SimStore: Efficient Data Management for Network Propagation Simulation

Dacheng Qu, Lin Zhang and Zhao Cao
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

***Abstract*—Simulation is widely adopted in large scale network propagation analytics. Plenty of analytics scenarios require to retrieve, review the simulation status of a given time points or interval. Unfortunately, it is unaffordable to re-run the simulation due to the long running time and other costs in many cases. In this paper, we introduce a system (SimStore) to enable efficient storage and retrieval of simulation snapshots. We present a novel technique to compress a series of simulation snapshots in order to reduce the storage cost. Experimental results demonstrate the efficiency and effectiveness of the proposed methods.**

Keywords-social network; propagation; simulation; data management; compression

I. INTRODUCTION

Large scale network propagation analytics, such as social networks[1][2][3] and epidemic networks[4][5][6], are essential in a large number of applications. To illustrate, let's consider the following analytics that may be conducted on the large scale networks:

- What-if analysis, which includes simulation such as "what are the most impacted vertices/edges if I change a property of a vertex or edge" or "show me the influence simulation result if I could control one or a subset of the vertices in a network". Such what-if analytics are simulations that run from an intermediate state of the simulation.

- Review/replay the propagation process, which includes scenarios such as "show me the intermediate state of 10 minutes ago" or "give me a quick reply of the influence propagation process since the 5th iteration". This class of scenarios requires getting the intermediate state in real-time.

Due to the large scale network and complexity of propagation rules, it would take a long time (e.g., few hours/days or even months) to reach the final state of the simulation. To enable the further quick analysis of the simulation process without run the simulation again, one of the viable approaches is to store the simulation data of important time points. For example, the analyzer would require storing the simulation snapshot in every 10 seconds for review/replay purpose.

Unfortunately, persist every simulation snapshot per few epics is unacceptable due to the heavy I/O cost; it even further affects the simulation performance. We aim to provide a system (SimStore), which could compress the snapshots, to efficiently persist and retrieve the snapshots of network simulation. This is challenging because: (1) vertices properties change slightly and frequently caused by the propagation; (2) network structure might change in the simulation either caused by the propagation rules or external input. We make the following contributions: (1) we design a system to manage the network simulation snapshot data efficiently; (2) we propose a novel technique to compress a series of simulation snapshots efficiently; (3) we design a query processing approach; (4) our contributions also include experiments over synthetic dataset to demonstrate the efficiency of the system.

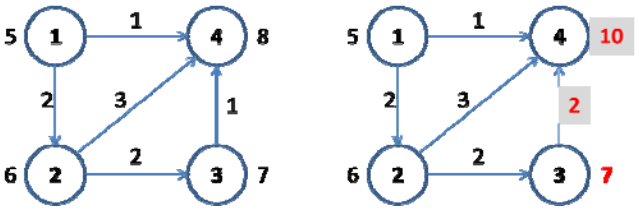
The remainder of this paper is organized as follows: We present a detailed problem statement and give an overview of the SimStore in Section II. The proposed simulation snapshots compression algorithm and query processing approach are presented in Section III. Experimental evaluation results are reported in Section IV. We review related work and conclude this paper in Section V and VI respectively.

II. PROBLEM STATEMENT AND SYSTEM OVERVIEW

A. Problem Statement

In network propagation simulation, a network is considered as a direct acyclic graph G=(V, E), where V and E are the sets of vertices and edges respectively. Each vertex and edge could have a set of properties. For a given vertex $v \in V$, let $A(v)=\{a_i=v_i|1 \leqslant i \leqslant |AV|\}$ be the set of properties for vertex v, where |AV| is number of attributes for each vertex. For an edge $e \in E$, let $A(e)=\{a_i=v_i|1 \leqslant i \leqslant |AE|\}$ be the set of properties for edge e, where |AE| is the number of attributes for each edge. Vertex $v \in V$ propagates its property value to its neighbor vertices, lets denote it as N(v), through edges. In the simulation, propagation consists of a series of iterations to reach the final state. At each iteration, both of V and E, including properties of each vertex in V and edge in E, would be changed according to propagation rules R and external user input. We consider each iteration of the propagation state as a snapshot of the intermediate result, which is also a graph, let's denote it as G_i as the result of *i-th* iteration. G_0 is the initial network structure. Note that the network structure would be updated in the simulation run.

