LSGraph: A Locatility-centric High-Performance Streaming Graph Engine

Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, Jin Zhao

Presented By: Sanjana Mupparaju

Streaming Graph Systems

- Dynamic graph data structure to support graph analytics applications
  - social networks, machine learning, bioinformatics
  - algorithms such as PageRank, BFS, connected components
- Graph updates arrive in batches to update state. Then analytics performed.
- Components of Design
  - Efficient support for graph algorithms (graph analytics)
  - Efficient support for graph updates

Streaming Graph Systems

- Dynamic graph data structure to support graph analytics applications
  - social networks, machine learning, bioinformatics
  - algorithms such as PageRank, BFS, connected components
- Components of Design
  - Efficient support for graph algorithms (graph analytics)
    - Quick access to neighbors of a vertex
    - Ordered vertex neighbors. Why?
  - Efficient support for graph updates

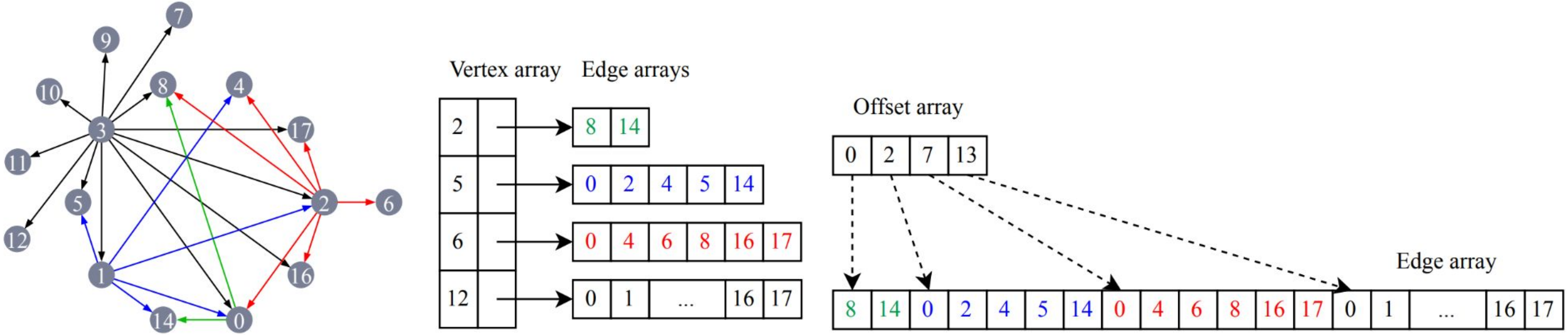
Streaming Graph Systems

- Dynamic graph data structure to support graph analytics applications
  - social networks, machine learning, bioinformatics
  - algorithms such as PageRank, BFS, connected components
- Components of Design
  - Efficient support for graph algorithms (graph analytics)
    - Necessitates quick access to neighbors of a vertex
    - Ordered vertex neighbors. Why?
      - Many graph algorithms rely on ordered neighbors for low computational complexity
        - Efficient set operations (intersection checks, perform merges, etc).
        - PageRank is an example
  - Efficient support for graph updates

Streaming Graph Systems

- Dynamic graph data structure to support graph analytics applications
  - social networks, machine learning, bioinformatics
  - algorithms such as PageRank, BFS, connected components
- Components of Design
  - Efficient support for graph algorithms (graph analytics)
    - Necessitates quick access to neighbors of a vertex
    - Ordered vertex neighbors. Why?
      - Many graph algorithms rely on ordered neighbors for low computational complexity
        - Exploits cache locality for fewer memory lookups when sequentially scanning neighbors
        - Efficient set operations (intersection checks, perform merges, etc).
  - Efficient support for graph updates
- **THEREFORE:** updates must *efficiently* maintain order by (1) searching to find position within representation of graph (2) minimize overhead data movement when preserving ordering

Static Graph Representations



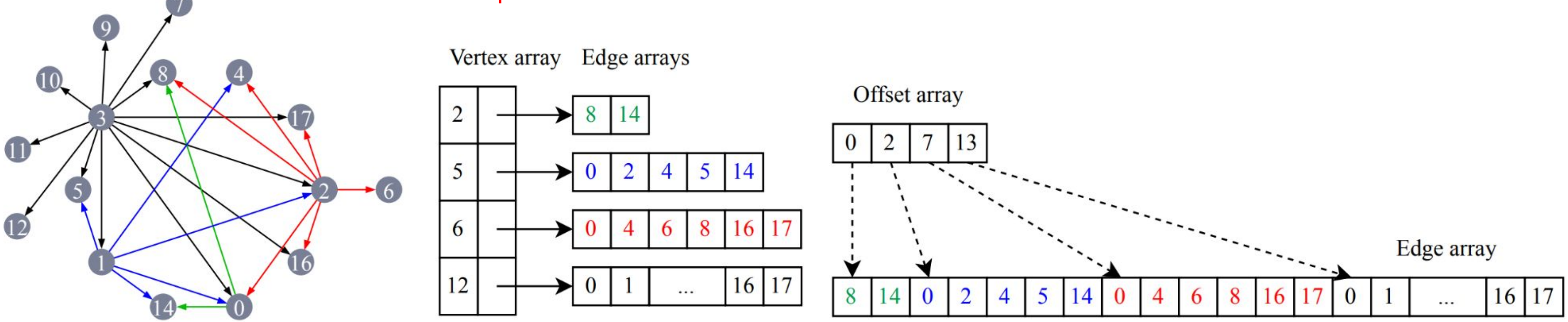
(a) An example of a directed graph

(b) Adjacency list format

(c) CSR format

Static Graph Representations

update efficient



(a) An example of a directed graph

(b) Adjacency list format

(c) CSR format

space efficient

Prior Work

- Authors mention Aspen + Terrace: Terrace performs better with graph analytics algorithms, but does not handle large insertions particularly well.

Prior Work

- Authors mention Aspen + Terrace: Terrace performs better with graph analytics algorithms, but does not handle large insertions particularly well.
- Terrace: Uses multiple structures to store edges
    1. High-degree vertices stored in B-tree
    2. Low degree vertices stored in *Packed Memory Array* (PMA) + Vertex Blocks

Prior Work

- Authors mention Aspen + Terrace: Terrace performs better with graph analytics algorithms, but does not handle large insertions particularly well.
- Terrace: Uses multiple structures to store edges
    1. High-degree vertices stored in B-tree -> fast updates
    2. Low degree vertices stores in *Packed Memory Array* (PMA) + Vertex Blocks. -> enable locality

Prior Work

- Authors mention Aspen + Terrace: Terrace performs better with graph analytics algorithms, but does not handle large insertions particularly well.
- Terrace: Uses multiple structures to store edges
    1. High-degree vertices stored in tree -> fast updates
    2. Low degree vertices stores in *Packed Memory Array* (PMA). -> enable locality
- Authors find that the PMA accounts for 97% of total update time.

