systems that use multiple machines. It is (i) 74.3, 1.3, 2.4 and 404.7 times faster than PowerGraph [36] that uses 6 machines for LCC, PR, SubIso and CC, respectively; it is faster than (ii) GRAPE [4, 33] for LCC, SubIso, CC and RPQ using 2, 1, 5 and 1 machines, respectively; (iii) Gemini [80] for LCC, SubIso and CC using 6, 3 and 6 machines, respectively; and (iv) LA3 [6] for LCC, SubIso, CC and RPQ using 6, 6, 6 and 1 machines, respectively. (d) Our incremental contraction algorithm is effective. It is faster than batch contraction even when $|\Delta G|$ is up to 25%$|G|$.

Related work. We categorize the related work as follows.

(1) Reducing big graphs. Graph compression [23, 28, 42, 57] and graph summarization [49, 54, 73] aim to reduce big graphs by merging nodes or subgraphs into supernodes. Graph compression computes query-equivalence relations and merges "equivalent" nodes; *e.g.,* [28, 42] answer reachability queries by merging nodes in the same transitive closure. Graph summarization generates summaries for nodes or subgraphs; *e.g.,* [49] aggregates node clusters into supernodes labeled with the numbers of edges within and between the clusters, for adjacency, degree and centrality queries.

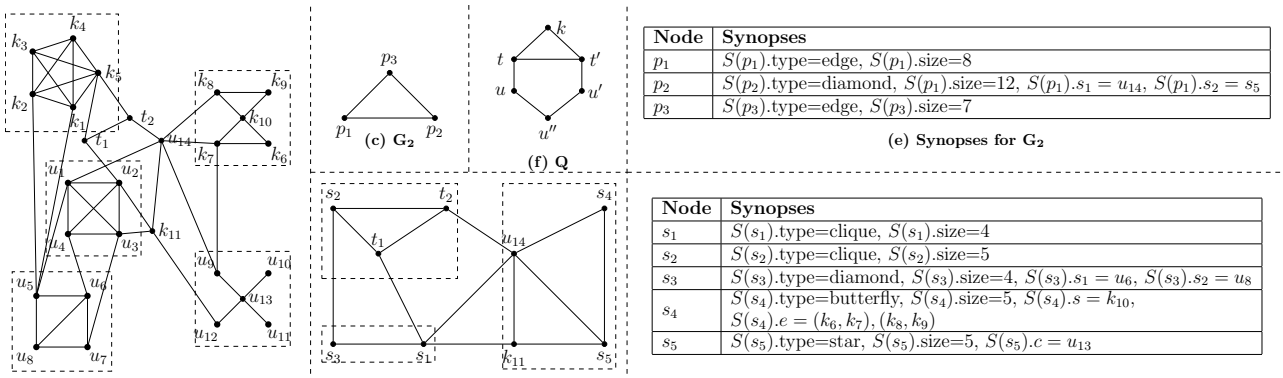
As opposed to the prior work, (a) our scheme is *generic and lossless*. In contrast, graph compression is query dependent and summarization is usually lossy. (b) We can adapt existing algorithms to the scheme by using their logic and data structures while new algorithms often have to be developed for compression and summarization. (c) Our scheme aims to reduce graphs to fit into memory, which is not guaranteed by compression and summarization.

Closer to this work is [29], which proposes to contract graphs as an optimization strategy. This work substantially extends [29]. (a) We contract graphs under a memory constraint to support multiple applications on a single machine. In contrast, [29] does not have to conform to a predefined memory bound. (b) To cope with memory bound, we contract a graph into a hierarchy, as opposed to a single contracted graph of [29]. (c) We contract a variety of frequent regular structures based on the types of graphs (see Section 3), and develop their synopses. In contrast, [29] adopts an one-size-fit-all solution and contracts only cliques, paths and stars for all types of graphs. (d) We experiment with larger graphs to evaluate the capacity of a single machine for querying big graphs.

(2) Querying big graphs. To scale with big graphs, several parallel graph computation systems have been developed, *e.g.,* PowerGraph [36, 55], Giraph++ [72], GRAPE [5, 33], Gemini [80], LA3 [6] and Pregel+ [77]. These systems partition a big graph into small parts and process all parts in parallel. However, parallel computations are not effective for all problems [30]; many parallel systems incur heavy communication cost and cannot beat the best single-threaded implementation [60]. Worse still, a large cluster of machines is often beyond the reach of those companies with limited resources.

A different approach is to query big graphs with a single machine, *e.g.,* GraphChi [47], Mosaic [56], GridGraph [22] and NXgraph [77]. Below we discuss representative GraphChi and Mosaic. GraphChi splits an input graph into disjoint intervals, which are stored in disks by using disk as memory extension. It adopts vertex-centric model and parallel sliding windows (PSW) to extract induced subgraph with non-sequential disk accesses to the intervals. PSW runs the user-defined update-function for each vertex, modifies values and writes updated values back to the disk in parallel. Mosaic uses a 2-level scheme to partition a graph into disjoint sets of edges, called *tiles*. Such tiles are compressed with local vertex identifier to fit into the last level cache and are processed following the Hilbert order. Mosaic also adopts the vertex-centric model, while exploiting the massive parallelism provided by modern heterogeneous hardware.

GraphChi, Mosaic and this work aim to query big graphs with a single machine. Our scheme differs from GraphChi and Mosaic



Node	Synopses
p_1	$S(p_1)$.type=edge, $S(p_1)$.size=8
p_2	$S(p_2)$.type=diamond, $S(p_1)$.size=12, $S(p_1).s_1 = u_{14}$, $S(p_1).s_2 = s_5$
p_3	$S(p_3)$.type=edge, $S(p_3)$.size=7

Node	Synopses
s_1	$S(s_1)$.type=clique, $S(s_1)$.size=4
s_2	$S(s_2)$.type=clique, $S(s_2)$.size=5
s_3	$S(s_3)$.type=diamond, $S(s_3)$.size=4, $S(s_3).s_1 = u_6$, $S(s_3).s_2 = u_8$
s_4	$S(s_4)$.type=butterfly, $S(s_4)$.size=5, $S(s_4).s = k_{10}$, $S(s_4).e = (k_6, k_7), (k_8, k_9)$
s_5	$S(s_5)$.type=star, $S(s_5)$.size=5, $S(s_5).c = u_{13}$

Figure 1: Hierarchical contraction scheme

in the following. (a) Our scheme annotates supernodes with synopses. As will be seen in Section 4, the synopses often have enough information to answer queries without the need for decontraction and hence, without disk accesses. (b) GraphChi splits nodes into disjoint intervals and stores edges with the same head together in the same interval. Mosaic partitions a graph into tiles following the Hilbert order and yields a compact representation for each tile. In contrast, our scheme contracts regular structures into supernodes, including their edges. Hence, when decontraction is necessary, our scheme needs only sequential disk accesses to load a subgraph, while GraphChi and Mosaic require non-sequential disk accesses. (c) One can often adapt existing single-machine graph algorithms to the hierarchical scheme by reusing their logic and data structures, while GraphChi and Mosaic require users to recast the algorithms into a vertex-centric model. (d) As a result of using PSW, loading the neighborhood of a single node on GraphChi [47] has to scan all edges in disk; similarly for Mosaic since it is also vertex-centric. In contrast, our scheme is "graph-centric" and can accommodate existing single-machine graph algorithms.

(3) Bounded evaluation. Bounded evaluation is another approach to making big graphs small [18, 21, 26]. Given a query Q on a big dataset D, it is to access a subset D_Q of D by leveraging an access schema $\mathcal{A}$, such that $Q(D_Q) = Q(D)$ and the cost of identifying and fetching D_Q is decided by the sizes of Q and $\mathcal{A}$, regardless of the size of D. The idea has been extended to graph pattern matching [19].

This work is quite different. (1) Bounded evaluation requires to build and maintain access schema. As indicated in [19], the size of access schema schema could be as large as the original graph. (2) To support multiple applications, for each of the applications, bounded evaluation requires a different access schema, which is prohibitively costly. Hence bounded evaluation does not suffice to support multiple graph applications under limited resources.

2 A HIERARCHICAL GRAPH SCHEME

In this section, we present the hierarchical contraction scheme.

Preliminaries. We start with basic notations.

Graphs. Assume an infinite set Θ for labels. We consider undirected graphs $G = (V, E, L)$ with labels on nodes and edges, where (a) V is a finite set of nodes, (b) $E \subseteq V \times V$ is a bag of edges, and (c) for each node $v \in V$ (resp. edge $e \in E$), $L(v) \in \Theta$ (resp. $L(e)$) is a label.

Queries. A graph query is a computable function from a graph G to, *e.g.,* a Boolean value, a graph, or a relation. For instance, a query of *label-constrained connectivity (*LCC*)* [10] on G consists of two nodes u, v in G and a label set L; it is a Boolean function that returns true iff there exists a path from u to v on which all labels of the nodes are covered by L. A *query class* $\mathcal{Q}$ is a set of queries of the same "type", *i.e.,* all LCC queries. We also refer to $\mathcal{Q}$ as an *application*. In practice, multiple applications often run on the same graph G.

Problem. We study *the problem of querying graphs* G when (1) we can afford at most main memory of size M to store the data of G, and a disk of an unbounded size. The disk is formatted into disjoint blocks, each of which has size B; and (2) the size of the entire graph G exceeds M. We use a buffer of size B in the main memory for loading subgraphs (blocks) from disk, at most one at a time.

- *Setting*: A graph G stored in disk, query classes $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ in use, buffer size B and memory capacity M such that $|G| > M$.
- *Objective*: A hierarchy $\mathcal{H}$ of contracted graphs such that
  - the contracted graph G_k at the top level of $\mathcal{H}$ can reside in the memory, *i.e.,* $|G_k| \leq M$;
  - for all $i \in [1, n]$ and queries $Q \in \mathcal{Q}_i$, exact answers $Q(G)$ can be computed in $\mathcal{H}$ (possibly with decontraction); and
  - the cost of computing $Q(G)$ is minimized.

We next define the hierarchical contraction scheme.

Hierarchy. A *hierarchical contraction scheme* $\mathcal{H}$ consists of k levels $\langle f_C^1, \mathcal{S}^1, f_D^1, f_R^1 \rangle, \ldots, \langle f_C^k, \mathcal{S}^k, f_D^k, f_R^k \rangle$ with contracted graphs $G_1, \ldots, G_k$. We refer to G_i and $\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i \rangle$ as *the contracted graph* and *contraction scheme* at level i, respectively, such that

(1) $G_1 = f_C^1(G)$ and $G_i = f_C^i(G_{i-1})$, *i.e.,* G_i is the contracted graph of G_{i-1} by *contraction function* f_C^i at level i; f_C^i contracts certain subgraphs H of size at most B in G_{i-1} into *supernodes* v_H in G_i;
(2) for each query class $\mathcal{Q}$ in use, a *synopsis function* $S_{\mathcal{Q}}^i \in \mathcal{S}^i$ is associated for answering queries of $\mathcal{Q}$, which extends $S_{\mathcal{Q}}^{i-1}$;
(3) f_D^i is *the decontraction function* at level i that restores each supernode v_H in G_i to its contracted subgraph H in G_{i-1};
(4) f_R^i is a function at level i that keeps track of nodes in subgraph H of G_{i-1} when H is contracted into a supernode v_H in G_i; as opposed to f_D^i, f_R^i does not restore the structure of H; and
(5) the contracted graph G_k at the top level has size at most M.

The *depth* k of the hierarchy is determined by the size of G and the resources available. Only G_k resides in the main memory. Labels of both nodes and edges are only stored once as synopses of G_1.

