Temporal Graph Cube

Guoren Wang, Yue Zeng, Rong-Hua Li, Hongchao Qin, Xuanhua Shi, *Senior Member, IEEE*, Yubin Xia, Xuequn Shang, and Liang Hong

***Abstract*—Data warehouse and OLAP (Online Analytical Processing) are effective tools for decision support on traditional relational data and static multidimensional network data. However, many real-world multidimensional networks are often modeled as temporal multidimensional networks, where the edges in the network are associated with temporal information. Such temporal multidimensional networks typically cannot be handled by traditional data warehouse and OLAP techniques. To fill this gap, we propose a novel data warehouse model, named Temporal Graph Cube, to support OLAP queries on temporal multidimensional networks. Through supporting OLAP queries in any time range, users can obtain summarized information of the network in the time range of interest, which cannot be derived by using traditional static graph OLAP techniques. We propose a segment-tree based indexing technique to speed up the OLAP queries, and also develop an index-updating technique to maintain the index when the temporal multidimensional network evolves over time. In addition, we also propose a novel concept called similarity of snapshots which shows a strong correlation with the efficiency of indexing technique and can provide a good reference on the necessity of building the index. The results of extensive experiments on two large real-world datasets demonstrate the effectiveness and efficiency of the proposed method.**

***Index Terms*—Data warehouse, OLAP, temporal multidimensional network, temporal graph cube, segment tree.**

I. Introduction

DATA warehouses and OLAP (Online Analytical Processing) techniques are helpful tools for knowledge workers (executive, manager, analyst) to analyze and make decisions, because Data warehouses and OLAP together can efficiently provide summarized information in different resolutions by specifying different views (different combinations of data dimensions) of large-scale real-world data through OLAP operations such as roll-up, drill-down and slice-and-dice [1]. Data warehouse built on traditional relational database (RDB) data is called data cube [2]. In 2011, Zhao et al. [3] proposed Graph Cube which extends data warehouses and OLAP techniques to analyze static multidimensional networks. For each query, static graph cube returns a static network with summarized information in its structure and statistical values on vertices and edges.

However, many real-life networks, such as human proximity networks, scientific collaboration networks, and biological networks, can be modeled as temporal networks [4], where each relationship or interaction has a timestamp. Also, vertices in temporal networks usually contain attributes of multiple dimensions. For example, vertices in human proximity networks have attributes such as name, gender, nationality, hobbies, and so on. Such an attributed network can be modeled as a temporal multidimensional network. There are many studies focusing on the management and analysis of temporal networks [5], but none of them have tried to extend OLAP techniques to temporal multidimensional networks, i.e., developing approaches to provide users in real time with summarized information on the temporal multidimensional networks in different resolutions and time ranges.

Example 1: Fig. 1 shows a sample temporal transaction network, presenting the transactions between individuals of different countries of birth and professions in each day. Table in Fig. 1(a) is the vertex table of the temporal network, showing the information of the 8 individuals. Each individual has a primary key ID and 3 dimensions (3 discrete attributes): gender, country (country of birth), profession. Income is a numeric attribute of individuals. Fig. 1(b) shows the network structure of the temporal network. There are 20 temporal edges, illustrating the transactions between individuals in 5 days. Temporal edges in each day can be organized into a snapshot. The numeric attribute on each temporal edge is the amount of each transaction. The static multidimensional information of individuals in Fig. 1(a) and the temporal information of network structure in Fig. 1(b) form a temporal multidimensional network.

In static graph cube [3], users can specify different views to obtain summarized information of the static multidimensional network and combine the information under these views through OLAP operations such as roll-up, drill-down and slice-and-dice interactively for decision support and business intelligence. For example, a company tries to use static graph cube to analyze the interaction characteristics of people with different attributes (name, gender, profession, income level, etc.) in a large-scale social network in order to support their marketing strategy

Manuscript received 15 June 2022; revised 31 January 2023; accepted 8 April 2023. Date of publication 26 April 2023; date of current version 8 November 2023. This work was supported in part by the National Key Research and Development Program of China under Grant 2020AAA0108500, in part by the Key R&D Program of Hubei under Grant 2020BAA020, in part by NSFC under Grants U2241211, 62072034, U1809206, and 72074172, and in part by CCF-Huawei Populus Grove Fund. Recommended for acceptance by Y. Tong. *(Corresponding author: Guoren Wang.)*

Guoren Wang, Yue Zeng, Rong-Hua Li, and Hongchao Qin are with the Beijing Institute of Technology, Beijing 100811, China (e-mail: wanggrbit@126.com; bruceez@163.com; lironghuabit@126.com; qhc.neu@gmail.com).

Xuanhua Shi is with the Huazhong University of Science and Technology, Wuhan, Hubei 430074, China (e-mail: xhshi@hust.edu.cn).

Yubin Xia is with the Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: xiayubin@sjtu.edu.cn).

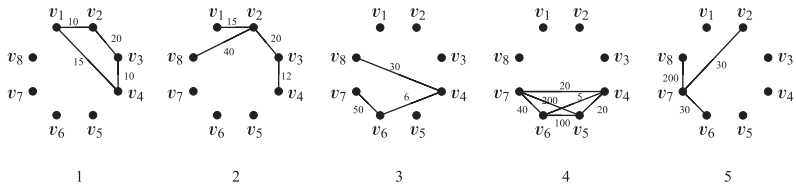
Xuequn Shang is with the Northwestern Polytechnical University, Xi'an, Shaanxi 710072, China (e-mail: shang@nwpu.edu.cn).

Liang Hong is with the Wuhan University, Wuhan, Hubei 430072, China (e-mail: hong@whu.edu.cn).

Digital Object Identifier 10.1109/TKDE.2023.3270460

ID	Gender	Country	Profession	Income
v_1	male	cn	teacher	5,000
v_2	female	us	shopowner	6,000
v_3	female	uk	lawyer	8,000
v_4	male	us	shopowner	10,000
v_5	male	jp	student	1,000
v_6	male	cn	student	1,000
v_7	female	uk	lawyer	6,000
v_8	female	jp	doctor	9,000

(a) Vertex table



(b) Network structure

Fig. 1. Running example: a temporal multidimensional network with its vertex table and network structure.

making. However, edges in real-world networks usually contain timestamps. For example, in social networks, interactions between people occur at specific times, such as a particular day. In this way, static graph cube does not work when users only need information for a specific time range. Considering the previous example, the company only needs data from the last few years, because data that is too old is less informative. Or, they want to study what the characteristics of people's interactions are during particular time ranges, such as holidays or COVID-19 periods. Or, they wish to compare several different time ranges to see how people's interaction patterns change over time. Static graph cube cannot meet the above requirements. As a result, we need to propose novel approaches to make data warehouses and OLAP capable of analyzing temporal multidimensional networks.

In this paper, we propose a new data warehouse model, called Temporal Graph Cube; and we extend the definition of OLAP queries by specifying a time range so that they can be applied to Temporal Graph Cube. Users can explore summarized information on different views of temporal multidimensional networks in any time range using Temporal Graph Cube. The challenge of our problem is: how to efficiently merge snapshots in a specific time range for each online OLAP query. A basic approach is to merge snapshots in the time range one by one. Such a basic method, however, is clearly inefficient when the time ranges are very large or a large number of OLAP queries come. The above problem can be seen as range query on snapshot arrays. Unfortunately, among the existing works for summarizing temporal networks [6], there is no work which focuses on summarizing snapshots within a certain time range online. In this paper, we investigate the problem of how to speed up merging snapshots in certain time ranges. Our solution is to build an index on snapshot arrays to reduce the query processing time. Specifically, we propose a segment-tree based index to support the range query on snapshot arrays. Since new edges are constantly inserted into the temporal multidimensional network, we also propose an index updating technique to handle such an edge-insertion case. In addition, similar to the static graph cube, the implementation of the Temporal Graph Cube also requires determining the materialization strategy of views in order to achieve a balance between time and space. To this end, we adopt a strategy called MinLevel proposed in [3] to handle the materialization problem in Temporal Graph Cube, as it fits our model best.

To summarize, the main contributions of this work are as follows:

1) We propose a new data warehouse model Temporal Graph Cube, which supports decision making on the basis of temporal multidimensional networks. The key difference compared to static graph cube is that Temporal Graph Cube supports querying summarized information for any time range of temporal multidimensional networks so that it supports more diverse analysis.
2) We extend the classic segment tree that tailored for traditional range query problems to our range query problems on snapshot arrays to reduce the OLAP query processing time. We also develop an index maintainable technique to handle the case when new edges are added to the temporal networks.
3) We propose a new metric, called similarity of snapshots, to measure the overall similarity of snapshots in a snapshot array, or, the degree to which the edges are shared by different snapshots. We show that this metric has a strong correlation with the effectiveness of the indexes and it can guide us to decide whether an index should be built.
4) We conduct extensive experiments on two large-scale real-world datasets. The results demonstrate the effectiveness and efficiency of the Temporal Graph Cube. The results also confirm the correlation between the efficiency (time and space) of the index and similarity of snapshots.

II. Temporal Graph Cube

Definition 1 (Temporal Multidimensional Network): Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, A, W)$ be an undirected temporal multidimensional network where $\mathcal{V}$ and $\mathcal{E}$ are the set of nodes and edges respectively. Edges in $\mathcal{E}$ are in the form of (u, v, t, a). u and v are nodes in $\mathcal{V}$. t and a are timestamp and numeric attribute respectively attached to edges. $A = \{A_1, A_2, \ldots, A_n\}$ represents the dimensions of the network or n discrete attributes of nodes in $\mathcal{V}$, i.e., $A(u) = \{A_1(u), A_2(u), \ldots, A_n(u)\}$ where $A_i(u)$ is the actual value of u on the ith attribute. $W(u)$ is the numeric value of u.

For convenience, we abbreviate temporal multidimensional network and static multidimensional network to temp-multi-network and static-multi-network respectively. Since we only consider undirected edges in this paper, if not specified, we assume without loss of generality $u \leq v$ for (u, v, t, a) in all following definitions. As shown in Fig. 1, $A = \{Gender, Country, Profession\}$ and $W = Income$. We do not consider ID to be an attribute of individuals because ID

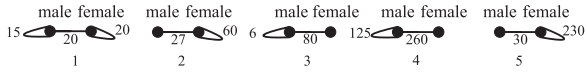


Fig. 2. Snapshots of the first temporal aggregate network in Example 2.

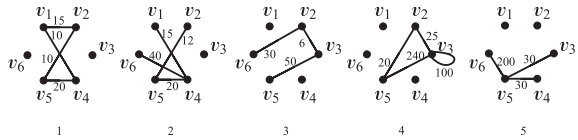


Fig. 3. Snapshots of the second temporal aggregate network in Example 2. v_1 = "male, teacher", v_2 = "male, shopowner", v_3 = "male, student", v_4 = "female, shopowner", v_5 = "female, lawyer", v_6 = "female, doctor".

is only the identification of individuals. $\mathcal{V}$ is the set of IDs of individuals in Fig. 1(a) and $\mathcal{E}$ is the set of temporal edges in Fig. 1(b). A temp-multi-network can also be represented by a sequence of snapshots.

Definition 2 (Snapshot): Let $\mathcal{T} = \{t|(u, v, t, a) \in \mathcal{E}\}$ be the set of timestamps. For each $t_i \in \mathcal{T}$, we can obtain a snapshot $S_i = \{(u, v, a)|(u, v, t_i, a) \in \mathcal{E}\}$.

For example, in Fig. 1(b), we can extract 5 snapshots; and the snapshot at timestamp 1 is $S_1 = \{(v_1, v_2, 10), (v_2, v_3, 20), (v_3, v_4, 10), (v_1, v_4, 15)\}$. With the definition of snapshot, we can define temp-multi-network in Definition 1 as $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$ where $\mathcal{S} = \{S_1, S_2, \ldots, S_{|\mathcal{T}|}\}$ and S_i is the snapshot at t_i.