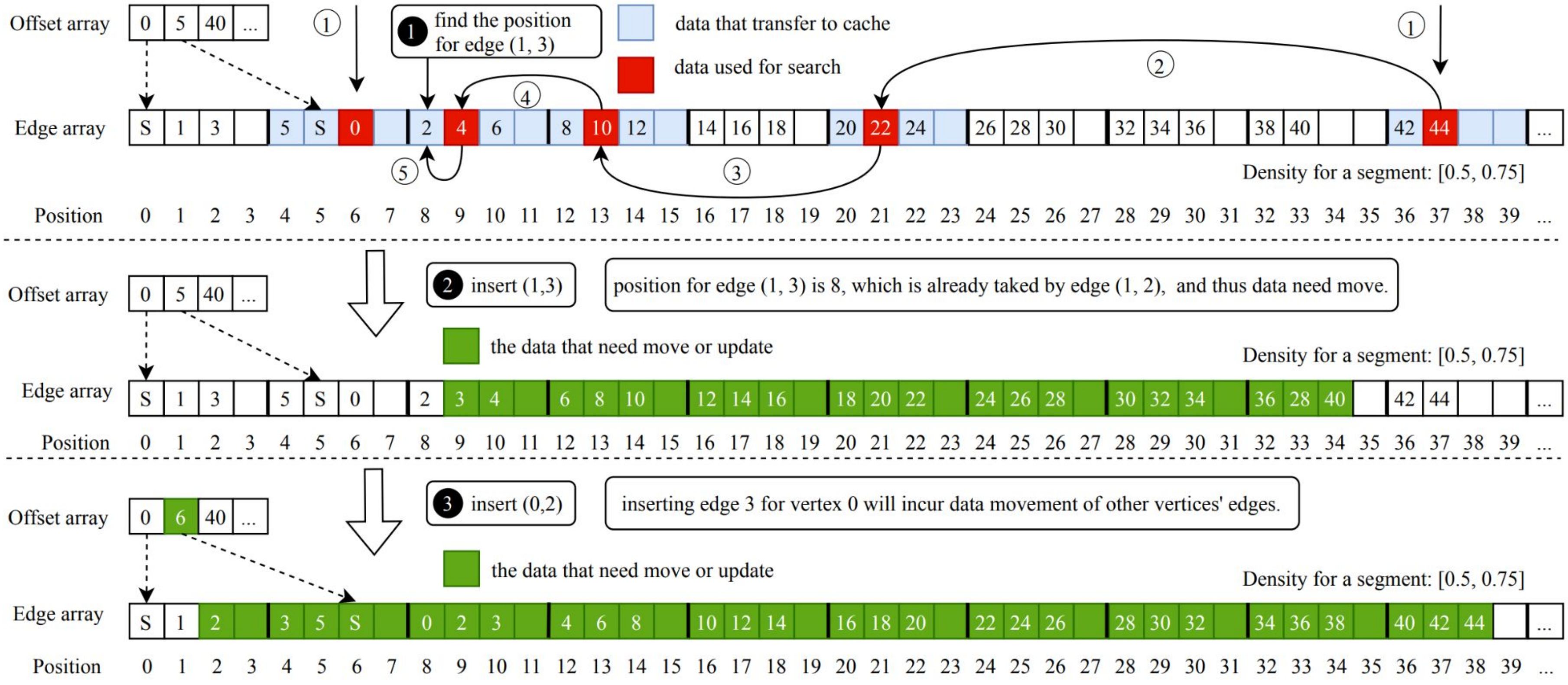


Figure 2. An example to illustrate inserting data into PMA. The "*S*" in the edge array is a sentinel entry.

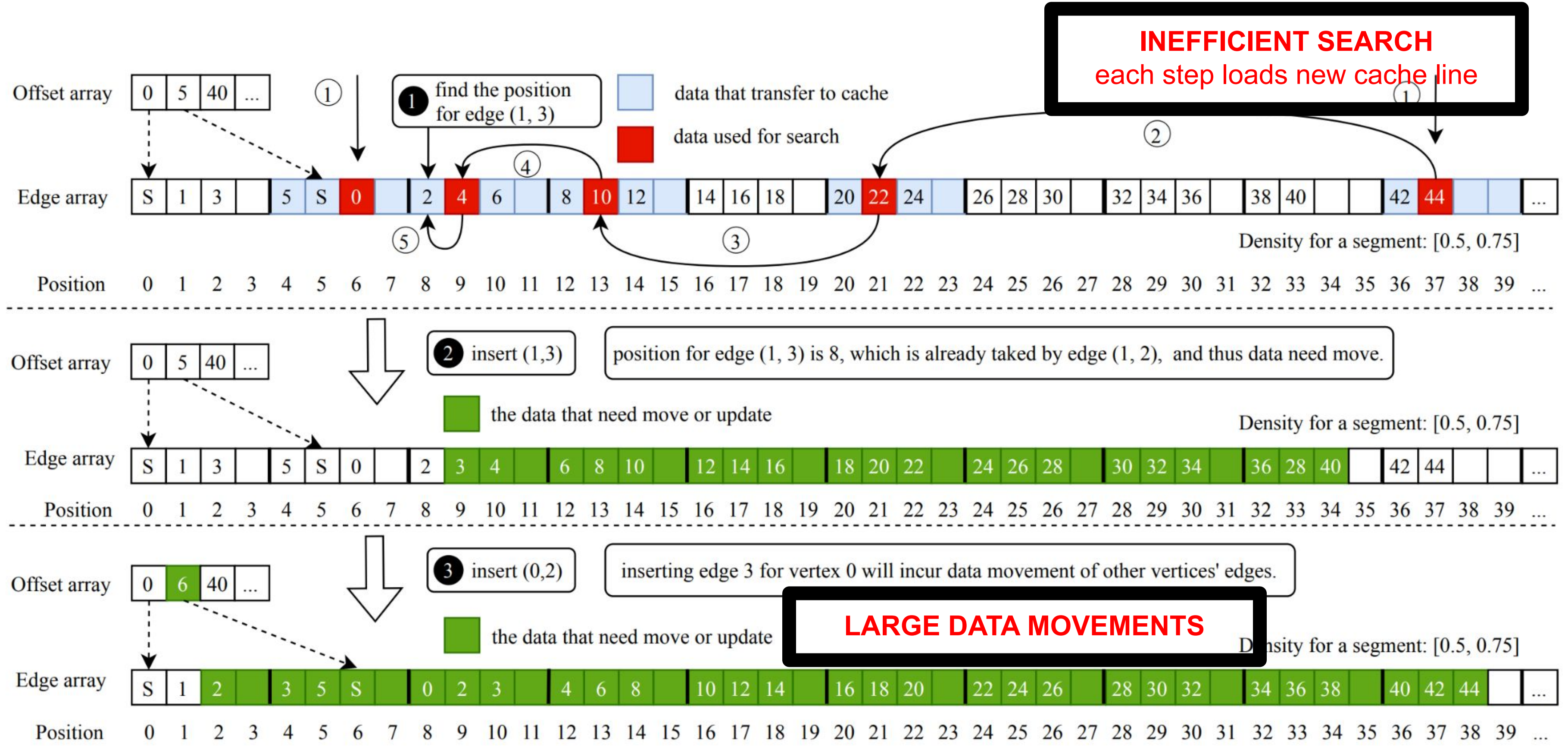


Figure 2. An example to illustrate inserting data into PMA. The "*S*" in the edge array is a sentinel entry.