Symbols	Notations
$G(V, E, L)$	labeled graph
$\mathcal{Q}$, Q	query class $\mathcal{Q}$, query in $\mathcal{Q}$
M, B	main memory size and buffer size
$\mathcal{H}$, G_i	hierarchical contraction scheme, contracted graph at level i
f_C^i, $\mathcal{S}^i$, f_D^i, f_R^i	contraction, synopsis and decontraction functions at level i
N	the largest number of edges contracted to a superedge
v_H, H	subgraph H in G_{i-1} contracted into a supernode v_H in G_i
$V(H), E(H)$	the node set and edge set of subgraph H, respectively
k_l, k_u	the lower and upper size bound of the regular structures

Table 1: Notations

Example 1: Figure 1 shows a 2-level contraction hierarchy. Graph G_0 (Fig. 1(a)) is a fraction of Twitter network, in which a node denotes a user (u), a tweet (t) or a keyword (k). An edge indicates the following: (1) (u, u'), a user follows another; (2) (u, t), a user posts a tweet; (3) (t, t'), a tweet retweets another; (4) (t, k), a tweet tags a keyword; (5) (k, k'), two keywords are highly related; or (6) (u, k), a user is interested in a keyword. In G_0, subgraphs in dashed rectangles are contracted into supernodes, and the remaining nodes are mapped to themselves, yielding contracted graph G_1 at level 1 (Fig. 1(b)). In a similar manner, contracting subgraphs in dashed rectangles in G_1 yields contracted graph G_2 (Fig. 1(c)). Graph G_2 fits in memory and is at the top level of the hierarchy. Synopses are shown in Figure 1(d)-(e) and will be illustrated in Sections 3 and 4. □

As will be seen in Section 4, the hierarchy is *generic, i.e.,* the *same* hierarchy is used to answer *different* classes of queries, and the same synopses in $S_{\mathcal{Q}}^i$ are used to answer all queries in $\mathcal{Q}$. Moreover, it is *lossless, i.e.,* we can compute *exact query answers* without loss of information, by means of the synopses and decontraction functions.

Contraction. We now outline how we contract a graph G into a hierarchy $\mathcal{H}$. Starting from $G_0 = G$ and $i = 1$, we "recursively" contract G_{i-1} into G_i until we reach level k such that $|G_k| \leq M$, in two phases. (1) At each level i of $\mathcal{H}$, we contract regular structures only. (2) Phase (1) proceeds until we reach a level where $|G_i|/|G_{i-1}| > t_p$ and $0<t_p<1$ is a predefined threshold, *i.e.,* when G_i has no substantial improvement on G_{i-1} by contracting regular structures; we then further conduct edge contraction and stop at level i, *i.e.,* $i = k$ and G_k is at the top level of $\mathcal{H}$. Below we elaborate the two phases.

Topology contraction. The contraction at level i is carried out by function f_C^i, to build contracted graph G_i. (1) It maps each node v in graph G_{i-1} to a *supernode* in G_i, which either contracts a regular structure H into v_H or is node v itself. (2) It includes a *superedge* (v_{H1}, v_{H2}) in G_i if there exist nodes v_1 and v_2 in G_{i-1} such that $f_C^i(v_1) = v_{H1}$, $f_C^i(v_2) = v_{H2}$ and (v_1, v_2) is an edge in G_{i-1}.

Edge contraction. When $|G_i|/|G_{i-1}| > t_p$ and $|G_{i-1}| > M$, we further contract edges in G_{i-1} until $|G_i| \leq M$. More specifically, we repeatedly contract edges into supernodes as long as (a) such edges are connected, and (b) each contracted subgraph has a size of at most B.

We will present our contraction algorithm in Section 3.1.

Synopses. For each query class $\mathcal{Q}$ in use, a synopsis function $S_{\mathcal{Q}}^i$ retains necessary features of the contracted subgraphs for answering queries of $\mathcal{Q}$. The function $S_{\mathcal{Q}}^i$ at level i extends $S_{\mathcal{Q}}^{i-1}$ as follows. Let v_H be a supernode in G_i. Then synopsis $S_{\mathcal{Q}}^i(v_H)$ is generated by composing and aggregating synopses $S_{\mathcal{Q}}^{i-1}(v)$ of nodes v in G_{i-1} that are contracted to v_H by f_C^i, *i.e.,* $f_C^i(v) = v_H$. For LCC, *e.g.,* the synopsis of a supernode is the set of node labels in the contracted subgraphs, and the aggregation is simply set union. We will give more details about synopses $S_{\mathcal{Q}}$ in Sections 3 and 4. Moreover, we will see that different query classes often share common synopses.

Figure 2: Frequent regular structures

Decontraction. As remarked earlier, we decontract a supernode v_H or a superedge (v_{H1}, v_{H2}) with decontraction function f_D^i only when necessary, to compute exact query answers. Consider a subgraph H that is contracted to a supernode v_H, where $V(H)$ and $E(H)$ denote the sets of nodes and edges in H, respectively.

There are two forms of decontraction. *Decontraction of supernode* v_H restores subgraph H that is contracted to v_H. *Decontraction of superedge* (v_{H1}, v_{H2}) just restores edges between $V(H_1)$ and $V(H_2)$ ($f_D^i(v_{H1}, v_{H2})$ fetches only edges contracted to the superedge from disk). We also use f_R^i that retrieves only nodes in H (no edges).

We ensure that the decontracted subgraph H can fit in buffer B. The decontraction of a supernode v_H in G_i consists of three parts: (1) restoring the structure of subgraph H in G_{i-1} that is contracted into v_H by f_C^i; (2) recovering the synopses of nodes in $V(H)$; and (3) recovering edges in G_{i-2} that are contracted to some edges in H. For (1), (a) when H is a regular structure, its structure can be recovered by synopsis without decontraction and the size $|E(H)|$ is easy to known (see Section 3); (b) when H is contracted by edges, decontraction restores $|E(H)|$ edges. For (2), the size can be calculated by aggregating corresponding synopses. For (3), we maintain a number N that indicates the largest number of edges contracted to a superedge. In each iteration to contract G_{i-1} into G_i, N is updated by the superedges constructed. Then, an upper bound of edges that may be loaded into memory is $N \times |E(H)|$. Note that if the number of edges in H being decontracted simultaneously is bounded by a constant c, then the edges to load into memory is bounded by $N \times c$. We contract a subgraph into a supernode if its decontraction and associated synopses have a total size at most B.

When answering queries, supernode and superedge decontraction can be conducted at any level of the hierarchy. We will show in Section 4 that regular structures typically do not need decontraction, while superedge decontraction is often needed.

The notations of this paper are summarized in Table 1.

3 CONTRACTING BIG GRAPHS

In this section, we first develop an algorithm to construct a hierarchy of contracted graphs (Section 3.1). We then show how synopses are defined for various regular structures (Section 3.2).

3.1 Contraction Algorithm

Different types of graphs contain different frequent regular structures, *e.g.,* cliques are ubiquitous in social graphs while paths are only effective in chemical graphs and road networks. As shown in Fig. 2, we identify frequent regular structures for different types of real-life graphs, and contract those that could improve contraction ratio of each level without incurring heavy I/O cost.

Contraction order. Table 2 shows how these structures appear as subgraphs in 10 *types* of graphs, ordered by their supports from high to low. We contract such structures H in different types of graphs

Graph type	Regular structures (ordered by support)
social graphs	clique, diamond, star, butterfly
communication networks	star, bipartite graph
citation networks	clique, star, diamond, butterfly, cycle
Web graphs	star, clique, diamond
knowledge graphs	star, claw, doubleclaw
collaboration networks	clique, bipartite graph, star
biomedical graphs	star, cycle, clique, path
economic networks	star
chemical graphs	cycle, claw, path
road networks	star, path, doubleclaw, cycle

Table 2: Common structures in different types of graphs

G following the order of Table 2, denoted as $T(G)$, to ensure that important structures are contracted earlier and preserved. Within H, the only edges are those that form H. Edges are allowed from nodes in H to nodes outside H, except from intermediate nodes on a path.

Note that the order $T(G)$ is determined by *the type* of G, *e.g.,* social graphs, in order to improve the efficiency of computations on the graph. It is learned *once offline* regardless of individual graph G. The contraction hierarchy is *stable*, by contracting nodes by their IDs and regular structures in a *deterministic* order. Besides, the contraction hierarchy of each graph is computed once offline, and incrementally maintained in response to updates (Section 5).

Size bounds. We contract a regular structure H such that the number of its nodes is in the range $[k_l, k_u]$. The reason is twofold. (1) We set a lower bound since if H is too small, the contracted graphs would be over-contracted with an excessive number of supernodes, and leads to a deep hierarchy and high I/O overhead for decontraction. (2) We deduce an upper bound k_u based on the buffer size B (Section 2), such that when we decontract v_H, the subgraph H fits in buffer B. (3) We experimentally find that the best k_l and k_u for our datasets tested are around 4 and 500, respectively.

These bounds apply to regular structures cliques, paths and stars, while diamonds and butterflies have a constant size. The upper bound for contracting edges is deduced similarly. To fit in memory, edge contraction (at the top level) has no lower bound.

Contraction algorithm. Adopting the pre-computed order $T(G)$, we present an algorithm to construct a hierarchy for a given graph G, denoted as HCon and shown in Fig. 3. HCon repeatedly calls procedure LCon, which generates G_i and $\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle$ from the contracted graph G_{i-1} at level $i-1$ (lines 2-5). More specifically, LCon preprocesses G_{i-1} by partitioning it into subgraphs g and then iteratively loads subgraphs g of G_{i-1} into memory and contracts regular structures in g into supernodes following the order of $T(G)$, until $|G_i| \le M$ or $|G_i|/|G_{i-1}| > t_p$ (see Section 2). Here g can be an arbitrary subgraph as long as it can fit in the memory. All nodes that are not contracted into regular structures are further partitioned into subgraphs for contraction. If the size of G_{i-1} still exceeds bound M, HCon further contracts edges in G_{i-1} by calling procedure TCon (not shown) to make $|G_i| \le M$ (lines 6-7).

Given G_{i-1}, procedure LCon builds G_i. Initially, all nodes in G_{i-1} are marked as "uncontracted". It then contracts frequent regular structures in G_{i-1} by following the pre-computed order $T(G)$, one by one (lines 2-5). For instance, it extracts a clique by repeatedly picking an uncontracted node that is adjacent to all selected ones, subject to bounds k_l and k_u; it extracts a star by first picking a central node v_c, and then repeatedly selecting an uncontracted node as a leaf that is (a) connected to v_c and (b) disconnected from all selected leaves, again subject to k_l and k_u; similarly for the other structures of Fig. 2. Each of the regular structures consists of uncontracted nodes only, *i.e.,* nodes in G_{i-1} are contracted at most once. Moreover, the size of each of the structures is in the range of $[k_l, k_u]$. The process proceeds until all nodes in G_{i-1} are processed (line 4). It then deduces $\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle$ based on the contraction (line 6).

Algorithm HCon
Input: A graph G, memory capacity M, threshold t_p and buffer size B.
Output: The hierarchical scheme $\langle f_C^1, \mathcal{S}^1, f_D^1, f_R^1\rangle, \ldots, \langle f_C^k, \mathcal{S}^k, f_D^k, f_R^k\rangle$.
1. $G_0 := G$; $i := 1$; $T(G)$:= precomputed ordered set of structures of G;
2. **while** $|G_{i-1}| > M$ Do
3. $\quad\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle := \mathsf{LCon}(G_{i-1}, T(G))$;
4. $\quad$**if** $|G_i|/|G_{i-1}| \le t_p$ **then** break;
5. $\quad i := i+1$;
6. **if** $|G_{i-1}| > M$ **then**
7. $\quad\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle := \mathsf{TCon}(G_{i-1})$; /* edge contraction*/
8. return $\langle f_C^1, \mathcal{S}^1, f_D^1, f_R^1\rangle, \ldots, \langle f_C^k, \mathcal{S}^k, f_D^k, f_R^k\rangle$;

Procedure LCon
Input: A graph G_{i-1}, order $T(G)$ on structures and buffer size B.
Output: The contracted graph at level i $\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle$.
1. partition G_{i-1} into subgraphs g;
2. **repeat** load subgraphs g of G_{i-1} such that $|g| < M$;
3. $\quad$contract regular structures in the range $[k_l, k_u]$ in g in order;
4. **until** all nodes in G_{i-1} are processed;
5. repeat lines 1-4 for all uncontracted nodes;
6. deduce and return $\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle$ from the contracted structures;

Figure 3: Algorithm HCon

For every contracted structure with k nodes, the size of its edge set is as follows: (1) cliques: $|E(H)| = k(k-1)/2$; (2) stars and paths: $|E(H)| = k-1$; (3) diamonds: $|E(H)| = 5$; and (4) butterflies:$|E(H)| = 6$. With the size of synopses for H (see Section 3.2), one can deduce upper bound k_u and estimate the space cost of H such that $|H| \le B$, *i.e.,* H can be loaded to the buffer when recovering H.