We would like to store these snapshots into a distributed key-value store, which support efficient retrieval of specified one or series of snapshots, with flexible scalability support. A retrieval query is of form *q=(start, end)* to specify the start round and end iteration id. A straightforward approach is to store each snapshot directly; unfortunately, it requires a huge and unaffordable size of storage. One of the major technique challenges is how to persist a series of snapshots $\{G_0, G_1, G_2,\ldots\}$ with as less storage cost as possible.



(a) Baseline graph (b) next graph

FIGURE I. AN ILLUSTRATIVE EXAMPLE

Since the graph structure and properties would not change so significantly among nearby iterations, our intuitive idea is to remember the changed part of the properties and structures based on a selected baseline snapshot, instead of the full graph. For example, Figure 1(a) is the baseline snapshot, the snapshot in Figure 1(b) is changed a little bit based on graph Figure 1(a). We only need to remember that the property for vertex 4 and weight for edge 3->4 are changed as marked with gray background. Comparing to store the entire graph in Figure 1(b), storing only the changed part could reduce the storage cost significantly. For a series of snapshots S={G_1, G_2, G_3,…, G_N}, we would like to: (1) split S into batches BS_i={$G_{|B|*i+x}$|0 ⩽x<|B|}; (2) For each batch, we select the first snapshot as baseline and compress the subsequent snapshots by only remembering the changed parts and the delta values of the changes.

B. System Overview

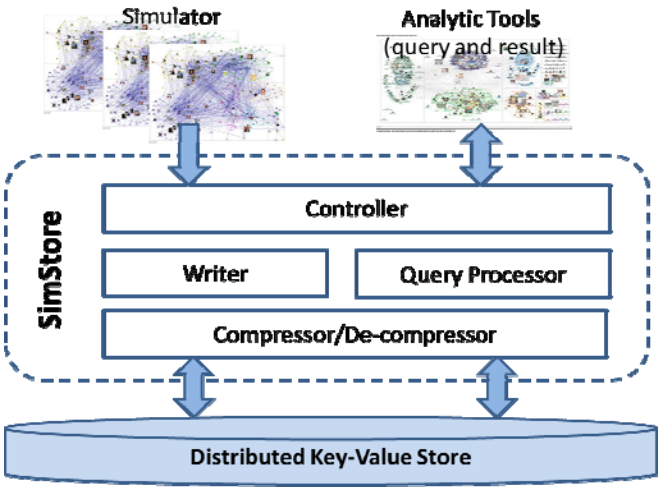


FIGURE II. SIMSTORE SYSTEM OVERVIEW

Figure 2 provides a high-level overview of SimStore and its logical architecture. Data enters through SimStore API invocation from the simulator. The data ingestion API is of the form *ingest(time t, Graph g)*. Snapshot is accessed via queries and the return (synchronously or asynchronously) of their results. The SimStore *Controller* in Figure 2 is the logical entry point for data ingestion and user requests. The *Writer* module is called by controller to cache the graph data in SimStore memory. The *Compressor* module is periodically called to compress the cached snapshots and persist the compressed result into the backend key-value store, for example Apache Cassandra is employed in our prototype system. The *Query Processor* module is called by controller to answer user requests. More specifically, it retrieves related compressed snapshots from backend key-value store; then calls the *decompressor* module to decompress and finally composes query results.

