Pagrol: PArallel GRaph OLap over Large-scale Attributed Graphs

Zhengkui Wang[§‡1], Qi Fan[§2], Huiju Wang[‡3], Kian-Lee Tan[§‡4], Divyakant Agrawal[†5], Amr El Abbadi [†6]

[§] NUS Graduate School of Integrative Science and Engineering, National University of Singapore, Singapore
{[1] wangzhengkui,[2] fan.qi}@nus.edu.sg

[‡] School of Computing, National University of Singapore, Singapore
{[3] wanghj,[4] tankl}@comp.nus.edu.sg

[†] Department of Computer Science, University of California at Santa Barbara, CA, USA
{[5] agrawal,[6] amr}@cs.ucsb.edu

***Abstract*—Attributed graphs are becoming important tools for modeling information networks, such as the Web and various social networks (e.g. Facebook, LinkedIn, Twitter). However, it is computationally challenging to manage and analyze attributed graphs to support effective decision making. In this paper, we propose, *Pagrol*, a parallel graph OLAP (Online Analytical Processing) system over attributed graphs. In particular, *Pagrol* introduces a new conceptual *Hyper Graph Cube* model (which is an attributed-graph analogue of the data cube model for relational DBMS) to aggregate attributed graphs at different granularities and levels. The proposed model supports different queries as well as a new set of graph OLAP Roll-Up/Drill-Down operations. Furthermore, on the basis of *Hyper Graph Cube*, Pagrol provides an efficient MapReduce-based parallel graph cubing algorithm, MRGraph-Cubing, to compute the graph cube for an attributed graph. *Pagrol* employs numerous optimization techniques : (a) a self-contained join strategy to minimize I/O cost; (b) a scheme that groups cuboids into batches so as to minimize redundant computations; (c) a cost-based scheme to allocate the batches into bags (each with a small number of batches); and (d) an efficient scheme to process a bag using a single MapReduce job. Results of extensive experimental studies using both real Facebook and synthetic datasets on a 128-node cluster show that *Pagrol* is effective, efficient and scalable.**

I. Introduction

The expressive power of *attributed graphs* makes them attractive in modeling a variety of information networks, such as the Web, sensor networks and social networks (e.g. Facebook, LinkedIn, Twitter) [1]. Attributed graphs model these information networks as follows: each individual object with its associated information is represented as a vertex with vertex attributes, and the relationships between two objects are captured as edges between two vertices with associated edge attributes. By analyzing the attributed graph of an information network, we may acquire accurate and more explicit insight of the real world, predict pattern evolution and make better decisions.

Example 1: Consider a social network which typically contains a wealth of individual user information (like profiles) and relationship information (like the connection between different users). Fig. 1 provides a simple attributed graph that models such user information. Fig. 1 (a) presents the underlying graph structure involving 9 vertices each of which represents one individual user with a user ID, and 17 edges each of which indicates one relationship between two users. Fig. 1 (b) shows a vertex attribute table describing each individual's profile information including Gender, Nation and Profession. Fig. 1 (c) shows an edge attribute table describing the relationship information between different individuals including the date they connected, their relationship types and strength, where sV and tV are the two vertex IDs of each edge. ■

While much research has been devoted to different traditional graph analysis (e.g. shortest path, centrality, pattern matching), attributed graphs offer additional opportunities to support a wider variety of OLAP (Online Analytical Processing) queries for information discovery and decision making. We identify several categories of queries that users may be interested in:

Category 1: Discovering knowledge over the vertex or edge attributes. This category involves queries that can be answered from either the vertex or edge attributes. Referring to our running example in Fig. 1, sample queries are "*What is the percentage of users among different professions in this network?*" or "*How many relationships appeared in 2012?*"

Category 2: Integrating knowledge over both the vertex and edge attributes. Queries in this category require integrating information from both the vertex and edge attributes. An example query is "*What is the trend of the number of relations appearing between USA and SG (Singapore) in last 3 years?*"

Category 3: Investigating an aggregated graph over multi-dimensional spaces. In this category of queries, users may be interested in viewing a coarse-grained summarized/aggregate graph along some dimensions, such as "*What is the graph structure as grouped by users' gender as well as relationship type?*" Different to category 1 and 2, the expected answer to such a query is an aggregate graph as shown Fig. 2 (a). Such queries are useful as the base underlying graphs are too massive to be observed, while the summarized graphs offer greater ease in information discovering.

Now, for large attributed graphs, it is computationally expensive to evaluate these queries from the base graphs. As such, it is critical to develop better OLAP query and decision making support over attributed graphs. In this paper, we adopt a two-pronged approach to address this challenge. First, we observe that traditional data cubes have been successfully

deployed to speed up OLAP query processing in RDBMS [2]. However, traditional data cube model is not applicable to graphs as it does not capture the graph structures. Thus, we need to design a conceptual graph cube model that supports the queries in all the three categories. Second, for large attributed graphs, parallelism is an effective and promising approach to ensure acceptable response time. As such, we seek to develop scalable parallel solutions for evaluating queries represented under our graph cube model.

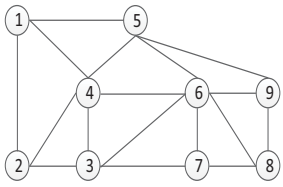
Recently, numerous distributed graph processing systems have been proposed, such as Pregel [3], GraphLab [4], PowerGraph [5]. These systems are vertex-centric and follow a bulk synchronous parallel model (where vertices send messages to each other through their connections or IDs). They are tailored and efficient for graph processing operators that require iterative graph traversal, such as page rank, shortest path, bipartite matching and semi-clustering[3]. However, it is costly to support graph cube computation. In fact, it will incur high overhead for message (carrying the attributes values) passing across the graph to find the vertices/edges with the same attribute values. Compared to vertex-centric processing model, the MapReduce (MR) computation paradigm has been shown to be efficient and scalable in supporting aggregate-like operators which do not require multiple iterations [6]. As we shall see later, graph cube computation is exactly such an operator. As such, MR, which has been successfully demonstrated to be effective for large graph mining [7][8], turns out to be a better fit for OLAP cube computation.

We are thus motivated to develop a new graph OLAP and warehousing model over attributed graphs as well as to develop parallel graph OLAP system over MR that is able to support an efficient graph cube computation algorithm. Our major contributions are summarized as follows:

- We propose a parallel graph OLAP system, *Pagrol*, to provide a good decision making support over large-scale attributed graphs. To support graph OLAP, *Pagrol* adopts a new conceptual graph cube model, *Hyper Graph Cube*, to extend decision making services on attributed graphs. *Hyper Graph Cube* is able to capture queries of all the aforementioned three categories into one model. Moreover, the model supports a new set of OLAP Roll-Up/Drill-Down operations on attributed graphs.
- We propose various optimization techniques to tackle the problem of performing an efficient graph cube computation under MR. The techniques include *self-contained join, cuboid batching, cost-based batch processing*.
- We introduce an efficient cube materialization approach, *MRGraph-Cubing*, that employs these techniques to process large-scale attributed graphs in *Pagrol*. *MRGraph-Cubing* is able to complete the entire materialization in 2 MR jobs. To the best of our knowledge, this is the first parallel graph cubing solution over large-scale attributed graphs.
- We conduct extensive experimental evaluations based on both real and synthetic datasets. The experimental results demonstrate that Pagrol is effective, efficient and scalable.

The rest of this paper is organized as follows: Section II presents our Hyper Graph Cube model used in *Pagrol* , followed by discussions on the query support and OLAP operations. In Section III, we introduce a naive MR-based graph cube computation scheme. Section IV provides the MRGraph-Cubing approached adopted in *Pagrol*. In Section V, we report the experimental results. Finally, Section VI and Section VII provide the related works and conclusion of this paper respectively.

II. HYPER GRAPH CUBE MODEL



(a) Graph Structure

ID	Gender	Nation	Profession
1	male	USA	professor
2	female	USA	doctor
3	male	China	engineer
4	male	SG	engineer
5	female	SG	professor
6	male	SG	doctor
7	female	China	engineer
8	female	China	doctor
9	male	USA	doctor

(b) Vertex Attribute Table

sV	tV	Date	Type	Strength
1	2	2008	Family	9
1	4	2010	Friend	7
1	5	2011	Colleague	9
2	3	2011	Friend	7
2	4	2011	Friend	4
3	4	2008	Friend	8
3	7	2011	Colleague	8
3	6	2012	Family	3
4	5	2009	Family	9
4	6	2010	Friend	6
5	6	2010	Family	8
5	9	2012	Friend	8
6	7	2008	Friend	7
6	8	2012	Colleague	9
6	9	2012	Friend	5
7	8	2011	Friend	5
8	9	2012	Colleague	8

(c) Edge Attribute Table

Fig. 1. A running example of an attributed graph

An attributed graph is able to model various information networks by adding attributes to each vertex and edge. We first provide a formal definition of an attributed graph.