Example 2: Given graph G_0 of Fig. 1(a), algorithm HCon builds the hierarchy of Fig. 1(a) as follows. (1) Since G_0 is a social network, HCon contracts cliques, diamonds, butterflies and stars in this order (Table 2). (2) In G_0, it finds cliques $(u_1, \ldots, u_4)$ and $(k_1, \ldots, k_5)$. It then contracts other regular structures and constructs G_1 (see Fig. 1(b)). (3) In G_1, it finds only one diamond $(s_4, s_5, u_{14}, k_{11})$ to contract. It then contracts edges and yields G_2 (see Fig. 1(c)). □

Contracting regular structures saves space. Taking star as an example, (1) we only store its central node and leaves, without storing any edges in the star, *e.g.,* we save the space of 4 edges by contracting a star with 5 nodes and 4 edges, and (2) edges between the nodes in the star and outside the star are contracted into superedges.

Complexity. Algorithm HCon takes $O(|G|^2)$ time. Indeed, (1) to extract a clique for a node v, HCon maintains its node set C, initialized as $\{v\}$, and a set P, initialized as the neighbor set of v. HCon iteratively adds a node u from P to C if it connects to all nodes in C, in time linear to its degree, if there is one. Since each clique contains at most k_u nodes, it takes $O(|G|)$ time to contract each clique and $O(|G|^2)$ time for all cliques. (2) Paths can be built in $O(|G|)$ time. (3) Similarly, the other regular structures are contracted in $O(|G|^2)$ time. Along the same lines as [29], HCon can be parallelized.

3.2 Deducing Synopses

We next present synopses for contracted subgraphs. For each structure, here we focus on *common features* that are shared by all applications. For a specific query class Q, its synopsis function S^i_Q may be further extended to include features specific to Q (Section 4). Each contracted component has a specific structure; the synopses encode its key features to answer queries without decontraction.

The synopsis function S^i at each level i is generated by composing and aggregating synopses S^{i-1} at the lower level. Assume that at level 0, each node has a synopsis of $S^0(v).size = 1$. We define synopsis for each structure and the function S^i at each level i as follows. For supernode v_H that contracts a subgraph H of type τ, *e.g.,* clique, we denote $S^i(v_H)$.type = τ. At the top level G_k, we denote $S^k(v_H)$.type = edge for supernodes v_H built by edge contraction.

- cliques: $S^i(v_H)$.size = $\sum_{v \in v(H)} S^{i-1}(v)$.size is the total number of level-0 nodes contracted in v_H; for the other structures below, $S^i(v_H)$.size is defined and aggregated in the same way;
- stars: $S^i(v_H).c$ is the central node id;
- paths: $S^i(v_H)$.list stores the ids of all its nodes in order;
- diamonds: $S^i(v_H).s_1$ and $S^i(v_H).s_2$ store the two shared nodes of the two triangles; and
- butterflies: $S^i(v_H).s$ is the node shared by the two triangles, and $S^i(v_H).e$ stores the two disjoint edges.

As an example, the synopses of regular structures in G_1 and G_2 of Figure 1 are shown in Figures 1(d) and (e), respectively. Note that space $|S^i(v_H)| \leq |V(H)|+2$, including (1) a single value for type; (2) at most $|V(H)|$ for .c, .list, .s and .e; and (3) a single value for size.

3.3 APIs

The scheme inherits the logic and data structures of existing sequential algorithms. It provides new APIs for generating synopses for new applications and loading contraction functions, synopses and decontraction functions. The APIs support the following.

For decontracting supernodes/superedges, the scheme provides functions **LoadDeconNode**(*int* depth, *)/**LoadDeconEdge**(*int* depth, *) to load corresponding supernodes/superedges based on the depth of the hierarchy. Besides, the scheme supports user-defined functions **GenSynopsis** () to generate synopses for a new application, and **GetSynopsis**() to access synopses stored in the disk. For example, **GetSynopsis**() of LCCA_h for supernode u returns labels of all the nodes contracted to supernode u. Similar to decontraction, the scheme provides functions **LoadConNode**(*int* depth, *) and **LoadConEdge**(*int* depth, *) to load contracted nodes and edges.

4 COMPUTING EXACT ANSWERS

We next show how to adapt existing algorithms to the hierarchy $\mathcal{H}$, by presenting (1) PageRank (PR), for unlabeled and local online queries; (2) label-constrained connectivity (LCC), for labeled and non-local online cases; (3) subgraph isomorphism (SubIso), for intractable cases (Sections 4.1-4.3). We also briefly list connected components (CC), clique decision (CD) and regular path query (RPQ) (Section 4.4). GraphChi [47] and Mosaic [56] can handle PR and CC well, but not LCC, SubIso, CD and RPQ.

The main conclusions of the section are as follows.

Theorem 1: *With a linear-time synopsis function, there exist algorithms for each of* PR, LCC, SubIso, CC, CD *and* RPQ *that can be adapted to the contraction hierarchy $\mathcal{H}$ such that*

(1) the adapted algorithms compute exact query answers;

(2) for PR, LCC, SubIso, CD *and* RPQ, *only superedges need decontraction,* ***not supernodes of regular structures;***

(3) for CC, ***neither superedges nor supernodes need to be decontracted.*** □

In a nutshell, given a query $Q \in Q$, we start the evaluation from the contracted graph G_k at the top level of $\mathcal{H}$. When we encounter a supernode v_H in G_k, we answer Q with its synopsis $S^k_Q(v_H)$ if possible, and drill down to the next level otherwise by recovering nodes and/or edges in G_{k-1} that are contracted to v_H. When we drill down, we evaluate Q with the data in the contracted graph at the next level, until we get answers $Q(G)$ in the original graph G.

We highlight the following. (1) The hierarchy is *generic, i.e.,* the *same* hierarchy is used to answer *different* classes of queries, and the same synopses in S^i_Q are used to answer all queries in Q. (2) It is *lossless, i.e.,* we can compute *exact query answers* without loss of information. (3) In principle the hierarchy is able to handle arbitrary queries on a graph with a single machine, but it may incur heavy I/O cost for decontraction. This said, in practice it often speeds up query answering for the following reasons. (a) Synopsis in $S^k_Q(v_H)$ often provides enough information for us either to process Q at v_H as a whole or safely skip v_H; it also often suffices to decontract superedges, not supernodes, as indicated in Theorem 1. (b) Query processing is conducted on smaller contracted graphs. (c) As will be seen in Section 6, the depth of a hierarchy is usually small. (d) Supernodes are decontracted only when necessary. (4) When a new query class is given, its synopses are computed offline over the contracted graph directly. As will be seen in Section 6, the cost of synopses computation is quite small compared to the contraction cost.

4.1 PageRank

We start with PageRank [12, 62], which has been widely used in applications including Web search [8] and recommendation [79].

For a graph G, its PageRank vector pr iteratively assigns scores to nodes that represent the stationary distribution of a stochastic process; hence $\sum_{u \in V} pr(u) = 1$. In the process, in each iteration, (1) each node u distributes its score evenly to its neighbors v through edges (u, v); and (2) node u aggregates scores received from its neighbors v via edges (v, u). More specifically, in the r-th iteration, $pr^r(u) = \sum_{(v,u) \in E} pr^{r-1}(v)/d(v)$, where $pr^r(v)$ and $d(v)$ denote the PageRank value in the r-th iteration and the degree of node v, respectively. The process reaches a stationary state when $|p^r(u) - p^{r-1}(u)|$ is smaller than a threshold ϵ for all nodes u.

The *PageRank problem*, denoted as PR, is to compute, given a graph G and a threshold ϵ, the PageRank vector pr.

PR is *unlabeled, i.e.,* labels have no impact on its query answer. It is *local* since the PageRank score of each node relies only on the scores and degrees of its neighbors.

As shown in [47], GraphChi can efficiently handle PR.

As a proof of Theorem 1 for PR, we adapt a conventional algorithm PRA [62] for PR to the contraction hierarchy $\mathcal{H}$.

4.1.1 Contraction for PR. The shared synopsis S^i suffices for us to answer PR queries. Indeed, for each subgraph H contracted to a

supernode v_H in G_i, we can check the existence of an edge in H by using synopsis $S^i(v_H)$ and f_R^i that records only nodes in H.

4.1.2 PageRank algorithm. Below we first review PRA [62]. We then adapt the algorithm to the hierarchy $\mathcal{H}$, referred to as $\mathsf{PRA_h}$.

PRA. Given a graph G and a threshold ϵ, PRA iteratively computes the PageRank vector pr. It initializes $pr^0(u) = 1/|V|$ for all nodes u, and updates $pr^i(u)$ by its neighbors in each iteration. PRA terminates as soon as $|pr^{i+1}(u) - pr^i(u)| < \epsilon$ for all nodes u.

Algorithm $\mathsf{PRA_h}$. $\mathsf{PRA_h}$ is the same as PRA except minor adaptations to deal with supernodes in hierarchy $\mathcal{H}$. A node v is called *converged* if $|pr^r(v) - pr^{r-1}(v)| < \epsilon$ in the r-th iteration. Each supernode v_H is associated with a Boolean variable v_H.cvg, initialized as false, to indicate whether all nodes contracted to v_H have converged. As claimed in [12], the difference $|pr(u) - pr^r(u)|$ decreases monotonically; hence once a node u is converged, it will never become unconverged later on; it is the same for supernodes. In the r-th iteration, (1) if v_H.cvg = true, we skip updating v_H as a whole; more specifically, for nodes u contracted to v_H, $\mathsf{PRA_h}$ neither updates $pr^r(u)$ nor distributes $pr^r(u)/d(u)$ to the neighbors of u; (2) otherwise, we recursively decontract an edge that exists in subgraph H contracted to v_H to update level-0 nodes u with u.cvg = false; and (3) if $|pr^r(u) - pr^{r-1}(u)| < \epsilon$, $\mathsf{PRA_h}$ sets u.cvg as true. Moreover, $\mathsf{PRA_h}$ updates v_H.cvg to be true as long as v.cvg = true for all nodes v contracted to v_H in the next level.

Note that cases (1) and (3) are checked with f_R^i and synopses without decontraction, and case (2) needs to decontract only superedges. In the entire process no contracted regular structures need to be restored, *i.e.,* no decontraction of supernodes is needed.

Example 3: Given the graph G in Fig. 1(a) and $\epsilon = 10^{-4}$, $\mathsf{PRA_h}$ first initializes $pr^0(v) = 1/27$ and v_H.cvg = false for all nodes and supernodes. In the first few iterations, $\mathsf{PRA_h}$ recursively decontracts superedges in H contracted to supernodes v_H to update $pr(v)$ for nodes v since v_H.cvg = false. In the 5-th iteration, $|pr^5(u_1) - pr^4(u_1)| < \epsilon$; hence $\mathsf{PRA_h}$ sets u_1.cvg = true. In the 26-th iteration, $\mathsf{PRA_h}$ sets u_2.cvg = true, which further updates s_1.cvg = true since nodes u_1, u_2, u_3 and u_4 have converged. In the following iterations, no update for s_1 is needed. After 40 iterations, v_H.cvg = true for all supernodes, hence $\mathsf{PRA_h}$ returns PageRank vector pr. □

Analyses. $\mathsf{PRA_h}$ is correct as it follows the same logic as PRA. It improves the efficiency of PRA by skipping a supernode v_H as a whole as long as all nodes contracted to v_H are converged. In each step, at most one superedge is decontracted; no supernode is decontracted. Even when restoring edges contracted to a supernode at the top level, its size is at most B; hence a small buffer suffices.