Biased Timestamp: To store snapshots in a snapshot array and access snapshots directly by corresponding timestamps, we replace the original timestamps of snapshots with *biased timestamps*. For each temp-multi-network, suppose t_1 is the smallest timestamp, $t_{|\mathcal{T}|}$ is the biggest timestamp and $t_{base} = t_1 - 1$, biased timestamp of snapshot S_i is computed by $t_i - t_{base}$. Through biased timestamp we can map timestamps in $[t_1, t_{|\mathcal{T}|}]$ to $[1, t_{end}]$, where $t_{end} = t_{|\mathcal{T}|} - t_{base}$. The above strategy is useful when timestamps in temp-multi-network are dense. If timestamps are sparse, i.e., $|\mathcal{T}| \ll t_{|\mathcal{T}|} - t_1$, there will be many empty snapshots in snapshot array. We can sacrifice a little in efficiency, hashing all the timestamps into a denser space and relocating time range $[l, r]$ of each query (we will discuss queries in Section III) to the actual time range. However, in large scale real-world temp-multi-networks, temporal relations can be built at almost all time, so the timestamps are unlikely to be sparse, which is confirmed in our datasets. All algorithms in this paper are designed based on the definition of temp-multi-network $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$, where $\mathcal{S}$ is a snapshot array with biased timestamps of snapshots start from 1.

Definition 3 (Temporal Aggregate Network): Given a temporal multidimensional network $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$ and an aggregation $A' = (A'_1, A'_2, \ldots, A'_n)$ where A'_i equals A_i or $*$, the obtained temporal aggregate network is another temporal multidimensional network $\mathcal{G}' = (\mathcal{V}', \mathcal{S}', B, W_{\mathcal{G}'})$ where $\mathcal{V}' \subseteq \mathcal{V}$, $B = \{A'_i|A'_i \neq *\}$ and

1) Let $[v]$ be an equivalence class of v, where $v \in \mathcal{V}$ and $[v] = \{u|B(u) = B(v), u \in \mathcal{V}\}$. For each $v \in \mathcal{V}$, $\exists v' \in \mathcal{V}'$ satisfying $v' \in [v]$ and $\nexists u' \in \mathcal{V}', u' \neq v'$ satisfying $u' \in [v]$. $W_{\mathcal{G}'}(v')$ is the aggregation result of $W(v)$ for $v \in [v']$ obtained by aggregation function upon vertices specified by user.
2) $\forall u', v' \in \mathcal{V}'$ where $u' \leq v'$ and any $\mathcal{S}[i]$ for $i \in [1, t_{end}]$, if there exists a nonempty maximum edge set $E = \{(u, v, a)|(u, v, a) \in \mathcal{S}[i], (u \in [u'] \wedge v \in [v']) \vee (u \in [v'] \wedge v \in [u'])\}$, then $\exists (u', v', a') \in \mathcal{S}'[i]$ where a' is the aggregation result of numeric attributes of edges in E obtained by aggregation function upon edges specified by user.

In short, conducting aggregations on temp-multi-network contains two stages: aggregating, or group-by on the vertex table as 1) in Definition 3 and aggregating each snapshots in the temp-multi-network as 2) in Definition 3.

Example 2: Fig. 2 shows the snapshots of a temporal aggregate network, which is obtained by aggregating temp-multi-network in Fig. 1 on dimension "Gender". The obtained temporal aggregate network is also a temp-multi-network. There are 5 snapshots in Fig. 2, each of them is corresponding to the snapshot with the same timestamp in Fig. 1(b). For each snapshot in Fig. 2, each edge e' is the aggregated edge of a set of edges E in the corresponding snapshot. Each edge in E have two vertices aggregated to the two vertices of e', as 2) in Definition 3. In this example, the attribute in e' is the amount of the attributes of edges in E, i.e., the total transactions between individuals of two genders in each day. We can also choose other aggregation functions for edges like COUNT($*$), MAX($*$), and so on. Similarly, Fig. 3 shows another temporal aggregate network by aggregating temp-multi-network in Fig. 1 on dimension "Gender" and "Profession".

Obviously, the temporal aggregate network is not the direct answer of OLAP query on temp-multi-network, because temporal aggregate network contains aggregated snapshot at each timestamp, but OLAP query on temp-multi-network requires a summarized snapshot of a specified time range. However, temporal aggregate network can provide a cheaper way to conduct queries compared to conducting queries directly on the original temp-multi-network. We will explain in detail in Section III.

We devise a basic algorithm, as outlined in Algorithm 1, to construct the temporal aggregate network with aggregation A' from the original network $\mathcal{G}$. Note that there are two things to be concerned about aggregation functions:

- We assume f_v and f_e are not AVERAGE($*$), since the result of AVERAGE($*$) can be easily computed by SUM($*$)/COUNT($*$).
- If f_v = COUNT($*$), we assign 1 to all $W(u), u \in \mathcal{V}$, because 1 is the correct attribute value of each vertex when counting is needed. The same assignment should be done to numeric attributes of all edges if f_e = COUNT($*$).

Algorithm 1 first conducts the specified aggregation on all vertices of the original network. In line 1, we first create a hash structure, h, to maintain a mapping from all possible $B(u), u \in \mathcal{V}$ to vertices of the temporal aggregate network. In lines 2–6, we group all vertices in $\mathcal{V}$ with the specified aggregation A' and compute the aggregation result of numeric attributes

Algorithm 1: TemporalAggregateNetworkConstruction.

Input: A temp-multi-network $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$; Aggregation A'; Aggregation function f_v and f_e upon vertices and edges respectively
Output: Temporal aggregate network $\mathcal{G}' = (\mathcal{V}', \mathcal{S}', B, W_{\mathcal{G}'})$, $B = \{A'_i | A'_i \neq *\}$

1 $\mathcal{V}' \leftarrow \{\}$, let h be hash structure: $\{B(u) | u \in \mathcal{V}\} \rightarrow \mathcal{V}'$;
2 **foreach** $u \in \mathcal{V}$ **do**
3 **if** $h(B(u)) = NULL$ **then**
4 $\mathcal{V}' \leftarrow \mathcal{V}' \cup \{u\}$; $W_{\mathcal{G}'}(u) \leftarrow W(u)$; $h(B(u)) \leftarrow u$;
5 **else**
6 $u' \leftarrow h(B(u))$; $W_{\mathcal{G}'}(u') \leftarrow f_v(W_{\mathcal{G}'}(u'), W(u))$;
7 $\mathcal{S}' \leftarrow$ a snapshot array with $t_{end} + 1$ empty snapshots;
8 **foreach** $i \in [1, t_{end}]$ **do**
9 $EP \leftarrow \{\}$;
10 **foreach** $(u, v, a) \in \mathcal{S}[i]$ **do**
11 $u' \leftarrow h(B(u))$; $v' \leftarrow h(B(v))$;
12 **if** $u' > v'$ **then** $u', v' \leftarrow v', u'$;
13 **if** $(u', v') \notin EP$ **then**
14 $EP \leftarrow EP \cup \{(u', v')\}$; $\mathcal{S}'[i] \leftarrow \mathcal{S}'[i] \cup \{(u', v', a)\}$;
15 **else**
16 Let (u', v', a') be edge in $\mathcal{S}'[i]$ which has u', v' as vertices;
17 $\mathcal{S}'[i] \leftarrow \mathcal{S}'[i] - \{(u', v', a')\}$;
18 $\mathcal{S}'[i] \leftarrow \mathcal{S}'[i] \cup \{(u', v', f_e(a', a))\}$;
19 **return** $\mathcal{G}' = (\mathcal{V}', \mathcal{S}', B, W_{\mathcal{G}'})$;

of vertices with the specified aggregation function f_v. Each snapshot $\mathcal{S}[i]$ of $\mathcal{G}$ is aggregated into a smaller representation, i.e., the corresponding snapshot $\mathcal{S}'[i]$ at the same timestamp in $\mathcal{G}'$ (lines 8–18). For each edge (u, v, a), we first map the two vertices u, v to u', v' using h (line 11-12). If there exists an aggregated edge (u', v', a') in $\mathcal{S}'[i]$, we update a' with a using f_e (lines 16–18), otherwise we insert (u', v', a) into $\mathcal{S}'[i]$ directly (line 14).

It is easy to derive that the time complexity of Algorithm 1 is $O(|\mathcal{V}| + \sum_{i=1}^{t_{end}} |\mathcal{S}[i]|)$. The space used to maintain $h, W_{\mathcal{G}'}$ is $O(\mathcal{V}')$ and we need $O(|\mathcal{V}'| + \sum_{i=1}^{t_{end}} |\mathcal{S}'[i]|)$ space to maintain $\mathcal{G}'$. Moreover, we can easily derive that $|\mathcal{V}'| \leq |\mathcal{V}|$ and $|\mathcal{S}'[i]| \leq |\mathcal{S}[i]|$. As a result, the space complexity of Algorithm 1 is $O(|\mathcal{V}| + \sum_{i=1}^{t_{end}} |\mathcal{S}[i]|)$.

Definition 4 (Temporal Graph Cube): Given a temporal multidimensional network $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$, the temporal graph cube is obtained by decomposing A into all possible aggregations. Each aggregation A' is a node in the temporal graph cube and it corresponds to a temporal aggregate network $\mathcal{G}'$ as defined in Definition 3.

In [3], Zhao et al. used equivalently the terms *cuboid*, *view* and *aggregation*. In this paper, however, we only use *view* and *aggregation* equivalently, while *temporal cuboid* is used to refer to temporal cuboid query. In following sections, computing a view A' means materializing the corresponding temporal aggregate network $\mathcal{G}'$ in memory. If A' is precomputed, then for simplicity, A' also refers to the corresponding temporal aggregate network $\mathcal{G}'$.

For a view A' in the temporal graph cube, $dim(A')$ denotes the set of non-$*$ dimensions of A', i.e., $dim(A') = B$ where B is the discrete attributes of vertices in $\mathcal{G}'$, the corresponding temporal aggregate network of A'. For two views A' and A'', A' is an *ancestor* of A'' and A'' is a *descendant* of A' if $dim(A') \subset dim(A'')$, denoted as $A' \preceq A''$. Especially, the

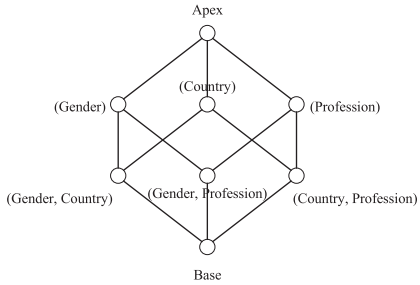


Fig. 4. A sample cube lattice.

base view A_{base} is a view where $dim(A_{base}) = A$, A is the dimensions of the original temp-multi-network. It is easy to see that A_{base} is a descendant of all other views. Another special view is $A_{apex} = (*, *, \ldots, *)$; and it is an ancestor of all other views in the temporal graph cube.

Example 3: Fig. 4 shows the temporal graph cube lattice built on the temp-multi-network in Fig. 1, each node on the lattice represents a view. For each edge in the lattice, the upper view is an ancestor of the lower view.

Temporal graph cube has the same form with static graph cube. The only difference is that each node in temporal graph cube is corresponding to a temporal aggregate network, not a static aggregate network. For each temporal aggregate network corresponding to a view, it contains more coarse-grained information than that of temporal aggregate network corresponding to a descendent view at each timestamp. In temporal graph cube, users can obtain summarized information in different resolution of the original temp-multi-network in a certain period by traversing the temporal graph cube lattice with specified time range. Users can also stay at some nodes in the lattice and query the summarized information in different time ranges. In these ways, summarized information of temp-multi-network in different resolutions and time ranges can be analyzed for decision support and business intelligence purposes.

III. Temporal OLAP Queries

Cuboid and crossboid are two important OLAP queries on static-multi-networks, or in static graph cubes [3]. In this section, we propose a generalized definition of OLAP queries in temporal graph cube, by extending cuboid and crossboid to temporal cuboid and temporal crossboid respectively.

A. Temporal Cuboid Query

The inputs of temporal cuboid query $Q = (A', [l, r])$ in temporal graph cube contain a specified view $A' = (A'_1, A'_2, \ldots, A'_n)$ and a time range $[l, r]$. The output is a static aggregate network obtained by the query.

Algorithm 2 is a basic algorithm to conduct temporal cuboid query on the original temp-multi-network (or on A_{base}). Same as Algorithm 1, Algorithm 2 first conducts the specified aggregation on the whole vertex table of the original network, even those vertices who do not exist in any edge of snapshots in $[l, r]$ (lines 1–6). In lines 7–15, we merge all snapshots in $[l, r]$ by

Algorithm 2: TemporalCuboidQuery.