Definition 1 (Attributed Graph): An attributed graph, G, is a graph denoted as G=(V, E, A_v, A_e), where V is a set of vertices, E $\subseteq$ V $\times$ V is a set of edges, and A_v = (A_{v1}, A_{v2}, ..., A_{vn}) is a set of n vertex-specific attributes, i.e. $\forall u \in V$, there is a multidimensional tuple $A_v(u)$ denoted as $A_v(u)$ = ($A_{v1}(u)$, A_{v2}(u), ..., A_{vn}(u)), and A_e=(A_{e1}, A_{e2}, ..., A_{em}) is a set of m edge-specific attributes, i.e. $\forall e \in E$, there is a multidimensional tuple $A_e(e)$ denoted as $A_e(e)$= $(A_{e1}(e), A_{e2}(e), ..., A_{em}(e))$.

To develop graph OLAP and warehousing, we formally define two types of dimensions in attributed graphs as follows:

Definition 2 (Vertex Dimensions): With regard to the attributed graph defined in Definition 1, the set of n vertex-specific attributes (A_{v1}, A_{v2}, ..., A_{vn}) are called the vertex dimensions, or V-Dims for short.

Definition 3 (Edge Dimensions): With regard to the attributed graph defined in Definition 1, the set of m edge-specific attributes (A_{e1}, A_{e2}, ..., A_{em}) are called the edge dimensions, or E-Dims for short.

For instance, Fig. 1 indicates an attributed graph, where each vertex is associated with three V-Dims (A_{v1}, A_{v2}, A_{v3}) - (Gender, Nation, Profession) and each edge is associated with three E-Dims (A_{e1}, A_{e2}, A_{e3}) - (Date, Type, Strength).

In a graph warehousing context, the graph structure-related characteristics can also be extracted as the dimensions for

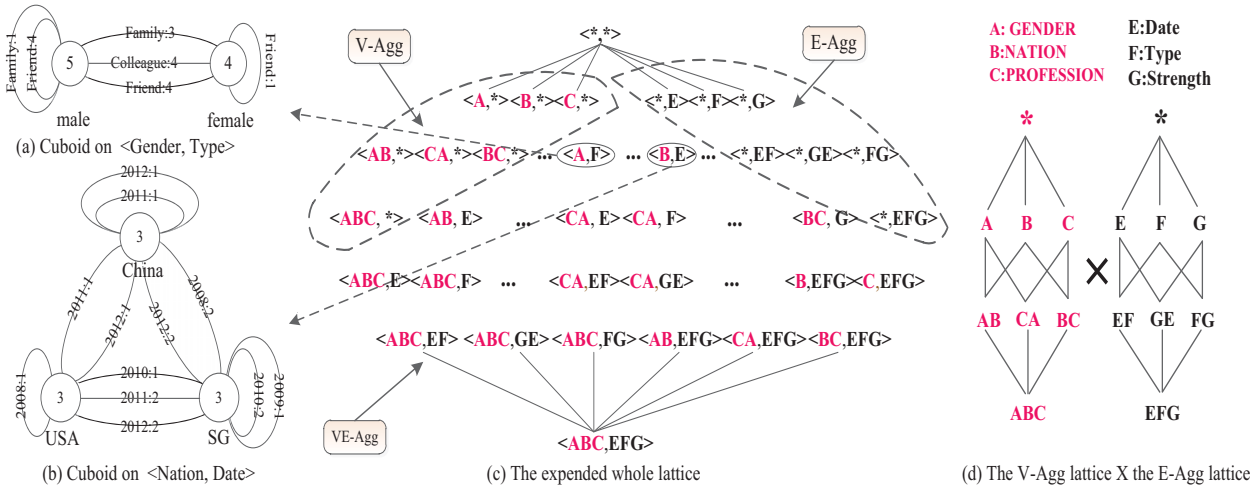


Fig. 2. The Hyper Graph Cube lattice and aggregate cuboids

analysis. For instance, the vertex degree can be extracted as one V-Dim for aggregation. This may extend the utility of the graph warehousing to support more queries.

To support queries in category 1, the data can be aggregated along the V-Dims or the E-Dims alone while omitting the graph structure, which is very similar to the traditional data cubes. In particular, the tuples with the same dimension values are grouped together to calculate the view under a specific aggregate function. Here, the aggregate functions can be vertex count, edge count, centrality, degree, diameter etc. Basically, this is to construct two small cubes along either the V-Dims or the E-Dims. We refer to the aggregation based on V-Dims or E-Dims as V-Agg or E-Agg respectively. For instance, all the V-Aggs and E-Aggs of the given vertex and edge attribute tables in Fig. 1 can be represented as two lattices as shown in the LHS and RHS of Fig. 2 (d).

However, to support those queries in categories 2 and 3, the graph can be aggregated from two aspects: aggregating the vertices along the V-Dims and aggregating the edges along the E-Dims while maintaining the graph structure. In so doing, the aggregation of graphs will be an aggregate graph which can be formally defined as follows.

Definition 4 (Aggregate Graph): Given an attributed graph G=(V, E, A_v, A_e) and a possible vertex aggregation $A_v^{'}$=($A_{v1}^{'}$, $A_{v2}^{'}$, ..., $A_{vn}^{'}$) and a possible edge aggregation $A_e^{'}$=($A_{e1}^{'}$, $A_{e2}^{'}$, ..., $A_{em}^{'}$) where $A_{vi}^{'}$ equals to A_{vi} or * and $A_{ei}^{'}$ equals to A_{ei} or *, the aggregate graph w.r.t. $A_v^{'}$ and $A_e^{'}$ is a weighted graph $G^{'}$=($V^{'}$, $E^{'}$, $W_{v'}$, $W_{e'}$) where

- $V^{'}$ is a set of condensed vertices each of which is associated with a group of vertices $G_v(v^{'})$ such that $\forall v \in V$, there exists one and only one $v^{'} \in V^{'}$ such that $A_v^{'}(v)$=$A_v^{'}(v^{'})$ and $v \in G_v(v^{'})$. The weight of $v^{'}$, $w(v^{'}) = \Gamma_v(G_v(v^{'}))$, where $\Gamma_v(\cdot)$ is an vertex aggregate function.
- $E^{'}$ is a set of condensed edges each of which is associated with a group of edges $G_e(e^{'})$ such that $\forall e = (u, v) \in E$, there exists one and only one $e^{'} = (u^{'}, v^{'}) \in E^{'}$ such that $u \in G_v(u^{'})$, $v \in G_v(v^{'})$, $A_e^{'}(e) = A_e^{'}(e^{'})$ and $e \in G_e(e^{'})$. The weight of $e^{'}$, $w(e^{'})$=$\Gamma_e(G_e(e^{'}))$, where $\Gamma_e(\cdot)$ is an edge aggregate function.

Note that this definition requires that at least one of the dimensions is not * in $A_v^{'}$ and $A_e^{'}$,.

We refer to such an aggregation based on both V-Dims and E-Dims as VE-Agg. According to Definition 4, we also refer to $A_v^{'}$ and $A_e^{'}$ as the vertex group-by dimensions(denoted as VD) and the edge group-by dimensions (denoted as ED) respectively. With regard to our running example in Fig. 1, Fig. 2 (a) and (b) illustrate two examples of the aggregate graph based on $<$ {Gender,*,*}, {*,Type,*}$>$[1] and <Nation, Date> respectively. In these examples, COUNT(.) is the aggregate function for both V-Dims and E-Dims. In a graph OLAP, these functions can also be Average Degree, Diameter, Min/Max Degree, Max/Min Centrality, the Most Central Vertex, Containment and so on, besides the traditional functions like SUM, AVG etc. Note that the aggregation functions for vertex and edge can be different.

Figure 2 (a) provides a *high level aggregate graph* which corresponds to the answer to the example query in category 3. There are two condensed vertices including male and female in the aggregate graph. The vertices for male and female are weighted as 5 and 4, since there are 5 males and 4 females in the original graph. The weighted edges indicate the number of relationships in the original graph. For instance, "Family:3" between male and female indicates that there are 3 family edges where one vertex is male and another vertex is female. Note that the aggregate graph can also be used to answer the queries in categories 2. For instance, the weighted edges between USA and SG provide the answer to the example query in categories 2 indicating the trend of the number of relationships from 2010 to 2012 directly.

Now we provide the formal definition of Hyper Graph Cube.

Definition 5 (Hyper Graph Cube): Given an attributed graph G=(V, E, A_v, A_e) with n V-Dims and m E-Dims,

[1]For simplicity, we omit * in the rest of the paper.