4.2 Label Constrained Connectivity

We next study label-constrained connectivity [10, 74], which has been used in regular path queries [9, 11] and program analysis [67].

In a graph G, for a label set $L \subset \Theta$, a path $p = \langle v_0, v_1, \ldots, v_l \rangle$ is an *L-path* if $L(v_i) \in L$ for all $0 \le i \le l$, *i.e.,* the label of each node v on the path is contained by L. We say that two nodes u and v are *L-connected* if there exists an L-path from u to v.

The *label-constrained connectivity problem*, denoted by LCC, is to determine, given a query Q consisting of a label set L and a pair (u, v) of nodes in G, whether u and v are L-connected.

Unlike PR, LCC is *labeled*, *i.e.,* paths between u and v are constrained by labels in L. It is *non-local*, *i.e.,* it has to traverse the entire graph when answering a query. As acknowledged in [47], LCC is inefficient for GraphChi since it requires to traverse the graph.

As a proof of Theorem 1 for LCC, we adapt a conventional algorithm LCCA [10, 16] for LCC to the contraction hierarchy $\mathcal{H}$, and show that the adapted algorithm works efficiently on $\mathcal{H}$. Below we first present synopses for LCC (Section 4.2.1), and then show how to adapt LCCA to the hierarchy (Section 4.2.2).

4.2.1 Contraction for LCC. We first extend the notion of L-connection to supernodes in the hierarchy. A node u is *L-connected* to a supernode v_H if it is L-connected to all the nodes in G that are contracted to v_H. Similarly, a supernode v_{H1} is *L-connected* to v_{H2} if all nodes contracted to v_{H1} are L-connected to v_{H2}.

To reduce decontraction, we use synopsis $S^i_{\mathsf{LCC}}(v_H)$ of supernode v_H in the contracted graph G_i for LCC. It extends the shared synopsis $S^i(v_H)$ (Section 3.2) with an extra tag labels that collects the labels of all the nodes in G that are contracted to v_H, where

- at level 0, $S^0_{\mathsf{LCC}}(v)$.labels = $\{L(v)\}$;
- at level i, $S^i_{\mathsf{LCC}}(v_H)$.labels = $\bigcup_{f_C^i(v)=v_H} S^{i-1}_{\mathsf{LCC}}(v)$.labels.

Example 4: In Fig. 1, $S^i_{\mathsf{LCC}}(v_H)$ extends $S^i(v_H)$ as follows. (1) In G_0, $S^0_{\mathsf{LCC}}(v)$.labels=$\{L(v)\}$ for all $v \in V$. (2) In G_1, $S^1_{\mathsf{LCC}}(v)$.labels = $\{u\}$ for $v \in \{s_1, s_3, s_5, u_{14}\}$; $S^1_{\mathsf{LCC}}(v)$.labels=$\{k\}$ for $v \in \{s_2, s_4, k_{11}\}$; and $S^1_{\mathsf{LCC}}(t_1)$.labels = $\{t\}$ for $v \in \{t_1, t_2\}$. (3) In G_2, $S^2_{\mathsf{LCC}}(p_1)$.labels = $\{u\}$, $S^2_{\mathsf{LCC}}(p_2)$.labels = $\{k, u\}$ and $S^2_{\mathsf{LCC}}(p_3)$.labels = $\{k, t\}$. □

4.2.2 LCC algorithm. Below we first review LCCA [10, 16]. We then adapt the algorithm to hierarchy $\mathcal{H}$, referred to as $\mathsf{LCCA_h}$.

LCCA. Given a graph G and a query $Q = (L, u, v)$, LCCA applies breadth-first-search (BFS) starting from u. It maintains a set S of nodes that are L-connected from u, initialized as $S = \{u\}$. At each step, LCCA selects a node $s \in S$, and expands the search from s to a newly visited node w via edge (s, w); if $L(w) \in L$, then node w is L-connected from u and is added to the set S. It terminates with true if v is visited and added to S, *i.e., u is L-connected to v*, and it stops with false otherwise if the search cannot be further expanded.

Algorithm $\mathsf{LCCA_h}$. $\mathsf{LCCA_h}$ adopts the same logic as LCCA. It expands the search from v'_H, a supernode that is L-connected from u, to a newly visited supernode v_H in graph G_i at level i if one of the following conditions is satisfied: (1) v'_H is contracted into v_H with $S^i_{\mathsf{LCC}}(v_H)$.labels $\subseteq L$; (2) $f_C^{i+1}(v'_H) = f_C^{i+1}(v_H) = v_S$ and there exists an L-path from v'_H to v_H; it checks such an L-path using $S^{i+1}_{\mathsf{LCC}}(v_S)$, *e.g.,* when v'_H and v_H are two leaves of a star, and the labels of the central node and v_H are covered by L; if v_S is at the top level, it decontracts v_S, which is essentially edge decontraction; or (3) there exists a supernode v''_H such that (a) $(v'_H, v''_H) \in G_{i+1}$, (b) the set $S^{i+1}_{\mathsf{LCC}}(v''_H)$.labels is not covered by L, (c) decontraction of superedge (v'_H, v''_H) includes an edge with endpoint v_H, and (d) $S^i_{\mathsf{LCC}}(v_H)$.labels $\subseteq L$. $\mathsf{LCCA_h}$ stops when (1) it reaches v or a supernode v_H that contracts v, or (2) the search cannot be expanded.

Note that conditions (1) and (2) are checked by using f_R^i and synopses without decontraction of any regular structures, and condition (3) needs to decontract only superedges. In the entire process no decontraction of supernodes of regular structures is needed.

Example 5: Given a query $(\{u\}, u_8, u_{10})$ on hierarchy $\mathcal{H}$ of Fig. 1, $\mathsf{LCCA_h}$ starts from G_2 at the top level: (1) initially, $S = \{u_8\}$; (2) it expands the search to p_1 in G_2 since $S^2_{\mathsf{LCC}}(p_1).\mathsf{labels} = \{u\} \subseteq L$; (3) it skips p_3 as its labels contain no elements in L; (4) since the labels of p_2 are not covered by L, $\mathsf{LCCA_h}$ decontracts superedge (p_1, p_2) and finds that u_{14} in G_1 is L-connected from u_8; (5) it further finds an L-path $\langle u_{14}, s_5\rangle$ in p_2 by checking synopses; (6) the search terminates since u_{10} is contracted to s_5, and $\mathsf{LCCA_h}$ returns true. □

Analyses. One can verify the correctness of $\mathsf{LCCA_h}$ by induction on the depth of $\mathcal{H}$ as it follows the same logic as LCCA. It is efficient since (a) it operates on contracted graphs, much smaller than the original G; and (b) it checks synopses for L-paths, which reduce expansion and validation costs, *e.g.,* it may find a supernode v_H as a whole that is L-connected from the source u if $S^i_{\mathsf{LCC}}(v_H).\mathsf{labels} \subseteq L$.

4.3 Graph Pattern Matching

We next study subgraph isomorphism [24, 39], which is widely used in graph queries [7, 34, 75, 76] and graph dependencies [31, 32].

Pattern matching. A *graph pattern* is a graph $Q = (V_Q, E_Q, L_Q)$. A *match* of pattern Q in graph G is a subgraph $G' = (V', E', L')$ of G that is isomorphic to Q, *i.e.,* there is a *bijective function* $h : V_Q \to V'$ such that (1) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (2) $e = (u, u')$ is an edge in Q iff $(h(u), h(u'))$ is an edge in G and $L_Q(u') = L(h(u'))$. We denote by $Q(G)$ the set of all matches of Q in G.

To simplify the discussion, we consider connected patterns Q. This said, our algorithm can be adapted to disconnected ones.

The *graph pattern matching problem*, denoted by SubIso, is to compute, given a pattern Q and a graph G, the set $Q(G)$ of matches.

Similar to LCC, SubIso is labeled. In contrast to LCC, SubIso is local. Denote by d_Q the *diameter* of Q, *i.e.,* the maximum shortest distance between any two nodes in Q; then any v_1 and v_2 in a match G' of Q in G are within d_Q hops. GraphChi cannot handle SubIso well.

The graph pattern matching problem is known NP-complete (cf. [35]). We adapt a conventional algorithm VF2 [24] for SubIso to the contraction hierarchy $\mathcal{H}$. Other algorithms, *e.g.,* TurboIso [39] with indices, can also be adapted to the contraction hierarchy.

4.3.1 Contraction for SubIso. Observe that if a node v in graph G matches a node u in pattern Q, then the labels of v's neighbors must cover the labels of u's neighbors. The synopsis $S^i_{\mathsf{SubIso}}(v_H)$ extends LCC synopsis $S^i_{\mathsf{LCC}}(v_H)$ with an extra tag nlabels that collects the labels of the neighbors of the nodes contracted to v_H, where

- at level 0, $S^0_{\mathsf{SubIso}}(v).\mathsf{nlabels} = \{L(v') \mid (v, v') \in E\}$;
- at level i, $S^i_{\mathsf{SubIso}}(v_H).\mathsf{nlabels} = \bigcup_{f^i_C(v)=v_H} S^{i-1}_{\mathsf{SubIso}}(v).\mathsf{nlabels}$.

Example 6: Consider $\mathcal{H}$ of Fig. 1. (1) In G_0, $S^0_{\mathsf{SubIso}}(u_1).\mathsf{nlabels} = \{u\}$, $S^0_{\mathsf{SubIso}}(u_2).\mathsf{nlabels} = \{u, k, t\}$, $S^0_{\mathsf{SubIso}}(k_1).\mathsf{nlabels} = \{u, k\}$; similarly for other nodes. (2) In G_1, $S^1_{\mathsf{SubIso}}(s_1).\mathsf{nlabels} = \{u, k, t\}$; the same for s_2, t_1, t_2 and u_{14}; $S^1_{\mathsf{SubIso}}(s_3).\mathsf{nlabels} = \{u, k\}$; the same for s_4 and s_5; and $S^1_{\mathsf{SubIso}}(k_{11}).\mathsf{nlabels} = \{u\}$. (3) In G_2, $S^2_{\mathsf{SubIso}}(p_1) = \{u, k, t\} = S^2_{\mathsf{SubIso}}(p_2) = S^2_{\mathsf{SubIso}}(p_3) = \{u, k, t\}$. □

4.3.2 SubIso algorithm. Below we first review VF2 [24]. We then adapt the algorithm to the hierarchy $\mathcal{H}$, referred to as $\mathsf{SubA_h}$.

VF2. Given a graph G and a pattern Q, algorithm VF2 computes $Q(G)$ by backtracking. It expands a partial mapping, denoted as s and initialized as $\emptyset$, by iteratively adding node pairs to obtain an isomorphism between pattern Q and a subgraph G' of G. Node pairs, maintained in a set P, are identified by checking (a) syntactic feasibility that depends only on the structure of the graphs, and (b) semantic feasibility that depends on labels. VF2 (1) backtracks if $P = \emptyset$, *i.e.,* no node pairs can be added to s to obtain a complete mapping, or (2) branches from each node pair in P to expand s. It expands $Q(G)$ with valid complete mappings identified in the process.

Algorithm $\mathsf{SubA_h}$. $\mathsf{SubA_h}$ adopts the same logic as VF2 except the following minor adaptions. (1) It adds a node pair (u, v_H) to P if (a) $L_Q(u) \in S^i_{\mathsf{SubIso}}(v_H).\mathsf{labels}$; (b) $L_Q(u') \in S^i_{\mathsf{SubIso}}(v_H).\mathsf{nlabels}$ for all neighbors u' of u in Q; and (c) for each $(u', v') \in s$ such that $(u, u') \in E_Q$, v_H either contains v' or connects to some supernode containing v'. (2) To branch from a node pair (u, v_H) in P to expand s, $\mathsf{SubA_h}$ recursively replaces (u, v_H) by (u, v'_H), where v'_H is a supernode in the next level contracted to v_H and can match u by satisfying condition (1), using nodes in f^i_R and synopsis S^i_{SubIso}. Conditions (1a) and (1b) are checked with synopsis S^i_{SubIso} and f^i_R; and condition (1c) is checked by superedge decontraction and f^i_R. In this way it adds node pairs (u, v) for level-0 nodes v in G.