Input: A temp-multi-network $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$; Aggregation A'; Time range $[l, r]$; Aggregation function f_v and f_e upon vertices and edges respectively
Output: A static aggregate network $G = (V, E, B, W_G)$, $B = \{A'_i | A'_i \neq *\}$

1 $V \leftarrow \{\}$, let h be hash structure: $\{B(u)|u \in \mathcal{V}\} \rightarrow V$;
2 **foreach** $u \in \mathcal{V}$ **do**
3 **if** $h(B(u)) = NULL$ **then**
4 $V \leftarrow V \cup \{u\}$; $W_G(u) \leftarrow W(u)$; $h(B(u)) \leftarrow u$;
5 **else**
6 $u' \leftarrow h(B(u))$; $W_G(u') \leftarrow f_v(W_G(u'), W(u))$;
7 $E' \leftarrow \{\}$; $EP \leftarrow \{\}$; $l \leftarrow l - t_{base}$; $r \leftarrow r - t_{base}$;
8 **foreach** $i \in [l, r]$ **do**
9 **foreach** $(u, v, a) \in \mathcal{S}[i]$ **do**
10 **if** $(u, v) \notin EP$ **then**
11 $EP \leftarrow EP \cup \{(u, v)\}$; $E' \leftarrow E' \cup \{(u, v, a))\}$;
12 **else**
13 Let (u, v, a') be edge in E' which has u, v as vertices;
14 $E' \leftarrow E' - \{(u, v, a')\}$;
15 $E' \leftarrow E' \cup \{(u, v, f_e(a', a))\}$;
16 $EP \leftarrow \{\}$; $E \leftarrow \{\}$;
17 **foreach** $(u, v, a) \in E'$ **do**
18 $u' \leftarrow h(B(u))$; $v' \leftarrow h(B(v))$;
19 **if** $u' > v'$ **then** $u', v' \leftarrow v', u'$;
20 **if** $(u', v') \notin EP$ **then**
21 $EP \leftarrow EP \cup \{(u', v')\}$; $E \leftarrow E \cup \{(u', v', a)\}$;
22 **else**
23 Let (u', v', a') be edge in E which has u', v' as vertices;
24 $E \leftarrow E - \{(u', v', a')\}$;
25 $E \leftarrow E \cup \{(u', v', f_e(a', a))\}$;
26 **return** $G = (V, E, B, W_G)$;

collecting all edges of those snapshots and summing the numeric attributes of edges sharing the same vertices using f_e. Then, in lines 17–25, we map the two vertices of all edges in E' using h and compute the aggregation result of edges sharing the same mapped vertices using f_e, which is similar to what Algorithm 1 does in each snapshot $\mathcal{S}[i]$ of $\mathcal{G}$ (lines 8–18 of Algorithm 1).

Before analyzing the time and space complexity of Algorithm 2, we first give a brief definition of summarized snapshot.

Definition 5 (Summarized Snapshot): Given a temporal multidimensional network $\mathcal{G} = (\mathcal{V}, \mathcal{S}, A, W)$, the summarized snapshot S of $\mathcal{G}$ is a special snapshot of $\mathcal{G}$ and it is the network structure in the result of a temporal cuboid query $Q = (A, [1 + t_{base}, t_{end} + t_{base}])$ on $\mathcal{G}$.

In other words, summarized snapshot S of $\mathcal{G}$ can be obtained by merging all snapshots in a snapshot array $\mathcal{S}$.

The time complexity of Algorithm 2 contains three parts: conducting aggregation on all vertices (lines 1-6), merging all snapshots in specified time range into E' (lines 7–15) and aggregating on E' (lines 17–25). Thus, the time complexity of Algorithm 2 is $O(|\mathcal{V}| + \sum_{i=l-t_{base}}^{r-t_{base}} |\mathcal{S}[i]| + |E'|)$. With the definition of summarized snapshot, we can rewrite the time complexity as $O(|\mathcal{V}| + \sum_{i=l-t_{base}}^{r-t_{base}} |\mathcal{S}[i]| + |S|)$, since we can easily find $|E'| \leq |S|$, where S is the summarized snapshot of $\mathcal{G}$. The memory consumed by V, h and W_G is linearly with respect to $|V|$ and $|V| \leq |\mathcal{V}|$. The size of E, E' and EP are all smaller than $|S|$, so the space complexity of Algorithm 2 is $O(|\mathcal{V}| + |S|)$.

Given a temporal cuboid query $Q = (A', [l, r])$, if we have precomputed the view A' with the same aggregation functions, we can derive the result of Q from $\mathcal{G}'$ by merging snapshots of $\mathcal{G}'$ in $[l, r]$ instead. Due to the space limit, all the proofs of this paper are omitted.

Theorem 1: A temporal cuboid query $Q = (A', [l, r])$ can be directly answered from A' if A' is precomputed with the same aggregation functions.

Theorem 1 gives another way to conduct Q. The static aggregate network as the result of Q contains two parts: vertex table and a snapshot presenting the network structure. When A' is precomputed, the vertex table of the resulting static aggregate network is the same with vertex table of $\mathcal{G}'$, the corresponding temporal aggregate network of A', and the network structure can be obtained by merging all snapshots of $\mathcal{G}'$ in $[l, r]$.

The algorithm of conducting Q on $\mathcal{G}'$ is exactly the subprocess in lines 7–15 of Algorithm 2 if we replace $\mathcal{G}$ with $\mathcal{G}'$. The time complexity of the simpler algorithm is $O(\sum_{i=l-t_{base}}^{r-t_{base}} |\mathcal{S}'[i]|)$. It is easy to see that $|\mathcal{S}'[i]| \leq |\mathcal{S}[i]|$, so this simpler algorithm is better than the baseline algorithm.

If the view A' is not precomputed while some other view A'' is precomputed with the same aggregation functions and $dim(A') \subset dim(A'')$, we can use the corresponding temporal aggregate network $\mathcal{G}''$ of A'' to get the result of Q. The algorithm is exactly Algorithm 2, where we only need to replace $\mathcal{G}$ with $\mathcal{G}''$.

Theorem 2. A temporal cuboid query $Q = (A', [l, r])$ can be answered from A'' where $dim(A') \subset dim(A'')$ and A'' is precomputed with the same aggregation functions.

The time complexity of conducting Q on $\mathcal{G}''$ is $O(|\mathcal{V}''| + \sum_{i=l-t_{base}}^{r-t_{base}} |\mathcal{S}''[i]| + |S''|)$, where $|\mathcal{V}''|$, $|\mathcal{S}''[i]|$ and $|S''|$ are vertex set, snapshot at i and summarized snapshot of $\mathcal{G}''$ respectively. We also have $|\mathcal{V}''| \leq |\mathcal{V}|$, $|\mathcal{S}''[i]| \leq |\mathcal{S}[i]|$ and $|S''| \leq |S|$; and in practice, $|\mathcal{V}''|$, $|\mathcal{S}''[i]|$ and $|S''|$ are much smaller than $|\mathcal{V}|$, $|\mathcal{S}[i]|$ and $|S|$ respectively.

If we have a set of precomputed views $\{A''_1, A''_2, \ldots\}$ with the same aggregation functions and for each A''_i, $dim(A') \subset dim(A''_i)$, which one should we choose in conducting Q using Algorithm 2? There is a similar problem in static graph cube, in which we only need to choose the view in $\{A''_1, A''_2, \ldots\}$ with the smallest size, since the time complexity of conducting a query $Q = (A')$ on the basis of A''_i in static graph cube equals the size of A''_i. It is easy to obtain the size of precomputed views of static graph cube in $O(1)$ time.

However, in temporal graph cube, extra time range should be specified in each query, so the time complexity of conducting a query $Q = (A', [l, r])$ on A''_i is not the size of A''_i. It is hard to compute the accurate time complexity of Algorithm 2 in $O(1)$ time. The reasons are as follows:

- If we want to compute the accumulation part $(\sum_{i=l-t_{base}}^{r-t_{base}} |\mathcal{S}[i]|)$ in the time complexity of Algorithm 2 in $O(1)$ time, an extra prefix array should be maintained.
- The size of summarized snapshot $|S|$ is just an upper bound of the number of edges visited in E' of Algorithm 2. To get the accurate value of $|E'|$ as fast as possible, we may have to build some indexes to get E' first. However, it is also difficult to get $|E'|$ in $O(1)$ time using indexes (we will discuss building indexes in Section IV).

As a result, in this paper we choose an arbitrary A''_i with least $|dim(A''_i)|$ in those precomputed views.

After discussing the temporal cuboid query, we can describe the OLAP operations in temporal graph cube, such as roll-up, drill-down and slice-and-dice. Suppose that we have conducted a temporal cuboid query $Q_0 = (A_0, [l, r])$. Roll-up means conducting another query $Q_1 = (A_1, [l, r])$ where A_1 is an ancestor of A_0, so that we get a coarser resolution of summarized information of temp-multi-network in the same time range. Drill-down is a contrary operation to obtain a finer summarized information in the same time range by conducting $Q_2 = (A_2, [l, r])$ where A_2 is a descendent of A_0. Slice-and-dice can be performed by selecting a subset of vertices in the vertex table of Q_0's result and generating the induced network. For example, $Q_0 = ((Profession), [2, 5])$ in temporal graph cube built on temp-multi-network in Fig. 1, we can choose to build an induced network based on the result of Q_0 with only "teacher" and "student" being selected to show the transactions between "teacher" and "student" from day 2 to 5.

We can also extend roll-up and drill-down according to the specified time ranges. A time related roll-up means conducting a query $Q_1 = (A_0, [l_1, r_1])$ where $l_1 \leq l$ and $r_1 \geq r$, so that we can get a coarser resolution of summarized information in time axis. On the contrary, time related drill-down means conducting a query $Q_2 = (A_0, [l_2, r_2])$ where $l_2 \geq l$ and $r_2 \leq r$.

B. Temporal Crossboid Query

The inputs of temporal crossboid query $Q_{cross} = (A'_1, A'_2, [l, r])$ in temporal graph cube contain two specified views $A'_1 = (A'_{11}, A'_{12}, \ldots, A'_{1n})$, $A'_2 = (A'_{21}, A'_{22}, \ldots, A'_{2n})$ and a time interval $[l, r]$, where $A'_1 \neq A'_2$. The output is a static aggregate bipartite network with two types of aggregated vertices corresponding to aggregations A'_1 and A'_2 respectively. We can conduct a temporal crossboid query directly on the original temp-multi-network. However, it is inefficient due to the large size of the original temp-multi-network. We omit the algorithm to conduct a temporal crossboid query on the original temp-multi-network, but give a more efficient way to conduct the query.

Definition 6 (Nearest Common Descendant): Given two different views A'_1 and A'_2, $cd(A'_1, A'_2)$ is common descendant of A'_1 and A'_2 and it is also a view in temporal graph cube satisfying $dim(A'_1) \cup dim(A'_2) \subseteq dim(cd(A'_1, A'_2))$. $ncd(A'_1, A'_2)$ is the nearest common descendant of A'_1 and A'_2 and it is one of the common descendants satisfying $dim(ncd(A'_1, A'_2)) = dim(A'_1) \cup dim(A'_2)$.

Theorem 3: Given a temporal crossboid query $Q_{cross} = (A'_1, A'_2, [l, r])$, Q_{cross} can be answered from the result of the temporal cuboid query $Q = (ncd(A'_1, A'_2), [l, r])$ where Q, Q_{cross} share the same aggregation function upon vertices and edges.

With Theorem 3, a temporal crossboid query can be turned into a temporal cuboid query, and thus can be solved by our previous techniques.

Algorithm 3 is an algorithm to compute the result of $Q_{cross} = (A'_1, A'_2, [l, r])$ from the result of $Q = (ncd(A'_1, A'_2), [l, r])$. Unlike Algorithm 2, we need aggregate each vertex on two different aggregations A'_1 and A'_2 (lines 3–11). For each edge (u, v, a) in G, we need to create or update two aggregated edges $(u', v'', *)$ and $(v', u'', *)$ (lines 12–27), because the two directions of (u, v, a) represent two interactions of aggregated vertices respectively (note that we do not assign $u' \leq v''$ or $v' \leq u''$ in E_B). The time complexity of Algorithm 3 is $O(|V| + |E|)$. We can easily derive that $|V_1| + |V_2| \leq 2|V|$ and $|E_B| \leq 2|E|$ in Algorithm 3, so the space complexity of Algorithm 3 is also $O(|V| + |E|)$.