Hyper Graph Cube constructs 2^{n+m} cuboids to aggregate the graph based on all possible V-Dims and E-Dims. It consists of three different types of aggregations: V-Agg represented as < VD, *>, E-Agg represented as <*,ED > and VE-Agg represented as <VD, ED>. There are 2^n-1 V-Agg cuboids and 2^m-1 E-Agg cuboids each of which is similar to the view calculated along either V-Dims or E-Dims in traditional data cubes. There are 2^{n+m} -2^n-2^m+1 VE-Agg cuboids each of which is an aggregate graph defined in Definition 4 along both V-Dims and E-Dims. Note that we reserve <*,*> as a special cuboid.

Intuitively, all the cuboids in Hyper Graph Cube can be represented as the Cartesian product of the V-Agg lattice and the E-Agg lattice. For instance, the hyper cube lattice of the example graph in Fig. 1 includes 64 cuboids represented as the Cartesian product between V-Agg and E-Agg as shown in Fig. 2 (d). Fig. 2 (c) shows the expanded lattice from Fig. 2 (d). Considering two cuboids C_1=<VD_1, ED_1> and C_2=<VD_2, ED_2>, C_1 is an ***ancestor*** of C_2 (denoted as $C_1 \prec C_2$), if $VD_1 \subseteq VD_2 \bigwedge ED_1 \subseteq ED_2$. Meanwhile, C_2 is the ***descendant*** of C_1 (denoted as $C_2 \succ C_1$). For instance, <Gender, Type> $\prec$ <{Gender, Nation}, Type>.

Query Support: On the basis of Hyper Graph Cube, queries can be easily supported by using their corresponding cuboids. The V-Agg or E-Agg cuboids contain the aggregation along either the V-Dims or E-Dims. Thus, they can be directly used to answer the queries in category 1. Furthermore, each VE-Agg cuboid is a high level aggregate graph based one specific dimension space. It is the direct answer to the queries in the category 3. As each aggregate graph also contains the aggregating information over both V-Dims and E-Dims, it can be used to support the queries in category 2 as well. For instance, the cuboid <Nation, Date> in Fig. 2 (b) can be used to answer the query mentioned in category 2.

Roll-Up/Drill-Down OLAP Operations: Roll-Up/Drill-Down are two of the most important OLAP operations to generate views in different levels and granularities. In graph OLAP, each V-Dim or E-Dim may be a dimension associated with a conceptual hierarchy where the Roll-Up/Drill-Down operation can be performed. For instance, the dimensions Birth Place and Time may be associated with geographic or time hierarchies respectively as follows:

- Birth Place: City $\rightarrow$ State $\rightarrow$ Country$\rightarrow$ all
- Time: Month $\rightarrow$ Year $\rightarrow$ Decade $\rightarrow$ all

The Roll-Up/Drill-Down operations along V-Agg or E-Agg are quite similar to the traditional OLAP in the literature [9]. We will not discuss them here. We mainly focus on OLAP operations on the VE-Agg cuboids. Due to the unique feature of VE-Agg, we introduce four different types of Roll-Up/Drill-Down operations in graph OLAP as follows:

Vertex-Up: For Vertex-Up, we fix the E-Dims while we roll up along the V-Dims to aggregate the graph into a more summarized level, such as navigating the aggregate graph from <City, Year> to <State, Year>.

Edge-Up: For Edge-Up, we fix the V-Dims while we roll up along the E-Dims to aggregate the graph into a more summarized level, such as navigating the aggregate graph from <City, Year> to <City, Decade>.

Vertex-Up-Edge-Up: For Vertex-Up-Edge-Up, we roll up along both the V-Dims and E-Dims to aggregate the graph into a more summarized level, such as navigating the aggregate graph from <City, Year> to <State, Decade>.

Vertex-Up-Edge-Down: For Vertex-Up-Edge-Down, we roll up along the V-Dims and drill down along the E-Dims to aggregate the graph, such as navigating the aggregate graph from <City, Year> to <State, Month>.

Similarly, there are four corresponding operators - **Vertex-Down**, **Edge-Down**, **Vertex-Down-Edge-Down** and **Vertex-Down-Edge-Up** - that operate in the opposite direction. Note that we can speed up OLAP operations for distributive and algebraic measures in two ways. First, the Roll-Up/Drill-Down can be conducted over the intermediate aggregate graph instead of the base graph. Second, the Roll-Up/Drill-Down operations using the closest aggregate graph is more efficient than others. For instance, to get an aggregate graph <Country, Decade>, Roll-Up operation is more efficient using the aggregate graph <State, Year> than <City, Month>.

III. A Naive MR-based Scheme

On the basis of Hyper Graph Cube, we are now to introduce the graph cube materialization over MR. We first briefly review the computation paradigm of MR. The computation of MR follows a fixed model with a map phase, followed by a reduce phase. The map function running on a mapper is used to process (key, value) pairs (k1,v1) of one input data chunk read from the distributed file system (DFS). After applying the map function, it then emits a new set of intermediate (k2,v2) pairs. The MR library sorts and partitions all the intermediate (k,v) pairs based on k. In the reduce phase, the library merge-sorts all the (k,v) pairs and supplies the globally sorted data to the reduce function running on a reducer. After the reduce process, the reducer emits new (k3,v3) pairs to DFS.

Algorithm 1: The Naive Cubing MR Job

```
1  Function Map()
2    # t is a tuple in the data
3    If t is a vertex (or an edge or a joined edge) then
4      foreach cuboid C_i ∈ V-Agg (or E-Agg or VE-Agg)
5        Project C_i's group-by attributes from t ⇒ k
6        Other information ⇒ v;
7        emit(k,v);
8  Function Reduce( k, v_0, v_1, ..., v_k)
9    let M be the measure function
10   emit(k, M(v_0, v_1, ..., v_k));
```

Under MR, a naive algorithm may conduct the graph cubing in two steps. Recall that the vertex and edge attributes are typically stored in two separate tables in the original data format as shown in Fig. 1. Therefore, in the first step, we join the two tables to obtain a flat table that contains all the dimensions. This join operation can be performed using one MR job (referred as a *Join-Job*), and the joined output is written back to DFS.

The second step performs the cube computation. One approach is to compute all the cuboids in one MR job, referred

as a *Cubing* job, where each cuboid is processed independently. Obviously, calculating the cuboids in V-Agg and E-Agg based on the original vertex and edge attribute tables is more suitable than the joined data. Thus, this cubing job can take both original tables and joined data as input. Recall that each cuboid in the VE-Agg is an aggregate graph as defined in Definition 4. To construct such an aggregate graph, each condensed vertex actually can be calculated while processing the cuboids in V-Agg since they would group the vertices in the same way. Thus, we only need to calculate the weighted edges between two condensed vertices in the aggregate graph based on the joined data. The algorithm is provided in Alg. 1. This naive algorithm extracts, shuffles and processes each cuboid's group-by attributes independently.

However, for large attributed graph, such a naive algorithm is expected to perform poorly for two reasons:

Size of joined data: As one vertex may be associated with multiple edges, the join output of the Join-Job can be very large. Assume that the average degree of each vertex, the size of vertex attribute table and the size of the edge attribute table are d, $|V|$ and $|E|$ respectively. The size of the output of the Join-Job is almost $d * |V| + |E|$.

Size of intermediate data: Processing each cuboid independently may generate a large amount of intermediate data, since each cuboid extracts its own (k,v) pairs which will incur high sorting/shuffling overheads.

IV. MRGraph-Cubing Graph cube materialization

In this section, we introduce an efficient Hyper Graph Cube computation approach, MRGraph-Cubing to handle large attributed graphs in *Pagrol*. We first introduce the overall process of MRGraph-Cubing: 1) It joins the vertex and attribute tables. For this, we propose a **self-contained join** to avoid writing and reloading the joined data to/from DFS (section IV-A); 2) It groups cuboids into batches so that the intermediate data and computation can be shared. Our **cuboids batching** scheme identifies the cuboids that can be batched (section IV-B); 3) It further bundles batches into bags so that we can process each bag in a single MR job (section IV-C); 4) To ensure optimal bundling of batches into bags in (3), we further develop a **cost-based execution plan optimizer** that can generate an execution plan to minimize the cube computation time (section IV-D).

A. Self-Contained Join

To reduce the high overhead incurred by the large joined data from the first Join-Job, we propose a *self-contained join* technique to postpone the join operation to the map side of the second cubing MR job. In so doing, after the join, the data will be directly used for graph cube computation and do not need to be written/reloaded via DFS. However, a self-contained join requires the data in each mapper to contain the edges and their corresponding vertices. Thus, instead of a Join-Job, we issue a *Blk-Gen* MR job to reorganize the original data into a series of self-contained data files.

In particular, the Blk-Gen job reads both the vertex and edge attribute tables in the map phase. Then it partitions the edges to different reducers according to its two vertex IDs.