In the entire process, no contracted subgraphs are restored.

Example 7: Query Q in Fig. 1(f) is to find potential friendships based on retweets, keywords and common friends. Nodes u, u' and u'' have label u. Given Q, $\mathsf{SubA_h}$ starts from G_2 at the top level: (1) it chooses t as the start node, to which only p_3 can match, hence it adds (t, p_3) to P; (2) to branch from (t, p_3), $\mathsf{SubA_h}$ replaces (t, p_3) with (t, t_1) and (t, t_2); (3) for (t, t_1), $\mathsf{SubA_h}$ next chooses k as the node to match and adds (k, p_3) and (k, p_2) to P; (4) $\mathsf{SubA_h}$ then replaces (k, p_3) by (k, k_5) and skips (k, p_2) since decontraction of superedge (p_3, p_2) cannot find a node contracted to p_2 having an edge with t_1; (5) in a similar manner, it matches t, k, t', u, u', u'' with $t_1, k_5, t_2, u_2, u_{14}, u_1$, respectively; (6) for (t, t_2), it matches t, k, t', u, u', u'' with $t_2, k_5, t_1, u_{14}, u_2, u_1$, respectively. □

Analyses. $\mathsf{SubA_h}$ is correct since it follows the same logic as VF2 excepts it adopts pruning strategies. While the two have the same worst-case complexity, by checking synopses, $\mathsf{SubA_h}$ can reduce expansion and validation costs as well as skipping supernodes as a whole. In each step, at most one superedge is decontracted.

4.4 CC, CD and RPQ

We next study connected component (CC), clique decision (CD) and regular path query (RPQ). For the lack of space, below we present only main ideas of adapting algorithms to the scheme for the three.

CC. The *connected component problem* (CC) [25, 71] is to compute the set of pairs (s, n) for a given graph G, where (s, n) indicates that there are n connected components in G, each consisting of s nodes. CC is non-local and non-labeled. It is widely used in pattern recognition [40, 43], graph partition [72] and random walk [41].

Observe that each subgraph H contracted to a supernode v_H is connected, no matter whether H is a regular structure (by topology contraction) or a set of edges (by edge contraction; see Section 2). Based on this, we adapt the algorithm of [71] for CC to the hierarchy. The synopses defined in Section 3.2 suffice for us to compute CC,

Algorithm IncHCon
Input: A contraction hierarchy $\mathcal{H}$ consisting of k levels $\langle f_C^i, \mathcal{S}^i, f_D^i, f_R^i\rangle$ and k contracted graphs G_i of a graph G, and updates ΔG to G.
Output: New contracted graphs $G_i \oplus \Delta G_i$ at each level.
1. $i := 0$; $\Delta G_i := \Delta G$; $T(G) :=$ ordered set of regular structures of G;
2. **while** $i < k$ and $\Delta G_i \neq \emptyset$ **do**
3. $V_s := \emptyset$; $\Delta G' := \emptyset$;
4. group edges of updates in ΔG_i; reduce ΔG_i;
5. **for each** group of edges $\Delta E \subset \Delta G_i$ with a representative (u, v) **do**
6. $v_H := f_C^{i+1}(u)$; /* $f_C^{i+1}(u) = f_C^{i+1}(v)$ */
7. recover subgraph H contracted to supernode v_H;
8. IncCR $(H, \Delta E, T(G), v_H, G_{i+1}, V_s, \Delta G')$;
9. contract $(V_s, G_i, T(G), \Delta G')$; $\Delta G_{i+1} = \Delta G'$; $i := i + 1$;
10. return $G_i \oplus \Delta G_i$;

Procedure IncCR
Input: a subgraph H, updates ΔE, ordered regular structures $T(G)$, a supernode v_H, a contracted graph G_{i+1}, singleton vertex set V_s and updates $\Delta G'$.
Output: Updated G_{i+1}.
1. $H := H \oplus \Delta E$;
2. **if** H is regular and H can fit in buffer size
3. **then** write the buffered edges to disk;
4. **else** contract regular structures in H, and update f_D^{i+1};
5. add singleton nodes to V_s, and collect updated superedges in $\Delta G'$;

Figure 4: Algorithm IncHCon

and we need to decontract neither supernodes nor superedges.

CD. A *clique* in a graph G is a subgraph C in which there are edges between any two nodes. It is a t-clique if the number of nodes in C is t (*i.e.,* $|V(C)| = t$). The *clique decision problem* (CD) [17, 45] is to find whether there exists a t-clique in G for a given natural number t. CD is non-labeled and local. It finds applications in community search [63], team formation [48] and anomaly detection [13, 53].

We adapt the algorithm of [45] for CD to the hierarchy, using (1) cliques in G contracted into supernodes in G_1 to find an initial maximum clique, and (2) the degree of node v as an upper bound of the maximum clique containing v, for pruning. The algorithm decontracts superedges only, but decontracts no supernodes.

RPQ. Consider a regular expression r. On a graph G with edge labels, a *regular path* r specifies a set of paths such that the labels on the edges of each path form a word in the language of r. A *regular path query* r (RPQ) [58] is to return all node pairs (u, v) such that there exists such a matching path from u to v.

RPQ is non-local and is labeled (it imposes constraints on edge labels). It is of vital importance in semantic web [37, 64], biological networks [51] and social network analysis [68].

We adapt the algorithm of [78] for RPQ to the hierarchy, by defining synopses in terms of (1) the labels of edges contracted into supernodes, and (2) the labels of edges that are adjacent to some nodes contracted into the supernodes. The algorithm decontracts superedges only, but decontracts no supernodes.

5 INCREMENTAL CONTRACTION

Real life graphs are often dynamic. In light of this, we next develop an incremental algorithm, denoted by IncHCon, to maintain the contraction hierarchy $\mathcal{H}$ in response to updates ΔG to graph G. Hierarchy $\mathcal{H}$ is built once offline by algorithm HCon (Section 3.1). It is then incrementally maintained by IncHCon online.

Problem. We consider batch updates ΔG, which are sequences of edge insertions, deletions and label updates. Vertex updates are a dual of edge updates [46] and can be processed accordingly.

Given contraction hierarchy $\mathcal{H}$ of k levels $\langle f_C^1, \mathcal{S}^1, f_D^1, f_R^1\rangle$, $\ldots$, $\langle f_C^k, \mathcal{S}^k, f_D^k, f_R^k\rangle$ with contracted graphs $G_1, \ldots, G_k$, and batch updates ΔG, the *incremental hierarchical contraction problem*, denoted as IHP, is to compute (a) changes ΔG_i to G_i at each level i such that $G_1 \oplus \Delta G_1 = f_C^1(G \oplus \Delta G)$ and $G_i \oplus \Delta G_i = f_C^i(G_{i-1} \oplus \Delta G_{i-1})$, *i.e.,* to get the contracted graph at level i of the updated $G_{i-1} \oplus \Delta G_{i-1}$, where $G_i \oplus \Delta G_i$ applies ΔG_i to G_i; (b) the updated synopses $\mathcal{S}^i$; and (c) functions f_D^i and f_R^i of the new contracted $f_C^i(G_{i-1} \oplus \Delta G_{i-1})$.

Criterion. Following [66], we measure the complexity of an incremental hierarchical contraction algorithm in terms of the size of the *affected area*, denoted by AFF. Here AFF includes (a) the changes ΔG to graph G, *i.e.,* the changed region of the input, (b) the changes ΔG_i to each level in the contraction hierarchy, *i.e.,* changes to the output $\mathcal{H}$, and (c) edges with at least one endpoint in (a) or (b).

An incremental algorithm is said to be *bounded* if its complexity is determined by $|\text{AFF}|$ (the three cases above), not by size $|G|$ of the entire graph G. An incremental problem is *bounded* if there exists a bounded incremental algorithm for it, and is *unbounded* otherwise.

Intuitively, ΔG is typically small in practice. When ΔG is small, so is ΔG_i for G_i at each level. Hence when ΔG is small, a bounded incremental algorithm is often far more efficient than a batch algorithm that recomputes each G_i of $\mathcal{H}$ starting from scratch, since the cost of the latter depends on the size of possibly big G.

Challenges. Problem IHP is nontrivial. (1) Regular structures are fragile. For instance, when inserting an edge between two leaves of a star H, H is no longer a star, and its nodes may need to be contracted into other regular structures. (2) When a contracted graph G_i is changed, so are its synopses and decontraction. (3) Edge insertions may make some contracted parts exceed the buffer size. (4) Maintenance of the hierarchy $\mathcal{H}$ may incur heavy I/O cost.

Main result. Despite the challenges, we show that bounded incremental hierarchical contraction is within the reach in practice.

Theorem 2: *Problem* IHP *is bounded for* PR, LCC, SubIso, CC, CD *and* RPQ*; it takes at most* $O(k|\text{AFF}|^2)$ *time. Here* k *is the depth of hierarchy* $\mathcal{H}$*, which is usually a small constant 2–4 in practice.* □

We give a constructive proof of Theorem 2, consisting of two parts: (1) the maintenance of the contracted graph G_{i+1} and its associated functions f_D^{i+1} and f_R^{i+1} at each level $i+1$ in the hierarchy $\mathcal{H}$ in response to updates ΔG_i to G_i; and (2) the maintenance of the synopses of affected supernodes in each G_{i+1}.

(1) Incremental algorithm. We provide an incremental algorithm, denoted by IncHCon, in Fig. 4. Since edge labels have no impact on the contraction algorithm, IncHCon deals with edge insertions and deletions only. In each iteration, IncHCon incrementally maintains the contracted graph G_{i+1} in response to ΔG_i to G_i (lines 2-9). Initialized as $\Delta G_0 = \Delta G$ (line 1), ΔG_i is obtained in each iteration (line 9). It has three steps as follows. To simplify the discussion, we focus on how to update G_{i+1} with ΔG_i; the maintenance of f_R^{i+1} and f_D^{i+1} for supernodes are similar, as a byproduct.

(a) Preprocessing. IncHCon recalls the order $T(G)$ on regular struc-

tures (line 1), which is decided by the type of G, not by G itself (see Section 3.1). The pre-computed $T(G)$ is used when a structure is no longer regular due to updates or when it exceeds the buffer size.

In each iteration, IncHCon identifies an initial area affected by update ΔG_i, maintains it in a set V_s, and tracks the changes to G_{i+1} in $\Delta G'$ for the next iteration (line 3). It then groups updates in ΔG_i such that two updates of edges (u_1, v_1) and (u_2, v_2) are in the same group if $f_C^{i+1}(u_1) = f_C^{i+1}(u_2)$ and $f_C^{i+1}(v_1) = f_C^{i+1}(v_2)$ (line 4). Each group, denoted by ΔE, is represented by an arbitrary (u, v) in ΔE. By grouping, updates to a supernode or a superedge are aggregated, to be carried out by few sequential disk accesses, to minimize the I/O cost. IncHCon next removes "unaffected" updates from ΔG_i that have no impact on G_{i+1} (line 4), *i.e.,* groups with representative (u, v) having $f_C^{i+1}(u) \neq f_C^{i+1}(v)$. These updates are made to superedges of G_{i+1} that are recorded in f_D^{i+1} and written to the disk.