Algorithm 3: TemporalCrossboidQuery.

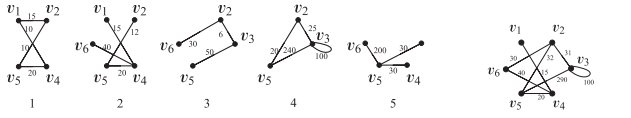
Input: $G = (V, E, B, W_G)$, result of $Q = (ncd(A'_1, A'_2), [l, r])$; A temporal crossboid query $Q_{cross} = (A'_1, A'_2, [l, r])$; Aggregation functions f_v and f_e
Output: A bipartite graph $G_B = (V_1, V_2, E_B, W_1, W_2, B_1, B_2)$

1 $B_1 \leftarrow dim(A'_1)$, $B_2 \leftarrow dim(A'_2)$;
2 Let h_1 be hash structure: $\{B_1(u) | u \in V\} \rightarrow V_1$, h_2 be hash structure: $\{B_2(u) | u \in V\} \rightarrow V_2$;
3 **foreach** $u \in V$ **do**
4 **if** $h_1(B_1(u)) = NULL$ **then**
5 $V_1 \leftarrow V_1 \cup \{u\}$; $W_1(u) \leftarrow W_G(u)$; $h_1(B_1(u)) \leftarrow u$;
6 **else**
7 $u' \leftarrow h_1(B_1(u))$; $W_1(u') \leftarrow f_v(W_1(u'), W(u))$;
8 **if** $h_2(B_2(u)) = NULL$ **then**
9 $V_2 \leftarrow V_2 \cup \{u\}$; $W_2(u) \leftarrow W_G(u)$; $h_2(B_2(u)) \leftarrow u$;
10 **else**
11 $u'' \leftarrow h_2(B_2(u))$; $W_2(u'') \leftarrow f_v(W_2(u''), W(u))$;
12 $EP \leftarrow \{\}$; $E_B \leftarrow \{\}$;
13 **foreach** $(u, v, a) \in E$ **do**
14 $u' \leftarrow h_1(u)$; $v'' \leftarrow h_2(v)$;
15 **if** $(u', v'') \notin EP$ **then**
16 $EP \leftarrow EP \cup \{(u', v'')\}$; $E_B \leftarrow E_B \cup \{(u', v'', a)\}$;
17 **else**
18 Let (u', v'', a') be an edge in E_B which has u', v'' as vertices;
19 $E_B \leftarrow E_B - \{(u', v'', a')\}$;
20 $E_B \leftarrow E_B \cup \{(u', v'', f_e(a', a))\}$;
21 $v' \leftarrow h_1(v)$; $u'' \leftarrow h_2(u)$;
22 **if** $(v', u'') \notin EP$ **then**
23 $EP \leftarrow EP \cup \{(v', u'')\}$; $E_B \leftarrow E_B \cup \{(v', u'', a)\}$;
24 **else**
25 Let (v', u'', a'') be an edge in E_B which has v', u'' as vertices;
26 $E_B \leftarrow E_B - \{(v', u'', a'')\}$;
27 $E_B \leftarrow E_B \cup \{(v', u'', f_e(a'', a))\}$;
28 **return** $G_B = (V_1, V_2, E_B, W_1, W_2, B_1, B_2)$;

IV. The Index-Based Approach

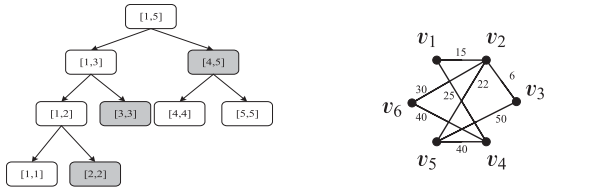
In this section, we propose an index-based approach to process the temporal cuboid query (temporal crossboid queries can be transformed to temporal cuboid queries). Recall that for a temporal cuboid query $Q = (A', [l, r])$, we need to merge snapshots of a certain temp-multi-network in time range $[l, r]$ whether A' is precomputed or not (see lines 8–15 of Algorithm 2). Merging snapshots in time ranges is similar to the classic range query problem: answering online queries $q = (func, [l, r])$ on a numeric array Arr, where $func$ might be SUM($*$), AVERAGE($*$), MAX($*$) or MIN($*$), specifying the needed statistical results of values in range $[l, r]$ of Arr. We can regard merging snapshots in snapshot array as a range query problem on a snapshot array.

Example 4. Fig. 5(a) is a snapshot array of temporal aggregate network of precomputed view $(Gender, *, Profession)$ in Fig. 3. Here we renumber all vertices and omit the isolated vertices in each snapshot for simplicity. Suppose that



(a) A snapshot array extracted from Fig. 3 (b) Result of range query Q in Example 4

Fig. 5. Range query on a snapshot array.



(a) The STree (b) Snapshot in the node $[1, 3]$

Fig. 6. A STree built on the snapshot array in Fig. 5.

a range query on the snapshot array in Fig. 5(a) is $Q = ((Gender, *, Profession), [2, 4])$ with SUM($*$) specified upon edges. We can derive that the result of Q as shown in Fig. 5(b).

A. The Segment Tree Index

Here we focus mainly on extending the classic segment tree (STree) structure to support range query on snapshot array when f_e is MAX($*$) or MIN($*$) and the maintenance of STree when adding new edges. For the other aggregation functions, including SUM($*$) and AVERAGE($*$), can be processed in a similar way. Below, we first briefly describe the basic properties of STree.

1) STree is a full binary tree, maintaining values for an array. If the length of the array is N, the height of STree is $\lceil \log(N) \rceil + 1$.
2) Each node in STree maintains statistical result of values in a sub-range of the array. The root of STree maintains the whole range of the array, i.e., $[1, N]$. The left child and the right child maintain range $[1, \lfloor (1 + N)/2 \rfloor]$ and range $[\lfloor (1 + N)/2 \rfloor + 1, N]$ respectively, which split $[1, N]$ equally. Both left child and right child can be regarded as roots of sub-STrees and they both have their own childs sharing their ranges equally. The above process for a node ends when its range can not be divided (length equals 1).
3) The time complexity of building a STree is $O(N)$. The space complexity of STree is also $O(N)$. The time complexity of processing a query $q = (func, [l, r])$ is $O(\log(N))$.

Unlike traditional range query problem, there are two differences in our range query problem: 1) elements in the array and statistical results in nodes of STree are not numeric values but snapshots, with edges in form of $(u, v, \max, \min)$; and 2) the range of STree might be expanded because of inserting new edges. Merging statistical information of nodes is a basic and frequent operation in both building and querying of STree, in which two statistical results of two nodes produce a new result. For the first difference, all we need to do is replacing the original operation on numeric values with MergeSnapshotExtre procedure (see Algorithm 4). We will address the second difference in the **Update** algorithm (see Algorithm 5). Below, we first briefly describe the index building procedure, followed by the query processing procedure and index updating procedure.

STree Building. We omit the detailed algorithm to build the STree for a snapshot array, because we only need to replace the merging operation of two numeric values in the traditional algorithm of building STree for numeric array with the proposed

Algorithm 4: STreeQuery(treeNode, timel, timer, ans).

Input: $treeNode$, node to be visited; Time range $[timel, timer]$; ans, a snapshot holding the current result

1 **if** $treeNode = NULL$ **then return**;
2 **if** $timel = treeNode.timel \wedge timer = treeNode.timer$ **then**
3 MergeSnapshotExtre($ans, treeNode.snapshot$);
4 **return**;
5 $mid \leftarrow (treeNode.timel + treeNode.timer)/2$;
6 **if** $timel > mid$ **then**
$STreeQuery(treeNode.rchild, timel, timer, ans)$;
7 **else if** $timer \leq mid$ **then**
$STreeQuery(treeNode.lchild, timel, timer, ans)$;
8 **else**
9 $STreeQuery(treeNode.lchild, timel, mid, ans)$;
10 $STreeQuery(treeNode.rchild, mid + 1, timer, ans)$;

11 **Procedure** MergeSnapshotExtre(S_a, S_b)
12 **foreach** $e_b \in S_b$ **do**
13 **if** $\exists e_a \in S_a, e_a.u = e_b.u \wedge e_a.v = e_b.v$ **then**
14 $e_a.max \leftarrow MAX(e_a.max, e_b.max)$;
15 $e_a.min \leftarrow MIN(e_a.min, e_b.min)$;
16 **else**
17 $S_a \leftarrow S_a \cup \{(e_b.u, e_b.v, e_b.max, e_b.min)\}$;
18 **return**;

MergeSnapshotExtre procedure (see Algorithm 4). For example, Fig. 6(a) illustrates a STree $STree$ for the snapshot array in Fig. 5. Note that each node of $STree$ maintains a snapshot. Fig. 6(b) shows the snapshot of the node $[1, 3]$ of $STree$ in Fig. 6(a).

As we have mentioned before, the time complexity for building a traditional STree is $O(N)$. Since MergeSnapshotExtre will be called in building each node, the time complexity of building STree for snapshot array of length N is $O(N|S|)$, where S is the summarized snapshot of the snapshot array. Also, it is easy to derive that the space usage of the STree building procedure is $O(N|S|)$.

Query Processing: Algorithm 4 conducts the range query on snapshot array using STree, and it shares the same framework with original query algorithm on STree but uses MergeSnapshotExtre to merge snapshots. Merging snapshots only happens when the current split range $[timel, timer]$ equals the range held by the current node $treeNode$ (line 2 in Algorithm 4). In the worst case, it happens at each depth of STree, so the time complexity of Algorithm 4 is $O(\log(N)|S|)$. For example, suppose we conduct a query $Q = (*, [2, 5])$ on snapshot array in Fig. 5(a) (the view in Q is omitted), we have to merge snapshots in nodes with range $[2, 2]$, $[3, 3]$ and $[4, 5]$ as in grey nodes in Fig. 6(a).

Index Updating. Let $STree.root.timer$ be the current largest timestamp in the STree. When a new edge (u, v, t, a) with $t >$

Algorithm 5: UpdateSTree.

Input: STree $STree$ built for $\mathcal{G}$; Mapped edge (u', v', a) of new edge $e = (u, v, t, a)$ used to update $\mathcal{G}$;
Output: Updated STree $STree$

1 **while** $STree.root.timer < t$ **do**
2 $newRoot \leftarrow$ a new empty STree node;
3 $newRoot.timel \leftarrow STree.root.timel$;
4 $newRoot.timer \leftarrow 2 \times STree.root.timer + 1 - STree.root.timel$;
5 $newRoot.lchild \leftarrow STree.root$;
6 $newRoot.snapshot \leftarrow STree.root.snapshot$;
7 $STree.root \leftarrow newRoot$;
8 STreeInsertEdge$(STree.root, (u', v', t, a))$;
9 **return** $STree$;
10 **Procedure** STreeInsertEdge$(treeNode, e' = (u', v', t, a))$
11 MergeSnapshotExtre$(treeNode.snapshot, \{(u', v', a, a)\})$;
12 **if** $treeNode.timel = treeNode.timer$ **then return**;
13 $mid \leftarrow \lfloor (treeNode.timel + treeNode.timer)/2 \rfloor$;
14 **if** $t \leq mid$ **then**
15 **if** $treeNode.lchild = NULL$ **then**
16 $newNode \leftarrow$ a new empty STree node;
17 $treeNode.lchild \leftarrow newNode$;
18 $newNode.timel \leftarrow treeNode.timel$;
19 $newNode.timer \leftarrow mid$;
20 STreeInsertEdge$(treeNode.lchild, e')$;
21 **else**
22 **if** $treeNode.rchild = NULL$ **then**
23 $newNode \leftarrow$ a new empty STree node;
24 $treeNode.rchild \leftarrow newNode$;
25 $newNode.timel \leftarrow mid + 1$;
26 $newNode.timer \leftarrow treeNode.timer$;
27 STreeInsertEdge$(treeNode.rchild, e')$;

$STree.root.timer$ comes, we need to expand the range of STree by creating new nodes to maintain larger range and switch the root of STree to one of the new nodes.