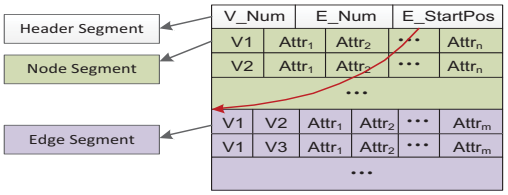


Fig. 3. The self-contained file format

Meanwhile, it also partitions the vertex information to the corresponding reducers whose edges contain the same vertex IDs. Note that each vertex is shuffled to multiple reducers as needed. In the reduce phase, the Blk-Gen job generates a series of self-contained data blocks and outputs each block as one file whose format is described in Fig. 3. Under this scheme, in each file, the vertex can be shared by multiple edges instead of being replicated multiple times.

As MR does not partition the input file if it is not bigger than one block, each self-contained file will be supplied to one mapper directly to perform a join in the second cubing join.

B. Cuboids Batching

To build a graph cube, computing each cuboid independently is clearly inefficient. A more efficient solution, which we advocate, is to combine cuboids into batches so that intermediate data and computation can be shared and salvaged.

For the scheme to be effective, we first identify the cuboids that can be combined and batched together. Here, we assume that we materialize the complete cube. Our scheme can be easily generalized to materialize a partial cube (compute only selected cuboids).

Recall that there are three different types of cuboids: V-Agg, E-Agg and VE-Agg. For processing the cuboids in either V-Agg or E-Agg, we have developed a solution in [10]. As such, we shall give an overview here as our focus in this work is on the VE-Agg; interested readers are referred to the paper for details. The collection V-Agg/E-Agg cuboids can be batched together under MR if they satisfy the following *combine criterion*:

Criterion 1: Among the multiple V-Agg or E-Agg cuboids, any two of them have the ancestor/descendant relationship and share the same prefix.

For instance, given 3 V-Agg cuboids C_1, C_2 and C_3, if their vertex group-by dimensions are VD_1=A, VD_2=AB and VD_3=ABC respectively, then $C_1 \prec C_2 \prec C_3$ and they share the same prefix with each other. In this given example, these 3 cuboids can be combined and processed together to facilitate the MR sorting feature (MR sorts all the intermediate (k,v) pairs according to the key) using the approach we proposed as follows: When the mapper reads each tuple, it emits one (k,v) pair to serve C_1,C_2 and C_3 where VD_3 is set as the key and VD_1 is used to partition the intermediate pairs, instead of the emitting three pairs. This would guarantee that the tuples with the same group-by values of C_1, C_2 and C_3 are shuffled to the same reducer, and thus can be processed together. More correctness proof can be found in [10].

The benefits of this approach are: 1) In the reduce phase, the group-by dimensions are all in sorted order for every cuboid in the batch, since MR would sort the data before supplying to the reduce function. This is an efficient way of cube computation since it obtains sorting for "free" and no other extra sorting is needed for aggregation. 2) All the ancestor cuboids do not need to shuffle their own intermediate data but use their descendant's. This would significantly reduce the intermediate data size, and thus remove a lot of data sorting/partitioning/shuffling overheads.

In Graph cube computation, we claim that the cuboids in VE-Agg can be batched together under MR if they satisfy the following *combine criterion*:

Criterion 2: Among the multiple VE-Agg cuboids, any two of them have the ancestor/descendant relationship. In addition, the V-Dims between any two of the cuboids share the same prefix, as well as their E-Dims.

Intuitively, to obtain the aggregate graph of each VE-Agg cuboid, the procedure can be divided into two parts: 1) In part 1, the vertices with the same V-Dims are grouped into condensed vertices and each condensed vertex's weight is computed using the vertex aggregate function. 2) In part 2, the edges with the same V-Dims of the two vertices and the same E-Dims are grouped into condensed edges weighted using the edge aggregate function. As we have mentioned before, the first part can be conducted with the V-Agg cuboids since they share the same vertex grouping condition. Note that if the aggregate function of V-Agg is the same as the one in the VE-Agg, the result of V-Agg can be directly used for constructing the VE-Agg. Thus, we focus on introducing the approach of calculating the weighted edges here.

As an example, suppose we are given three VE-Agg cuboids C_1, C_2 and C_3, where their vertex group-by dimensions are VD_1=A, VD_2=AB and VD_3=ABC and edge group-by dimensions are ED_1=E, ED_2=EF and ED_3=EF respectively, then $C_1 \prec C_2 \prec C_3$ and the V-Dims (as well as the E-Dims) share the same prefix order. In this example, C_1 and C_2 can be processed and combined together with C_3 into one batch. Note that calculating the weighted edges is based on the joined edges each of which (e) is a triple-tuple: V-Dims of sV (VD(sV)), V-Dims of tV (VD(tV)) and E-Dims of e (ED(e)).

Under MR, C_1, C_2 and C_3 can be processed in one batch as follows: When one mapper parses one joined edge e, it emits one (k,v) pair to serve C_1, C_2 and C_3 where the concatenation of VD_3(sV), VD_3(tV) and ED_3(e) (which is ABCABCEF) as the key. And the concatenation of VD_1(sV), VD_1(tV) and ED_1(e) (which is AAE) is used to partition the (k,v) pairs. This guarantees that all the edges with the same group-by values of C_1, C_2 and C_3 are shuffled to the same reducer, thus they can be processed together.

Now, we formally define two special cuboids within one batch.

Definition 6 (Project_Cuboid): Given one batch, if all other cuboids are the ancestors of cuboid A, A is defined as the Project_Cuboid, denoted as Pro_Cubd. During the cube computation, the aggregation dimensions of A are projected as the key to serve all other cuboids within the batch under MR.

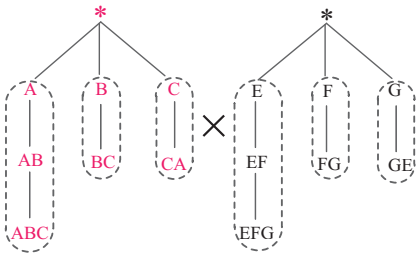


Fig. 4. The generated batches

Definition 7 (Partition_Cuboid): Given one batch, if cuboid A is the ancestor of all other cuboids, A is defined as the Partition_Cuboid, denoted as Par_Cubd. During the cube computation, the aggregation dimensions of the A are used to partition the key.

For instance, given one batch with three V-Agg cuboids {<A,*>, <AB,*>, <ABC,*>}, <ABC,*> and <A,*> are called the Pro_Cubd and Par_Cubd respectively. Likewise, given one batch with three VE-Agg cuboids {<A,E>, <AB,EF>, <ABC,EF>}, <ABC,EF> and <A,E> are called the Pro_Cubd and Par_Cubd respectively.

Basically, the more cuboids we combine into one batch, the less intermediate data will be generated and the more computation sharing we can get. Based on the aforementioned principles, one cuboid can be batched with all its descendant cuboids satisfying criterion 1 or 2. Therefore, to generate batches, we first search the cuboids in V-Agg and E-Agg and generate the batches within V-Agg and E-Agg according to criterion 1. For instance, given the lattice in Fig. 2 (d), it generates three batches within V-Agg and E-Agg as shown in Fig. 4 using the dotted lines.

For VE-Agg, the batch plan can be generated by combining the V-Agg batches with the E-Agg batches through cartesian product. Let us still take Fig. 2 (d) as an example. The final execution plan will generate 9 batches whose Pro_Cubds are <ABC,EFG>, <ABC,FG>, <ABC, GE>, <BC,EFG>, <BC,FG>, <BC, GE>, <CA,EFG>, <CA,FG> and <CA, GE>. Each batch consists of multiple cuboids. For instance, the batch <ABC, EFG> includes the 9 cuboids <A,E>, <A,EF>, <A,EFG>, <AB,E>, <AB,EF>, <AB,EFG>, <ABC,E>, <ABC, EF> and <ABC,EFG>.

C. Batch Processing

Since processing V-Agg, E-Agg and VE-Agg are all based on different input data, we propose to process the batches in these three different types separately. The MR-based algorithm of processing V-Agg and E-Agg is similar to the cube computation in relational database [10]. Thus, we omit it here. As mentioned in section IV-B, calculating the condensed vertices can be integrated with V-Agg. Therefore, in this section, we mainly focus on how the batches in VE-Agg can be processed to get all the condensed and weighted edges.

Given a computing cluster and a set of batches, there are multiple plans to process all the batches.

