Efficient and Effective Path Compression in Large Graphs

Yilun Huang[†], Dong Wen[‡], Longbin Lai[§], Zhengping Qian[§], Lu Qin[†] and Ying Zhang[†]

[†]*The Australian Artificial Intelligence Institute, University of Technology Sydney, Australia*
[‡]*The University of New South Wales, Australia*
[§] *Alibaba Group, China*
[†]{ying.zhang, lu.qin}@uts.edu.au; yilun.huang@student.uts.edu.au;
[‡]dong.wen@unsw.edu.au;
[§]{longbin.lailb, zhengping.qzp}@alibaba-inc.com

***Abstract*—A path in a graph is a walk from one vertex to the other via edges. Many tasks for graph analytics may produce numerous paths, which record critical intermediate information or results. On the platform of Alibaba Cloud, a transaction (e.g., user purchase and money transfer) usually involves network communication via multiple servers. The server communication history is recorded as a path, where each vertex is an IP address. It is of significance to record such paths in Alibaba Cloud for daily maintenance tasks, such as anomaly server detection and network routing optimization. Motivated by the considerable data scale of IP paths, this paper proposes a compression method Overlap-Free Frequent Subpath (OFFS) to reduce the overall size. Meanwhile, the compressed paths should allow retrievals of any individual path, which is required by applications in our scenarios. We build a lookup table to match a series of frequent common subpaths to supernodes. Each path is shortened by replacing subpaths with corresponding supernodes in the table. We adopt a bottom-up framework to construct the lookup table in given iterations. Several optimizations are proposed to improve the compression ratio and speed. We conduct extensive experiments to show our effectiveness and efficiency based on several real datasets from Alibaba Cloud.**

***Index Terms*—Compression, Frequent subpaths**

I. Introduction

The graph data model is widely used to capture complex relationships between entities in everyday applications, such as social analysis, e-commerce transactions, the word wide web, cybersecurity and protein interactions. A (simple) path in a graph is a sequence of vertices where each pair of adjacent vertices are connected by an edge in the graph. In many graph-based analysis tasks, paths are generated by various queries. It can be a shortest path on road networks, a planning route in public-transaction networks, a routing record in telephone networks, or a message transmission in social networks. It is inevitable to record a set of paths in these applications. Real-life graphs usually contain millions or billions of edges and vertices. Sophisticated analytics over such big graphs enables valuable insights into the underlying data but may produce an abundance of paths. Note that many overlaps among paths may make the total path size significantly greater than the graph size. It is crucial to study how to store paths effectively and reduce space usage. In this paper, given a set of paths as input, we aim to develop a strategy to compress them.

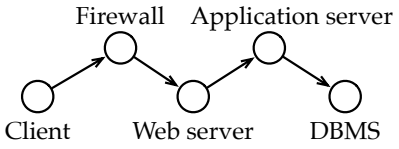


Fig. 1: A simplified transaction in a purchase action

Vast amounts of daily transactions and services take place on the platform of Alibaba Cloud. Figure 1 presents a simplified network communication in a purchase action in Alibaba e-commerce systems. Whenever a buyer (client) submits an order via a phone app or a browser, a network message will be sent to a series of middle-tier servers via the Internet. A typical middle tier contains a firewall, a web server to process web requests, and an application server to operate data. In the end, the carefully prepared messages arrive at a DBMS server. We leverage a graph model to facilitate daily analysis and monitoring. Each server/machine is represented by a vertex, while each communication is denoted by an edge. Therefore, a purchase action can be described by a path in the graph. Note that each step (e.g., web server) usually involves multiple machines deployed in the cloud. A web request is forwarded to a particular machine for processing based on various factors such as machine workloads, network status, and dispatching strategy. Maintaining a transaction path for a user action enables locating affected users when an anomaly is detected in a specific machine.

A more general case is illustrated in Figure 2. Many devices are deployed on the platform Alibaba Cloud to support a wide range of services. Network messages between machines are recorded as IP pairs in system logs. An IP address may match a host server, a portable device, a client machine, or a gateway. The topological relationship between IPs in a certain task is denoted as IP hops (path). Below are two common instances of utilizing IP paths in daily system maintenance.

Case 1. Identifying Affected Nodes. Once there is an

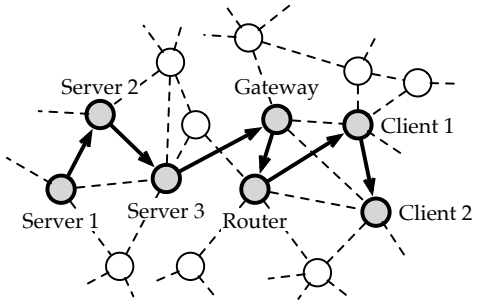


Fig. 2: An example in Alibaba Cloud Service

anomaly in a host server due to unexpected maintenance or crashes, it is of high priority to identify all affected segments for altering routing plans. One may consider performing a search from the issue server IP by scanning all neighbour IPs. The search space may be exponentially growing and contains many useless results. By retrieving all indexed IP paths containing the issue node, we can fetch all affected IP nodes accurately.

Case 2. Locate Anomalies. Many network issues are reported by customers every day. Given a user client IP and a terminal IP (e.g., an application server) related to the reported issue, we need to investigate all intermediate IP nodes of network transactions for troubleshooting. This can be done by collecting all IP paths with given terminals.

Challenges. Storing IP paths facilitates many services in Alibaba Cloud, while the number of paths can be numerous. For instance, a table to record IP hops usually accumulates more than 50 GB of data from nearly one million data transmissions in one day. And there are massive data to be collected by more tables every day. Motivated by this, we aim to develop a method to compress all paths. One may try to organize all paths as a whole and apply generic compression methods [1]–[3]. However, they are designed for general data, and their compression quality for path compression is poor compared with our tailored approach according to our experiments. In addition, generic compression methods do not support decompressing part of data, which is an essential need in aforementioned applications. We may divide paths into several fine-granularity blocks and compress them individually, but the compression performance would drop dramatically.

Our Approach. We propose a lightweight (de)compression method Overlap-Free Frequent Subpath (OFFS) for a set of paths in this paper. Our idea is to build a lookup table based on global information. The table contains a set of supernodes, and each supernode matches a frequent common subpath. Based on that, we contract subpaths into supernodes so that many paths are shortened. It is easy to see that the compression quality heavily depends on how to select supernodes. An intuitive idea is to compute a certain number of frequent long subpaths for supernode compression given that frequent pattern mining has been extensively studied in the literature [4]–[6]. However, many overlaps may exist between the identified frequent subpaths. There often come two overlapped frequent subpaths in the same path. Once replacing a subpath with a supernode in data, the other frequent subpath becomes useless for compression. We call this *match collision* issue, which produces results with bad compression performance. To address this challenge, we design a new algorithm to construct a supernode table. We adopt a bottom-up framework that runs in given iterations. Specifically, the algorithm starts by identifying frequent subpaths with only two vertices. We collect and judge each frequent subpath (supernode) carefully. The key is to eliminate the effectiveness of vertices that suffer match collisions with other supernodes. In each iteration, we generate new candidate frequent subpaths by two operations, namely merge and expansion. The merge operation concatenates two adjacent existing frequent subpaths appearing back to back in the same path, while the expansion operation adds the next vertex of each matched frequent subpath in data paths. We keep a certain number of frequent subpaths as candidates in each iteration. The algorithm terminates when the number of iterations reaches a predefined threshold.

We conduct extensive experiments based on real datasets from Alibaba Cloud. Compared with the straightforward methods, our solution achieves a high compression ratio in a shorter time. To further show our performance, we also report our results on several public real datasets.

Organization. The rest of the paper is organized as follows. Section II introduces background knowledge and related works. Section III introduces our computing framework and a straightforward solution. Section IV presents our algorithm for supernode table construction. Section V analyses the theoretical performance. Section VI presents the practical performance, and Section VII concludes the paper.