(b) Updating. Algorithm IncHCon then updates G_{i+1} (lines 6-9). For each group ΔE of edges with representative (u, v) that have $f_C^{i+1}(u) = f_C^{i+1}(v) = v_H$ (line 6), IncHCon recovers the subgraph H contracted to v_H either by synopsis or by edge decontractions when v_H is at the top level (line 7). It then invokes procedure IncCR to update H by ΔE (line 8). Now some nodes may become "singletons" when a regular structure is decomposed by the updates, *e.g.,* leaves of a star. It collects such singleton nodes in the set V_s.

More specifically, procedure IncCR applies updates in ΔE to H (line 1). When H is still regular, *e.g.,* adding an edge into a diamond makes it a clique, if the updated H can fit in the buffer, updating H is done by writing the buffered edges to disk (line 2-3). If H is non-regular, IncCR contracts regular structures in H as in HCon (line 4). Here updated superedges include those inside H and those adjacent to v_H in G_{i+1}. All singleton nodes are added to V_s, and those added/removed superedges are collected in $\Delta G'$ (line 5).

(c) Contraction. Finally, IncHCon processes nodes in V_s (line 10). It (a) iteratively loads into memory the nodes in V_s and their adjacent edges, and contracts regular structures following the order $T(G)$, or (b) leaves nodes v as singletons, *i.e.,* $f_C^{i+1}(v) = v$. Moreover, it updates decontraction f_D^{i+1} and writes the buffered edges to disk.

Example 8: For an edge (u, v), denote by $(u, v)^+$ and $(u, v)^-$ updates of inserting (u, v) and deleting (u, v), respectively. Consider updating graph G_0 of Fig. 1(a) with $\Delta G = \{(k_{11}, u_9)^+, (k_8, k_9)^-, (k_6, k_7)^-, (k_1, k_2)^-, (k_2, k_3)^-\}$. IncHCon works as follows.

In the preprocessing phase, IncHCon adopts $T(G)$ = [clique, diamond, butterfly, star] since G is a social network. It then groups updates in ΔG into three groups: $\Delta E_1 = \{(k_{11}, u_9)^+\}$, $\Delta E_2 = \{(k_8, k_9)^-, (k_6, k_7)^-\}$ and $\Delta E_3 = \{(k_1, k_2)^-, (k_2, k_3)^-\}$. Now ΔE_1 is reduced and written to the disk to update $f_D^1(k_{11}, s_5)$.

In the first iteration (to maintain G_1 in response to ΔG to G), (1) the updating step first applies ΔE_2 to the butterfly contracted to s_4, denoted as H. Then H becomes a star, which is still regular; (2) it then applies ΔE_3 to the clique contracted to s_2, denoted as H'. Now H' becomes non-regular. It is contracted into a clique consisting of nodes $\{k_1, k_3, k_4, k_5\}$, along with a singleton node k_2. A new superedge (k_2, s_2) is built and the original superedge (s_2, s_3) is replaced by (s_2, s_3) and (s_3, k_2). Such updates are written to the disk. Moreover, $V_s = \{k_2\}$ and $\Delta G_1 = \{(k_2, s_2)^+, (s_3, k_2)^+\}$. No structures can be contracted from the nodes in V_s.

In the second iteration (for maintaining G_2 in response to ΔG_1 to G_1), all updates in ΔG_1 are reduced in preprocessing. As a consequence, no updating and contraction are needed. □

Analyses. Recall the definition of AFF earlier in this section. Algorithm IncHCon takes $O(|\mathsf{AFF}|^2)$ time. Indeed, (a) its preprocessing step is in $O(|\mathsf{AFF}|)$ time, since each iteration is in $O(|\Delta G_i|)$ time; (b) the updating step takes $O(|\mathsf{AFF}|)$ time, since each subgraph H has a bounded size; and (c) the cost of the step for contracting V_s is in $O(|\mathsf{AFF}|^2)$ time, since (1) this step considers only nodes in V_s and their adjacent edges, and the total size is bounded by $|\mathsf{AFF}|$; and (2) the dominating part in this step is contracting cliques and stars; one can verify that it takes $O(|\mathsf{AFF}|^2)$ time along the same lines as the analysis of the contraction algorithm HCon (Section 3.1).

The algorithm is (a) bounded, since its cost is determined by $|\mathsf{AFF}|$ alone [66], and (b) *local* [27], *i.e.,* the changes are confined to affected supernodes and their neighbors in each G_i. The contracted graphs G_i incrementally maintained by IncHCon may differ from those of HCon since singleton nodes in V_s may be contracted in different orders. Nonetheless, we find that the differences are small. Moreover, such G_i's are compact and cannot be further contracted.

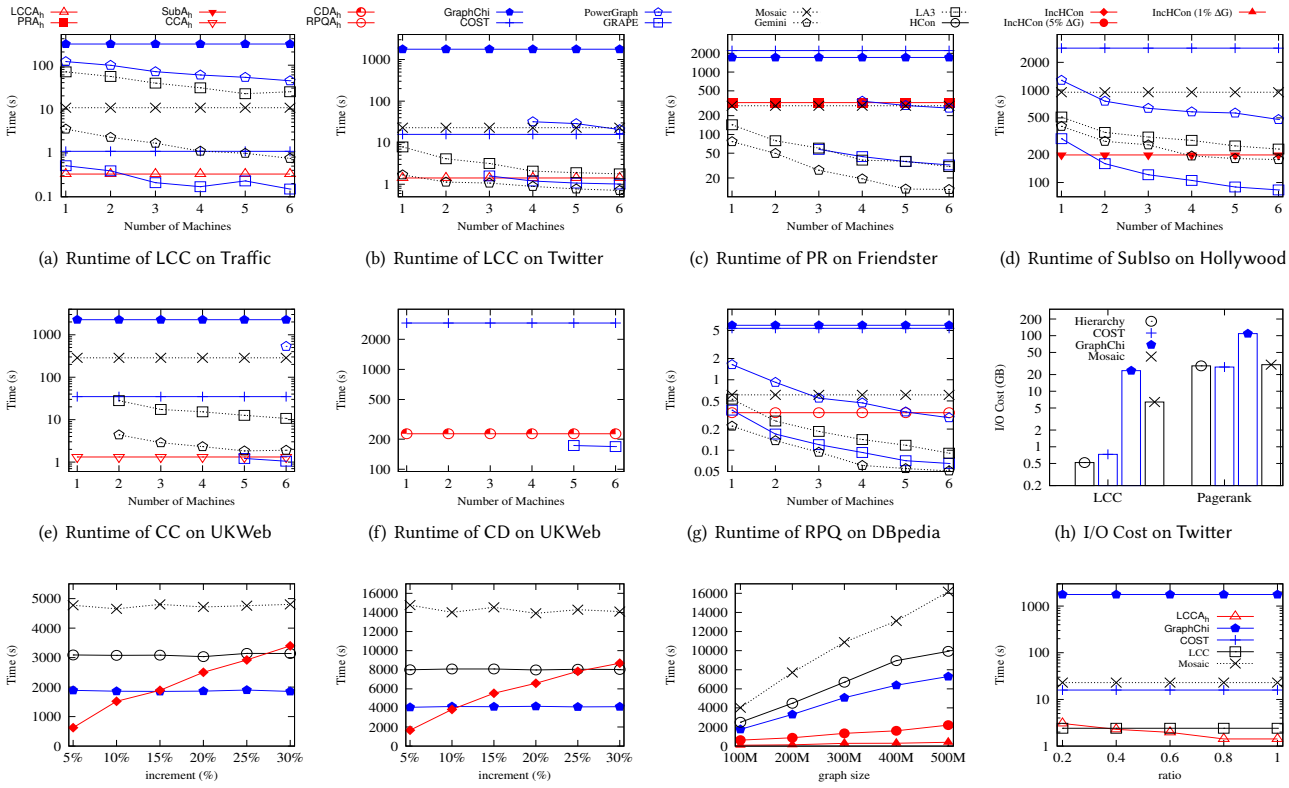
(2) Synopses maintenance. We next show that for LCC, SubIso, PR, CC, CD and RPQ, (a) at most $O(k|\mathsf{AFF}|)$ supernodes have affected synopses, where k is the depth of hierarchy $\mathcal{H}$, and (b) the synopsis for each supernode can be updated in $O(|\mathsf{AFF}|)$ time. These also apply to edge label updates, which affect only supernodes that contain or are adjacent to the updated edges. Thus incremental synopses maintenance for each of these is in $O(k|\mathsf{AFF}|^2)$ time.

To see these, consider a supernode v_H in G_i. The shared synopsis $S^i(v_H)$ stores the type, key feature and total number of level-0 nodes contracted to v_H (Section 3.2). One can verify that the number of supernodes whose synopses are affected is at most $k|\mathsf{AFF}|$ (as a node in AFF can influence at most k supernodes at higher levels). Moreover, $S^i(v_H)$ is confined to v_H, and can be updated in $O(1)$ time. Thus the maintenance of shared synopses S takes at most $O(k|\mathsf{AFF}|)$ time. In addition, the maintenance of extended synopses for each of the six query classes takes at most $O(k|\mathsf{AFF}|^2)$ time. For example, for LCC, $S^i_{\mathsf{LCC}}(v_H)$ extends $S^i(v_H)$ with $S^i_{\mathsf{LCC}}(v_H)$.labels; note that $S^i_{\mathsf{LCC}}(v_H)$.labels is updated in a similar manner as $S^i_{\mathsf{LCC}}(v_H)$.size; hence it takes at most $O(k|\mathsf{AFF}|)$ time to maintain S_{LCC}.

Example 9: Continuing with Example 8, we show how to maintain $S^i_{\mathsf{SubIso}}(v_H)$.nlabels for supernodes v_H in G_i; $S^i(v_H)$, $S^i_{\mathsf{LCC}}(v_H)$ and $S^i_{\mathsf{PR}}(v_H)$ are simpler. (1) For (unaffected) ΔE_1, $S^1_{\mathsf{SubIso}}(v_H)$.nlabels remains the same for all v_H in G_1. (2) By updates ΔE_2, s_4 becomes a star, while $S^1_{\mathsf{SubIso}}(s_4)$.nlabels remains the same. (3) Updates ΔE_3 make s_2 a 4-clique; $S^1_{\mathsf{SubIso}}(s_2)$.nlabels and $S^1_{\mathsf{SubIso}}(s_3)$.nlabels remain the same while $S^1_{\mathsf{SubIso}}(k_2)$.nlabels $= \{u, k\}$. Update of $S^1_{\mathsf{SubIso}}(k_2)$.nlabels has no impact on $S^2_{\mathsf{SubIso}}(p_3)$.nlabels. □

6 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we experimentally evaluated (1) the effectiveness of the hierarchical contraction scheme, (2) the impact of contracting each regular structure, (3) the space cost of the hierarchy, (4) the efficiency of the contraction and incremental contraction algorithms, and (5) the scalability of our scheme.



(a) Runtime of LCC on Traffic (b) Runtime of LCC on Twitter (c) Runtime of PR on Friendster (d) Runtime of SubIso on Hollywood

(e) Runtime of CC on UKWeb (f) Runtime of CD on UKWeb (g) Runtime of RPQ on DBpedia (h) I/O Cost on Twitter

(i) HCon, IncHCon on Friendster (j) HCon, IncHCon on UKWeb (k) Scalability on $|G|$ (l) Scalability on $M/|G|$

Figure 5: Performance evaluation

Experiment setting. We start with the setting.

(1) Graphs. We used 7 real-life graphs: (a) DBpedia [50], a knowledge base with 4.9M entities and 54M relationships, (b) Traffic [1], a road network with 23M nodes and 29M edges, (c) Twitter [59], a social network with 42M users and 1.5B links, (d) Friendster [3], a social network with 65M users and 1.8B links, (e) UKWeb [2], a Web graph with 106M nodes and 3.7B edges, (f) Hollywood [15], a collaboration network with 1.1M nodes and 56M edges, and (g) Patent [52], a citation network with 3.8M nodes and 16.5M edges.

Here DBpedia (resp. Traffic, Hollywood, Patent) is the largest knowledge base (resp. road, collaboration and citation network) that we could find. We set memory limit M as 200MB for these four (100MB for Patent), so that the graphs do not fit in the memory. For the other three graphs, we set M as 4GB. We set buffer size B as 100MB. To accurately evaluate the impact of M, we limited caching space used by the operating system within 5% of memory M.