01 **Redundant Indexed Array**

for low degree vertices

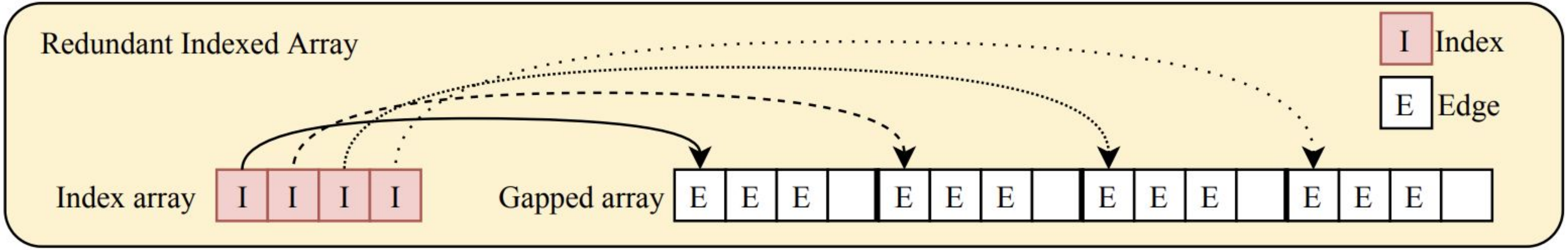
02 **Learned Indexed Array**

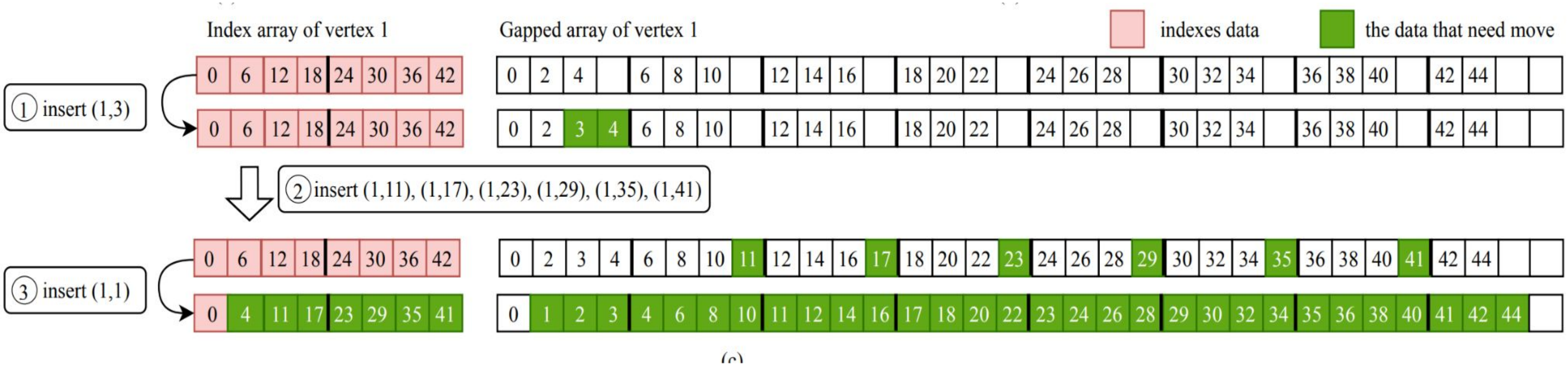
for high degree vertices

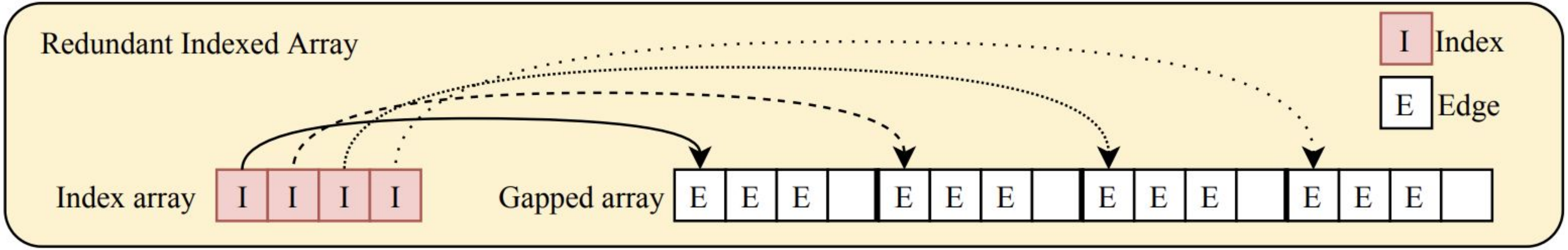
03 **Hybrid Indexed Tree (HITree)**

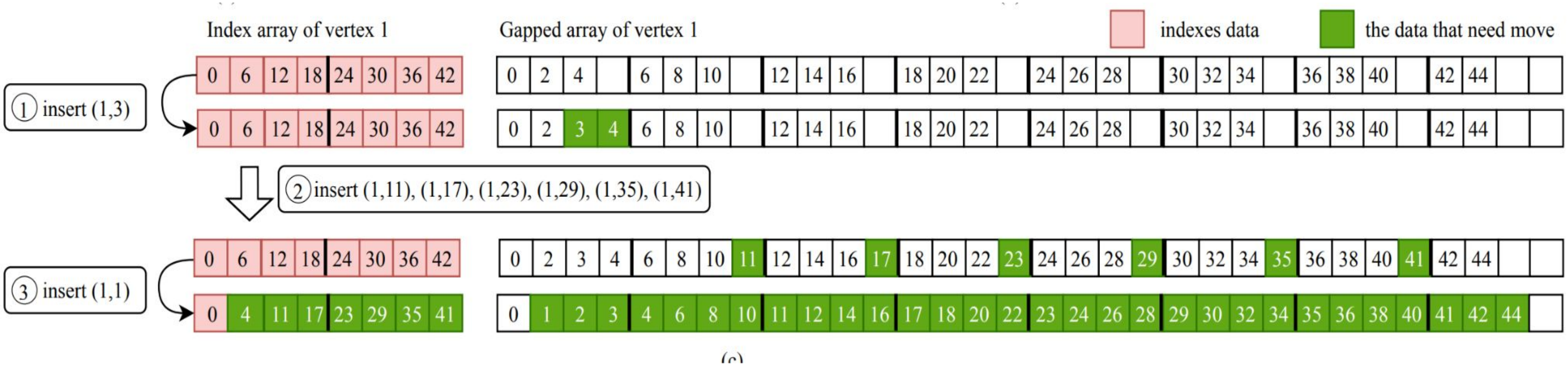
for vertical data movement

04 **LS-GRAPH**

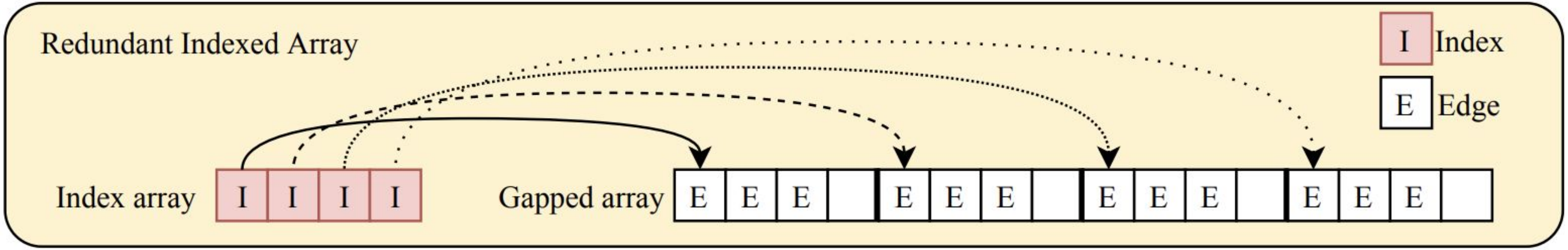

Redundant Indexed Array
I
Index
E
Edge
Index array
I I I I
Gapped array
E E E E E E E E E E E E

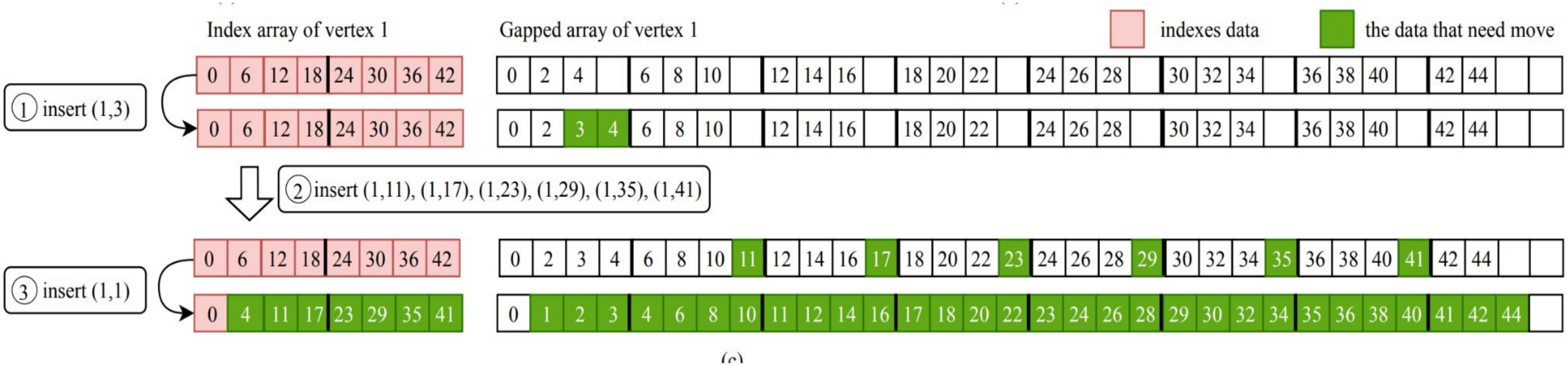

Index array of vertex 1
Gapped array of vertex 1
indexes data
the data that need move
1 insert (1,3)
0 6 12 18 24 30 36 42
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44
0 6 12 18 24 30 36 42
0 2 3 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44
2 insert (1,11), (1,17), (1,23), (1,29), (1,35), (1,41)
3 insert (1,1)