III. SIMULATION SNAPSHOTS COMPRESSION

Given a batch Bi={$G_{|B|*i+x}$|0⩽x<|B|}, we will describe how to compress a batch in this section. To compress the entire snapshots series, we just need to apply the same approach to each batch separately. We select the first snapshot $G_{|B|*i}$ in batch B_i as the baseline, let's denote it as BSL_i. As shown in Figure 3, for each snapshot, the snapshot data is split into three parts, including: network structure, vertices properties and edges properties. For the baseline, we store the graph data completely. However, for each subsequent snapshot, we only persist what are changed comparing to the baseline in structure, vertices properties and edges properties respectively. The following subsection A, B and C will present the compression strategy for each part. Subsection D will describe how to answer a query based on the compressed snapshots data.

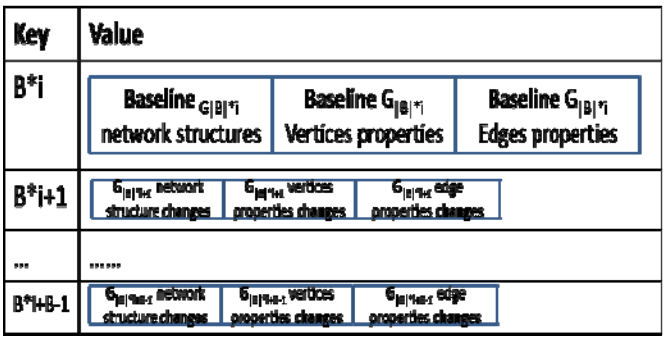


FIGURE III. COMPRESSED SNAPSHOTS LAYOUT IN KEY-VALUE STORE

We would like to exploit key-value store as the backend data store for its flexible scalability and efficient value retrieval. The iteration id of the snapshot is chosen as key in key-value store. For the baseline snapshot, we set the serialized baseline graph as value (as the value for key B*i shown in Figure 3). But for the subsequent snapshots, we set the serialized compressed snapshot (as the value for key B*i+1 shown in Figure 3).

A. Graph Structure Compression

The graph structure includes a list of vertices and edges between vertices. Consequently, for baseline network structures, we store a list of sorted vertices ids and a list of sorted edges in the form of (u, u') denotes an edge from vertex u to u', where the edges is sorted by u and then sorted by u' if two edges have the same u.

For each consecutive snapshot, one of the observations is that graph structure doesn't change so frequently. Hence, we would like to remember the changes only. For both of vertices and edges, there are two types of changes: add and remove. Consequently, we use four lists to maintain network structure

changes, including: (1) new added vertices; (2) removed vertices; (3) new added edges; (4) removed edges.

Algorithm 1 Graph structure compression algorithm

```
Input: a batch of snapshots Bi={G|B|*i+x|0≤x<|B|}
Output: four lists regarding to changes to graph structure
1: AddedV, RemovedV, AddedE, RemovedE←Φ
2: BL←G|B|*i
3: for each subsequent graph g∈Bi
4:     for each vertex v in V(g)
5:         if v does not exist in BL     AddedV.add(v)
6:     for each vertex v in V(BL)
7:         if v does not exist in g      RemovedV.add(v)
8:     for each edge e in E(g)
9:         if e does not exist in BL     AddedE.add(e)
10:    for each edge e in E(BL)
11:        if e does not exist in g      Removed.add(e)
12: return (AddedV, RemovedV, AddedE, RemovedE)
```

As shown in line 3 of Algorithm 1, for each subsequent snapshot, we compare it with the baseline snapshot. If a vertex or edge is in the subsequent snapshot but not in the baseline, it should be a new added vertex (line 5) or edge (line 9). In contrast, if a vertex or edge is in the baseline but not in the subsequent snapshot, it should be a removed vertex (line 7) or edge (line 11).