Suppose that $l = STree.root.timel, r = STree.root.timer$, if $t > r$, we first create a new node $newNode$ with range $[l, 2r - l + 1]$, which is twice the length of $[l, r]$, so that $STree.root$ can be a valid left child of $newNode$. Then, we let $STree.root$ be the left child of $newNode$. Finally, we let $newNode$ be the new root of $STree$. If $[l, 2r - l + 1]$ still can not cover t, we continue the process above until t is coverd. The updated $STree$ is not a fully binary tree, since there is only a left subtree for the new root which violates the basic properties of STree. However, it can be seen as a part of a standard STree, which might be completed by continuously added new edges.

Algorithm 5 is used to update STree. In line 1, the condition in the while loop is true which means that the current root of $STree$ can not cover t. So, we need to create a new root, which has a double size of range compared to the current root (line 3-4). In line 6, we directly copy the snapshot of the current root to the new root, since there is no possible right child for the new root before new edge is added. If the range of the candidate new root can not cover t (it hardly happens because of the density of timestamps in real-word large scale temporal network), then the loop in line 1 continues. When $STree.root.timer \geq t$ is true, we can use STreeInsertEdge procedure to update STree. In line 11, we update the snapshot of $treeNode$ with the mapped edge, since t is in $[treeNode.timel, treeNode.timer]$. In line 15 and line 22, we need to test whether the left child or the right child exists or not and create left child (lines 16-19) or right child (lines 23–26).

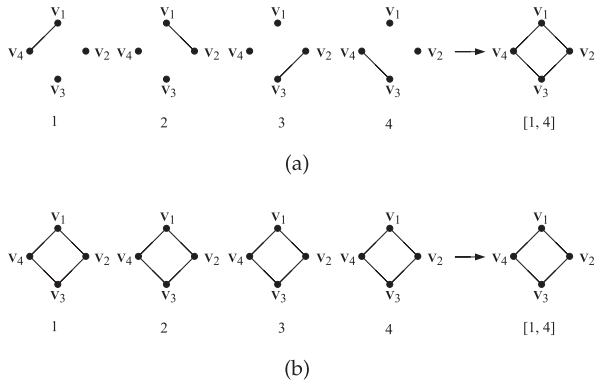


Fig. 7. Two cases of range query on snapshot array.

The time complexity of Algorithm 5 contains two parts: creating new root in lines 1-7 and inserting new edge using STreeInsertEdge procedure. Suppose the length of range maintained by the root of STree is $len_1 = STree.root.timer - STree.root.timel + 1$ before new edge is added, and the length of new range of timestamps in $\mathcal{G}$ becomes $len_2 = t - STree.root.timel + 1$ after $\mathcal{G}$ is updated. To cover len_2, we need repeat the loop in line 1 at least x times, and we have

$$len_1 * 2^x \geq len_2, \quad x \geq \log\left(\frac{len_2}{len_1}\right). \tag{1}$$

For each loop, the cost mainly comes from line 6. Suppose that the snapshot maintained in root is S_{root} before a new edge is added, the time complexity of creating the new root is $O(\log(\frac{len_2}{len_1}) \times |S_{root}|)$. In practice, $len_2 > len_1$ is hardly true because len_1, the length of range maintained by root, grows exponentially (lines 3-4) but len_2 grows linearly since timestamps in real-world temporal graph are dense. In STreeInsertEdge procedure, let the length of range in the final new root is len, the procedure go through the $\log(len) + 1$ layers of STree, in each layer it updates the snapshot of node covering t or creates necessary node first. Since updating a snapshot and creating a new empty node both contain a constant number of operations, the time complexity of STreeInsertEdge procedure is $O(\log(len) + 1)$.

V. Effectiveness of the Index

In traditional range query problems, indexes are helpful to accelerate the query processing. However, similar indexes cannot always work in our problem, since the elements in the array are not numeric values but snapshots.

Consider an example in Fig. 7(a), there is no single edge shared by any two snapshots (edge e is shared by snapshots means that there exists an edge that share the vertices with e in each of those snapshots). In this case, if we merge snapshots in $[t_1, t_4]$, the best choice is the baseline method, i.e., traversing all snapshots in the range and computing the result one by one. In baseline method, only 4 edges will be visited and each of them is visited only once. The edges in the result (snapshot at the right of the arrow in Fig. 7(a)) are exactly those 4 edges, thus

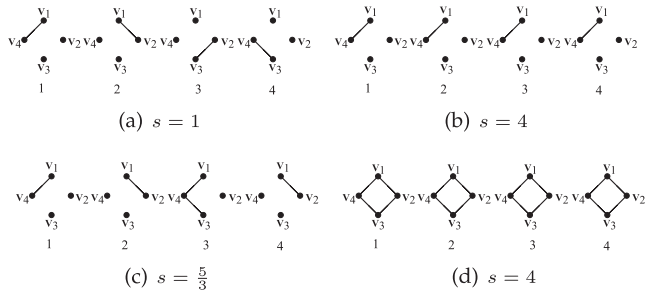


Fig. 8. Four snapshot arrays with different similarity of snapshots.

the baseline method achieves the highest efficiency and there is no need to build indexes in this case.

Consider the second example in Fig. 7(b), all snapshots share the same edges. In this case, if we still merge snapshots in $[t_1, t_4]$, each edge will be visited 4 times in the baseline method, therefore it is necessary to build index. Next, we describe the measurement of the necessity of index and the key factor we found which affects the necessity of index.

Definition 7 (Acceleration Ratio of Index): For an array Arr with numeric values or snapshots, we build a segment tree index on it. For a query $q = (func, [l, r])$, assume that the basic components are visited C_b times using the baseline method and C_i times using the index-based method, the acceleration ratio of the index on q is $\frac{C_b}{C_i}$.

The basic components in snapshot array and index built on snapshot array are edges. We use the number of visited edges as the measure of time consumption when conducting a query using the baseline method and using the index-based method in Definition 7.

Definition 8 (Similarity of Snapshots): Given a snapshot array $snapArr$ starting from 1 and containing n snapshots. Suppose the summarized snapshot of all snapshots in $snapArr$ is S and $E = \sum_{i=1}^{n} |snapArr[i]|$, the similarity of snapshots in $snapArr$ is $s = \frac{E}{|S|}$.

In other words, the similarity of snapshots is the ratio of the total number of edges in snapshot array to the size of the summarized snapshot. For simplicity, we assume that each snapshot in $snapArr$ has at least one edge. Clearly, for a snapshot array of length n, similarity of snapshots s is in $[1, n]$.

Example 5: Consider examples in Fig. 8. There are 4 snapshot arrays having the same length of 4 but different similarities of snapshots s. In Fig. 8(a), $s = 1$, and we can see that there is no edge shared by any two snapshots, or, snapshots in Fig. 8(a) are not similar to each other. In Fig. 8(b) and (d), $s = 4$, all snapshots share the same edges. In Fig. 8(c), $s = \frac{5}{3}$, and there are edges shared by some snapshots.

From the above examples, we can see that with the similarity of snapshots grows, there will be more snapshots in snapshot array which are similar to each other, or, more edges are shared by snapshots.

Theorem 4: For two arrays of length n and starting from 1, snapshot array $snapArr$ and numeric array $numArr$, if similarity of snapshots in $snapArr$ equals n, then for any query $q = (func, [l, r])$ on both $snapArr$ and $numArr$, the acceleration ratios of the same index built on them are equal.

Theorem 5: For a snapshot array $snapArr$ of length n and starting from 1. If similarity of snapshots in $snapArr$ equals 1, then for query $q = (func, [l, r])$ on $snapArr$, the acceleration ratio of STree is less than or equal to 1.

The above two theorems show the necessity of building index in two extreme cases. As in Theorem 4, if similarity of snapshots in snapshot array equals the length of the snapshot array, the index-based solution achieves the designed acceleration ratio as they perform in range query on numeric arrays. As in Theorem 5, if similarity of snapshots in snapshot array equals 1, STree has no acceleration effect. The baseline method is the best approach to conduct any query in this case, because the number of edges in the result is exactly the number of edges being visited in the baseline method.

It is difficult to model the relationship between similarity of snapshots and acceleration ratio of the index-based solution on all possible queries precisely, because similarity of snapshots describes the overall similarity of snapshots in the snapshot array, but in specific queries, snapshots in sub-arrays specified by ranges of the queries may not show the same similarity as the overall similarity. However, it is possible to model the relationship roughly. If the similarity of snapshots in snapshot array equals 1, there is no edge shared by two snapshots and visited more than once in baseline method. If the similarity of snapshots in snapshot array equals the length of the array, all edges are shared by all snapshots. In this case, each edge will be visited repeatedly in each cycle of baseline method (line 8 in Algorithm 2), while the index stores summarized snapshots of some sub-arrays of the snapshot array in advance to reduce the number of repeated visits on edges. With the similarity of snapshots grows from 1 to the length of snapshot array, more edges will be shared by snapshots, making the efficiency of baseline method lower and efficiency of indexes higher. As a result, the acceleration ratio of the index grows when similarity of snapshots grows, which will be confirmed in our experiments in Section VII-C.

The similarity of snapshots also affects the space consumption of the index. Since the elements in all snapshot arrays and the STree built on them are snapshots, we use the total number of edges in all snapshots to indicate the space consumption. For example, if we build two STrees $STree_a$ and $STree_b$ on both snapshot arrays in Fig. 8(a) and (b), the snapshot in each node of $STree_b$ has only one edge, while the snapshot in each node of $STree_a$ may have to store more than one edge, even if Fig. 8(a) and (b) have the same number of edges.

Definition 9 (Space Consumption Ratio): Consider an array Arr holding numeric values or snapshots. If Arr is a snapshot array, suppose the total number of edges in Arr is E_a and the total number of edges in the index is E_i. If Arr is a numeric array, E_a, E_i are the total number of numeric values in Arr and in the index respectively. The space consumption ratio of the index is $\frac{E_i}{E_a}$.

Theorem 6: For two arrays of length n and starting from 1, snapshot array $snapArr$ and numeric array $numArr$, if similarity of snapshots in $snapArr$ equals n, then the space

consumption ratios of the same index built on the two arrays are equal.

Similar to the acceleration ratio, the space consumption ratio of the index is also negatively affected by the decrease of similarity of snapshots.

Theorem 7: For a snapshot array $snapArr$ of length n and starting from 1. If similarity of snapshots in $snapArr$ equals 1, the space consumption ratio of STree built on $snapArr$ is in $(\lceil \log(n) \rceil, \lceil \log(n) \rceil + 1]$.

From Theorem 7, if similarity of snapshots equals 1, we need more space to store STree, which may be $\log(n) + 1$ times the space of the original snapshot array, while only 2 times the space is needed to store STree if similarity of snapshots equals the length of the original snapshot array.

Based on the above analysis, we can conclude that the necessity of building indexes, which can be determined by acceleration ratio and space consumption ratio of indexes, varies with the similarity of snapshots. This is the main difference of range query on a snapshot array compared to range query on a numeric array. Based on this observation, it is better to build index on the snapshot array with high similarity of snapshots. This is compatible with the implementation strategy of Temporal Graph Cube presented in the next section, which requires precomputing views with high similarity of snapshots in their snapshot arrays.

VI. Partial Materialization

To implement a temporal graph cube, we need precompute, or materialize some or all views in the temporal graph cube, since precomputed views can reduce the response time of OLAP queries and operations like roll-up, drill-down and slice-and-dice as we discussed in Section III.

In this work, we adopt a partial materialization strategy, i.e., we select a set of views to be precomputed in order to balance the space consumption and average response time according to the probability distribution of all possible queries. *View selection* in traditional data cube scenario is a NP-hard problem [7]. It is also a NP-hard problem in static graph cube scenario because traditional data cube can be regarded as a special case of static graph cube [3]. Similarly, static graph cube can also be regarded as a special case of temporal graph cube, if only one timestamp exists in the temporal graph cube. Thus, *view selection* problem in temporal graph cube is also a NP-hard problem. In traditional data cube and static graph cube, *view selection* problem is usually solved by heuristic sub-optimal solutions [3], [7], such as greedy algorithm and its variations [8]. Those heuristic solutions always measure the effectiveness of the selected views by benefit those views bring. Here the concept of *Benefit* was first introduced in [9], we briefly review the definition of *benefit* below.