0 6 12 18 24 30 36 42
0 2 3 4 6 8 10 11 12 14 16 17 18 20 22 23 24 26 28 29 30 32 34 35 36 38 40 41 42 44
0 4 11 17 23 29 35 41
0 1 2 3 4 6 8 10 11 12 14 16 17 18 20 22 23 24 26 28 29 30 32 34 35 36 38 40 41 42 44





O(1) block level search after binary searching index array.





O(1) block level search after binary searching index array.

Still inefficient if vertex has high degree.

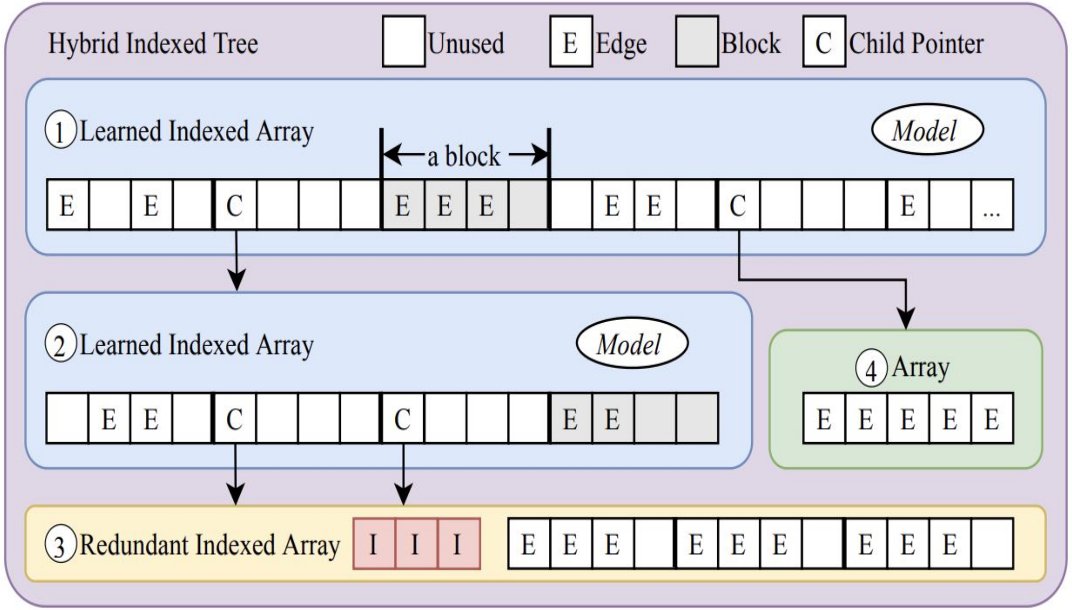


Figure 8. The design overview of HITree

KEY

Model takes key (edge dest) as input and outputs a position in the array

-> linear regression

Unused position can be used for inserts

Edge stores destination index of edge

Block position conflict but space in the block

Child too many edges mapped to block by the model, so points to child which can be

(1) an array
(2) another LIA
(3) RIA

Can be larger because we are using model instead of binary search.