II. Background and related work

A. Problem Statement

Given a directed graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges, $e(u, v)$ denotes a directed edge from u to v. A path P is a sequence of vertices $\{v_1, ..., v_l\}$ such that $\forall 1 < i \leq l, e(v_{i-1}, v_i) \in E$. A path P is simple if all vertices in P are distinct. When the context is clear in the rest, we use the term "path" to represent "simple path" for short. We use $|P|$ to denote the number of vertices in P. Given two integer x and y with $x < y \leq |P|$, we use $P[x : y]$ to denote the subpath from x-th vertex to $(y-1)$-th vertex, and x is starting from 0. Likely, we use $P[x]$ to denote x-th vertex in P. For example, given a path $P = \{1, 2, 3, 5, 8, 13\}$, we have $P[1 : 4] = \{2, 3, 5\}$ and $P[4] = \{8\}$.

Lossless compression and decompression. In this paper, we study the problem of lossless compression of a set of simple paths with access to any individual path. Specifically, given a set of paths P, our aim is to find a compression scheme $f : P \Rightarrow (P', R)$, where P' is a set of contracted paths and R is

the compression rule. Meanwhile, there exists a decompression scheme $f^T : (P', R) \Rightarrow P$ that restores P from P' with R losslessly. Given any subset $Q' \subseteq P'$, $f^T : (Q', R) \Rightarrow Q$ returns its corresponding uncontracted paths $Q \subseteq P$ without decompressing the other compressed paths in $P' - Q'$.

B. Related Work

We review existing works that are closely related to this paper in this section.

Generic Compression. When it comes to compression scheme, a straightforward idea is generic compression, such as zlib [1], lz4 [3], zstd [2]. Most of them reduce the sizes of files using classic LZ77 [7], LZ78 [8] or their variants (e.g., LZW [9] and DEFLATE [10]). However, they are commonly used in the default block mode only considering duplication within the local block. The effective minimal size of a block, 1 KB, is much larger than a path. Given that only one path within the block is queried instead of the whole block, it is dramatically expensive and inefficient for data retrieval.

Lightweight Compression. In contrast to heavyweight algorithms [7], [11], [12], lightweight algorithms demand a less computational cost to achieve comparable or even better compression rates in finer granularity. In a recent survey [13], there are five basic categories of lightweight techniques: frame-of-reference (FOR) [14], [15], delta coding (DELTA) [16], [17], dictionary compression (DICT) [15], [18], run-length encoding (RLE) [17], [18] and null suppression (NS) [17], [18]. Generally, they focus on different data levels. FOR, DELTA, DICT, and RLE consider the logical data level, while NS addresses the physical level of bits or bytes. To give a brief introduction, FOR represents each value as the difference to a given certain value, DELTA represents each value as the difference to its predecessors, and DICT replaces values as symbols in the dictionary. It is easier for NS to omit unnecessary zeros after representing smaller values with the aforementioned three schemes. Finally, RLE deals with continuous sequences of occurrences of the same value, i.e., runs and each run is represented as its value and length.

Database Management System. Modern systems usually use data compression engines to deal with the exponentially growing data volume. There are several works [19]–[21] aiming at performing read-only operations directly on compressed data. They apply the idea of lazy decompression to avoid redundant compression and decompression operations. The latest CompressDB by Zhang et al. [22] has explored more than read-only query processing. It proposes an efficient technique to support write operations like updates, inserts and deletes directly on compressed data. However, the finest granularity of (de)compression of them is usually 1 KB to be compatible with existing systems. It is much larger than that of a path and not suitable in our case.

Trajectory Simplification. With the development of location-based services, it is becoming increasingly costly to collect, store and transmit trajectory data. There have been works on line simplification to reduce the trajectory size while preserving necessary information. They can be classified into two classifications, batch solutions, like the classic Douglas-Peucker algorithm [23], Bellman algorithm [24] and Top-Down Time Ratio [25], and online solutions like Open Window [26] and Dead Reckoning [27]. It is worth noting that all of them exploit temporospatial metrics such as Synchronous Euclidean Distances [28], directions, speeds etc. to compress trajectories, which are not available in our case. In addition, lossy compression is usually acceptable for trajectories but does not work for our application.

String Compression. String columns serve as necessary components in modern database systems and there are plenty of works on string compression. Classical string compression methods usually consider the special quality or operation of string, several works [29], [30] study the order-preserving compression of strings, [31] proposes an method specifically for compressing keys in B-trees and [32] explores string matching of LZ compressed string. More recent works focus on Single Instruction Multiple Data (SIMD), the CPU-level parallelism, to improve the efficiency of compression. BRPFC [33] adopts strong dictionary compression based on Re-Pair Front Coding, and FSST [34] adopts a lightweight method based on DICT aforementioned. They usually come with a hard-coded solution or require specific constraints of order-persevering in compressed data like lexicographical order, which is inappropriate in our case.

Graph Summarization. Graph summarization aims at reducing big graphs to smaller ones by contracting vertices or edges into more compacted supernodes (merged nodes) and superedges (edges between supernodes) [35]. Several works [36], [37] aim at answering queries on graphs with no correction set. Namely, a correction set is the extra information of edge correction needed for recreating graphs from supernodes and superedges. Recently, Fan et al. [38] propose a scheme to contract obsolete components, stars, cliques, and paths into supernodes and prioritize up-to-date data. It mainly focuses on specific graph queries and stores extra synopsis in the disk. Some works come with correction sets to restore the input graph. In another latest work, Yong et al. [39] propose an algorithm called LDME, which summarizes the input graph into a summary graph with correction sets at billion-scale. The idea of merging nodes into supernodes is quite useful in our case, which aims to reduce the space cost in memory.

C. Straightforward Methods by Generic Compression

We discuss straightforward methods by generic compression in this section.

As mentioned in the previous section, generic compression methods in the default block mode do not support partial decompression, i.e., they cannot retrieve part of the data while leaving the rest in the block compressed. A naive and

straightforward idea is to divide all paths into a set of blocks and compress each block individually. However, it has several shortcomings as follows, (1) it fails to detect the duplication of paths among blocks, (2) it costs much more space and time to decompress a whole block even if we only need one path, and (3) the absence of global information hinders the compression performance. In our observations, the compression quality drops dramatically as we allocate a block for each path and then compress it.

Instead, we turn to another mode of lz4, a popular generic compression method. Specifically, the stream mode of lz4 can process data block by block with a flexible size as small as that of a path. Besides, it learns from other blocks to improve the compression quality. The latest API of lz4 considers the assistance of a global dictionary, which is trained and formatted by zdict, a library in zstd, as the official documentation suggests. One may question how a compression framework for general files can handle paths composed of several integers (vertex IDs). We consider integers as a sequence of bits. For instance, in modern machines, a sequence of eight vertices is stored as 256 consecutive bits in physical level if each vertex is stored as an integer with 32 bits. We call this baseline method dictionary with lz4, Dlz4. Primarily it needs a dictionary trained from data by zdict. In the compression phase, it initializes the stream structure for encoding with the dictionary and then encodes a path interpreted as a byte array. In the decompression phase, it decodes a byte array with the assistance of the same dictionary and reinterprets it as an integer array. Since type reinterpretation is a built-in feature for most modern languages like C++, we can rest assured the time cost is dominated by encoding or decoding.

Although Dlz4 seems acceptable in our case, it still has the following drawbacks. (1) It does not guarantee independence between blocks in nature. Namely, it learns from prior blocks during encoding and requires extra information from them during decoding. Thus, it is essential to refresh the stream structure every time to avoid dependency issues, which affects the speed of (de)compression. (2) Interpreting paths as byte arrays and encoding them into byte buffers loses necessary information from raw data. It becomes a hurdle for future data mining, if we cannot tell whether an encoded buffer is a simple path. (3) The compression quality, in our observation, is still not good enough in our case. It sounds good to reduce the input size by half for quick compression methods like lz4, but it is not that satisfying for our application in view of the data scale of Alibaba Cloud. Therefore, it is significant to dig further into this problem.

III. The Framework

To improve the compression quality for our problem, we adopt the framework of dictionary compression DICT, which contracts each input path into a shorter path with the assistance of a lookup table. The paths in our problem come from the same graph, and common subpaths appear repeatedly in different paths. Our basic idea is to merge those common subpaths into supernodes. We record the corresponding subpath for each supernode in the lookup table.

A. Decompression and Compression

We first introduce the decompression scheme as shown in Algorithm 1. Given a supernode table $\mathcal{ST}$ and an arbitrary compressed path, we scan the compressed path. When meeting a supernode, we look up the supernode table and replace the supernode with the corresponding subpath. In line 4, $\mathcal{ST}[P_c[pos]]$ derives the subpath for the supernode $P_c[pos]$, and $\oplus$ is a concatenation operation. If $P_c[pos]$ is not a supernode, we keep the original vertex and continue to scan. To deal with an overwhelming number of path decompression queries, we can process paths in parallel to improve efficiency.