B. Vertices Property Compression

We observe that vertex properties are changed the most frequently in network propagation, however, the vertex property don't change significantly among consecutive iterations. Therefore, we would store the original vertex property values for the baseline graph. Then, for the consecutive snapshots, one design point is that we compute the property value changes. More specifically, we for each vertex, we would like to remember the delta comparing to the value in baseline graph. For example, a vertex v's property value is 1000 in the baseline graph, and its property value is updated to 1001 in the next iteration, we would only remember the delta(v)=1001-1000=1, which will use less bytes (e.g., 2 bytes in our prototype system) in storage. Another point is that we don't store the delta values as a list of (vertex id, delta value) tuples. Since the vertex list is sorted, we just store the delta values in the same order as vertex id list to save storage cost. However, for the new added vertices, we still persist their delta values as a list of (vertex id, delta value) tuples.

C. Edges Property Compression

We also observe that edge property values are changed much less frequently than vertex property. Meanwhile, the property value would not change significantly comparing to the nearby iterations. Hence, we decide to represent these changes in the form of (edge, new value) tuple list, mainly because the edge property changes are rare. We could save time cost to decompress the edge changes comparing to keep the delta only.

D. Query Processing

The basic idea for query processing is as follows: (1) compute the required batches according to the start and end iteration ids of the request; (2) for each required snapshot, we decompress the snapshot as shown in Algorithm 2, which returns the required snapshots.

Algorithm 2 Decompression algorithm

```
Input: iteration id in the request,iid
Output: a snapshot
1: CBL←retrieve compressed baseline from key-value store
2: CChange←retrieve compressed changes from key-value store
3: RS←Φ   //result snapshot
4: //decompress structure
5: RS.V←CBL.V
6: RS.E←CBL.E
7: RS.V←RS.V ∪ CChange.AddedV
8: RS.V←RS.V-CChange.RemovedV
9: RS.E←RS.E ∪ CChange.AddedE
10: RS.E←RS.E-CChange.RemovedE
11: //decompress vertices properties
12: for each vertex v's properties in CChange.VDelta
13:     RS.V.property(v)+=CChange.VDelta.property(v)
14: for each vertex v's properties in CChange.VAdded
15:     RS.V.property(v) = CChange.VAdded.property(v)
16: //decompress edge properties
17: for each edge e's properties  in CChange.Edges
18:     RS.E.property(e) =CChange.Edges.property(e)
19: return RS
```

As shown in Algorithm 2, for each required snapshot, we first decompress its network structure by merging the new added vertices and edges to the baseline graph (line 7 and 9), and eliminating the removed vertices and edges from the baseline graph (line 8 and 10). Then, we decompress the vertices properties, as shown from line 11-15 in Algorithm 2, for properties of the vertices existing in baseline snapshot, we add the delta value to the value in baseline graph; but for properties of new added vertices, the value is what is got from the compressed snapshot. Finally, edges properties values are calculated by merging the values in baseline graph and changed values in compressed snapshot as shown in line 17 and 18.

IV. EXPERIMENTAL EVALUATION

We next present an experimental study of SimStore system using synthetic data.

A. Experimental Setup

Hardware and platform: All our experiments were performed on an Intel(R) Core(TM) i3-5010U 2.1GHz CPU machine running Ubuntu14.04 with 4GB of RAM. The system was implemented in Java and used Apache Cassandra as backend key-value store.

Datasets: We implemented a network simulator to: (1) generate the initial network structure and vertices/edges properties, where the values of vertices/edges properties are all integers; (2) produce propagation simulation snapshots, which are the input to SimStore system, based on the configured propagation rules.

Effectiveness/efficiency measurements: We measure the storage effectiveness by the compression ratio comparing to persisting the graph vertices and edges directly. The retrieval efficiency is measured by the retrieval query latency.