We also generated synthetic graphs with up to 500 million nodes and 6 billion edges, to test the scalability of our scheme.

Updates. We randomly generated updates ΔG controlled by size $|\Delta G|$. We keep the ratio of edge insertions to deletions as 1 unless stated otherwise, *i.e.,* the sizes of updated graphs remain unchanged.

(2) Implementation. We implemented the following, all in C++. (1) Algorithms PRA_h (Section 4.1.2), LCCA_h (Section 4.2.2), SubA_h (Section 4.3.2), CCA_h for CC, CDA_h for CD, and RPQA_h for RPQ (Section 4.4). (2) Our contraction algorithm HCon and incremental IncHCon. (3) Baselines PRA, LCCA, VF2, CCA, CDA and RPQA that run on competitor systems.

We tested another 7 systems as baselines: (a) GraphChi [47] and Mosaic [56], two disk-based single-machine graph systems (see Section 1); (b) COST [60], another disk-based single-thread system; and (c) four parallel systems: graph-centric GRAPE [4, 33] and vertex-centric PowerGraph [36], Gemini [80] and LA3 [6]. All the parallel systems are in-memory solutions.

On a single machine with 16 cores, we implement the query answering algorithms to explore on-chip parallelism as follows: (1) we edge-cut [69] a graph into fragments, such that each core can operate on its designated fragment in parallel; and (2) the cores synchronize and communicate with each other via shared memory.

The hierarchy $\mathcal{H}$ is stored as follows: (1) contracted graphs G_i are maintained by adjacency lists; (2) functions f_C^i and f_R^i are maintained by hashmaps; (3) decontraction functions f_D^i are stored as lists such that the edges in the same contracted subgraph are stored sequentially; and (4) synopses are stored as user-defined structures.

(3) Environment. Experiments were conducted on a HPC cluster, with machines powered by Xeon 2.5GHz, 64GB RAM and 10Gbps NIC. We tested HCon, IncHCon, GraphChi, Mosaic and COST on a single machine with $c = 16$ cores, while parallel systems used up to $n = 6$ machines (96 cores). Note that COST can use only one core since it is a single-thread system. We used SSD to store data. Each experiment was run 5 times, and the average is reported here.

Experimental results. We now report our findings.

Exp-1: Effectiveness. We first tested the performance of query answering with the contraction scheme. As remarked earlier, none of the seven graphs fits into memory of size M above. We fix $t_p = 0.7$

Graph	LCC			PR			SubIso		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
DBpedia	47.1	-	-	4.5	-	-	20.7	-	-
Traffic	26.2	9.9	-	2.5	1.7	-	13.5	3.9	-
Twitter	30.8	19.3	8.0	4.6	2.5	2.1	10.7	8.1	2.3
Friendster	22.4	3.7	4.5	3.6	1.0	0.8	21.5	6.7	5.1
UKWeb	34.1	20.3	2.7	4.6	1.5	0.9	30.4	12.0	6.6
Hollywood	31.5	16.3	8.2	4.9	2.1	0.6	19.1	15.2	10.3
Patent	51.3	7.2	4.9	7.1	1.5	1.0	39.2	8.1	5.2

Table 3: Slowdown(%) by disabling certain regular structures

and show that a single machine is able to compute exact answers with a hierarchy of depth of 2-4. In contrast, when n is small, parallel systems ran into memory overflow when querying some graphs, *e.g.,* as shown in Table 4, GRAPE needs 283G memory to load UKWeb alone (with labels), which do not fit in four machines.

Label-constrained connectivity. Figures 5(a)–5(b) report runtime of LCCA_h on the contraction hierarchy using a single machine compared with baselines, by varying n from 1 to 6. As shown there, on average, (1) LCCA_h is 1083.5, 24.3 and 7.3 times faster than GraphChi, Mosaic and COST on Traffic and Twitter, respectively. (2) Single-machine LCCA_h is even faster than parallel PowerGraph, GRAPE, Gemini and LA3 that are equipped with 6, 2, 1 and 6 machines, respectively. It is on average 132.1 times faster than PowerGraph when n varies from 1 to 6, respectively.

Figure 5(h) reports the I/O cost for superedge/node decontraction of LCCA_h, compared with the I/O cost of COST, GraphChi and Mosaic. On average, LCCA_h takes only 2.3%, 71.5% and 8.1% of the I/O cost of GraphChi, COST and Mosaic, respectively. We do not compare the I/O cost of LCCA_h with the communication cost of parallel systems since the former is the amount of graph data read into memory while the latter is the amount of intermediate data transferred between machines, which are significantly different.

Intuitively, LCCA_h is efficient since (a) it operates on smaller contracted graphs; (b) it inherits graph-level optimization of LCCA; and (c) it finds an L-connected supernode from source node as a whole, and skips a supernode in the hierarchy without decontraction as long as its synopsis (encoding the label set of the nodes contracted to the supernode) contains no label in the query.

PageRank. Figure 5(c) shows that on average, (1) PRA_h is 5.3 and 6.8 times faster than GraphChi and COST on Friendster, respectively. (2) It is 12.4% slower than Mosaic on Friendster, since PRA_h has to decontract superedges. (3) Parallel systems ran faster than PRA_h when using sufficient machines, since I/O dominates the cost of PR. (4) As shown in Fig. 5(h), the I/O cost of PRA_h is only 23.5% of that of GraphChi, and is comparable to those of COST and Mosaic.

Subgraph isomorphism. As shown in Fig. 5(d), (1) SubA_h beats COST and Mosaic by 14.4 and 4.8 times on Hollywood. (2) Single-machine SubA_h is faster than GRAPE, PowerGraph, Gemini and LA3 when the latter used $n = 1, 6, 3, 6$ machines, respectively. GraphChi ran out of memory or spends more than 2 hours, since it requires substantial graph traversal and can not adopt graph-level optimization. SubA_h does well by using label synopses to prune supernodes.

Connected component, clique decision and regular path query. We find the following. (1) Figure 5(e) shows that on average, (a) CCA_h beats COST, GraphChi and Mosaic by 26.4, 1716.8 and 217.0 times on UKWeb, respectively. (b) CCA_h is 404.8, 1.4 and 8.1 times faster than PowerGraph, Gemini and LA3 when they use $n = 6$ machines, respectively, and it is faster than GRAPE with $n = 5$. CCA_h is efficient since it only uses G_k at the top of $\mathcal{H}$. In addition, it decontracts neither supernodes nor superedges, *i.e.,* it incurs no I/O.

(2) As shown in Figure 5(f), CDA_h is 12.9 times faster than COST on UKWeb, by using synopses to find an initial maximum clique that is near-optimal in size, while GraphChi, Mosaic, PowerGraph, Gemini and LA3 ran out of memory for CD.

(3) As shown in Figure 5(g), RPQA_h is 15.7, 17.3 and 1.8 times faster than COST, GraphChi and Mosaic on DBpedia, by using synopses to prune supernodes. Moreover, RPQA_h is faster than PowerGraph, GRAPE and LA3 when they use 5, 1 and 1 machines, respectively.

These also show that to perform comparably with memory-based approaches, our scheme could support query classes of different types even when the memory is only 7.6% of the graph size. For PR, LCC, SubIso, CC, CD and RPQ, it outperforms not only prior single-machine solutions COST, GraphChi and Mosaic, in both runtime and I/O cost, but also parallel systems that use more machines.

In principle, our hierarchical scheme is able to handle arbitrary queries on a graph of an arbitrary size with a single machine (subject to constraints of the underlying operating system), *i.e.,* there is no lower bound for the main-memory capacity for the input graphs. Nonetheless, when the memory is too small, I/O cost may substantially increase. We experimentally find that when the memory is 3% of the graph size, the query evaluation efficiency of our scheme is on average 9.1 times slower than memory-based system GRAPE with 6 machines, except CC that basically does not decontract.

Graph	HCon	GraphChi	Mosaic	COST	PowerGraph	GRAPE	Gemini	LA3
DBpedia	1.3G,159M	953M	979M	854M	5.3G	2.1G	6.1G	8.7G
Traffic	928M,152M	629M	831M	625M	3.1G	1.1G	4.9G	10.2G
Twitter	28.1G,2.98G	21G	27.4G	18.3G	126G	104G	72.1G	61.7G
Friendster	34.6G,3.89G	37G	39.1G	28.5G	148G	131G	90.7G	93.3G
UKWeb	60.5G,3.92G	62G	46G	51.3G	307G	283G	142.4G	221G
Hollywood	722M,185M	885M	801M	861M	3.4G	1.9G	5.2G	7.4G
Patent	285M,66M	254M	366M	237M	1.6G	0.7G	2.4G	4.7G

Table 4: Space cost of HCon and competitor systems

Exp-2: Impact of each structure. We next evaluated the impact of contracting each regular structure. Based on Table 2, we took contraction of the first 3 types of regular structures as the baseline, and tested the impact of each component on query evaluation by disabling it, using all the datasets. As shown in Table 3, the average slowdown by disabling each of the first 3 structures is (a) 34.8%, 12.8% and 5.7% for LCC, (b) 4.6%, 1.7% and 1.1% for PR, and (c) 22.1%, 9.0% and 5.9% for SubIso, respectively. We can see that the impact of each regular structure is consistent with the contraction order $T(G)$ (Section 3.1). The results on CC, CD and RPQ are consistent.

We also studied the impact of the contraction order of Section 3.1. We tested the impact of (a) RE, by reversing the order, and (b) EX, by exchanging between different types of graphs, *e.g.,* we used the order for social networks to contract road networks, which may not preserve the frequent regular structures of road networks. The average slowdown of RE and EX is 5.6% and 18.4%, respectively. These justify the contraction order proposed in Section 3.1.

Exp-3: Space. We tested space cost of our scheme by measuring its disk and memory usage, including the cost of storing node and edge labels. The contracted graphs G_1–G_{k-1} with their contraction

schemes are stored in disk, and G_k at the top level resides in memory. We measured the disk usage of disk-based GraphChi, Mosaic and COST, and the memory usage of memory-based ones, *i.e.,* all parallel systems. Table 4 shows the results in which the two numbers in the column for HCon report its disk and memory usage, respectively. As shown there, (1) HCon takes on average 22.8% more space than GraphChi, COST and Mosaic, while its query answering is much faster as shown earlier. (2) PowerGraph, GRAPE, Gemini and LA3 take on average 3.4 times more space than HCon.

Exp-4: Efficiency of (incremental) contraction. We next evaluated the efficiency of HCon and IncHCon versus GraphChi and Mosaic. Varying the size $|\Delta G|$ of updates from 5% of $|G|$ up to 30%, Figures 5(i)–5(j) report the results on Friendster and UKWeb, respectively. On average, (1) HCon is 1.8 times slower than GraphChi on Friendster and UKWeb, since contraction is more complicated than graph partition of GraphChi; nonetheless, query processing on the hierarchy contracted by HCon is much faster as shown earlier; this justifies the one-time offline contraction cost of HCon. HCon is 1.6 times faster than Mosaic, since Mosaic compresses edges by Hilbert order, which requires one global sorting. (2) The cost of synopses computation for the six query classes is low; it only accounts for 33.1% of the total cost of HCon. (3) Incremental IncHCon is faster than HCon even when $|\Delta G|$ is up to 25%$|G|$. It is 4.94 and 4.82 times faster on the two graphs when $|\Delta G| = 5\%|G|$, and is 28.3 and 32.2 times faster when $|\Delta G| = 1\%|G|$. The results are consistent when ΔG consists of insertions only or deletions only (not shown). These justify the need of incremental contraction.

Preprocessing. To explore on-chip parallelism of our hierarchy scheme, graphs are first edge-cut partitioned into fragments and then contracted into hierarchy by HCon, while GraphChi and Mosaic directly split graphs into disjoint intervals without edge-cut partitioning. We find that on average, the preprocessing time (including the costs of partitioning and HCon) of our scheme is 1.5 times faster than Mosaic, and is 1.9 times slower than GraphChi.