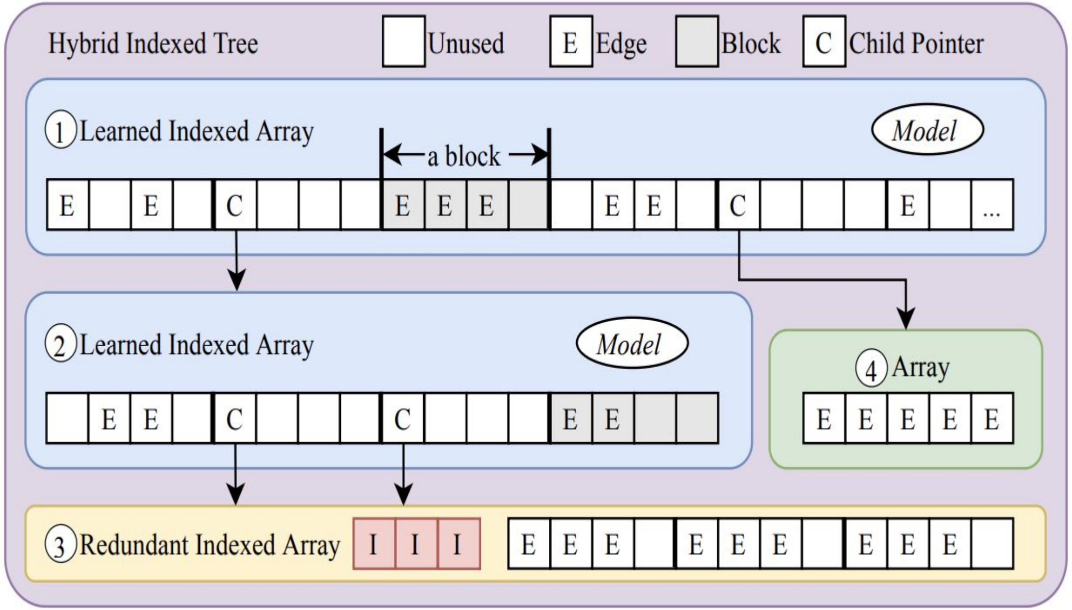


Figure 8. The design overview of HITree

KEY

Model takes key (edge dest) as input and outputs a position in the array

-> linear regression

Unused position can be used for inserts

Edge stores destination index of edge

Block position conflict but space in the block

Child too many edges mapped to block by the model, so points to child which can be

(1) an array

(2) another LIA

(3) RIA

Can be larger because we are using model instead of binary search.

Large horizontal data movement.