Lemma 1: Given a compressed path P_c, decompressing P_c takes $O(|P|)$ time, where P is the decompressed path.

Assume that we already have a supernode (lookup) table. The compression scheme is presented in Algorithm 2. We use $\mathcal{ST}^{-1}$ to represent the inverted lookup table, which returns a supernode given a subpath. In lines 3–9, we adopt a greedy strategy to match subpaths to supernodes. We use a parameter δ to denote the longest length among all compressed subpaths. Starting from the first vertex $pos = 0$, we find the longest subpath in P such that a matching supernode exists. If a matched subpath exists (line 5), we replace it with the corresponding supernode in line 6. Otherwise, we skip the vertex at the current position (line 8) and start matching from the next vertex in the next iteration.

Lemma 2: Given a path P, compressing P takes $O(|P| \cdot \delta^2)$ time, where δ is the longest subpath length in the lookup table.

Note that the square of δ comes from δ times possible hashes of up to δ elements, but never mind since δ is small in practice. In our evaluation and real deployment, δ is set as 8.

Based on that, we will give straightforward ideas on how to build $\mathcal{ST}$ and $\mathcal{ST}^{-1}$ in the following subsections.

Algorithm 1: Decompress$(P_c, \mathcal{ST})$

Input: A compressed path P_c and a supernode table $\mathcal{ST}$
Output: An orginal path P

1 $P \leftarrow \emptyset$;
2 $pos \leftarrow 0$;
3 **for** $0 \le pos < |P_c|$ **do**
4 **if** $P_c[pos] \in \mathcal{ST}$ **then** $P \leftarrow P \oplus \mathcal{ST}[P_c[pos]]$;
5 **else** $P \leftarrow P \oplus P_c[pos]$;
6 **return** P;

B. Frequent Subpaths

One naive solution for table construction comes from brute-force enumeration. The cost of testing the compression performance of each table is $O(|\mathbb{P}| \cdot \delta^2)$ as Lemma 2 shows. More

Algorithm 2: Compress($P, \mathcal{ST}^{-1}$)

Input: A path P and an inverted supernode table $\mathcal{ST}^{-1}$

Output: A compressed path P_c

```
1  P_c ← ∅;
2  pos ← 0;
3  while pos < |P| do
4      for min(δ, |P| − pos) ≥ l > 1 do
5          if P[pos : pos + l] ∈ ST^{-1} then
6              P_c ← P_c ⊕ ST^{-1}[P[pos : pos + l]];
7              break;
8      if l = 1 then P_c ← P_c ⊕ P[pos : pos + 1] ;
9      pos ← pos + l;
10 return P_c;
```

specifically, given the maximum supernode size δ, the number of candidates is bounded by $\delta \cdot |\mathbb{P}|$. The possible number of tables composed of c supernodes is $\binom{\delta \cdot |\mathbb{P}|}{c}$. It is bounded by $O(min((\delta \cdot |\mathbb{P}|)^c, (\delta \cdot |\mathbb{P}|)^{\delta \cdot |\mathbb{P}| - c}))$ and is beyond polynomial time. For instance, if we set δ as 8, and construct a table of the top 1,000 out from 12,500 paths, a toy case compared to real-life datasets, the estimated cost 100000^{1000} has too many digits to count and becomes unacceptable for applications.

Instead, we are exploring in another way via finding frequent patterns, which is one of the classic topics in data mining. There are several methods on that, such as Apriori [4] and FP-Tree [5]. As far as we know, the state-of-the-art solution for computing frequent subpaths is Apriori for Frequent Subpaths (AFS) [6]. AFS adopts a bottom-up framework as in Algorithm 3 to get the result sets L_i of length i through iterations. It starts from short frequent subpaths of length one in line 1. Next, it joins current frequent subpaths with outgoing edges of the last element in the subpath (suppose there is a graph as ground truth), and checks whether the extended subpath excluding the first element is in the last set L_{i-1} in line 4. After that, it counts their gains (i.e., the product of frequency and length) in data paths and keeps candidates with gains larger than a threshold k in line 5. Then it increases the count of iterations i and checks whether i has reached the target l, if so returns results and ends the process, otherwise goes to the next iteration.

However, (1) the time complexity of this method is unacceptable. Suppose that we gain $|L|$ subpaths of length i in i-th iteration and there are n nodes in the graph, then the cost to extend in each iteration is $O(i \cdot n \cdot |L|)$ due to checks in L_{i-1}. Assume the number of subpaths returned in line 12 is λ, it will cost $O(l^2 \cdot n \cdot \lambda)$ in total. Even worse, (2) unlike Apriori for sets, the generated longer subpaths in C_i are not guaranteed valid on data paths, as the constraint of order and adjacency of paths is stricter than that of sets. Therefore, it is essential to prune the useless subpaths by identifying them on datasets once more.

Algorithm 3: AFS($\mathbb{P}, l, k$)

Input: A set of paths $\mathbb{P}$, the maximum length l, and a threshold k

Output: A set of frequent subpaths L

```
// initialize the set with nodes
1  L_1 ← {v|v ∈ ℙ};
2  i ← 2;
3  while i ≤ l do
4      C_i ← JoinWithCheck(L_{i−1}, i − 1);
5      L_i ← CountGain(C_i, ℙ, k, i);
6      i ← i + 1;
7  return L ← L_1, ..., L_l;

1  Procedure JoinWithCheck(L, i)
2      C ← ∅;
3      foreach P = {v_0, v_1, ..., v_{i−1}} ∈ L do
4          foreach u as neighbor of v_{i−1} do
5              P' ← {v_0, v_1, ..., v_{i−1}, u};
6              if P'[1 : i + 1] ∈ L then
7                  C ← C ∪ {P'};
8  return C;

1  Procedure CountGain(C, ℙ, k, i)
2      L ← ∅;
3      foreach P ∈ ℙ do
4          j ← 0;
5          while j + i ≤ |P| do
6              if P[j : j + i] ∈ C then
7                  increase P[j : j + i].count;
8              j ← j + 1;
9      foreach subpath ∈ C do
10         if subpath.count ≥ k then
11             L ← L ∪ {subpath};
12 return L;
```

It requires another $O(l^2 \cdot |\mathbb{P}|)$ where $|\mathbb{P}|$ is the node number in $\mathbb{P}$. (3) Another serious issue is that the generated subpaths tend to be overlapped with each other and be covered by the longer subpaths. Namely, they cannot be matched effectively due to match collisions. According to our observation, (1) and (2) make it take too much time to generate a tiny table, whose compression quality is far from satisfactory. Instead, we come up with two more practical one-pass baselines named RSS and GFS. RSS stands for randomly sampling all candidates, which does not consider any measure. GFS stands for gross frequent subpaths, which picks top candidates in the order of a gross measure.

The process of one-pass baselines carries out the following steps as Algorithm 4 shows, where $\mathcal{H}$ is a hash table to maintain all vertex sequences and corresponding weights. The

Algorithm 4: TConstruct($\mathbb{P}$)

Input: A set of paths $\mathbb{P}$

Output: A lookup table $\mathcal{ST}$, and its inverted table $\mathcal{ST}^{-1}$

// collect and count $O(|P|^2)$ subpaths

1 **foreach** $P \in \mathbb{P}$ **do**

2 **foreach** $0 \leq i < |P|$ **do**

3 **foreach** $i < j \leq |P|$ **do** $\mathcal{H}$.add($P[i:j]$);

4 **if** $|\mathcal{H}| > \lambda$ **then** pick λ items in $\mathcal{H}$ via its rule;

5 create a table $\mathcal{ST}$ for decompression and an inverted table $\mathcal{ST}^{-1}$ for compression from $\mathcal{H}$;

1 **Procedure** $\mathcal{H}.add(seq)$

2 **if** $seq \in \mathcal{H}$ **then** $\mathcal{H}[seq] \leftarrow \mathcal{H}[seq] + 1$;