Exp-5: Scalability. Finally, we evaluated our (incremental) contraction scheme for its (1) scalability with graph size $|G|$, and (2) scalability with the ratio $M/|G|$ of memory limit to graph size.

Scalability on $|G|$. Varying the size $|G| = (|V|, |E|)$ of synthetic graphs from $(100M, 1.2B)$ to $(500M, 6B)$, we tested the scalability of HCon and IncHCon (fixing $|\Delta G| = 1\%|G|$ and $5\%|G|$ for IncHCon). We set memory limit M as 2GB. As shown in Fig. 5(k), (1) HCon and IncHCon still work well when $|G|$ is 22.2 times of the memory. (2) Both scale well when G grows. When $|\Delta G| = 1\%|G|$, IncHCon takes 96s on G with 100M nodes and 1.2B edges. In practice, $|\Delta G| \leq 0.1\%|G|$ for large G; in this case, IncHCon takes at most 26.1s.

Scalability on $M/|G|$. We evaluated the scalability of the contraction scheme with $M/|G|$ using Twitter. We varied $M/|G|$ from 0.2 to 1 (when $M/|G| = 1$, Twitter could fit in the memory without contraction). Here LCCA is a full-memory single-machine implementation [10, 16]. As shown in Fig. 5(l), (1) LCCA_h is still faster than COST, GraphChi and Mosaic when $M/|G|$ is 0.2. (2) When $M/|G|$ is 0.4, LCCA_h is faster than the full-memory computation of LCC. The results for the other five query classes and on the other graphs are consistent (not shown). These verify that our scheme can efficiently answer queries on big graphs with limited memory.

On-chip parallelism. On average, on a single machine with 16 cores, LCCA_h, PRA_h, CCA_h, CDA_h, SubA_h and RPQA_h are on average 5.9 times faster than on a machine with one core.

Summary. From the experiments we find the followings.

(1) The scheme enables a single machine to efficiently query big graphs G, even when memory is as small as 7.6% of $|G|$.

(2) On average, the scheme is (a) 7.3, 5.2, 14.4, 26.4, 12.9 and 15.7 times faster than COST for LCC, PR, SubIso, CC, CD and RPQ, (b) 1083.5, 5.3, 1716.8 and 17.3 times faster than GraphChi for LCC, PR, CC and RPQ (for SubIso and CD, GraphChi either ran out of memory or could not finish within 2 hours); and (c) 24.3, 4.8, 217.0 and 1.8 times faster than Mosaic for LCC, SubIso, CC and RPQ (Mosaic ran out of memory for CD), respectively. For the 6 algorithms, it reduces the I/O cost of these three systems by 40.5%.

(3) For some algorithms it outperforms parallel systems that use multiple machines. It is faster than (a) GRAPE for LCC, SubIso, CC and RPQ when GRAPE used 2, 1, 5 and 1 machines, (b) Gemini for LCC, SubIso and CC with 6, 3 and 6 machines, (c) LA3 for LCC, SubIso, CC and RPQ with 6, 6, 6 and 1 machines, and (d) it is 74.3, 1.3, 2.4 and 404.7 times faster than PowerGraph that used 6 machines for LCC, PR, SubIso and CC, respectively. PowerGraph, Gemini and LA3 ran out of memory on CD even with 6 machines; and for PR and CD, GRAPE requires at least 3 machines on Friendster.

(4) The total space cost of our scheme is comparable to that of disk-based COST, GraphChi and Mosaic; it is on average 3.4 times less than that of PowerGraph, GRAPE, Gemini and LA3.

(5) IncHCon is faster than HCon when $|\Delta G|$ is up to 25%$|G|$, and is 30.2 times faster when $|\Delta G|$=1%$|G|$. Moreover, HCon and IncHCon scale well with large graphs. On graphs with 6.5 billion nodes and edges, IncHCon takes at most 26.1s when $|\Delta G| \leq 0.1\%|G|$, while in practice, updates to large graphs rarely exceed 0.1%$|G|$.

7 CONCLUSION

We have proposed a hierarchical contraction scheme for a single machine to support *multiple applications* on graphs that do not fit in memory. We have shown how to adapt existing single-machine algorithms to the same scheme and compute *exact answers* for representative query classes, without decontracting supernodes of regular structures. We have also provided an incremental algorithm to maintain the hierarchy, and shown its boundedness. Our experimental study has verified that the scheme is effective.

One future work is to extend the scheme to train GCN [44] with a single machine. To reduce heavy sampling cost of GCN models, *e.g.,* GraphSAGE [38], one could develop an online contraction scheme. It iteratively contracts nodes that bear stable and similar embeddings into supernodes. The supernodes carry a synopsis to abstract key features of the contracted parts, to be reused in later sampling iterations. This reduces the memory cost and further speeds up the process by making big graphs smaller.

Acknowledgements. Fan, Li and Liu are supported in part by ERC 652976 and Royal Society Wolfson Research Merit Award WRM/R1/180014. Liu is also supported in part by EPSRC EP/L01503X/1. Lu is supported in part by NSFC 62002236.

REFERENCES

[1] 2006. Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*
[2] 2006. UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/.*
[3] 2012. Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*
[4] 2020. GRAPE. *https://github.com/alibaba/libgrape-lite.git.*
[5] 2020. GraphScope. *https://graphscope.io/.*
[6] Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Tamer Elsayed. 2018. LA3: A scalable link-and locality-aware linear algebra-based graph analytics system. *PVLDB* 11, 8 (2018), 920–933.
[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*. 1421–1432.
[8] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. 2004. Objectrank: Authority-based keyword search in databases. In *VLDB*, Vol. 4. 564–575.
[9] Pablo Barceló Baeza. 2013. Querying graph databases. In *PODS*. 175–188.
[10] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav Marathe, and Dorothea Wagner. 2008. Engineering label-constrained shortest-path algorithms. In *AAIM*. Springer, 27–37.
[11] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.
[12] Pavel Berkhin. 2005. A survey on PageRank computing. *Internet mathematics* 2, 1 (2005), 73–120.
[13] Nina Berry, Teresa Ko, Tim Moy, Julienne Smrcka, Jessica Turnley, and Ben Wu. 2004. Emergent clique formation in terrorist recruitment. In *AAAI Workshop on Agent Organizations: Theory and Practice*.
[14] Maciej Besta and Torsten Hoefler. 2018. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *CoRR* abs/1806.01799 (2018).
[15] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression techniques. In *WWW*. 595–602.
[16] Béla Bollobás. 2013. *Modern graph theory*. Vol. 184. Springer Science & Business Media.
[17] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.
[18] Yang Cao and Wenfei Fan. 2016. An Effective Syntax for Bounded Relational Queries. In *SIGMOD*.
[19] Yang Cao, Wenfei Fan, and Ruizhe Huang. 2015. Making Pattern Queries Bounded in Big Graphs. In *ICDE*.
[20] Yang Cao, Wenfei Fan, Yanghao Wang, and Ke Yi. 2020. Querying Shared Data with Security Heterogeneity. In *SIGMOD*. 575–585.
[21] Yang Cao, Wenfei Fan, Yanghao Wang, Tengfei Yuan, Yanchao Li, and Laura Yu Chen. 2017. BEAS: Bounded Evaluation of SQL Queries. In *SIGMOD*.
[22] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In *ICDE*. IEEE, 409–420.
[23] Sara Cohen. 2016. Data management for social networking. In *PODS*. 165–177.
[24] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI* 26, 10 (2004), 1367–1372.
[25] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[26] Wenfei Fan, Floris Geerts, Yang Cao, and Ting Deng. 2015. Querying Big Data by Accessing Small Data. In *PODS*.
[27] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *SIGMOD*.
[28] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *SIGMOD*. 157–168.
[29] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. 2021. Making Graphs Compact by Lossless Contraction. (2021). SIGMOD.
[30] Wenfei Fan, Xin Wang, and Yinghui Wu. 2014. Distributed graph simulation: Impossibility and possibility. *PVLDB* 7, 12 (2014), 1083–1094.
[31] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association rules with graph patterns. *PVLDB* 8, 12 (2015), 1502–1513.
[32] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *SIGMOD*.
[33] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *TODS* 43, 4 (2018), 18:1–18:39.
[34] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*. 1433–1445.
[35] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.
[36] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. 17–30.
[37] Claudio Gutierrez, Carlos A Hurtado, Alberto O Mendelzon, and Jorge Pérez. 2011. Foundations of semantic web databases. *J. Comput. System Sci.* 77, 3 (2011), 520–541.
[38] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. (2017).
[39] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{iso}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*.
[40] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. 2009. Fast connected-component labeling. *Pattern recognition* 42, 9 (2009), 1977–1987.
[41] Martin Szummer Tommi Jaakkola and Martin Szummer. 2002. Partially labeled classification with Markov random walks. *NIPS* 14 (2002), 945–952.
[42] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*. 595–608.
[43] U Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. 2010. Patterns on the connected components of terabyte-scale graphs. In *ICDM*. 875–880.
[44] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *ICLR* (2016).
[45] Ina Koch. 2001. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science* 250, 1-2 (2001), 1–30.
[46] Walter Kropatsch. 1996. Building irregular pyramids by dual-graph contraction. In *Vision Image and Signal Processing*.
[47] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*. 31–46.
[48] Theodoros Lappas, Kun Liu, and Evimaria Terzi. 2009. Finding a team of experts in social networks. In *KDD*.
[49] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph structure summarization. In *SDM*. SIAM, 454–465.
[50] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.
[51] Ulf Leser. 2005. A query language for biological networks. *Bioinformatics* 21, suppl_2 (2005), ii33–ii39.
[52] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *SIGKDD*.
[53] Kingsly Leung and Christopher Leckie. 2005. Unsupervised anomaly detection in network intrusion detection using clusters. In *ACSW*.
[54] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *ACM Comput. Surv.* 51, 3 (2018), 62:1–62:34.
[55] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB* 5, 8 (2012).
[56] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*. 527–543.
[57] Antonio Maccioni and Daniel J Abadi. 2016. Scalable pattern matching over compressed graphs via dedensification. In *SIGKDD*. 1755–1764.
[58] Wim Martens and Tina Trautner. 2018. Evaluation and enumeration problems for regular path queries. In *ICDT*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[59] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NIPS*.
[60] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.
[61] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science* 1 (2015).
[62] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.
[63] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in social media. *Data Mining and Knowledge Discovery* 24 (2012).
[64] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *TODS* 34, 3 (2009), 16:1–16:45.
[65] Ganesan Ramalingam and Thomas Reps. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21, 2 (1996), 267–305.
[66] Ganesan Ramalingam and Thomas Reps. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science* 158, 1-2 (1996), 233–277.
[67] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11-12 (1998), 701–726.
[68] Royi Ronen and Oded Shmueli. 2009. SoQL: A language for querying and creating data in social networks. In *ICDE*. IEEE, 1595–1602.
[69] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2017.

PuLP/XtraPuLP: Partitioning Tools for Extreme-Scale Graphs. Technical Report. Sandia National Lab (SNL-NM), Albuquerque, NM, US.
[70] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *WSDM*. 540–546.
[71] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.
[72] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *PVLDB* 7, 3 (2013), 193–204.
[73] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. 2008. Efficient aggregation for graph summarization. In *SIGMOD*. 567–580.
[74] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *SIGMOD*. 345–358.
[75] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A property graph query language. In *GRADES*.
[76] W3C Recommendation. 2008. SPARQL Query Language for RDF. *https://www.w3.org/TR/rdf-sparql-query/*.
[77] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*. 1307–1317.
[78] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17, 11 (1971), 712–716.
[79] Quan Yuan, Gao Cong, and Aixin Sun. 2014. Graph-based point-of-interest recommendation with geographical and temporal influences. In *CIKM*. 659–668.
[80] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI*. 301–316.