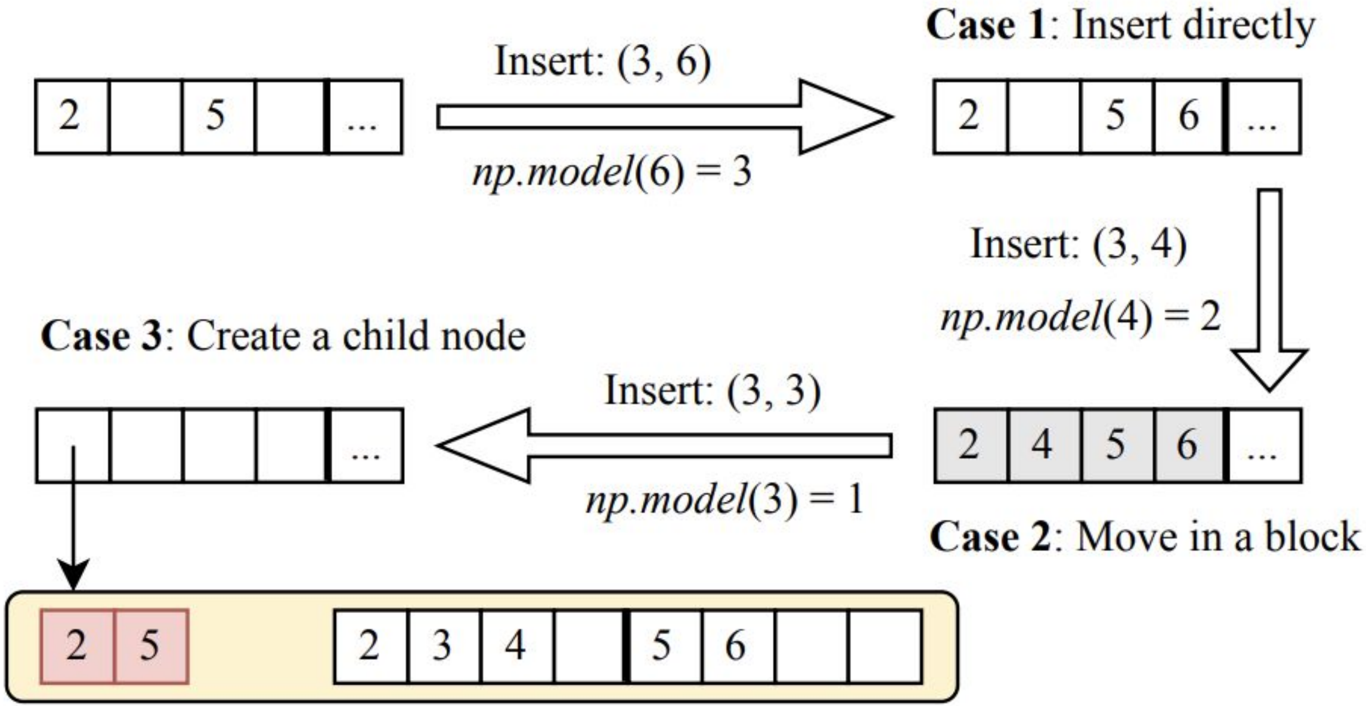


Figure 10. Insertions in the LIA

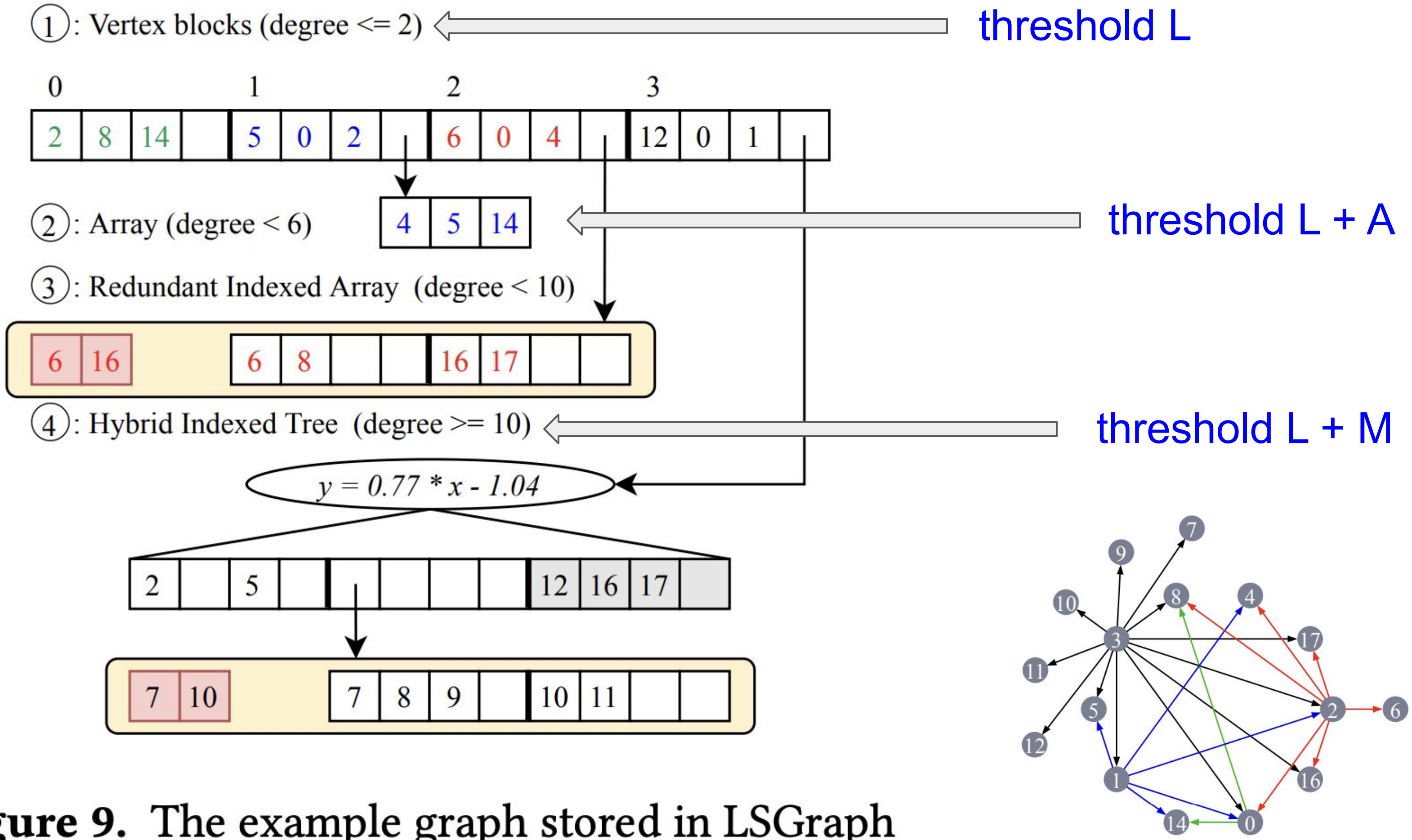


Figure 9. The example graph stored in LSGraph

(a) An example of a directed graph

Algorithm 1: The BulkLoad Algorithm for HITree

Input: *ns*: an array of ordered elements; α: the space amplification factor; *M*: the threshold using RIA or LIA; *BKS*: the size of each block.

Output: *np*: the current node pointer.

```
if ns.size <= M then
    np.gapped_array = malloc(ns.size * α);
    np.index_array = malloc(⌈ns.size * α / BKS⌉);
    DistributeData(np.gapped_array, ns);
    BuildIndex(np.index_array, np.gapped_array);
else
    np.array = malloc(ns.size * α);
    np.model = BuildModel(ns, np.array);
    poss = PredictedAllPositions(ns, np.model);
    foreach subns, subposs in a BKS of np.array ∈ ns, poss do
        ba = BlockAddress(subposs, np.array);
        if is_unique(subposs) then
            foreach u, pos ∈ subns, subposs do
                np.array[pos] = u;  SetType(pos, E);
        else if subns.size <= BKS then
            StoreBlock(np.array, ba, subns);
            SetTypes(ba, B);
        else if subns.size > BKS then
            child = BulkLoad(subns, α, M, BKS);
            np.array[ba] = child;  SetTypes(ba, C);
    MergeAdjacentChildren();
return np;
```

Algorithm 2: The Insert Algorithm for HITree

Input: *np*: the current node pointer; *u*: the insert element; α: the space amplification factor; *M*: the threshold using RIA or LIA; *BKS*: the size of each *Block*.

```
if np.size <= M then
    bid = SearchIndex(np.index_array, u);
    insert_ok = InsertBlock(np.gapped_array, bid, u, BKS);
    if insert_ok then
        UpdateIndex(np, bid, BKS);
    else
        move_ok, range = MoveNearBlocks(np, bid, BKS);
        if move_ok then
            UpdateIndexes(np, range, BKS);
        else
            ns = MergeData(np.gapped_array, u);
            np = BulkLoad(ns, α, M, BKS);
else
    pos = Predicted(np.model, u);
    type = GetType(pos);
    ba = BlockAddress(pos, np.array);
    if type == U then
        np.array[pos] = u;  SetType(pos,E);
    else if type == E or type == B then
        ns = MergeDataBlock(np.array, pos, u);
        if ns.size <= BKS then
            StoreBlock(np.array, ba, ns);  SetTypes(ba, B);
        else
            child = BulkLoad(ns, α, M, BKS);
            np.array[ba] = child;  SetTypes(ba, C);
    else if type == C then
        Insert(np.array[ba], u, α, M, BKS);
```

Algorithm 1: The BulkLoad Algorithm for HITree

```
Input: ns: an array of ordered elements; α: the space
       amplification factor; M: the threshold using RIA or
       LIA; BKS: the size of each block.
Output: np: the current node pointer.
if ns.size <= M then
    np.gapped_array = malloc(ns.size * α);
    np.index_array = malloc(⌈ns.size * α / BKS⌉);
    DistributeData(np.gapped_array, ns);
    BuildIndex(np.index_array, np.gapped_array);
else
    np.array = malloc(ns.size * α);
    np.model = BuildModel(ns, np.array);
    poss = PredictedAllPositions(ns, np.model);
    foreach subns, subposs in a BKS of np.array ∈ ns, poss
     do
        ba = BlockAddress(subposs, np.array);
        if is_unique(subposs) then
            foreach u, pos ∈ subns, subposs do
                np.array[pos] = u;  SetType(pos, E);
        else if subns.size <= BKS then
            StoreBlock(np.array, ba, subns);
            SetTypes(ba, B);
        else if subns.size > BKS then
            child = BulkLoad(subns, α, M, BKS);
            np.array[ba] = child;  SetTypes(ba, C);
    MergeAdjacentChildren();
return np;
```