Definition 10 (Benefit of Selected Views): For a sequence of selected views $u_1, u_2, \ldots, u_{k-1}$ in the order of being selected (u_1 is always A_{base}), the benefit of the candidate view u to be selected next is $B(u, S_{k-1})$, i.e., benefit brought by u w.r.t. $S_{k-1} = \{u_i | i \in [1, k-1]\}$. $B(u, S_{k-1})$ is defined as follows:

1) For each $w \preceq u$, define the quantity B_w by:
   a) Let v be the view of least cost in S_{k-1} and $w \preceq v$.
   b) If $C(u) < C(v)$, then $B_w = C(v) - C(u)$. Otherwise $B_w = 0$.
2) $B(u, S_{k-1}) = \sum_{w \preceq u} B_w$.

$C(*)$ returns the cost of views. In traditional data cube and static graph cube scenario, cost of views is the size of their corresponding precomputed tables or networks. The total benefit of selected view sequence is $\sum_{i=2}^{k} B(u_i, S_{i-1})$.

In greedy algorithm, among all candidate views, the view u with the largest $B(u, S_{k-1})$ is selected to be u_k. For the total benefit of view sequence generated by each heuristic solution, the closer to the benefit of the optimal view sequence, the more effective the solution is.

In Definition 10, the cost of views is used in representation of the benefit, because the cost of views is also the time complexity or cost of queries in data cube and static graph cube. However, cost of views in temporal graph cube (space consumption of the corresponding temporal aggregate networks of views) cannot be used as the cost of temporal cuboid queries. First, in Definition 10, $C(u)$ cannot be the cost of temporal cuboid query $Q = (w, [l, r])$ if we conduct Q on u. It is similar to $C(v)$. The reason is that there is a specified time range $[l, r]$ in Q, which does not exist in any query of traditional data cube or static graph cube. In implementation of temporal graph cube, we know nothing about the specified time ranges in all possible queries. Second, if we use an expected range $[l_e, r_e]$ as the representative of all ranges in all queries according to the probability distribution of specified ranges in all queries, we may be able to get the time complexity of conducting $Q = (w, [l_e, r_e])$ on u using the baseline method by precomputing the total size of snapshots in $[l_e, r_e]$ and the size of summarized snapshot of snapshots in $[l_e, r_e]$. Only in this way can we get a relatively accurate benefit for each u. Third, the above precomputing process is very costly, and it is based on conducting all queries with the baseline method.

As a result, the greedy algorithm and its variations are not suitable for view selection problem in temporal graph cube. In this paper, we adopt a much simpler solution to select views to be precomputed, MinLevel, which was originally introduced in [3]. The idea of MinLevel is simple: users are more willing to query with small number of dimensions, i.e., $|dim(A)|$ is small in $Q = (A, [l, r])$. In MinLevel, view A where $|dim(A)| = l_0$ is in the first batch of views to be selected (l_0 is an empirical value). If all the views with l_0 dimensions have been selected and the number of selected views or the total size needed for precomputing the selected views has not reach the limit, then we continue to select views with $l_0 + 1$ dimensions until we select enough number or size of views. In this paper, we select k views to be precomputed and k is also an empirical value.

VII. Experiments

A. Datasets

We conduct experiments on two real-world temp-multi-networks: DBLP (https://dblp.org/xml) and IMDB (https://www.imdb.com/interfaces), to evaluate the effectiveness and efficiency of the temporal graph cube respectively. Below we introduce the details of the two datasets.

TABLE I
BASIC INFORMATION OF DATASETS

Dataset	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	t_{end}	Time scale
DBLP	1,242,883	9,532,533	85	year
IMDB	3,497,300	39,682,231	151	year

TABLE II
DISCRETIZATION OF PUBLICATION NUMBER PER YEAR

Productivity	Publication number per year x
Excellent	$8 < x$
Good	$3 < x \leq 8$
Fair	$1 < x \leq 3$
Poor	$x \leq 1$

DBLP: The DBLP datasets covers 9 areas in computer science: CA (computer architecture), CN (computer network), NS (network security), SE (software engineering), DB (database), TH (computer science theory), CG (computer graphics), AI (artificial intelligence) and HI (human-computer interaction). Table I illustrates the statistic information of this dataset. For each vertex (author), there are three dimensions of information: Name, Area, Productivity. Since we classify all publications into 9 areas, Area of an author is one of the 9 areas where he or she has the most publications ever, which can be regarded as the representative of the focus of his or her papers. For Productivity, we discrete the number of publications per year of an author into four different buckets, which is shown in Table II. The number of publications per year of an author equals the total number of publications of the author divided by $2022 - y$, y is the first year when the author had his or her first publication. Productivity specifies the ability of producing academic papers of an author. Each temporal edge represents that two authors coauthored a paper at a certain time (year).

IMDB: The IMDB dataset contains various types of works like movie, short, video, etc. Each work has the following attributes: tile, time, titleType, genre, isAdult, rating, voteNum. rating is a numeric attribute, representing the average rating for the work from users. voteNum is the number of votes the work has received. There are some other files about principals of each work, which will be used as vertices of the temporal network. In summary, we extract several dimensions for each principal: Name, Pro, Field, Genre, isAdult, Averating, Hotness. Pro is the primary profession of a person, e.g., director, actor, producer, soundtrack, etc., which can be directly extracted from certain files. Field specifies the field that a person belongs to, e.g., movie, short, video, etc., which is obtained by counting the types of the works that the person participates in and choosing the type having the biggest count. Genre denotes the genre of works that a person usually participates in, which is obtained by the same process in obtaining Field. IsAdult represents if a person mainly works for adult works or not. Averating specifies the average rating of works that a person participates in, and it shows the average quality of works related to the person. Hotness denotes the hotness of works a person usually participates in, which is expressed by the average vote number of works related to the person. Both Averating and Hotness are discretized as shown in Table III.

TABLE III
ATTRIBUTED INFORMATION OF IMDB

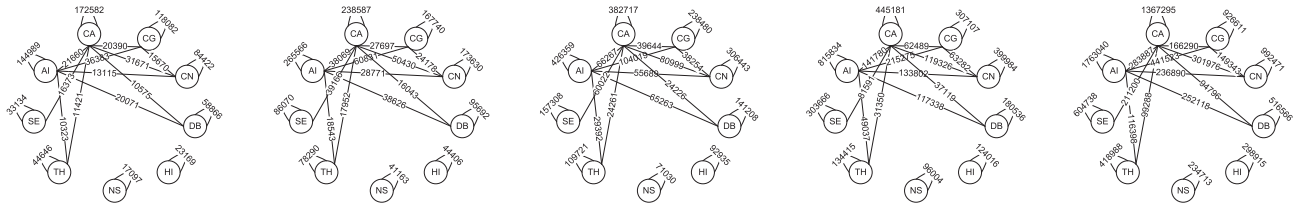
Averating	Averating value x	Hotness	Average votes of works w
5	$8 \leq x$	hot	$100,000 \leq w$
4	$6 \leq x < 8$	popular	$10,000 \leq w < 100,000$
3	$4 \leq x < 6$	fair	$1,000 \leq w < 10,000$
2	$2 \leq x < 4$	cold	$w < 1,000$
1	$x < 2$		

We refer to each person (each principal of the work) as a vertex in the temp-multi-network, and two vertices can form a temporal edge if they work together for the same work. The timestamp of the temporal edge is the release time of the work, and the numeric attribute of the temporal edge is the average rating of the work. The statistical information of temp-multi-network in IMDB is shown in Table I.

In the above datasets, temporal edges contain only single timestamp. There other types of temporal networks with temporal edges containing time range, which are not considered here. However, our method can be easily adapted for such datasets with similar performance. The key is to regard each temporal edge as a series of temporal edges containing only one timestamp which is in the original time range. We should also re-define aggregation function on temporal edges COUNT($*$) as asking for the total count of time units in connections between two vertices or two groups of vertices. For example, for a temporal edge (u, v, t_s, t_e), we can regard (u, v, t_s, t_e) as $(t_e - t_s)$ temporal edges with single timestamp $(u, v, t_s + 0.5), (u, v, t_s + 1.5), \ldots, (u, v, t_e - 0.5)$. For a query $[t_1, t_2]$ with COUNT($*$) satisfying $t_1 < t_s < t_2 < t_e$, the aggregation result of (u, v) should be $(t_2 - t_s)$. From another perspective, only $(u, v, t_s + 0.5), (u, v, t_s + 1.5), \ldots, (u, v, t_2 - 0.5)$ are in range $[t_1, t_2]$, so there are total $(t_2 - 0.5 - (t_s + 0.5) + 1) = (t_2 - t_s)$ time units in range $[t_1, t_2]$, which is consist with the previous result. In the practical we can let all timestamps be multiplied by 2. In this case, using query $[2t_1, 2t_2]$ with COUNT($*$) we can obtain the same result. Finally we can continue to use STree to accelerate the above query.

B. Effectiveness Evaluation

In this section, we evaluate the effectiveness of temporal graph cube as a decision-support tool on DBLP. First, we build temporal graph cube on DBLP and conduct a series of temporal cuboid queries $Q_1 = ((\text{Area}), [2001, 2006])$, $Q_2 = ((\text{Area}), [2006, 2011])$, $Q_3 = ((\text{Area}), [2011, 2016])$, $Q_4 = ((\text{Area}), [2016, 2021])$. We also build static graph cube on DBLP and conduct a cuboid query $Q_{static} = (\text{Area})$. Aggregation functions upon vertices and edges are both COUNT($*$). The network structures of Q_1-Q_4 and Q_{static} are shown in Fig. 9, illustrating the co-authorship patterns between authors grouped by different areas. For simplicity, in Fig. 9(a) we omit those co-authorships with COUNT($*$) values less than



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4 (e) Q_{static}

Fig. 9. Network structures in the results of queries $Q_1 - Q_4$ and Q_{static}. $Q_1 = ((\mathsf{Area}), [2001, 2006])$, $Q_2 = ((\mathsf{Area}), [2006, 2011])$, $Q_3 = ((\mathsf{Area}), [2011, 2016])$, $Q_4 = ((\mathsf{Area}), [2016, 2021])$ are temporal cuboid queries, $Q_{static} = (\mathsf{Area})$ is a cuboid query in static graph cube.

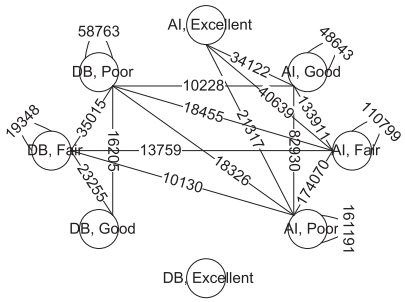


Fig. 10. Network structure in the result of $Q'_4 = ((\mathsf{Area}, \mathsf{Productivity}), [2016, 2021])$.

$10,000$, which are also omitted in Fig. 9(b), (c), (d), and (e), regardless of their COUNT$(*)$ values.

Some interesting observations can be obtained in these results. For example, authors in each area co-author most with authors in the same area all the time (Q_{static}) and in different time ranges ($Q_1 - Q_4$). We can clearly see that in recent 20 years ($Q_1 - Q_4$), the number of co-authorship between each pair of areas increases over time. However, such results can never be found in Q_{static}. More interestingly, in $[2001, 2006]$, researchers in TH co-authored most with researchers in CA, but with the time evolves, researchers in TH are more willing to co-author with researchers in AI. The reason may be that theoretical research on AI becomes more and more important with the wide applications of AI in recent years. However, in Q_{static} we can only know that researchers in TH co-authored most with researchers in AI, but ignore the trend above. In each time range of temporal cuboid queries we can zoom into a more fine-grained view by conducting a drill-down operation. For example, for $Q_4 = ((\mathsf{Area}), [2016, 2021])$, drill-down can be a query $Q'_4 = ((\mathsf{Area}, \mathsf{Productivity}), [2016, 2021])$. The network structure of Q'_4 is shown in Fig. 10. For simplicity, we drop edges with weight less than 10,000 and we only keep vertices related to areas in $\{\mathsf{AI}, \mathsf{DB}\}$ (it can be regarded as a slice-and-dice operation). In Fig. 9(d), there are 117,338 co-authorships between researchers of AI and DB in $[2016, 2021]$. In fine-grained view of Fig. 10, we can see that the most co-authorships belong to researchers of poor and fair productivity in AI and DB.