3 **else** $\mathcal{H}[seq] \leftarrow 1$;

weight of a vertex sequence seq in $\mathcal{H}$ is initialized by 1 once found. We increase the weight of seq in $\mathcal{H}$ every time it is matched elsewhere. At the start, it traverses paths and counts frequencies of all subpaths (lines 1–2). Then it picks the top λ with its rule if the number of candidates in $\mathcal{H}$ is more than the given threshold λ (line 4). Consequently, it builds a lookup table based on the remaining candidates(line 5). It is easy to realize that AFS serves as lines 1—4 in picking top candidates, however, the cost makes it impossible to build a large enough table in practice. Instead, we exploit RSS and GFS. RSS is a naive solution that randomly samples c out of candidates without considering any measure. GFS is named after the measure *gross weighted frequency* (for subpaths), the product of frequency and size. It works as the equivalent of AFS if we set the maximum size of candidates as l in lines 2—3. Better still, GFS solves both problems (1)(2) of AFS via collecting subpaths and counting their frequencies directly in data paths. Specifically, it takes $O(l^2 \cdot |\mathbb{P}|)$ to collect and count all subpaths and $O(l \cdot |\mathbb{P}| \cdot log(\lambda))$ to pick the top λ. Although GFS is suitable for pattern mining like AFS, we will show in the following that it is not an ideal measure for compression. Sometimes it is even worse than the most naive solution RSS, due to the problem (3). We will introduce a more efficient strategy focusing on the match collision issue in Section IV.

IV. Supernode Table Compression

A. Match Collision Issue

We clarify why the gross weighted frequency is far from a good choice for compression. As aforementioned, a straightforward and naive solution is to consider both frequency and size, as GFS does, to pick the most common subpaths. It is reasonable for frequent pattern mining, however, it is not an ideal measure for compression. Here is an instance to show its drawback due to match collision issues.

TABLE I: Supernode tables with different measures

$\mathcal{ST}$		$\mathcal{ST}^*$	
subpaths	supernodes	subpaths	supernodes
$v_2, v_3, v_5, v_8, v_{12}$	u_0	$v_2, v_3, v_5, v_8, v_{12}$	u_0^*
v_2, v_3, v_5, v_8	u_1	v_{13}, v_{21}	u_1^*
v_3, v_5, v_8, v_{12}	u_2	v_{17}, v_9	u_2^*
v_2, v_3, v_5	u_3	v_2, v_2	u_3^*
v_3, v_5, v_8	u_4		

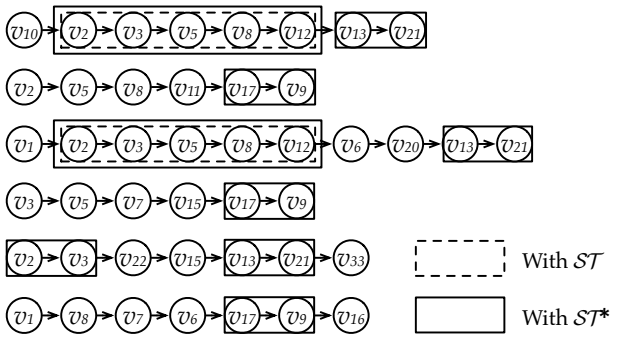


Fig. 3: Compression with tables of frequent subpaths

Example 1: Assume there is a path set as in Figure 3, the maximum size of supernodes is 5, and the capacity of the lookup table is 5. Note that if two candidates have the same weighted frequency, we prefer the longer one unless it has a frequency of 1. Following the scheme of collecting and selecting in Algorithm 4, the final supernode table $\mathcal{ST}$ is composed of $u_0 = \{v_2, v_3, v_5, v_8, v_{12}\}$, $u_1 = \{v_2, v_3, v_5, v_8\}$, $u_2 = \{v_3, v_5, v_8, v_{12}\}$, $u_3 = \{v_2, v_3, v_5\}$, $u_4 = \{v_3, v_5, v_8\}$. And the result of performing Algorithm 2 with that is as dotted boxes in Figure 3 shows, where only two paths get compressed. Suppose the issue is considered, and somehow the table $\mathcal{ST}^*$ is composed of $u_0^* = \{v_2, v_3, v_5, v_8, v_{12}\}$, $u_1^* = \{v_{13}, v_{21}\}$, $u_2^* = \{v_{17}, v_9\}$, $u_3^* = \{v_2, v_3\}$ under the same given constraints. Then the result after compression will be like solid boxes in Figure 3 with each path contracted, which has a better compression quality as well as a smaller table size.

The problem with GFS is that only one-fifth of the table is helpful for compression performance, while the rest contribute nothing since they have been covered as shown in Table I. In other words, the gross weighted frequency is inappropriate given a limited capacity λ. The result includes too many overlapped subpaths that it fails to give good compression performance. In contrast, candidates in a good lookup table are supposed to be complementary, like u_0^*, u_1^*, and u_2^* in $\mathcal{ST}^*$.

That suggests it unreasonable to generate and add every possible frequent subpath greedily in Algorithm 4. The naive solution has serious computational redundancy and misses potential complementary candidates due to the match collision

issue. More specifically, the overlapping problem lies in the process of extending and counting supernodes. For instance, suppose $\{v_2, v_3, v_5, v_8, v_{12}\}$ is matched in Figure 3, the frequencies of others appearing within that matched interval will never count under the compression framework in Algorithm 2. The practical frequencies of them during matching are definitely zero in this case. Following this idea, we put forward our measure to solve matched collision issues, *practical weighted frequency*, the product of practical frequency and size. The difference between gross and practical frequencies is that only the valid ones following the compression scheme count instead of counting occurrences at any position. Based on that, we will give a detailed description of how to construct a representative lookup table.

B. Identifying Supernodes by Merge and Expansion

We introduce our method to compute the supernode table. The key is selecting a set of vertex sequences that frequently appear in many paths.

Our idea is to start by considering some frequent pairs of vertices (i.e., edges) as supernodes. Then in each iteration, we extend the current candidate vertex sequences to a set of new longer sequences. The number of iterations is controlled by a parameter τ.

The formal pseudocode for table construction is described in Algorithm 5. Lines 1–2 add all edges in the path set to the hash table $\mathcal{H}$ as the initial candidate patterns (supernodes). Next, for each path P in each iteration, we start from the first node and identify the target sequence with the assistance of LongestPrefix, that matches a corresponding supernode in the candidate set $\mathcal{H}$ (line 5). Every time we identify a matched sequence (line 9), we increase the weight of the supernode in $\mathcal{H}$ (line 4 of Algorithm 6). At the end of each iteration, we generate new supernodes based on the current supernode (lines 10–15) and skip the sequence (line 16) to identify new patterns.

Merge and Expansion. We generate new candidate supernodes in each iteration using two strategies, merge and expansion, to leverage the latest matches as much as possible. Specifically, the merge strategy concatenates a pair of continuous supernodes found in a path (lines 10–13). Note that the length of a compressed sequence is bounded by δ. We cut the later part if the size of the merged sequence exceeds δ (lines 12–13). As for the expansion strategy, a complement to merging, it adds the following vertex to the sequence (lines 14–15). We only keep at most λ items in $\mathcal{H}$ in each iteration. Here we use the same input and constraints in Example 1 to show how to construct a supernode table from scratch.

Example 2: Suppose the input is as Figure 3, and we update the lookup table Table II in two iterations, along with initialization and finalization. Primarily, we initialize the table with all 26 edges with frequency one, where the weight suggests existence. In the first iteration, the maximum size of matched supernodes is two whenever calling LongestPrefix. If there are two adjacent successful matches, take $\{v_3, v_5\}$ and $\{v_8, v_{12}\}$ for instance, we not only increase their weights by one, but also count for the results of merge and expansion, $\{v_3, v_5, v_8, v_{12}\}$ and $\{v_3, v_5, v_8\}$. The top 5 candidates after the first iteration are shown in Table II. Then, the maximum size of supernodes increases to four, which is also the value of l in Algorithm 6 for the next iteration. Likewise, we keep matching, counting and picking the top 5 at the second iteration. After that, the candidate table is updated to include more representative sequences like $\{v_2, v_3, v_5, v_8, v_{12}\}$ and exclude overlapped ones like $\{v_3, v_5, v_8\}$. During finalization, we drop the useless ones with weight one, like $\{v_3, v_5, v_8, v_{12}\}$, and generate the same $\mathcal{ST}^*$ as in Table I.