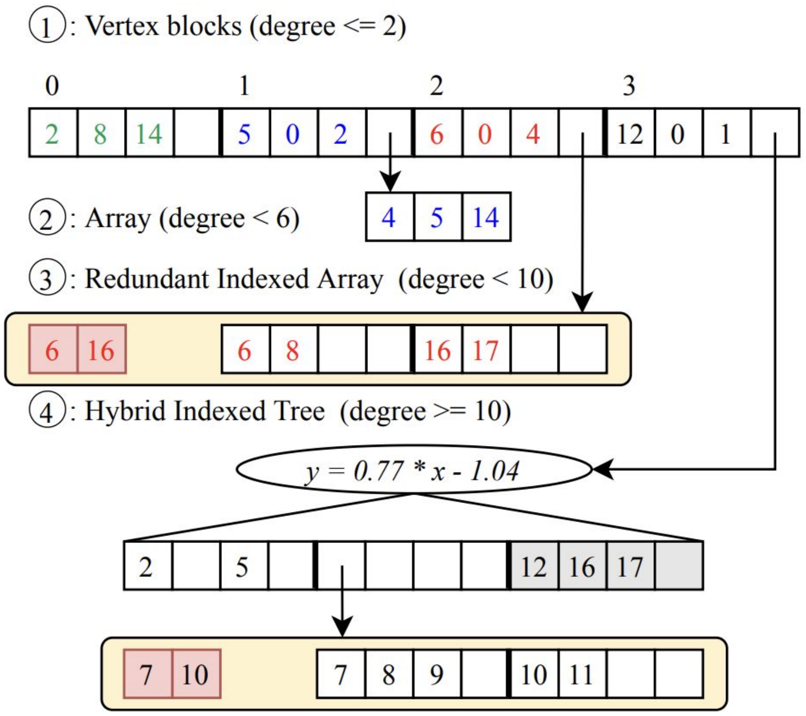


Figure 9. The example graph stored in LSGraph

Algorithm 2: The Insert Algorithm for HITree

```
Input: np: the current node pointer; u: the insert element;
       α: the space amplification factor; M: the threshold
       using RIA or LIA; BKS: the size of each Block.
if np.size <= M then
    bid = SearchIndex(np.index_array, u);
    insert_ok = InsertBlock(np.gapped_array, bid, u, BKS);
    if insert_ok then
        UpdateIndex(np, bid, BKS);
    else
        move_ok, range = MoveNearBlocks(np, bid, BKS);
        if move_ok then
            UpdateIndexes(np, range, BKS);
        else
            ns = MergeData(np.gapped_array, u);
            np = BulkLoad(ns, α, M, BKS);
else
    pos = Predicted(np.model, u);
    type = GetType(pos);
    ba = BlockAddress(pos, np.array);
    if type == U then
        np.array[pos] = u;  SetType(pos,E);
    else if type == E or type == B then
        ns = MergeDataBlock(np.array, pos, u);
        if ns.size <= BKS then
            StoreBlock(np.array, ba, ns);  SetTypes(ba, B);
        else
            child = BulkLoad(ns, α, M, BKS);
            np.array[ba] = child;  SetTypes(ba, C);
    else if type == C then
        Insert(np.array[ba], u, α, M, BKS);
```

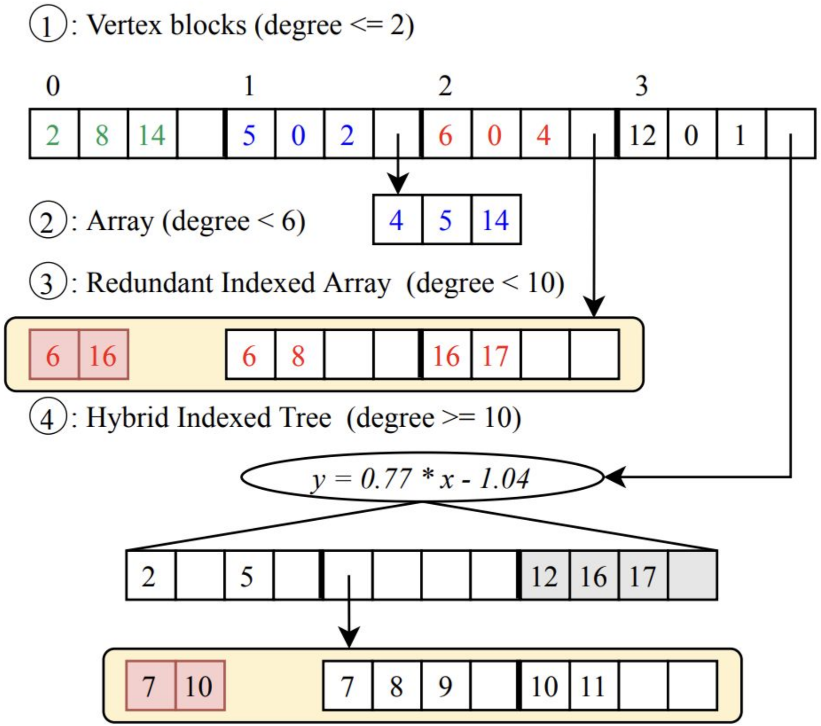


Figure 9. The example graph stored in LSGraph

EXPERIMENTAL SETUP

DATA

Table 1. A list of graph datasets with their number of vertices and edges, and their *average degree* (Avg.Deg) for evaluation

Graph	Vertices	Edges	Avg.Deg
LiveJournal (LJ)	4,847,571	85,702,474	17.7
Orkut (OR)	3,072,627	234,370,166	76.2
rMat (RM)	8,388,608	1,098,754,156	130.9
Twitter (TW)	61,578,415	2,405,026,092	39.1
Friendster (FR)	124,836,180	3,612,134,270	28.9

BASELINES

Terrace
PaC-Tree
Aspen

METRICS

Throughput of Graph Updates
Performance of Algorithms
Memory Footprints

GRAPH ALGORITHMS

BFS
Single Source Betweenness Centrality
Pagerank
Connected Components

THROUGHPUT

LSGraph: A Locality-centric High-performance Streaming Graph Engine — EuroSys ’24, April 22–25, 2024, Athens, Greece

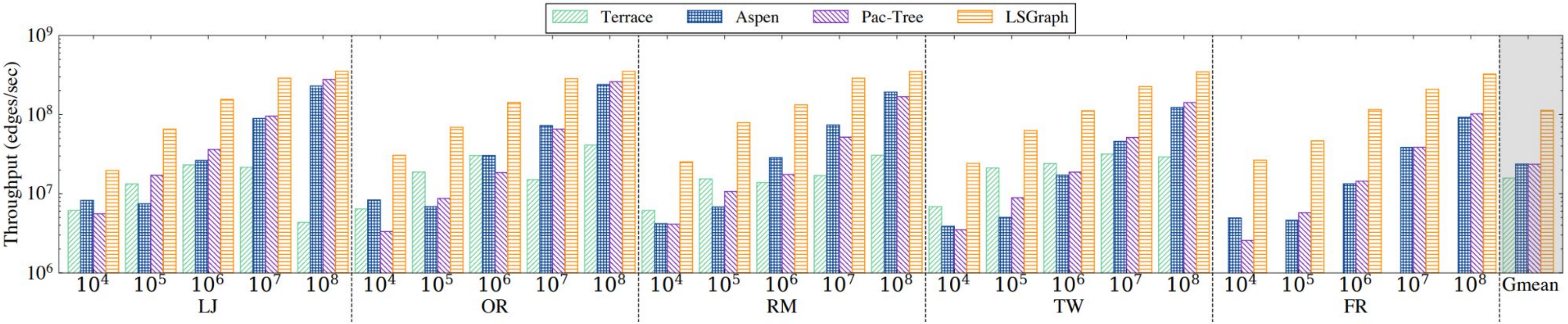


Figure 12. Throughput (edges/second) for insertion with varying batch sizes on all graphs in Terrace, Aspen, Pac-Tree, and LSGraph. Throughputs of the FR graph for Terrace are omitted because of time constraints.