We then examine the effectiveness of temporal crossboid query.

Fig. 11 shows the results of a series of temporal crossboid queries:

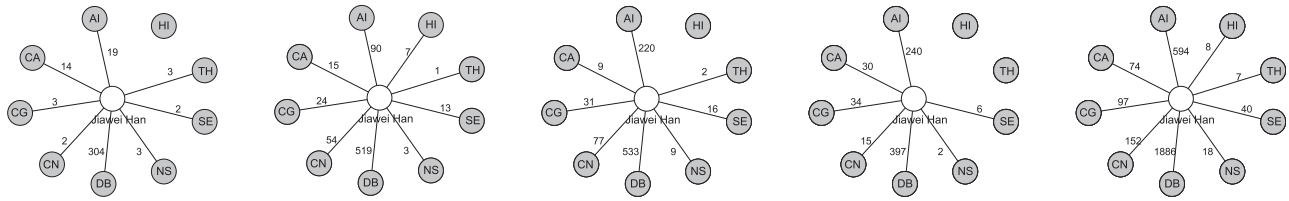
$$Q_{cross1} = ((\mathsf{Name}), (\mathsf{Area}), [2001, 2006]),$$
$$Q_{cross2} = ((\mathsf{Name}), (\mathsf{Area}), [2006, 2011]),$$
$$Q_{cross3} = ((\mathsf{Name}), (\mathsf{Area}), [2011, 2016]),$$
$$Q_{cross4} = ((\mathsf{Name}), (\mathsf{Area}), [2016, 2021]),$$

and a static crossboid query $Q_{cross-static} = ((\mathsf{Name}), (\mathsf{Area}))$. Above queries focus on the cross interaction between Name and Area, i.e., the co-authorship with each area for each researcher. We further slice-and-dice the results on Name to show the results related to "Jiawei Han". We can find that in all the queries, Jiawei cooperated with researchers in DB most. Meanwhile, in recent 10 years he has more and more cooperations with researchers in AI, which can be ignored compared to cooperations with researchers in DB in early 10 years. However, in static crossboid query $Q_{cross-static}$ (Fig. 11(e)) we can only find that Jiawei cooperated with researchers in DB most, but we ignore that the number of co-authorships between them is decreasing while Jiawei co-authored with researchers in AI more and more frequently.

C. Efficiency Evaluation

Exp-1: Performance of the Index-Based Method. Here we aim to evaluate how much the index-based solution can accelerate the merging operation of snapshots in time ranges while processing temporal cuboid queries, compared to the baseline algorithm (Algorithm 2). We will also compare other types of indexes with STree used in this paper later, such as prefix array and sparse table. As we analyzed before, the average acceleration ratio of the index-based solution can vary if the similarity of snapshots in temp-multi-networks changes. In this experiment, we mainly use IMDB dataset, since vertices in IMDB dataset have 7 dimensions and 2^7 views can be obtained. For convenience, we only materialize views in Table IV, among which we only materialize one view containing dimension Name, because Names of two persons are hardly the same in IMDB, and thus Name can be approximately regarded as a primary key and having only one view containing dimension Name is enough.

We conduct a series of temporal cuboid queries on materialized views to examine the efficiency of the index-based method (conducting temporal cuboid queries on materialized views are all about merging snapshots in time ranges as we



(a) Q_{cross1} (b) Q_{cross2} (c) Q_{cross3} (d) Q_{cross4} (e) $Q_{cross-static}$

Fig. 11. Network structures in the results of $Q_{cross1} - Q_{cross4}$ and $Q_{cross-static}$. $Q_{cross1} = ((\mathsf{Name}), (\mathsf{Area}), [2001, 2006])$, $Q_{cross2} = ((\mathsf{Name}), (\mathsf{Area}), [2006, 2011])$, $Q_{cross3} = ((\mathsf{Name}), (\mathsf{Area}), [2011, 2016])$, $Q_{cross4} = ((\mathsf{Name}), (\mathsf{Area}), [2016, 2021])$ are temporal crossboid queries, $Q_{cross-static} = ((\mathsf{Name}), (\mathsf{Area}))$ is a static crossboid query in static graph cube.

TABLE IV
MATERIALIZED VIEWS

No.	Materialized Views	Similarity of Snapshots
1	Pro	53.48
2	Pro, Field	18.16
3	Pro, Field, Genre	6.15
4	Pro, Field, Genre, isAdult	6.07
5	Pro, Field, Genre, isAdult, Averating	4.70
6	Pro, Field, Genre, isAdult, Averating, Hotness	3.97
7	Name, Pro, Field, Genre, isAdult, Averating, Hotness	1.12

TABLE V
TIME RANGES OF QUERIES

Length	Time ranges
4	[1878,1880],[1891,1895],[1906,1910], [1921,1925],[1936,1940],[1951,1955], [1966,1970],[1981,1985],[1996,2000],[2011,2015]
10	[1878,1883],[1888,1898],[1903,1913], [1918,1928],[1933,1943],[1948,1958], [1963,1973],[1978,1988],[1993,2003],[2008,2018]
20	[1878,1888],[1883,1903],[1898,1918], [1913,1933],[1928,1948],[1943,1963], [1958,1978],[1973,1993],[1988,2008],[2003,2023]
50	[1878,1903],[1878,1918],[1883,1933], [1898,1948],[1913,1963],[1928,1978], [1943,1993],[1958,2008],[1973,2023],[1988,2028]
100	[1878,1928],[1878,1943],[1878,1958], [1878,1973],[1888,1988],[1903,2003], [1918,2018],[1933,2028],[1948,2028],[1963,2028]

analyzed in Section III-A). We set a series of time ranges to be specified in the queries, which are shown in Table V. The earliest and the latest timestamps in IMDB are 1878 and 2028 respectively. We set 5 groups of time ranges of different length: 4, 10, 20, 50, 100. For each length, we set 10 ranges which are evenly distributed in $[1878, 2028]$ as shown in Table V. Ranges at two ends in each length group may be truncated since we try to keep each time range in $[1878, 2028]$. Specifically, we set a series of queries $QS_1, QS_2, \ldots, QS_7$, where QS_i is a set of queries specifying the No.i materialized view in Table IV and covering all time ranges of all lengths in Table V, i.e., $QS_1 = \{((Pro), [1878, 1880]), \ldots, ((Pro), [2011, 2015]), ((Pro), [1878, 1883]), \ldots, ((Pro), [2008, 2018]), \ldots, ((Pro), [1878, 1928]), \ldots, ((Pro), [1963, 2028])\}$.

TABLE VI
PERFORMANCE OF THE INDEX-BASED SOLUTION (K = 1,000, M = 1,000,000)

View No.	s	Baseline (sec.)	STree (sec.)	acce-ratio (ave)	View size	STree size	Space ratio
1	53.48	17	4	4.25	41K	94K	2.29
2	18.16	173	58	2.98	397K	1.1M	2.77
3	6.15	280	145	1.93	3.7M	13.8M	3.72
4	6.07	286	148	1.93	3.7M	13.9M	3.76
5	4.70	516	294	1.76	5.4M	22.1M	4.09
6	3.97	896	550	1.63	8.3M	35.8M	4.31
7	1.12	6,719	6,367	1.06	37.1M	285.6M	7.70

We set $f_e = \text{MAX}(*)$ and use the queries in $QS_1, \ldots, QS_7$ to test the efficiency of our index-based algorithm. The results are shown in Table VI. Similar results can also be observed for the other aggregation functions ($f_e = \text{COUNT}(*)$, $f_e = \text{SUM}(*)$, and $f_e = \text{MIN}(*)$). In Table VI, ViewNo. denotes the No. of materialized views in Table IV, and s is the similarity of snapshots. The third and the fourth columns are time used to process all the queries on each materialized view using the baseline method and the index-based algorithm respectively. For example, in the first line, processing all queries in QS_1 takes 17 s using the baseline method, but consumes 4 s using the segment-tree index based algorithm. The $\mathsf{acce-ratio(ave)}$ is the average acceleration ratio of the index-based algorithm on each query, compared to the baseline algorithm. In most cases, the index-based algorithm can accelerate merging snapshots in time ranges in conducting queries. However, the effect of acceleration varies with s. In the No.1 view, where s is the largest, we can see that the index-based solution achieves the best average acceleration ratio. We can observe that the average acceleration ratio of the index-based solution drops if s drops in general, which confirms our analysis in Section V.

Table VI also shows the space usage of the index-based solution (6th–8th columns). As can be seen, the size of the segment-tree based index is only several times larger than the original size of the views. For example, in the first line, the original view size is 41K, while our index-based solution takes only 94K space, with 2.27× more space usage. Also, we can see

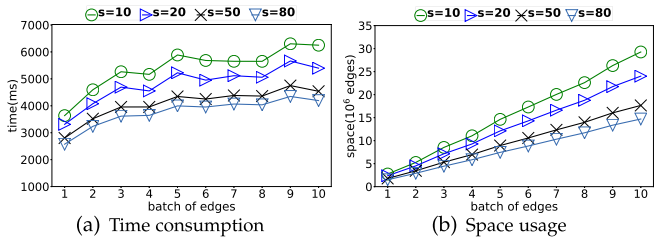
TABLE VII
PERFORMANCE OF PREFIX ARRAY AND SPARSE TABLE

View No.	s	acce-ratio preArr	Space ratio preArr	acce-ratio STable	Space ratio STable
1	53.48	5.37	1.51	12.92	9.38
2	18.16	3.20	2.58	7.87	14.45
3	6.15	1.76	5.76	5.15	26.78
4	6.07	1.74	5.84	5.15	27.04
5	4.70	1.56	7.14	4.84	31.74
6	3.97	1.45	8.33	4.58	35.85
7	1.12	0.85	27.38	OM	OM

that with s increasing, the space usage of the index-based solution decreases. These results demonstrate that our segment-tree based index structure is very space-efficient when s is relatively large. However, when s is small, e.g., near to 1, the space overhead of our index-based solution is high. Moreover, when s is near to 1, the average acceleration ratio is also near to 1 (i.e., the index-based solution is only slightly better than the baseline method). These results indicate that when s is very small, there is no need to build an index.

We also implement prefix array (preArr, only supports $f_e = \text{SUM}(*)$) and sparse table (STable, only supports $f_e = \text{MAX}(*)$ or $\text{MIN}(*)$) to conduct the same queries as a comparison. The results are shown in Table VII. We can see that the acceleration ratio of prefix array is close to our solution (STree) but with more space consumption in most cases. The performance of prefix array is also influenced by similarity of snapshots, as we analyzed in Section V. The acceleration ratio of sparse table is much higher than that of prefix array and STree, but consumes much more space. Building sparse table even leads to an out-of-memory on the No. 7 view. Similarly, the performance of sparse table is also influenced by similarity of snapshots. Overall, our index-based method, STree, is the best in these three methods considering both acceleration ratio and space consumption ratio.

Exp-2: Updating Performance of the Index-Based Solution. Here we evaluate the updating performance of our index-based solution. We divide all temporal edges in a materialized view into 10 batches equally according to the order of timestamps, e.g., the first batch of temporal edges are edges with the smallest timestamps, and they are added into the index one by one following the order of timestamps. When edges in a single batch are added into the index, we record both the time and space consumption of adding edges in the batch. The results are shown in Fig. 12. From Fig. 12(a), we can see that with the number of batch increasing, the time consumption of the index-updating algorithm increases. These results are consistent with our analysis in Section IV-A. Moreover, we can observe that for a large s, the time cost of our index-updating algorithm is low, while for a small s, the cost is often high. The reason could be that updating STree involves copying snapshot, i.e., snapshot in root, which maintains all snapshots in snapshot array, and thus gets larger if s gets smaller. As shown in Fig. 12(b), the space consumption of the index-updating algorithm is linear with respect to the number of batches, indicating that our algorithm



(a) Time consumption (b) Space usage

Fig. 12. The updating performance of the index-based solution.