Definition 8 (Execution Plan): Given a set of batches B=$\{B_1, B_2, ..., B_x\}$, the execution plan is a set of MR jobs

Algorithm 2: The MRGraph-Cubing Algorithm

```
1  Function Map()
2    Let t be an input tuple
3    If t is a vertex then
4      Cache t in memory;
5    ElseIf t is an edge then
6      t joins vertex ⇒ e
7      foreach B_i ∈ B
8        extracts k and v from e for B_i and emit(k,v) ;
9  Function Partition(k,v)
10     b ⇐ get batch number from v;
11     Par_Cubd ⇐ get partition value from k;
12     return Σ_{i=1}^{b-1} R(i) + Par_Cubd%R_b;
13 Function Reduce(k, v_0, v_1, ..., v_k)
14   # M ⇐ The measure function
15   B_i ⇐ Get batch from the batch identifier
16   C_i ⇐ The cuboid in B_i
17   For C_i in B_i
18     If the group-by cell in C_i receives all tuples it needs
19       v ⇐ M(v_1, ..., v_m, v_1', ..., v_k', ...)
20       emit(k,v);
21     Else
22       buffer the measure for aggregation
23 Function Combine(k, v_0, v_1, ..., v_k)
24   # M ⇐ The measure function
25   id ⇐ Get batch from identifier
26   v ⇐ M(v_1, ..., v_m, v_1', ..., v_k', ...)
27   v ⇐ v.append(id);
28   emit(k,v);
```

p=$\{j_0, j_1, ..., j_k\}$ where j_i is in charge of processing one bag which consists of one or more batches. The batches processed by j_i is denoted as $bag(j_i)$. And p satisfies the following condition: $bag(j_0) \bigcap bag(j_1) \bigcap .... \bigcap bag(j_k) = \phi$ and $bag(j_0) \bigcup bag(j_1) \bigcup .... \bigcup bag(j_k) = B$.

For instance, two straightforward execution plans are: 1) put each batch into one independent bag and process each bag using one MR job; 2) put all the batches into one bag and process this bag using one MR job. The advantage of the second plan is that the original data only need to be read once. However, each mapper has to replicate and emit all the intermediate data for all batches which incur high overhead to collect, partition and sort them. In contrast, under the first plan, the MR job only needs to emit the data for one batch resulting in more efficient data collection, partitioning and sorting in the map phase. However, each job needs to read the original data once, which can be costly when the number of jobs increases.

Clearly, there are many other possible plans. We defer the discussion on finding an optimal execution plan to the next subsection. Here, we shall focus on how a bag (containing a set of batches) can be processed using a single MR job.

Consider one bag B consisting of x VE-Agg batches $\{B_1, B_2, ..., B_x\}$. Suppose the number of reducers needed for each batch is R=$\{R_1, ..., R_x\}$, where R_i is the number of reducers assigned for batch B_i. Alg. 2 lists how our proposed MR-based scheme works.

Map Phase: The input data are the self-contained files output from the first Blk-Gen job each of which is supplied to one mapper. The mapper first conducts a join. Basically, it caches the vertex information in memory, as the vertices are supplied to mappers earlier than the edges (Lines 3-4). Note that caching the vertices consumes very little memory, since cached information is always smaller than one block (input file) size. When an edge arrives, it performs a join with the vertices. Whenever a joined tuple is produced, it constructs and emits x (k, v) pairs for the x batches in bag B (Lines 5-8). For each batch B_i (whose Pro_Cubd is <VD, ED>), the key is set as the concatenation of VD(sV), VD(tV) and ED(e) as described in Section IV-B. Meanwhile, other information can be put to the value. For instance, if the aggregate function is COUNT(), then 1 can be put into the value.

To distinguish which (k,v) pair is for which batch, we append one bitmap to the value. The size of the bitmap corresponds to the number of batches where the k^{th} bit is set to 1 if this pair belongs to batch B_{k+1}.

The partitioning function partitions the intermediate (k,v) pairs to their corresponding reducers according to the Par_Cubd and reducer allocation plan R (Lines 9-12).

Reduce Phase: In the reduce phase, all the (k,v) pairs are sorted based on the key. Each reducer obtains its computation tasks (the cuboids in the batch) by parsing the identifier in the value. For each input tuple, the reduce function extracts the measure and projects the group-by dimensions for each cuboid in the batch. For the Pro_Cubd, the aggregation can be conducted based on each input tuple, since each input tuple is one complete group-by cell. This case is captured in line 18. For other cuboids, the measures of the group-by cell are buffered until the cell receives all the measures it needs for aggregation (lines 13-22). Finally, the aggregation results for different cuboids are written to different destinations in DFS.

Note that if the (k,v) pairs can be pre-aggregated in the map phase, users can write a *combiner* which will reduce the shuffle data size. The combine function is listed in lines 23-28.

Using the Alg. 2, multiple batches in one bag can be processed in one MR job. In next section, we introduce our proposed batch execution optimization technique to generate the optimal bags.

D. Cost-based Execution Plan Optimization

Definition 9 (Batch Execution Plan Optimization): Given a computing cluster and x batches, the batch execution plan optimization problem is to find the best way to bag the batches such that the generated execution plan (as defined in Definition 8) has the smallest cube materialization time.

Intuitively, the optimizer consists of two key components: a) *Plan refinement* searches different execution plans; b) *Plan execution time estimation* estimates each plan's execution time so that the plan with the smallest execution time can be chosen.

Plan Refinement: Given x batches, $B : \{B_1, B_2, ..., B_x\}$, enumerating all the possible plans is equivalent to computing all partitions from B which has been well studied in [11]. The total number of partitions is $\Theta((\frac{x}{ln(x)})^x)$ [11]. Therefore, when x is large, an exhaustive enumeration is no longer applicable. As such, some heuristic algorithms can be used to find a suboptimal execution plan.

TABLE I. VARIABLES USED IN COST MODEL

Notations	Description
SF	Sort factor configured in MR cluster
S_{spill}	Spill size configured in MR cluster
IO_{lr}	I/O cost for reading from local disk per MB
IO_{lw}	I/O cost for writing from local disk per MB
NW	cost for network transfer per MB
IO_{hr}	I/O cost for reading from DFS per MB
IO_{hw}	I/O cost for writing to DFS per MB

In this paper, we adopt a greedy algorithm that is iterative in nature and follows the classical local search pattern. Let P_i denote the input execution plan for iteration i. Initially, P_0 corresponds to the case where there are x jobs (i.e., each bag has only one batch). Intuitively, the algorithm works as follows: In (i+1)th iteration, it evaluates all the plans obtained by bundling any two jobs in P_i together and finds the best plan as P_{i+1}. If P_{i+1} is better than P_i (based on a cost model to be discussed shortly), P_{i+1} is passed on to the next search iteration. If P_{i+1} is worse than P_i or P_i contains one job with all batches, the algorithm terminates. And P_i is chosen as the final execution plan. In the worst case, this algorithm needs to enumerate $\binom{x}{2}$+$\binom{x-1}{2}$...+$\binom{2}{2}$ possible plans, which is bounded in $O(x^3)$ time, where $\binom{i}{2}$ is the number of plans to evaluate in the $(x-i+1)^{th}$ iteration. In practice, this plan enumeration time is acceptable during the cube computation.

Plan Execution Time Estimation: Before introducing how to evaluate each plan's execution time, we first provide a cost model which is used to estimate the execution time of each mapper and reducer in one job. The cost model is aware of the cluster hardware, MR configurations, input file and the batches it needs to process. For simplicity, we assume no compression is adopted during the processing. In addition, considering that the I/O cost dominates performance, for simplicity, we omit the CPU cost in the model. Table I lists the variables related to the MR cluster.

Assume one MR job needs to process one bag with y batches $\{B_1, B_2, ..., B_y\}$ and m input files. The set of reducers to process B_i is R_i. In addition, we refer to S_i as the input file size for mapper M_i, C_{ij} as the j^{th} cuboid in batch B_i, CR_i as the combine ratio of batch B_i in the map phase, CR_{ij} as the combine ratio for C_{ij} in the reduce phase, P_i as the project ratio for B_i and P_{ij} as the project ratio for C_{ij}. Here, the combine (resp. project) ratio indicates the percentage of data that remains after combine/aggregate (resp. project) operations. If no combiner is used, then the batch combine ratio CR_i equals to 1.

For the map phase, we refer to $M_c = \sum_{i=1}^{i=y}(S_i \cdot P_i)$ as the size of (k,v) pairs for all batches after projection, $N_s = \lceil \frac{M_c}{S_{spill}} \rceil$ as the number of spills created in spill phase, $N_m = \lceil log_{SF} N_s \rceil$ as the number of merge passes, $M_o = \sum_{i=1}^{i=y}(S_i \cdot P_i \cdot CR_i)$ as the intermediate map output data after combining.

The total cost of mapper M_i is calculated as follows:

$$S_i \cdot IO_{hr} + M_o \cdot N_m(IO_{lr} + IO_{lw}) \quad (1)$$

where $S_i \cdot IO_{hr}$ is the cost for reading the input file from DFS; $M_o \cdot N_m(IO_{lr} + IO_{lw})$ is the local I/O cost for sorting and partitioning the intermediate data.