METHODOLOGY:

- batch of edge updates
- deleted right after
- updated edges generated by rMAT
- 5 trials

PERFORMANCE AND MEMORY

Table 3. Memory usage (GB) of graphs in Table 1 on the different systems (T, A, P denoted as Terrace, Aspen, PaC-tree), and the ratio of index overhead (denoted as I/L) to LSGraph. T/L is the ratio of Terrace's memory usage to LSGraph.

Graph	LSGraph	T	A	P	T/L	I/L
LJ	0.61	1.51	0.58	0.35	2.48	2.90%
OR	1.27	2.51	0.89	0.73	1.98	4.96%
RM	5.67	18.02	3.99	3.47	3.18	5.43%
TW	16.58	44.70	12.36	8.92	2.70	3.16%
FR	23.66	51.72	22.76	14.99	2.19	4.06%

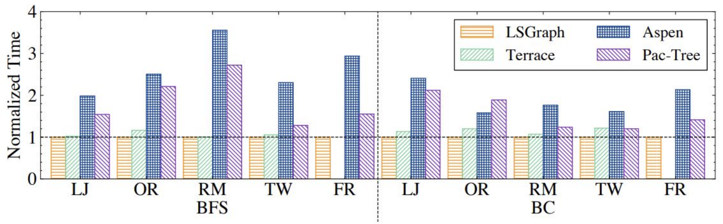


Figure 13. Time to run BFS and BC normalized to LSGraph in LSGraph , Terrace, Aspen, and Pac-Tree

Table 2. Execution times (in seconds) of LSGraph and Terrace on PR, CC, and TC. T/L denotes the speedup of LSGraph with respect to Terrace, Traversal and Tra/L denote the traversal time of LSGraph and the time ratio of traversal to LSGraph.

Graph	PR			CC			TC				
	LSGraph	Terrace	T/L	LSGraph	Terrace	T/L	LSGraph	Traversal	Terrace	T/L	Tra/L
LJ	0.184	0.239	1.30	0.053	0.055	1.04	1.335	0.148	2.031	1.52	10.99%
OR	0.352	0.593	1.69	0.099	0.152	1.53	3.535	0.689	5.116	1.45	19.48%
RM	1.782	2.205	1.24	0.309	0.348	1.13	13.151	2.351	22.130	1.68	17.88%
TW	8.553	11.902	1.39	2.187	2.677	1.22	792.797	5.044	3394.430	4.28	0.64%

SENSITIVITY ANALYSIS

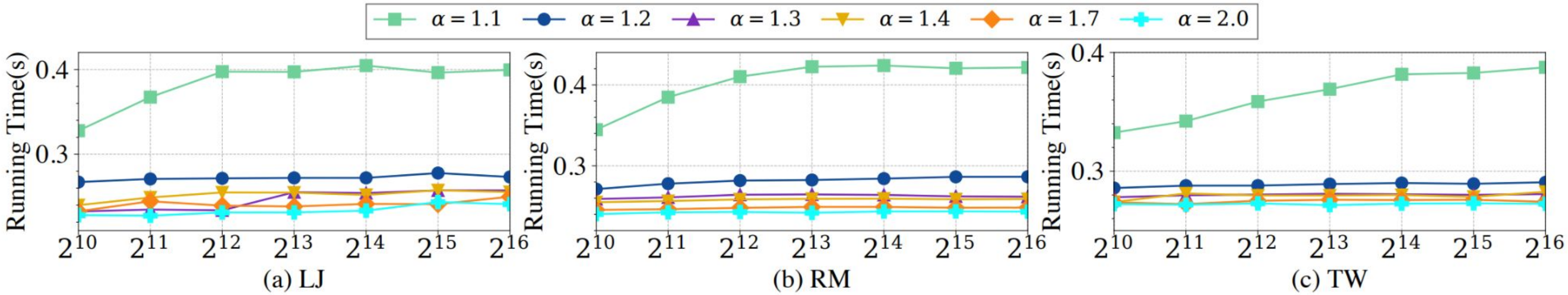


Figure 14. Running times (in seconds) of inserting 10^8 edges in LSGraph on LJ, RM, TW with different M and α

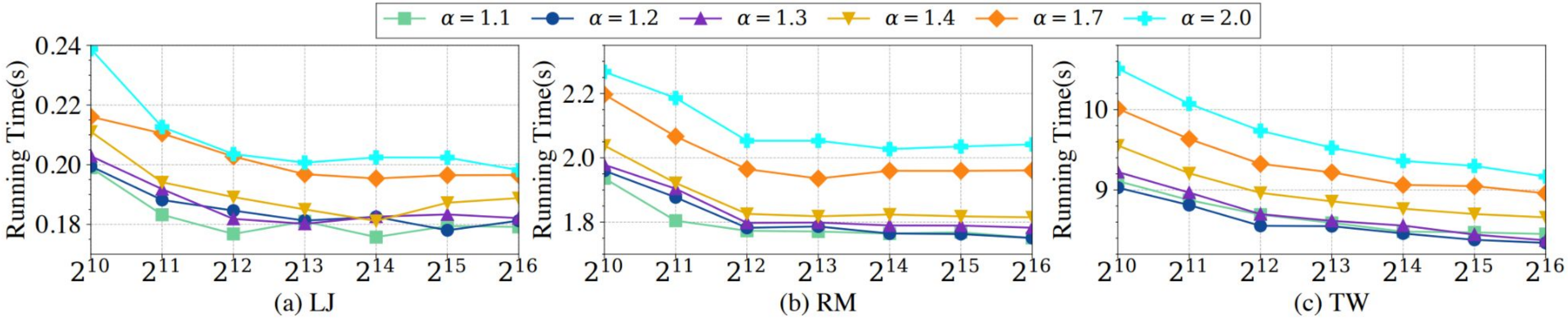


Figure 15. Running times (in seconds) of PR algorithm in LSGraph on LJ, RM, TW with different M and α

Strengths

- Good results -> were all improvements on previous results
- Strong experimental section
- Design choices were well explained.
- Thorough background explanation and exploration of bottlenecks.

Weaknesses

- Minor but did not include details of ablation study.
- Data structure is very complicated and seems hard to implement -> implementation details were not clear.
- Model learning details were not clear.
- No formal analysis on worst-case behavior or bounds (memory and runtime).

DISCUSSION QUESTIONS

(1) How sensitive if LSGraph's performance to the accuracy of the learned index?
(2) What are the theoretical worst case bounds for vertical and horizontal movements in the HITree?
(3) Does there exist an adversarial input in which LSGraph behaves worse than PMA or a B-tree?
(4) Could LSGraph be extended for dynamic weighted graphs?
(5) What are areas good for parallelizing in the implementation of LSGraph?