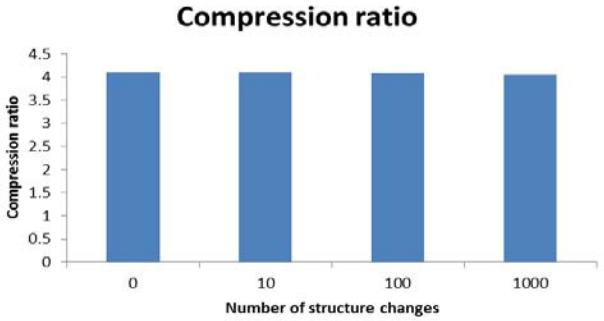


FIGURE IV. COMPRESSION RATIO EXPERIMENTAL RESULTS

B. Experimental Results

Storage effectiveness: As shown in Figure 4, SimStore could overall significantly reduce the storage cost by compression ratio of 4. We further evaluate the storage effectiveness by varying the number of graph structure changes. As shown in Figure 4, with the increase of structure change, the storage cost would not increase significantly.

Snapshot retrieval efficiency: We use randomly selected iteration id as point query to verify the snapshot retrieval efficiency. We also compare the query latency with baseline approach which stores each snapshot directly into key-value store. In our experimental result, we see that SimStore only introduces 20% of the additional query latency than the baseline approach.

V. RELATED WORK

Data compression: General data compression algorithms[7] were deeply investigated in the community and applied successfully[8]. All of these compression algorithms don't consider the characters of the data. In database management system area, e.g.,C-Store[9][10], HP Vertica[11], IBM DB2 BLU[12], a batch of compression techniques are employed to compress data based on data characteristics. These works are for column relational data management, but cannot used in graph data.

Graph data management: Graph data management is a hot topic in data management community. The major focus is to manage a very big graph, instead of managing a series of graphs.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented the design of a network propagation simulation data management system. Our major technical contributions include (1) designed a network simulation snapshot data management system; (2) proposed a novel algorithm to compress a series of snapshot and decompress it for query processing. Our experimental results demonstrate the efficiency of our techniques. In future work, we plan to extend our work to support distributed adapter, writer, compressor, and query processor. We would further explore series graph data compression for other simulation data management problems.

Acknowledgment

This project has been supported by the National Natural Science Foundation of China (No. 61370136).

References

[1] Yu Q. H., Zhang C. A Hybrid Algorithm for Relationship Recommendation in Social Networks. *Beijing 4th NCSC*. 132-137, 2012.

[2] Kumar R., Novak J., Tomkins A. Structure and evolution of online social networks[C]. *Proceedings of the 12th ACM SIGKDD. New York*: 611-617, 2006.

[3] Wu X. Y., Liu Z. H. How community structure influences epidemic spread in social network. *Physica A*. 387: 623, 2008.

[4] Small M., Tse C. K.. Clustering model for transimission of the SARS virus: application to epidemic control and risk assessment. *Physica A*. 351: 499, 2005.

[5] Lu T. Y. Early Experience in Analyzing the Human Flesh Search Model Based on Small World Model[C]. *Beijing, 4th NCSC*. 156-160, 2012.

[6] Liu Z. H., Hu B. Epidemic spreading in community networks[J]. *Euro phys. Lett*. 72: 315, 2005.

[7] Witten I, Neal R, Cleary J. Arithmetic coding for data compression[J]. *Communications of The ACM*, 30(6): 520-540, 1987.

[8] https://en.wikipedia.org/wiki/Gzip.

[9] Stonebraker M, Abadi D, Batkin A. C-store: a column-oriented DBMS[J]. *Very Large Data Bases*, 2005.

[10] Abadi D, Madden S, Ferreira M. Integrating compression and execution in column-oriented database systems[J]. *International Conference on Management of Data*, 2006.

[11] Lamb A, Fuller M, Varadarajan R. The vertica analytic database: C-store 7 years later[J]. *VLDB*, 5(12): 1790-1801, 2012.

[12] Raman V, Attaluri G, Barber R. DB2 with BLU acceleration: so much more than just a column store[J]. *Proceedings of The Vldb Endowment*, 6(11): 1080-1091, 2013.

[13] Angles R, Gutierrez C. Survey of graph database models[J]. *ACM Computing Surveys*, 2008.