A Distributed Approach for a Graph-Oriented Multidimensional Analysis

Benoît Denis
Université Catholique de Louvain
Louvain-la-Neuve, Belgium
benoit.denis.ph@gmail.com

Amine Ghrab, Sabri Skhiri
EURA NOVA R&D
Mont-Saint-Guibert, Belgium
{*amine.ghrab, sabri.skhiri*}*@euranova.eu*

***Abstract*—The importance of graphs as the fundamental structure underpinning many real world applications is no longer to be proved. Large graphs have emerged in various fields such as biological, social and transportation networks. The large volume of the network poses challenges to traditional techniques for storage and analysis of graph data. In particular, OLAP (On-Line Analytical Processing) analysis requires access to large portions of data to extract key information and to feed strategic decision making. OLAP provides multilevel, multiperspective views of the data. Most of current techniques are optimized for centralized graph processing. A distributed approach providing horizontal scalability is required in order to handle the analysis workload.**

In this paper, we focus on applying OLAP analysis on large, distributed graph data. We describe Distributed Graph Cube, our distributed framework for graph-based OLAP cubes computation and aggregation. Experimental results on large real-world datasets demonstrate that our method significantly outperforms its centralized counterparts. We also evaluate the performance of both Hadoop and Spark for distributed cubes computations.

***Keywords*-OLAP Cubes; Large Multidimensional Networks; Distributed Graph Processing**

I. Introduction

OLAP is a common analysis technique in data warehouses [1]. The data analysis follows a fact/dimension dichotomy to study a business fact according to a set of metrics. OLAP cubes embed aggregated data denoted as measures. Measures are the metrics for the analysis. Cubes are placed into the so-called multidimensional space, where dimensions are the factors influencing the values of the measures. In a relational setting, a cube for describing the sales of products would have the selling price as measure wile time, location and product would be the dimensions.
In the other hand, the greater expressive power of graphs encourages their use in extremely diverse domains, especially for modeling structural relationships. In biology graphs are used for modeling metabolic pathways, genetic regulations, known as the transduction signal network, and protein interactions in one single significant graph [2]. In Social networks they are used for modeling relationships between users and mining uncovered structural information such as churn prediction [3] or marketing strategies [4]. Other examples can also be found in transportation as illustrated by the UkGraph [5]. Those examples share a common challenge: they are composed by a significant number of nodes and edges. In the new era of Big Data, some large graphs can have over billions of vertices and trillions of edges. Therefore, two main questions rise: (1) How can we apply the functionally equivalent OLAP features to extract relevant information out of those graphs? (2) How can we deal efficiently with their significant sizes?

Traditional OLAP on relational data cannot efficiently support graph data analysis. Transformation of graph data into tabular representation is a complex process. It would also cause information loss, decrease performance and limit querying expressiveness. This proves the need to develop a new OLAP technique taking into account the specific nature of graph data.

On a graph data setting, Zhao et al. have designed an approach for constructing and aggregating OLAP cubes based on graph data [6]. They consider undirected, homogeneous, vertex-attributed graphs. To apply OLAP analysis on such a graph, vertex-attributes are considered as the dimensions. For instance, in a social network, a vertex would represent a member, dimensions could be the age, the sex and the location. An edge would join two vertices if the individuals they represent are friends. The graph is then viewed as a multidimensional network.
However, the method presented by Zhao et al., called Graph Cube, is challenged by a severe limitation: Graph Cube only considers a centralized approach when applying its materialization techniques. As demonstrated in Section II, the time complexity of Graph Cube algorithms depends on the size of the analyzed network. A key desirable quality in the OLAP workload is to execute quickly, with decent query throughput and response time [1]. The centralized approach of Graph Cube presents two weaknesses for large inputs:

1) The input network is read sequentially, linearly increasing the execution time of algorithms according to its size.
2) Graph Cube algorithms rely on a centralized in-memory store. However, if the main memory is limited and the input is massive, swapping issues can occur. Of course, a disk-based solution solves the problem but at the cost of a performance decrease due to slower disk accesses.