Algorithm 5: TConstruct$^*(\mathbb{P})$

Input: A set of paths $\mathbb{P}$
Output: A supernode table $\mathcal{ST}$ and an inverted supernode table $\mathcal{ST}^{-1}$

```
// initialize H
1  foreach P ∈ ℙ do
2      foreach e ∈ P do H.add(e);
3  for 1 ≤ i ≤ τ do
4      foreach P ∈ ℙ do
5          match ← LongestPrefix(0, P, i, H);
6          pos ← |match|;
7          while pos < |P| do
8              pre_match ← match;
9              match ← LongestPrefix(pos, P, H);
               // Merge
10             if |pre_match| + |match| ≤ δ then
11                 H.add(pre_match ⊕ match);
12             else
13                 H.add(pre_match ⊕ match[0 :
                     δ − |pre_match|]);
               // Expansion
14             if |match| > 1 ∧ |pre_match| < δ then
15                 H.add(pre_match ⊕ P[pos]);
16             pos ← pos + |match|;
17     if |H| > λ then keep top-λ items in H;
18 create a supernode table ST for decompression and an
     inverted table ST^{-1} for compression from H;
```

C. Practical Implementation

We give a detailed description of how to identify the longest matched supernodes in Algorithm 7.

As aforementioned, the upper bound for identifying the longest matched supernode at any position is $O(\delta^2)$, because we only store the exact key in the hash table and the cost of each hash is linear to the size of sequences. The drawback is that there will be much redundancy in hashing common

TABLE II: Candidates and weights during table updates

Stage	Initialization		1st iteration		2nd iteration		Finalization	
	candidates	weights	candidates	weights	candidates	weights	subpaths	supernode ids
Top-5	v_2, v_{10}	1	v_3, v_5, v_8, v_{12}	2	$v_2, v_3, v_5, v_8, v_{12}$	2	$v_2, v_3, v_5, v_8, v_{12}$	u_0^*
	v_2, v_3	1	v_3, v_5, v_8	2	v_{13}, v_{21}	3	v_{13}, v_{21}	u_1^*
	...	...	v_3, v_5	3	v_{17}, v_9	3	v_{17}, v_9	u_2^*
	v_{17}, v_9	1	v_{13}, v_{21}	3	v_2, v_3	2	v_2, v_3	u_3^*
	v_9, v_{16}	1	v_{17}, v_9	3	v_3, v_5, v_8, v_{12}	1		

Algorithm 6: LongestPrefix$(pos, P, i, \mathcal{H})$

Input: A position indicator pos, a path P, an iteration indicator i, the supernode table $\mathcal{H}$
Output: A prefix of P

1 $l \leftarrow \min(2^{i+1}, \delta, |P| - pos)$;
2 **while** $l > 1$ ***and*** $P[pos : pos + l] \notin \mathcal{H}$ **do**
3 $l \leftarrow l - 1$;
4 **if** $l > 1$ **then** $\mathcal{H}.add(P[pos : pos + l])$;
5 **return** $P[pos : pos + l]$;

Algorithm 7: LongestPrefix$^*(pos, P, i, \mathcal{H})$

Input: A position indicator pos, a path P, an iteration indicator i, the supernode table $\mathcal{H}$
Output: A prefix of P

1 $l \leftarrow \min(2^{i+1}, \delta, |P| - pos)$;
2 **if** $l \leq \alpha$ **then**
3 **while** $l > 1$ ***and*** $P[pos : pos + l] \notin \mathcal{H}_1$ **do**
4 $l \leftarrow l - 1$;
5 **if** $l > 1$ **then**
6 $\mathcal{H}_1.add(P[pos : pos + l])$;
7 **else if** $P[pos : pos + \alpha] \in \mathcal{H}_2$ **then**
8 **while** $l > \alpha$ ***and*** $P[pos + \alpha : pos + l] \notin \mathcal{H}_2[P[pos : pos + \alpha]]$ **do**
9 $l \leftarrow l - 1$;
10 **if** $l > \delta$ **then**
11 $\mathcal{H}_2[P[pos : pos + \alpha]].add(P[pos + \alpha : pos + l])$;
12 **else**
13 $\mathcal{H}_1.add(P[pos : pos + \alpha])$;
14 **else**
15 **return** LongestPrefix$^*(pos, P[0 : pos + \alpha], i, \mathcal{H})$;
16 **return** $P[pos : pos + l]$;

prefixes when we start identifying the prefix from the longest length.

Example 3: Assume the inputs i and pos of Algorithm 6 make $l \geq 8$ in line 1, P is $\{v_8, v_5, v_0, v_9, v_1, v_3, v_4, v_2\}$, and the return value is $\{v_8\}$. That is, we meet the worst case with no valid matched supernode. During identification, we compute and combine the hashes from v_8 to v_2 at the beginning. However, there is no matched key in $\mathcal{H}$, so we pop the back v_2. We repeat hashing and popping back until only v_8 remains and return it. The total cost for that is $(8 + 2)(8 - 2 + 1)/2 = 35$, where the common prefix are involved repeatedly.

It is not difficult to realize the problem lies in the storing data structure, namely, the hash table does not support prefix key matching. However, we cannot afford to store all possible prefixes either, as it requires $O(\delta^2)$ space for each supernode. It is a reasonable tradeoff between space and time costs to leverage a multi-level hash scheme. Specifically, all supernodes with sizes no larger than α are kept in a one-level hash map $\mathcal{H}_1$. The others are maintained in a two-level hash map $\mathcal{H}_2$ with the primary key $P[pos : pos + \alpha]$ and the secondary key $P[pos + \alpha : pos + l]$.

Based on that, we implement our scheme for identifying the longest matched supernodes in Algorithm 7. Line 1 starts with initializing the possible maximum size l for matching. If it is too small to have a secondary key for matching (line 2), the process is reduced to what Algorithm 6 does in lines 3—6, otherwise, we look it up in $\mathcal{H}_2$. If there is a match of the primary key $P[pos : pos+\alpha]$ in $\mathcal{H}_2$, we just move on matching the remaining suffix as the secondary key $P[pos+\alpha : pos+l]$ (lines 8—9) and increase the weight for successful matches of suffixes (lines 10—11) or prefixes (lines 12—13), otherwise, it falls back to $\mathcal{H}_1$ with its prefix $P[pos : pos + \alpha]$ (line 15). Besides, a small trick to optimize the matching in lines 7—13 is that there will always be a valid key of $P[pos : pos + \alpha]$ in $\mathcal{H}_1$ if it exists in $\mathcal{H}_2$. It guarantees that we never fall back to lines 3—6 whenever line 7 is satisfied. Line 16 returns the longest matched subpath.

Lemma 3: Given a path P, compressing or identifying longest matched supernodes in P takes $O(max(|P| \cdot \alpha^2, |P| \cdot (\delta - \alpha)^2))$ time, where α is the maximum length of the primary hash key and $\delta - \alpha$ is the maximum length of the secondary hash key.

As we can get in Lemma 3, the refined upper bound is less than $O(|P| \cdot \delta^2)$ and the optimal value of α is supposed to be approximately $\delta/2$.

Example 4: Assume the input is the same as in Example 3, and the return value is still the worst case $\{v_8\}$, and α is 5. Since $8 > 5$, there remain two possible cases, (1) the prefix $\{v_8, v_5, v_0, v_9, v_1\}$ is not matched in two-level hash table $\mathcal{H}_2$,

the total cost in this case is $(5+2)(5-2+1)/2 = 14$ in line 3—6, otherwise, (2) it is matched as primary key in $\mathcal{H}_2$, then the upper bound for hashing is $5 + (3+1) \cdot 3/2 = 11$. And the maximum value 14 is less than 35 in Example 3.

In practice, it works pretty fine in current datasets, however, it does not make a difference in polynomial time. And we will cover possible optimizations for more complicated cases in the following subsection.

D. Possible Optimizations

As a component of the real-life industrial system of Alibaba Cloud, it is essential to consider all possible edge cases for comprehensiveness and robustness sake. We will face challenges when the average length of paths extends to hundreds or even thousands. The cost of the current version might grow dramatically. The main drawback of the current bottom-up framework comes from the accumulation of candidates through iterations and the growth of costs of identifying matched supernodes.