For the reduce phase, we refer to R_{ij} as the j^{th} reducer processing B_i, R_{in}=$\frac{\sum_{k=1}^{k=m}(S_k \cdot P_i \cdot CR_i)}{|R_i|}$ as the input size for R_{ij},

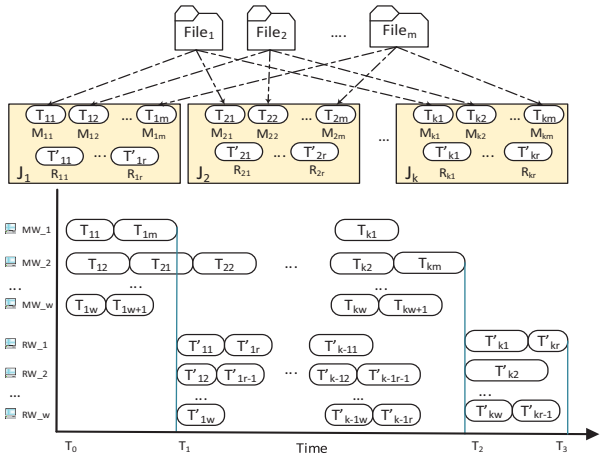


Fig. 5. The worker fitting model for multiple jobs execution

$R_{np} = \lceil log_{SF} m \rceil$ as the number of merge passes on R_{ij} and S_f= $\sum_{C_{ij} \in B_i}(R_{in} \cdot CR_{ij} \cdot P_{ij})$ as the output size in R_{ij}.

The total cost of the reducer R_{ij} is calculated as follows:

$$R_{in} \cdot NW + R_{in} \cdot R_{np} \cdot (IO_{lr} + IO_{lw}) + S_f \cdot IO_{hw} \quad (2)$$

where $R_{in} \cdot NW$ is the cost for shuffling data from mappers to R_{ij}; $R_{in} \cdot R_{np} \cdot (IO_{lr} + IO_{lw})$ is the cost for merge-sort, and $S_f \cdot IO_{hw}$ is the cost for writing the views to DFS.

Given a cluster, all the variables in Table I can be predetermined. Meanwhile, the input file size of each mapper can be easily obtained from DFS. The only parameters that we need to dynamically collect w.r.t. different datasets for the model are the combine ratio and project ratio for each batch and cuboid. To collect this information, a "mini" cubing is performed based on a small set of sampling data in memory in each reducer during the Blk-Gen job. The project ratio is related to the dimension size, thus it is easy to obtain. The batch combine ratio CR_i (resp. cuboid combine ratio CR_{ij}) can be obtained by recording the percentage of data in B_i (resp. C_{ij}) that remains after applying the combine (resp. aggregate) function. Since the data is already in memory, this graph information collection is, as we shall see in our experimental results in section V, efficient. The average value among these reducers can be input to the model for planning. We note that this cost model may not provide an accurate estimation. Instead, this is an approximate approach to evaluate the relative cost. The intuition is to avoid bad plans. We emphasize that our scheme is not restricted to this specific model; instead, other cost models can also be used.

Given one execution plan p=$\{j_0, j_1, ..., j_k\}$, assume that the execution time of each mapper and reducer has been obtained by the cost model for each job j_i. Estimating the execution time of multiple jobs is still non-trivial under MR. A very straightforward approach is to estimate p's execution time T(p) simply as the sum of each job j_i's execution time T(j_i) as follows: $T(p) = \sum_{i=1}^{k} T(j_i)$.

However, we claim that this naive approach is not accurate, since it omits the context of cluster resource and MR scheduling strategy. For instance, assume the number of mappers and reducers one cluster can run simultaneously is larger than the total number of mappers and reducers that j_1

TABLE II. CLUSTER CONFIGURATION

Parameter	Value
Hadoop Version	Hadoop-1.1.1
Mappers per Node	2
Reducers per Node	2
Replication Factor	3
io.sort.fact	20
JVM Size per Task	2GB
Size of Data Chunk	256MB
Default Node Number	64

and j_2 need, the execution time of running j_1 and j_2 will be $\text{MAX}(T(j_1), T(j_2))$ instead of $T(j_1) + T(j_2)$.

In this paper, we propose a *worker fitting model* to precisely estimate multiple jobs execution time which tightly simulates the scheduling mechanism in MR. We choose the FIFO scheduler as our illustration example in this section. According to MR, when multiple MR jobs are submitted, it maintains a map task and a reduce task queues separately. The map and reduce tasks of the first received job are maintained in the head of the queues. When the cluster has any free slot (mappers or reducers), it schedules the next unprocessed task in the corresponding queue for processing.

Based on this fact, the worker fitting model simulates the MR scheduling process as follows - illustrated via an example as shown in Fig. 5: Intuitively, the mappers and reducers that the cluster supports are considered as map workers and reduce workers which are used to consume the map and reduce tasks. As in Fig. 5, there are w map workers and w reduce workers. For a given plan p, k MR jobs are submitted to the cluster.

In Fig. 5, the example contains m mappers (as there are m input files) and r reducers. Using the cost model, the optimizer estimates the execution time of each mapper and reducer. For instance, in job j_1, T_{11} is the time needed to process the first input file and T'_{11} is the time needed to process a reduce task. The model captures the slowest reducer finishing time as the plan's execution time.

With a FIFO scheduler, the map tasks in j_1 are assigned to the map workers first, then the ones in j_2, j_3 and so on. The strategy of task fitting is to assign the map task to the worker with the "cheapest" map tasks. Here, by "cheapest", it means a smaller SUM value of each map's execution time one worker gets. For instance, as in Fig. 5, T_{1w+1} is assigned to MW_w since MW_w gets the smallest task from the first round.

On the other hand, the reduce tasks can only be scheduled to shuffle data when some of the mappers in the same MR job have finished. For simplicity, we assume the reduce task in one job starts when all map tasks finish. The optimizer starts to fit the reduce task to the reduce workers using the same task fitting strategy at the time point when the slowest map task finishes and there are "free" reduce workers. Otherwise, the reduce task has to wait to be scheduled. For instance, in Fig. 5, the slowest map task in j_1 finishes at time point T_1. Then the reduce tasks in j_1 start to be fitted to the reduce workers. At time point T_2, all the map tasks in j_k are finished, and then the reduce tasks in j_k can be scheduled. At time point T_3, all the jobs finish.

We emphasize that this technique is applicable to different schedulers, such as the fair scheduler or the capacity scheduler. With different schedulers, the task queue may be in a different order but still under the same methodology. With this cost-based multiple jobs execution estimation, the optimizer is able to identify the best plan to conduct the graph cube computation.

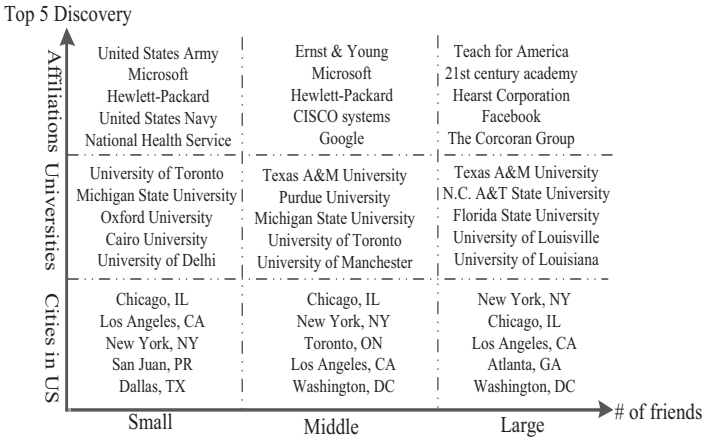


Fig. 6. OLAP query on V-Agg Cuboids

V. EXPERIMENT

We evaluate *Pagrol* on our local cluster with 128 nodes. Each node consists of a X3430 4(4) @ 2.4GHZ CPU running Centos 5.4 with 8GB memory and 2X 500GB SATA disks. Our evaluation is based on Hadoop [12], an open source equivalent implementation of MR. The detail configuration of the cluster is provided in Table II.

We perform our experimental studies on two kinds of datasets including one real Facebook dataset and a set of synthetic datasets.