Those limitations were observed experimentally (see Section IV) and cause scalability challenges to Graph Cube.

To solve these problems, we designed a distributed approach that provides horizontal scalability and support large workloads. We also enabled parallel processing of the input data, as described in Section III. Our strategy do not rely on a high capacity main memory. The distributed approach solves the previously elicited limitations and outperforms both in size and speed the current solution.

The contributions of this paper are twofold. First, we propose a distributed approach for OLAP cubes computation and aggregation. Second, we provide a scalable, efficient and fault-tolerant implementation. The comparison with its centralized counterpart proves the superiority of our approach. Moreover, Distributed Graph Cube is open-source and can be reused as basis for further research.

The remaining of this document is structured as follows. In Section II, the original Graph Cube paper is presented. Multidimensional networks, aggregate network, the cuboid query (the Graph Cube main algorithm) and the materialization challenge are described. At the end of this section, the limitations of the centralized approach are highlighted. Opportunities to overcome those limitations with a distributed approach and our distributed variant for the cuboid query are described in Section III. In Section IV, our distributed approach is validated by comparing its performances with a centralized implementation. Further experiments on the distributed implementations are presented and results are interpreted. Finally Section V describes opportunities for future work concludes the paper.

II. Centralized Cuboid Query and the Materialization Challenge

The Graph Cube approach [6] was the first main contribution for filling the gap between the graph and data warehouse worlds, especially for performing functionally equivalent OLAP queries. In this section, we first present the Graph Cube theory and introduce few key concepts. Then, we describe the centralized `cuboid query`, which is the main focus of our contribution. Finally, materialization issues are highlighted.

A. Graph Cube principles

Graph Cube techniques are inspired from the OLAP theory [7]. In traditional OLAP, multidimensional relational data is represented through data cubes where some dimensions can be aggregated. A cell on the cube contains a numerical value computed by an aggregate function, such as `COUNT(.)`, `SUM(.)` or `AVERAGE(.)`. Analysts explore the multidimensional data by performing queries, such as `roll-up`, `drill-down` and `slice-and-dice`. A roll-up makes the vertical transition to a coarser-grained cube where the information is less aggregated. Drill-down does the opposite, transitioning to a finer-grained view. Slice-and-dice allows to fix a dimension in the data cube.

In Graph Cube, `multidimensional networks` and `aggregate networks` are the equivalents of data cubes and their aggregated forms. The traditional OLAP queries are transposed into their corresponding queries within the Graph Cube model. The concepts of multidimensional network, aggregate network and queries are developed hereafter.

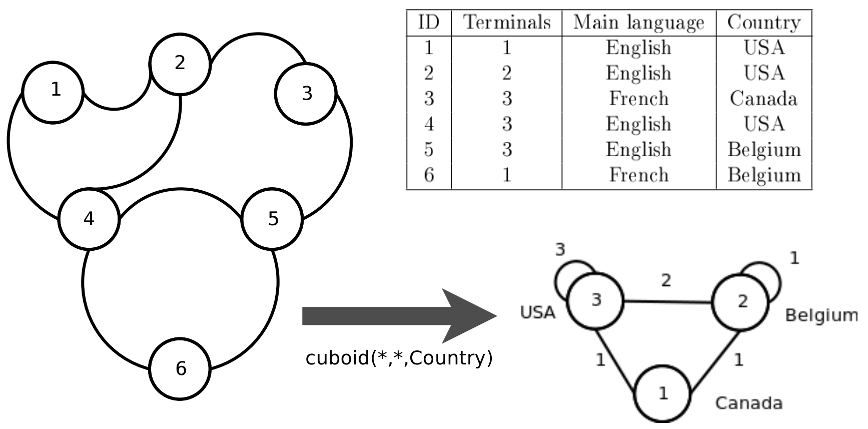
A multidimensional network is an undirected graph where the vertices are associated with a set of dimensions representing the vertices attributes. Formally, the authors of Graph Cube defined a multidimensional network as follows :