Focusing on that, here we put forward two possible optimizations. They have not been tested yet due to the absence of datasets with longer average lengths at the moment. (1) We can implement a hybrid framework combining top-down with bottom-up. Namely, the top-down framework cuts the least important nodes to generate shorter subpaths. It is triggered as a complement for the bottom-up scheme when the target length δ is large enough. (2) Following the idea of Lemma 3, it is possible to leverage a more complicated hierarchical structure. We can adopt a new data structure to serve as Prefix-Tree or Trie [40] for vertex sequences. Each node in the tree is composed of an index of the vertex and pointers to the next vertices in subpaths. This structure tailored for prefixes will identify matches faster with less run-time memory cost. Better still, the upper bound of each prefix match is optimized from $O(\delta^2)$ to $O(\delta)$ in this way, where the depth of tree structure equals δ.

V. Complexity analysis

A. Time complexity

The time cost for prefix matching during table construction (Algorithm 5) could be divided into initialization (lines 1—2), iteration (lines 3—17), and finalization (line 18). As for the main part during iteration, the cost to identify matched supernodes (line 9) for a path set of total $|\mathbb{P}|$ nodes is $O(|\mathbb{P}| \cdot \delta^2)$ bounded by constant times of hashes. Given the bound of the capacity of the candidate set λ, the cost to update the table (line 17) is $O(|\mathbb{P}| \cdot \ log(\lambda))$. It is derived from keeping top-λ elements in a min-heap of size λ, with at most $|\mathbb{P}|$ updates. Likewise, the overhead for the initialization with edges (lines 1—2) and the finalization to return the lookup table (line 18) is $O(|\mathbb{P}| + \lambda \cdot \delta)$. Better still, we could make it more efficient by sampling. In our observation, it is still effective to sample less than one percent of all paths during table construction. In sum, the total cost of i iterations with sample rate s, i.e., one in every s paths, is $O((i \cdot (|\mathbb{P}| \cdot \delta^2 + |\mathbb{P}| \cdot log(\lambda)) + |\mathbb{P}| + \lambda \cdot \delta)/s)$. Note that we set λ linear to $|\mathbb{P}|$ with a fixed factor β in practice. Therefore, that could be further simplified as $O(\gamma \cdot |\mathbb{P}|)$, where the factor $\gamma = (i \cdot (\delta^2 + log(|\mathbb{P}| \cdot \beta)) + \beta \cdot \delta)/s$. As we can see, the total time cost heavily relies on i, δ, β and s. In practice, δ is set as a constant, and β is large enough to trigger filters only in the last few iterations. There remain two significant parameters i and s, to which we will pay close attention in the following experiments.

As for the compression phase in Algorithm 2, what runs likewise as table construction. The difference is that the table becomes static, where matching does not lead to merging or expanding. Since it is one-pass travel in the whole dataset, the upper bound is $O(|\mathbb{P}| \cdot \delta^2)$ with no doubt. Better still, it is safe for us to run compression with the static table in parallel. More specifically, we are able to implement pleasing parallelism on a finer granularity as small as a path in $O(|\mathbb{P}| \cdot \delta^2/p)$ on a p-core machine. Likely, the decompression phase in Algorithm 1 is also easy to parallelize with the bound $O(|\mathbb{P}|/p)$, which makes it a lightweight and fast method.

B. Space complexity

The overhead during compression is under strict control, as stated in the section before. The runtime memory during table construction in Algorithm 5 only requires a tiny sample $O(|\mathbb{P}|/s)$. The space for the heap of candidates is bounded by $O(|\mathbb{P}| \cdot \beta \cdot \delta)$ in each iteration. Therefore, the total space cost is $O(|\mathbb{P}| \cdot \nu)$, where ν is bounded by $O(1/s + \beta \cdot \delta)$. In our observation, the supernode table is still effective when the overhead factor ν is less than 0.03.

As for phases of compression and decompression of Algorithm 2 and Algorithm 1, the required space can be as small as several I/O blocks plus dictionary size $O(|\mathbb{P}| \cdot \beta \cdot \delta)$ thanks to our finer granularity. The upper bound of output size is linear to $|\mathbb{P}|$ with a small overhead factor $\beta \cdot \delta$. In other words, the worst ratio of input size to output size is $(1/(1 + \beta \cdot \delta))$. Assume all supernodes are of length δ, and each subpath is exactly replaced by a supernode, then the ideal ratio for the best case is δ.

VI. Experimental Study

Using real-life data, we conducted three sets of experiments to evaluate the (1) impacts of parameters for compression, (2) comparison between OFFS and baselines in compression ratio and compression speed, and (3) retrieval and scalability.

A. Experimental setting

Datasets. We use four real-life datasets as shown in Table III. (a) The first one is a private real-life dataset that monitors the IP hops from servers to clients in the network of Alibaba Cloud. Our sample logs IP hops in its network on June 18th 2021. It has 171,024,135 paths with maximum length

TABLE III: Real-world paths

Dataset	path number	node number	id number	maximum length	average length
Alibaba Cloud	171,024,135	2,941,010,457	426,248	30	17.20
Rome	3,426,475	229,972,163	39,078	503	67.12
Porto	36,898,213	1,207,828,790	137,288	1,355	32.73
San Francisco	5,857,208	102,038,897	6,026	103	17.42

30, 426,248 distinctive ids, and 2,941,010,457 nodes in total. We also involve additional public datasets from real-life taxi trajectories to further demonstrate the compression performance. (b) Rome [41] is a dataset that covers the trajectories of approximately 320 taxis in February 2014 in Rome. It consists of 3,426,475 paths with maximum length 503, 39,078 distinctive ids, and 229,972,163 nodes. (c) Porto [42] is a real-life dataset that records traces of 442 taxis from 01/07/2013 to 30/06/2014 in Porto, Portugal. It contains 36,898,213 paths with maximum length 1355, 137,288 ids and 1,207,828,790 nodes in total. (d) San Francisco [43] is a trajectory dataset covering approximately 500 taxis collected over 30 days in San Francisco Bay Area. It has 5,857,208 paths with maximum length 103, 6,026 ids and 102,038,897 nodes in total. It is not difficult to observe that average lengths are less than 100 in all datasets, while the maximum length can be several hundred times those.

Please note that we make use of the following preprocessing to get datasets ready for further experiments.

New id. It is easy to assign new ids to IPs in the Alibaba Cloud network, i.e., integers from zero to indicate different vertices. Trajectories are recorded as sequences of pairs of $\{longitude, latitude\}$ in time order. Due to vehicle movements and GPS errors, it does not make sense to denote distinct pairs by new ids. Namely, it is abnormal for taxi drivers in the same city to never drive on the same road. Therefore, we need to increase spatial granularity by dividing the space into grids based on various time intervals. In this way, we merge nodes in the same grid into one.

Simple path. There can be duplicated nodes in the same path as a consequence of several reasons. (a) Noise. Whenever we encounter a sequence of adjacent duplicate vertices, we keep only the first one and drop the rest. (b) Cycle. We solve the loop issue by cutting before the first recurring node and generating two shorter paths. In addition, we prune the trivial data by discarding paths of size no more than 2. Having solved (a)(b), we can rest assured that the output paths always stay simple.

Group set. The generated path sets are grouped according to given rules for future mining in the dataset of Alibaba Cloud. Likewise, we organize paths of taxi drivers into sets. For instance, we classify them according to their starting and ending vertices, or passing vertices of interest. Note that although paths are distinct in a given set, they can recur among sets.

Implementation details. We then provide the necessary details about the baseline implementations.

The algorithms are implemented in C++ and compiled by clang++ 14.0.6 with -O3. We conducted all experiments on a Unix machine with eight processors of 3.20 GHz and 16 GB memory.

All competitors involved work under the framework of building lookup dictionaries and compressing data path by path. The sample rate for table construction is set to 128 during comparison. The dictionary of Dlz4 is constructed by zdict with enough samples, as the official documentation suggests. Specifically, we pick one in every 128 as sample, and divide them into blocks of 1 KB for training a dictionary. Note that we attempted to allocate blocks path by path or provide more samples as a favor, but it did not make a difference in our observation. As for RSS and GFS, the table capacity c is the same as for OFFS. In our observation, they still take too much time for the candidate collection given a limited supernode size. Therefore, we set the threshold $5 \cdot c$ to speed them up during table construction. For OFFS, we set the maximum length of primary keys α to 5, the maximum length of subpaths in the table δ to 8, and the factor between the table capacity and the input size β to 500. Last but not least, all of them are implemented with OpenMP default parallelism during compression and decompression for fairness' sake.