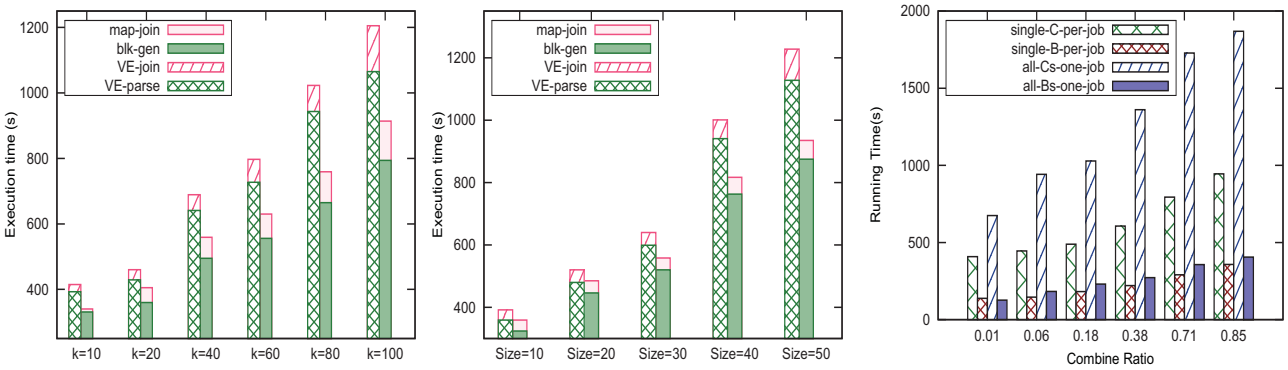
Facebook Dataset. This dataset contains a sampled Facebook data collected in April-May 2009 [13]. In the dataset, each vertex is one Facebook user and each edge indicates the relationship between two users. We further extract four dimensions for each vertex: *TotalFriends*, *School*, *Region* and *Affiliation*. And we extract one dimension for each edge: *Type* with three different values (Schoolmates, Colleagues and Friends). This dataset includes 957,359 vertices and 4.5 million edges.

Synthetic Dataset. For synthetic data, we use SNAP to generate the underlying graph structure without dimensions [14]. Since SNAP is slow in generating large attributed graphs, we develop a MR-based parallel synthetic attributed graph generator to add dimensions to each vertex and edge in parallel to the graph structure generated by SNAP.

A. Effectiveness

We first show the effectiveness of Hyper Graph Cube as a powerful decision making tool over the Facebook data. We present some interesting findings by issuing OLAP queries.

First, we want to identify the communities of active users in Facebook. Note that we classify the number of total friends into three categories: small (the number of friends is less than 150 friends), middle (between 150 to 800) and large (larger than 800). Then based on the V-Agg cuboid $< \{\text{Region,TotalFriends}\}, * >$, $< \{\text{School, TotalFriends}\}, * >$



(a) Impact of graph degree (b) Impact of V-Dim size (c) Impact of batch processing

Fig. 8. Evaluation for self-contained join and batch processing

and $< \{\text{Affiliation, TotalFriends}\}, * >$, we can easily obtain the top 5 communities which has the largest number of people in each category as shown in Fig. 6. We can see that "Teach for America", "Texas A&M University" and "New York, NY" have the largest number of people with more than 800 friends on Facebook. The result also indicates that "New York" belongs to the top 5 in each category which shows that people living in New York are very active in Facebook.

Next, we investigate how people are connected with each other between the cities in California: LA (Los Angles), OC (Orange City), SFO (San Francisco) and SV (Silicon Valley). We perform such a query based on the VE-Agg cuboid <Region, Type>. Fig. 7 provides an aggregate graph which gives us a much simpler and clearer picture of the original Fackbook data. We observe that there are only 4 cross-city colleague relationships (denoted as C:4 in the figure) between SV and SFO. Such a relationship suggests that some companies may be operating in these two cities. By drilling down, we discover that three of the relationships were built among the people working in Tellme Inc. which is a big company operating in both cities. Another observation is that cross-city schoolmate relationships (denoted as S:21) appear the most between LA and OC. Our drill-down operation allows us to figure out that these belong to the Universities in California like UCLA. There are still much more interesting information. Due to space constraint, we do not discuss them here. From the results, it is clear that our Hyper Graph Cube model offers an effective mechanism to study and explore the characteristics of large graphs.

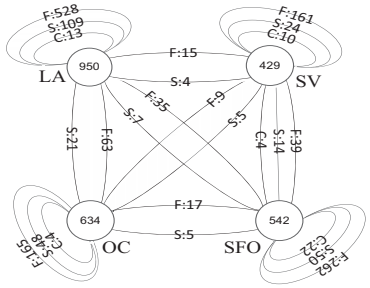


Fig. 7. OLAP query on VE-Agg cuboid <Region, Type>

For the following experiments, we use the synthetic dataset. Each graph contains 3 V-Dims and 3 E-Dims. By default, the average size of each V-Dim or E-Dim is 7 bytes. Meanwhile, we choose COUNT(·) as the measure function for both the vertex and edge aggregation. Note that if a graph contains N edges with degree K, there will be N*2/K vertices in the graph.

B. Self-Contained Join Optimization

In this experiment, we compare the proposed self-contained join optimization with the naive approach. For our proposed self-contained join optimization, we have two MR jobs: *blk-gen* job which reads the vertex and edge attribute tables and generates a series of self-contained files, and *map-join* job which reads the self-contained files, conducts the map side join and ends after the join operation in the map phase. For the naive approach, we also have two MR jobs: *VE-join* job which joins the vertex with edge attribute tables, and *VE-parse* job which reads the joined data and ends when it finishes parsing each joined data in the map phase.

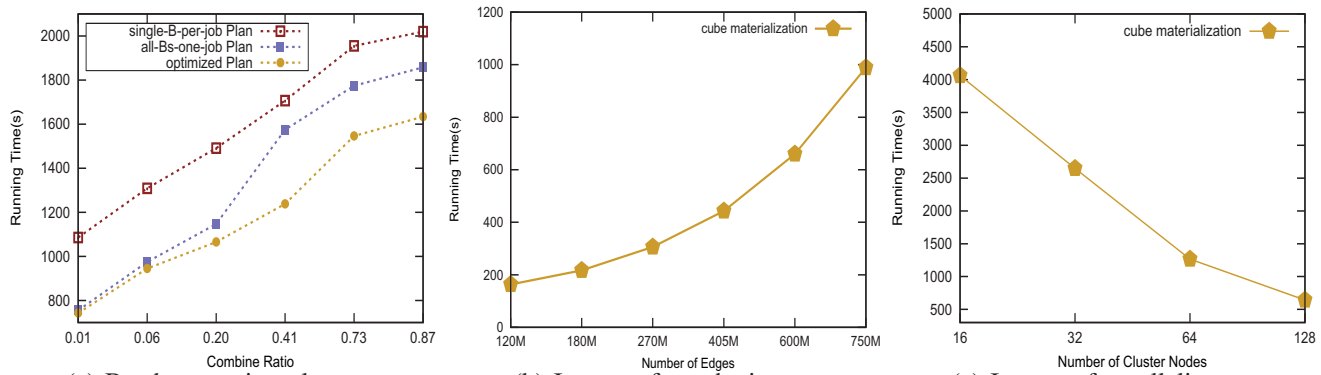
Figure 8 (a) shows the performance comparison on datasets with 200 million(M) vertices when we vary the graph degree from 10 to 100. The results indicate that our optimization performs, on average, 30% faster than the naive scheme.

Figure 8 (b) provides the comparison results by using the graphs with 200M vertices and with degree 40 when we vary the average size of each V-Dim from 10 bytes to 50 bytes. The results indicate that our scheme has an average performance improvement of 21% over the naive scheme as well.

The insights we gain are: 1) The self-contained join is superior over the naive join strategy over all the evaluated datasets; 2) The performance gain is more significant when the graph degree is high and the vertex attribute is large.

C. Cuboids Batching Optimization

Next, we study the benefit of our proposed cuboid batching optimization. We implement four algorithms. The first two are the baselines without batching the cuboids: single-C-per-job (computes each cuboid independently in one MR job) and all-Cs-one-job (computes all cuboids in one MR job). The next two are with our batching optimization: 1) single-B-per-job (compute each batch in one MR job) and all-Bs-one-job (compute all batches in one MR job).



(a) Batch execution plans (b) Impact of graph size (c) Impact of parallelism

Fig. 9. Evaluation of the plan optimizer and scalability

Figure 8 (c) provides the comparison results based on the datasets with 600M edges and with degree 60 when we vary the average combine ratio of the graph. The results indicate that the algorithms with batching optimization are 2.5X and 4X faster than single-C-per-job and all-Cs-one-job respectively. Meanwhile, we observe that the larger the intermediate data (with a bigger combine ratio) are, the higher the performance gain by the batching optimization.

D. Batch Execution Plan Optimization

We next evaluate the proposed batch execution plan optimizer. We compare the performance between the plan (*optimized plan*) generated by the optimizer with two basic plans: single-B-per-job and all-Bs-one-job. Fig. 9 (a) shows the results on the datasets with 600M edges with degree 60 while varying the combine ratio from 0.01 to 0.87. The execution time is the complete graph materialization time including the first Blk-Gen job. For the optimized plan approach, the graph information is collected by sampling 1% tuples in the memory in the Blk-Gen job.

The results show that the execution time of the optimized plan and all-Bs-one-job is much shorter than single-B-per-job when the combine ratio is low. When the combine ratio equals to 0.01, the optimized plan is the same as the all-Bs-one-job. This is reasonable, since when the map output size is small, combining all batches in one job is less costly. It also shows that the optimizer can generate much better plan than the two basic ones when combine ratio increases. From this experiment, we gather the following insights: 1) The in-memory graph information collection for cost model incurs very low overhead; 2) The optimizer is able to find reasonable execution plan in all cases even with a simple sampling approach.