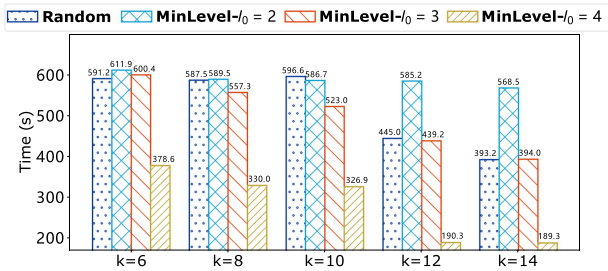


Fig. 13. Performance of strategies in partial materialization. k is the number of views to be materialized.

is space-efficient. Likewise, we can observe that the algorithm takes more space for a smaller s. These results further confirm our analysis in Section IV-A.

Exp-3: Performance of MinLevel *in Partial Materialization.* In this experiment we also use IMDB. The greedy strategy introduced in Section VI is not suitable for partial materialization in temporal graph cube due to the uncertain time range specified in queries, which we have analyzed in Section VI. Instead, we choose a random strategy, i.e., selecting k views to be materialized randomly, as a comparison to MinLevel. We use total response time of 40 temporal cuboid queries to evaluate the performance. In these queries, the number of queries in the format of $((A_1, \ldots, A_x), [l, r])$ is almost twice as many as the number of queries in the format of $((A_1, \ldots, A_{x+1}), [l, r])$, because users are more willing to query with less dimensions. The results are shown in Fig. 13. We can see that as k increases, all of the 4 strategies achieve better performance because more views are materialized. With proper setting of l_0, e.g., $l_0 = 4$, MinLevel achieves the best performance since most of the queries specified by users have no more dimensions than 4, and these queries are more possible to be accelerated compared to the case when Random strategy is used.

VIII. RELATED WORK

Our work extends data warehouse and OLAP techniques to temporal multidimensional networks. There are many works applying data warehouse and OLAP to other data types, such as stream data [10], spatio-temporal data [11], sequence data [12], [13], and text data [14]. Except the traditional relational data, Dehdouh et al. [15] tried to compute data cubes from column-oriented NoSQL data. As for graph data, except [3], some other works focus on heterogeneous networks [16], [17]. [18], [19]

improve efficiency of graph OLAP query by parallelization and distribution. However, none of them focus on temporal multidimensional networks.

Our work is also related to graph summarization and analysis. For static graphs, there are some representative works, including graph compression with MDL (Minimum Description Length) principle [20], attributed graph compression [21] and graph clustering based on vertices partitioning [22]. Recently, there exist several studies on static graph summarization, such as summarizing directed acyclic graph (DAG) [23], summarizing multi-relation graph [24] where multiple edges of different types may exist between any pair of vertices, and DPGS model [25], which can preserve properties like graph spectrum and the authorities and hubnesses of vertices while reconstructing graph. However, all of them ignore the temporal information of big graph data. For temporal graph and dynamic graph, TimeCrunch [26] summarizes a large dynamic graph with a set of important temporal structures using MDL, such as ranged full clique, periodic bipartite core, oneshot star, etc. Some works ([27], [28], [29], [30], [31], [32]) summarize graph stream into one sketche. [33], [34] use a sliding window of fixed length to only summarize the latest snapshots, and [35] only summarize the current snapshot of a graph stream incrementally and losslessly. However, in this paper, we can specify arbitrary time window to query summarized result in real time. Chen et al. [36] also study summarizing snapshots in arbitrary time window, but they ignored the attributes of vertices and different views determined by combinations of those attributes. We summarize graphs based on selected attributes of vertices, so we can get summarized information between vertices in different resolutions.

IX. Conclusion

In this article, we extend data warehouse and OLAP technology to temporal multidimensional networks by proposing a new data warehouse model Temporal Graph Cube, which provides knowledge workers with tools to analyze temporal multidimensional networks. We first extend the basic concepts in static graph cube and introduce two new queries, temporal cuboid and temporal crossboid, allowing Temporal Graph Cube to support OLAP queries on temporal multidimensional networks. Then, we propose a segment-tree based index method to accelerate the OLAP queries. We also present a new metric to measure the efficiency of the index. We conduct extensive experiments on two large real-world datasets, and the results demonstrate the effectiveness and efficiency of the proposed solutions.

References

[1] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod Rec.*, vol. 26, no. 1, pp. 65–74, 1997.

[2] J. Gray et al., "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining Knowl. Discov.*, vol. 1, no. 1, pp. 29–53, 1997.

[3] P. Zhao, X. Li, D. Xin, and J. Han, "Graph cube: On warehousing and OLAP multidimensional networks," in *Proc. Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 853–864.

[4] P. Holme, "Modern temporal network theory: A colloquium," *Eur. Phys. J. B*, vol. 88, no. 9, pp. 1–30, 2015.

[5] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm, "Management and analysis of big graph data: Current systems and open challenges," in *Handbook of Big Data Technologies*, Springer, 2017, pp. 457–505.

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

[7] H. Karloff and M. Mihail, "On the complexity of the view-selection problem," in *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, 1999, pp. 167–173.

[8] K. Morfonios, S. Konakas, Y. Ioannidis, and N. Kotsis, "Rolap implementations of the data cube," *ACM Comput. Surv.*, vol. 39, no. 4, pp. 12–es, 2007.

[9] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," *Acm Sigmod Rec.*, vol. 25, no. 2, pp. 205–216, 1996.

[10] J. Han et al., "Stream cube: An architecture for multi-dimensional analysis of data streams," *Distrib. Parallel Databases*, vol. 18, no. 2, pp. 173–197, 2005.

[11] L. Gómez, B. Kuijpers, B. Moelans, and A. Vaisman, "A survey of spatio-temporal data warehousing," *Int. J. Data Warehousing Mining*, vol. 5, no. 3, pp. 28–55, 2009.

[12] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung, "Olap on sequence data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 649–660.

[13] M. Liu et al., "E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 889–900.

[14] C. X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao, "Text cube: Computing ir measures for multidimensional text database analysis," in *Proc. IEEE 8th Int. Conf. Data Mining*, 2008, pp. 905–910.

[15] K. Dehdouh, F. Bentayeb, O. Boussaid, and N. Kabachi, "Columnar noSQL cube: Agregation operator for columnar noSQL data warehouse," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, 2014, pp. 3828–3833.

[16] M. Yin, B. Wu, and Z. Zeng, "Hmgraph olap: A novel framework for multi-dimensional heterogeneous network analysis," in *Proc. 15th Int. Workshop Data Warehousing OLAP*, 2012, pp. 137–144.

[17] P. Wang, B. Wu, and B. Wang, "Tsmh graph cube: A novel framework for large scale multi-dimensional network analysis," in *Proc. IEEE Int. Conf. Data Sci. Adv. Anal.*, 2015, pp. 1–10.

[18] Z. Wang, Q. Fan, H. Wang, K.-L. Tan, D. Agrawal, and A. El Abbadi, "Pagrol: Parallel graph OLAP over large-scale attributed graphs," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 496–507.

[19] H. Chen et al., "High performance distributed OLAP on property graphs with grasper," in *Proc. Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2705–2708.

[20] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 419–432.

[21] Y. Wu, Z. Zhong, W. Xiong, and N. Jing, "Graph summarization for attributed graphs," in *Proc. Int. Conf. Inf. Sci. Electron. Elect. Eng.*, 2014, pp. 503–507.

[22] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 718–729, 2009.

[23] X. Zhu, X. Huang, B. Choi, and J. Xu, "Top-k graph summarization on hierarchical DAGs," in *Proc. 29th ACM Int. Conf. Inf. Knowl. Manage.*, 2020, pp. 1903–1912.

[24] X. Ke, A. Khan, and F. Bonchi, "Multi-relation graph summarization," *ACM Trans. Knowl. Discov. From Data*, vol. 16, no. 5, pp. 1–30, 2022.

[25] H. Zhou, S. Liu, K. Lee, K. Shin, H. Shen, and X. Cheng, "DPGS: Degree-preserving graph summarization," in *Proc. SIAM Int. Conf. Data Mining*, 2021, pp. 280–288.

[26] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos, "Timecrunch: Interpretable dynamic graph summarization," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2015, pp. 1055–1064.

[27] O. Mudannayake and N. Ranasinghe, "kMatrix: A space efficient streaming graph summarization technique," in *Proc. IEEE 10th Int. Conf. Inf. Automat. Sustainability*, 2021, pp. 161–166.

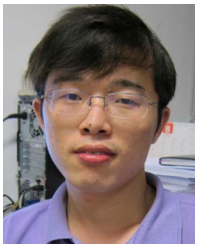
[28] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1481–1496.

[29] Z. Ma, J. Yang, K. Li, Y. Liu, X. Zhou, and Y. Hu, "A parameter-free approach for lossless streaming graph summarization," in *Proc. Database Syst. Adv. Appl.: 26th Int. Conf.*, Taipei, Taiwan, Apr. 11–14, 2021, pp. 385–393.

[30] N. Ashrafi-Payaman, M. R. Kangavari, S. Hosseini, and A. M. Fander, "GS4: Graph stream summarization based on both the structure and semantics," *J. Supercomput.*, vol. 77, pp. 2713–2733, 2021.
[31] X. Gou, L. Zou, C. Zhao, and T. Yang, "Fast and accurate graph stream summarization," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 1118–1129.
[32] M. Chen, R. Zhou, H. Chen, and H. Jin, "Scube: Efficient summarization for skewed graph streams," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 100–110.
[33] S. Fernandes, H. Fanaee-T, and J. Gama, "Dynamic graph summarization: A tensor decomposition approach," *Data Mining Knowl. Discov.*, vol. 32, no. 5, pp. 1397–1420, 2018.
[34] I. Tsalouchidou, F. Bonchi, G. D. F. Morales, and R. Baeza-Yates, "Scalable dynamic graph summarization," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 2, pp. 360–373, Feb. 2020.
[35] J. Ko, Y. Kook, and K. Shin, "Incremental lossless graph summarization," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 317–327.
[36] M. Chen, R. Zhou, H. Chen, J. Xiao, H. Jin, and B. Li, "Horae: A graph stream summarization structure for efficient temporal range query," in *Proc. IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 2792–2804.

Guoren Wang received the BS, MS, and PhD degrees from the Department of Computer Science, Northeastern University, China, in 1988, 1991, and 1996, respectively. Currently, he is a professor with the Beijing Institute of Technology (BIT), Beijing, China. His research interests include graph data management, graph mining, and graph computational systems.

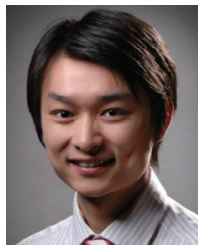
Yue Zeng is currently working toward the masters degree with the Beijing Institute of Technology (BIT), Beijing, China. His research interests include graph data management and social network analysis.



Rong-Hua Li received the PhD degree from the Chinese University of Hong Kong, in 2013. He is currently a professor with the Beijing Institute of Technology (BIT), Beijing, China. His research interests include graph data management and mining, social network analysis, graph computational systems, and graph-based machine learning.

Hongchao Qin received the BS degree in mathematics, ME and PhD degrees in computer science from Northeastern University, China, in 2013, 2015 and 2020, respectively. He is currently a postdoc with the Beijing Institute of Technology, China. His current research interests include social network analysis and data-driven graph mining.

Xuanhua Shi (Senior Member, IEEE) received the PhD degree in computer engineering from HUST, China, in 2005. He is a professor in Service Computing Technology and System Lab and Cluster and Grid Computing Lab, HUST (China). His current research interests focus on the scalability, resilience and autonomy of large-scale distributed systems, such as peta-scale systems, and data centers.



Yubin Xia (Affiliate, IEEE) received the deploma degree in software school, Fudan University, Shanghai, China, in 2004, and the PhD degree in computer science and technology from Peking University, Beijing, China, in 2010. He is now an associate professor in Shanghai Jiao Tong University since Sep. 2012. His research interests include computer architecture, operating system, and security.

Xuequn Shang received the PhD degree in computer science from the University of Magdeburg, Germany, in 2005. She is currently a professor in School of Computer Science, Northwestern Polytechnical University, China. Her research interests include data mining, bioinformatics and machine learning.

Liang Hong received the BS and PhD degrees in computer science from the Huazhong University of Science and Technology (HUST), in 2003 and 2009, respectively. Now, he is an associate professor in School of Information Management of Wuhan University. His research interests include graph database, spatio-temporal data management and social networks.