B. Measures

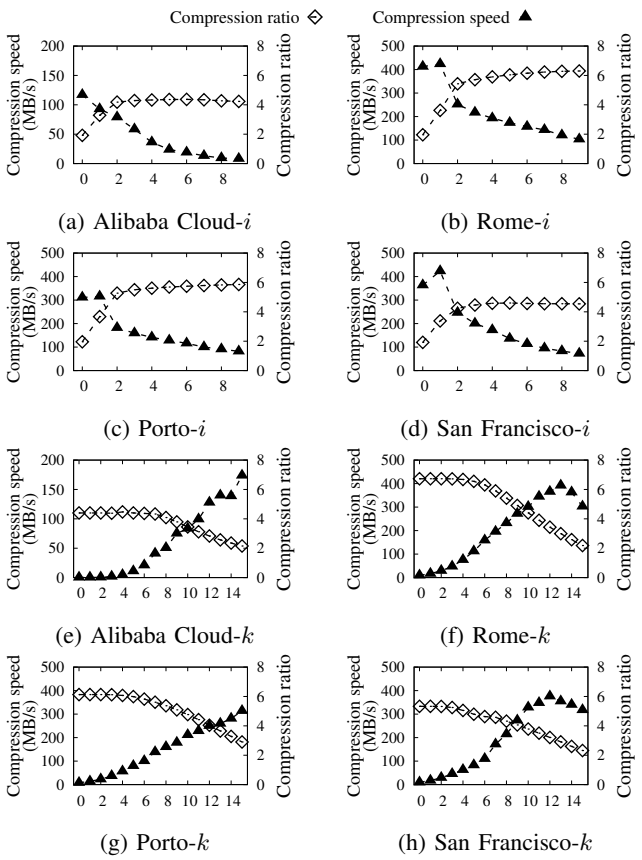
We next describe our measures of compression performance in experiments.

Compression ratio. Suppose the cost to store contracted paths and rule is $|P'| + |R|$ with the raw size of paths $|P|$, the compression ratio, CR of the corresponding compression scheme $f : P \Rightarrow (P', R)$ is $CR = \dfrac{|P|}{|P'| + |R|}$.

Compression speed. Given the time of compression and decompression T_c, T_d, the compression speed is defined as $CS = \dfrac{|P|}{T_c}$, likewise, the decompression speed is defined as $DS = \dfrac{|P|}{T_d}$. For any partial decompression $f^T : (Q', R) \Rightarrow Q, Q \subseteq P$ with decompression time T_{pd}, the partial decompression speed is defined as $PDS = \dfrac{|Q|}{T_{pd}}$.

C. Experimental results

Exp-1: Impacts of i and k. We first study the impacts of two significant parameters during table construction, the number



(a) Alibaba Cloud-i (b) Rome-i (c) Porto-i (d) San Francisco-i (e) Alibaba Cloud-k (f) Rome-k (g) Porto-k (h) San Francisco-k

Fig. 4: Impacts of parameters i (a–d) and k (e–h) on compression speed and compression ratio

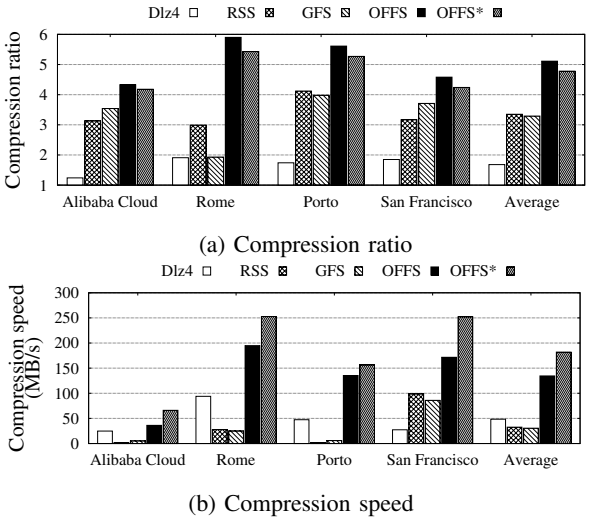
of iterations i and the exponent k of the base 2 as the sample rate. The results are shown in Figure 4. We provide a detailed analysis of their impacts in terms of CR and CS in the following.

(1) As i changes from 0 to 3, CR increases rapidly. Note that the maximum size of current candidates reaches δ at the third iteration. Thereafter, it grows gently as i changes from 3 to 9. On average, it increases by 3 as the maximum size of supernodes in the table grows, and by 0.6 as more iterations are involved to refine the candidates.

(2) As k increases from 0 to 7, CR slowly decreases. Meanwhile, the sample size goes from the full set to less than one percent. It then drops sharply as k changes from 7 to 15. On average, the drop from 0 to 7 is 0.7, while it becomes almost half from 7 to 15.

(3) CS becomes half as i changes from 0 to 4. Then it drops again by half as i increases from 4 to 9. Not surprisingly, the differences are significant when the main update in the table is to extend supernodes in the first three iterations. Thereafter, as the candidates in the table change less, the differences in CS become smaller.

(4) CS increases sharply by 20 times as k changes from 0 to 7. It then slowly doubles with k increasing from 7 to 15.



(a) Compression ratio

(b) Compression speed

Fig. 5: Comparison with baselines on compression ratio and compression speed

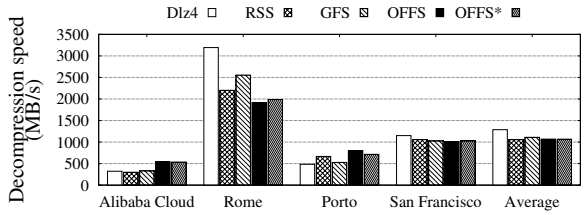
When k is small, the table construction phase accounts for most of the time cost, so CS changes considerably. While k becomes larger, the compression phase accounts for most of the time cost. Meanwhile, it might suffer from more useless matches during compression, which affects CS.

(5) Regarding the trade-off between CS and CR, we pick two sets of (i, k), the default mode $(4, 7)$ and the the fast mode $(2, 7)$. The default mode continues to refine candidates after the maximum size of candidates reaches δ, while the fast mode finishes the table construction after that. We denote them as OFFS and OFFS* for the following comparison.

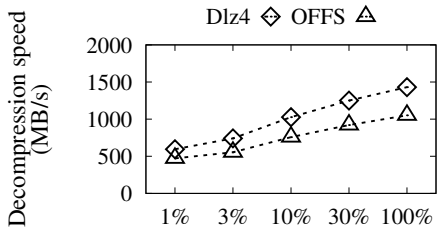
Exp-2: Comparison with baselines. We then compare the proposed OFFS with two sets of baselines. Dlz4 serves as representative of generic compression methods, while RSS and GFS work as naive DICTs. Experiments are conducted on all datasets focused on CS and CR. The results shown in Figure 5 tell us the following.

(1) CR of OFFS is 5.11 on average as shown in Figure 5a, which is more than 3 times that of Dlz4 and 1.5 times those of RSS and GFS. Note that all dictionaries are trained from sufficient samples, which means the larger samples will not make a difference to CR. The result shows that OFFS works much better than Dlz4 and enhances a lot compared with GFS. It comes from the optimization of practical weighted frequency. We are not surprised to find that the average CR of GFS is worse than that of RSS due to match collisions. As for the quick mode, OFFS* only loses 0.33 compared to OFFS. These observations in Figure 5a suggest that OFFS improves a lot from Dlz4, providing better and stabler CR compared to the naive DICT solutions.