E. Scalability

Finally, we evaluate the scalability of our MRGraph-Cubing approach from two different angles. The first set of experiments studies the scalability of the scheme with respect to the size of the datasets. Fig. 9 (b) shows the execution time of cube computation on the datasets with degree 60 when we vary the number of edges from 120M to 750M. We observe that when the dataset is 1.5X bigger, the execution time becomes almost 1.5X slower. This indicates that the algorithm scales well in terms of dataset size.

The second set of experiments aims at studying the impact of parallelism. Fig. 9 (c) provides the execution times of cube materialization on a dataset with 750M vertices with degree 60 when we vary the cluster size from 16 to 128 nodes. The results indicate that when the computation power is doubled, the execution time almost reduces to half from 16 to 64 nodes. This confirms that the algorithm scales linearly up to 64 nodes. Beyond 64 nodes, the benefit of parallelism decreases a little. This is reasonable since the MR framework setup time may reduce the benefits of increasing the computation resources when the cube computation time is not long enough.

VI. Related Work

Data Cubes: Data cubes have been widely used as a powerful tool in traditional data warehousing and OLAP [2]. Many researches have been devoted to developing efficient cube materialization algorithms under a centralized system or a small number of nodes [15][16] [17] as well as on MR [18] [10] [19]. However, since the traditional data cubes model ignores the graph structure, these existing algorithms cannot be directly applied for graph cube computation.

Graph Summarization and Simplification: Our work is related to the research that aims to summarize or simplify graphs. For instance, [20] proposed to simplify a graph by preserving its skeleton with topological features. As another example, [21] summarized a graph by extracting simple graph characteristics like degrees and hot plots. Graph clustering [22] is another proposed method that summarize the graphs based on different partitioned regions. However, most of these studies focused on unlabeled graphs and it is hard to navigate w.r.t the underlying networks without analysis from different perspectives as well. [23] is the closest work to graph OLAP which also summarized attributed graphs. The work introduced SNAP (Summarization by grouping Nodes on Attributes and Pairwise relationship) and a less restrictive k-SNAP which summarizes the graph to k clusters based on the user input attributes, relationships and k. The difference between k-SNAP and *Pagrol* is the roll-up/drill-down manner. In k-SNAP, these operations are performed by decreasing/increasing the number of k which may lead different number of clusters. Differently, these operations in *Pagrol* are performed along the hieratical dimensions upon the attributed graphs.

Graph OLAP: [24] [25] proposed a high-level graph OLAP framework by classifying OLAP into two types: informational OLAP (I-OLAP) and topological OLAP (T-OLAP) where I-OLAP was mainly discussed. I-OLAP overlays a set of graphs into a summarized graph. Differently, T-OLAP summarizes the graph according to its topological structure over one single graph. Note that our Hyper Graph Cube belongs to T-OLAP, where the algorithm is not comparable with [24] [25]. [26] introduced a T-OLAP framework and discussed two techniques (T-Distributiveness and T-Monotonicity) for OLAP operation. However, none of these works provided any specific graph cube model. [27] provided a graph cube model for multi-dimensional networks where only the vertex has attributes. [28] provided a HMGraph framework to provide more dimensions and operation over multi-dimensional information networks which mainly focused on the vertex dimensions. However, in reality, the edges are always associated with attributes as well. Hyper Graph Cube model is a more general model than [27] [28] over attributed graphs where both vertex and edges are associated attributes. Several other works have also contributed to graph OLAP from different perspectives. For instance, [29] presented one demo giving one topic-oriented, integrated and multi-dimensional organizational solution over information networks. [30] proposed a graph data model as well as a query language to support n-dimensional computations extended SPARQL. However, none of the existing works provided any parallel graph cube materialization algorithms.

VII. Conclusion

In this paper, we extended the OLAP techniques to attributed graphs towards a better query and decision making support. We proposed *Pagrol*, a parallel graph OLAP system over lage-scale attributed graphs. To support graph OLAP, we first proposed a new conceptual graph cube model-*Hyper Graph Cube* over attributed graphs. It is an effective approach to aggregate the graphs for users to better understand the characteristics of the underlying massive graphs from different granularities and levels. In addition, we then provided, to the best of our knowledge, the first parallel MapReduce-based graph cube computation solution, *MRGraph-Cubing* over large-scale attributed graphs with various optimization techniques. Experimental results showed the effectiveness, efficiency and scalability of *Pagrol*.

Hyper Graph Cube faces the similar challenge as traditional data warehousing and OLAP does while handling the high dimensional datasets. The existing solutions for traditional high-dimensional OLAP (e.g. partial cube materialization [27], shell-fragment [17]) can be extended to tackle the challenge here. We will further explore this as our future work.

Acknowledgement

Zhengkui Wang and Kian-Lee Tan are partially supported by the MOE/NUS grant R-252-000-500-112. Divyakant Agrawal and Amr EI Abbadi are supported by the NSF grants III 1018637, CNS 1053594, and NEC Labs America and NSF Grant IIS-1135389.

References

[1] S. Sakr, S. Elnikety, and Y. He, "G-sparql: a hybrid engine for querying large attributed graphs," ser. CIKM '12, 2012, pp. 335–344.

[2] J. Gray, A. Bosworth, A. Layman, D. Reichart, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," in *ICDE*, 1996, pp. 152–159.

[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD Conference*, 2010, pp. 135–146.

[4] "Graphlab," http://graphlab.org/.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *OSDI*, ser. OSDI'12, 2012, pp. 17–30.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," ser. OSDI, 2004, pp. 137–150.

[7] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[8] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system," in *ICDM*, 2009, pp. 229–238.

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

[10] Z. Wang, Y. Chu, K.-L. Tan, D. Agrawal, A. E. Abbadi, and X. Xu, "Scalable data cube analysis over big data," in *CoRR*, arXiv:1311.5663 2013.

[11] N. G. de Bruijin, "Asymptotic methods in analysis," New York: Dover, 1981, pp. 102–109.

[12] "Hadoop," http://hadoop.apache.org.

[13] M. Kurant, M. Gjoka, Y. Wang, Z. W. Almquist, C. T. Butts, and A. Markopoulou, "Coarse-Grained Topology Estimation via Graph Sampling," in *WOSN*, Helsinki, Finland, 2012.

[14] "Snap," http://snap.stanford.edu/.

[15] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates," in *VLDB*, 1996, pp. 506–521.

[16] K. S. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in *SIGMOD*, 1999, pp. 359–370.

[17] X. Li, J. Han, and H. Gonzalez, "High-dimensional olap: A minimal cubing approach," in *VLDB*, 2004, pp. 528–539.

[18] K. Sergey and K. Yury, "Applying map-reduce paradigm for parallel closed cube computation," in *DBKDA*, 2009, pp. 62–67.

[19] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan, "Distributed cube materialization on holistic measures," in *ICDE*, 2011, pp. 183–194.

[20] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *SIGMOD Conference*, 2008, pp. 419–432.

[21] D. Chakrabarti and C. Faloutsos, *Graph Mining: Laws, Tools, and Case Studies*, ser. Synthesis Lec on Data Mining and Know. Dis., 2012.

[22] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities." *PVLDB*, pp. 718–729, 2009.

[23] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient aggregation for graph summarization," in *SIGMOD Conference*, 2008, pp. 567–580.

[24] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu, "Graph olap: Towards online analytical processing on graphs," in *ICDM*, 2008, pp. 103–112.

[25] C. Chen, X. Yan, F. Zhu, J. Han, and P. Yu, "Graph olap: a multi-dimensional framework for graph data analysis," *Knowl. Inf. Syst.*, vol. 21, no. 1, pp. 41–63, 2009.

[26] Q. Qu, F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Li, "Efficient topological olap on information networks," in *DASFAA (1)*, 2011, pp. 389–403.

[27] P. Zhao, X. Li, D. Xin, and J. Han, "Graph cube: on warehousing and olap multidimensional networks," in *SIGMOD*, 2011, pp. 853–864.

[28] M. Yin, B. Wu, and Z. Zeng, "Hmgraph olap: a novel framework for multi-dimensional heterogeneous network analysis." in *DOLAP*, 2012, pp. 137–144.

[29] C. Li, P. S. Yu, L. Zhao, Y. Xie, and W. Lin, "Infonetolaper: Integrating infonetwarehouse and infonetcube with infonetolap." *PVLDB*, vol. 4, no. 12, pp. 1422–1425, 2011.

[30] S.-M.-R. Beheshti, B. Benatallah, H. R. M. Nezhad, and M. Allahbakhsh, "A framework and a language for on-line analytical processing on graphs," in *WISE*, 2012, pp. 213–227.