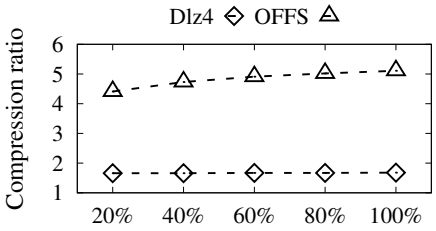
(2) CS of OFFS is 135 MB/s on average. It is 3 times faster than Dlz4 and 4 times faster than naive DICTs, while OFFS* further improves the speed of OFFS by 1.5 times. Figure 5b



(a) Decompression speed



(b) Scalability of decompression



(c) Scalability of dictionary

Fig. 6: Decompression comparison and scalability test

shows that naive DICTs are slower than Dlz4. Specifically, the gap could be an order of magnitude in large datasets like Porto and Alibaba Cloud. This suggests inefficiency and redundancy during the candidate collection process of naive DICTs in our case. As we notice in the Alibaba Cloud dataset, the speed drops when the space cost exceeds the available memory and causes a memory swap with I/O. Despite our attempts to avoid I/O, it cannot be neglected when the input size is large enough. It is preferable to adopt a more advanced stream mode that simultaneously handles reading and processing for that. However, here we leave it as a pressure test of an extreme case of limited memory space. The result shows that OFFS is still more robust compared to baselines, and it is faster in CS and much better in CR when I/O is inevitable.

Exp-3: Retrieval and scalability. We next evaluate the speed of data retrieval with decompression and the scalability of table construction. Note that all decompression processes start from the results in memory to avoid I/O impacts. For either full or partial decompression, the output is the same set or subset as the input before compression, which is lossless. To test the scalability of table construction, we randomly pick paths to simulate the real-world case where we build a table based on first arriving samples.

(1) We first compare the DS of whole datasets between OFFS and baselines. Note that the DS of Dlz4 is almost ideal without reallocating memory because the exact sizes of input paths are recorded. It is to avoid memory issues on datasets like Porto whose maximum length is hundreds of times of average length. The results in Figure 6a show that OFFS is competitive against Dlz4 with DS around 1000 MB/s. It is not surprising to find that all DICT-based methods have approximately the same DS since they follow the same decompression strategy bounded by $O(|\mathbb{P}|/p)$.

(2) We next compare PDS between Dlz4 and OFFS with scalability from 1 to 100 percent of whole datasets. As shown in Figure 6b, DS of OFFS is 0.75 on average of that of Dlz4, and the average PDS is around 500 MB/s when the sample rate is 1 percent. It suggests that OFFS can handle partial decompression for data retrieval efficiently.

(3) We then demonstrate the scalability of OFFS based on various sample rates for table construction. The test sample rates change from 20 to 100 percent. As shown in Figure 6c, CR changes from 4.4 to 5.1 with the sample rate changing from 20% to 100%. In other words, the relative loss of CR is less than 15% when building on a sample as tiny as 20%. Better still, the CR of OFFS is more than 2.5 times that of Dlz4 with 20% sampling in our observation. It shows that OFFS is capable to build a more representative table than that of Dlz4 under scalability tests.

VII. CONCLUSION

In this paper, we present Overlap-Free Frequent Subpath, a strategy inspired by real-life cases from Alibaba Cloud to effectively reduce the overall size of path sets with easy retrievals to compressed paths. Specifically, we take advantage of a bottom-up framework of merge and expansion during the table construction stage to count practical weighted frequencies and refine the lookup table with iterations. Moreover, the granularity as finer as a path makes it friendly for parallelism in both compression and decompression stages. We clarify how to solve the challenge of match collisions of overlapped subpaths during table construction, provide analyses of time and space complexity, and compare with baselines in evaluation to highlight the performance of OFFS. Experiments show that OFFS dramatically improves compression ratio and compression speed, has competitive decompression speed, and demonstrates the scalability of table construction based on tiny samples.

ACKNOWLEDGMENT

We thank the colleagues of Alibaba Cloud who provided insight and expertise that greatly assisted the research. We would also like to express our gratitude to our colleagues at Alibaba DAMO Academy who helped evaluate the results and provided valuable comments.

REFERENCES

[1] P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," Tech. Rep. 2070-1721, 1996.

[2] "Zstd," http://facebook.github.io/zstd.

[3] "Lz4," https://lz4.github.io/lz4.

[4] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *PVLDB*, vol. 1215, 1994, pp. 487–499.

[5] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[6] S. Guha, "Efficiently mining frequent subpaths," in *AusDM*, 2009, pp. 11–15.

[7] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[8] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[9] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 06, pp. 8–19, 1984.

[10] P. Deutsch, "Deflate compressed data format specification version 1.3," Tech. Rep. 2070-1721, 1996.

[11] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *CACM*, vol. 30, no. 6, pp. 520–540, 1987.

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. Inst. Radio Eng.*, vol. 40, no. 9, pp. 1098–1101, 1952.

[13] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner, "Lightweight data compression algorithms: an experimental survey (experiments and analyses)," in *EDBT*, 2017, pp. 72–83.

[14] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *ICDE*, 1998, pp. 370–379.

[15] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *ICDE*, 2006, pp. 59–59.

[16] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Softw., Pract. Exper.*, vol. 45, no. 1, pp. 1–29, 2015.

[17] M. A. Roth and S. J. Van Horn, "Database compression," *SIGMOD Rec.*, vol. 22, no. 3, pp. 31–39, 1993.

[18] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006, pp. 671–682.

[19] R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: experiments with compressing suffix arrays and applications." in *SODA*, vol. 4, 2004, pp. 636–645.

[20] P. Ferragina and G. Manzini, "Indexing compressed text," *JACM*, vol. 52, no. 4, pp. 552–581, 2005.

[21] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: enabling queries on compressed data," in *NSDI*, 2015, pp. 337–350.

[22] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du, "Compressdb: enabling efficient compressed data direct processing for various databases," in *SIGMOD*, 2022, pp. 1655–1669.

[23] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica*, vol. 10, no. 2, pp. 112–122, 1973.

[24] R. Bellman, "On the approximation of curves by line segments using dynamic programming," *CACM*, vol. 4, no. 6, p. 284, 1961.

[25] N. Meratnia *et al.*, "Spatiotemporal compression techniques for moving point objects," in *EDBT*, 2004, pp. 765–782.

[26] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *ICDM*, 2001, pp. 289–296.

[27] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, and D. Vaccaro, "On-line data reduction and the quality of history in moving objects databases," in *MobiDE*, 2006, pp. 19–26.

[28] M. Potamias, K. Patroumpas, and T. Sellis, "Sampling trajectory streams with spatiotemporal criteria," in *SSDBM*, 2006, pp. 275–284.

[29] G. Antoshenkov, "Dictionary-based order-preserving string compression," *VLDBJ*, vol. 6, no. 1, pp. 26–39, 1997.

[30] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD*, 2009, pp. 283–296.

[31] G. Antoshenkov, D. Lomet, and J. Murray, "Order preserving string compression," in *ICDE*, 1996, pp. 655–663.

[32] M. Farach and M. Thorup, "String matching in lempel-ziv compressed strings," in *STOC*, 1995, pp. 703–712.

[33] R. Lasch, I. Oukid, R. Dementiev, N. May, S. S. Demirsoy, and K.-U. Sattler, "Faster & strong: string dictionary compression using sampling and fast vectorized decompression," *VLDBJ*, vol. 29, no. 6, pp. 1263–1285, 2020.

[34] P. Boncz, T. Neumann, and V. Leis, "Fsst: fast random access string compression," *PVLDB*, vol. 13, no. 12, pp. 2649–2661, 2020.

[35] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *CSUR*, vol. 51, no. 3, pp. 1–34, 2018.

[36] K. LeFevre and E. Terzi, "Grass: Graph structure summarization," in *SDM*, 2010, pp. 454–465.

[37] M. Riondato, D. García-Soriano, and F. Bonchi, "Graph summarization with quality guarantees," *ICDM*, vol. 31, no. 2, pp. 314–349, 2017.

[38] W. Fan, Y. Li, M. Liu, and C. Lu, "Making graphs compact by lossless contraction," in *SIGMOD*, 2021, pp. 472–484.

[39] Q. Yong, M. Hajiabadi, V. Srinivasan, and A. Thomo, "Efficient graph summarization using weighted lsh at billion-scale," in *SIGMOD*, 2021, pp. 2357–2365.

[40] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[41] L. Bracciale, M. Bonola, P. Loreti, G. Bianchi, R. Amici, and A. Rabuffi, "CRAWDAD dataset roma/taxi (v. 2014-07-17)," Downloaded from https://crawdad.org/roma/taxi/20140717, Jul. 2014.

[42] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *TITS*, vol. 14, no. 3, pp. 1393–1402, 2013.

[43] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD dataset epfl/mobility (v. 2009-02-24)," Downloaded from https://crawdad.org/epfl/mobility/20090224, Feb. 2009.