Definition 1: A multidimensional network, N, is a graph denoted as $N = (V, E, A)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges and $A = A_1, A_2, ..., A_n$ is a set of n vertex-specific attributes, i.e., $\forall u \in V$, there is a multidimensional tuple $A(u)$ of u, denoted as $A(u) = (A_1(u), A_2(u), ..., A_n(u))$, where $A_i(u)$ is the value of u on i-th attribute, $1 \leq i \leq n$. A is called the dimensions of N.

We slightly modified the original model to allow the network to be weighted. Therefore two functions have to be defined additionally, $W_V : V \rightarrow \mathbb{N}$ and $W_E : E \rightarrow \mathbb{N}$, with W_V and W_E associating weights to vertices and edges, respectively.

To illustrate the concept of multidimensional network, an example is provided at the left of Figure 1. Vertices represent airports which are linked each other when a there is connecting flights between them. All weights are assumed to be equal to 1, i.e. $W_V = 1$ and $W_E = 1$. Three dimensions characterize the airports : the number of terminals, the main language and the country where it is located. Similarly to traditional OLAP where data cubes can be aggregated in order to present a summarized view of the data, aggregate networks are derived from the base multidimensional network in which some dimensions are aggregated. An aggregate network computed from the airport network, where all dimensions except `Country` are aggregated, is shown at the right side of Figure 1. Vertices and edges weights were computed with the `SUM(.)` aggregate function[1]. Such aggregate network corresponds to a `cuboid`. **A cuboid** is an aggregation of a multidimensional network. For instance, the aggregate network of Figure 1 is mapped to the cuboid $(*, *, Country)$, where $*$ designates a fully aggregated dimension. The size of a cuboid is $\#V + \#E$, V and E being the number of vertices and the number of edges of the corresponding aggregate network, respectively. The dimension of a cuboid, denoted `dim`, is the set of the non-aggregated dimensions of the related aggregate

[1]In the Graph Cube paper, `COUNT(.)` is used as the default aggregate function. As we allow the multidimensional network to be weighted, `SUM(.)` is a more adapted aggregate function.



ID	Terminals	Main language	Country
1	1	English	USA
2	2	English	USA
3	3	French	Canada
4	3	English	USA
5	3	English	Belgium
6	1	French	Belgium

Figure 1. A Cuboid Query on a Network of Airports

network. There are 2^{N-1} cuboids that can be generated from a multidimensional network, N being the number of dimensions. They are partially ordered in a lattice, where, for two cuboids A and A', A is an ancestor of A' if $dim(A) \subseteq dim(A')$ and inversely A' is a descendant of A. Traditional OLAP queries have an equivalent in the context of Graph Cube. For instance, from the $(*, *, country)$ cuboid, one could perform a `drill-down` to the finer-grained aggregate network represented by $(terminals, *, country)$ in order to see if large airports are more connected to other countries than smaller ones. A `roll-up` to the coarse-grained network from the cuboid $(terminals, *, *)$ can be made to determine whether considering only the airport's size provides information of interest. `Slice-and-dice` consists in looking only at portion of the aggregate network.

B. Centralized cuboid query

A cuboid query is a function that, given a multidimensional network and a cuboid, computes the corresponding aggregate network. An example of a cuboid query is given by Figure 1.

Cuboid queries are essential in the Graph Cube theory. Indeed, all the previously described OLAP-like queries are processed from aggregate networks produced by those queries. The pseudo-code of a centralized cuboid query is formulated in Figure 2. We provided minor modifications to the algorithm presented in the original paper, such as summing the weight together, implementing the aggregate function `SUM(.)` instead of the original `COUNT(.)`.

The principles can be described as follows : for each vertex from the input network, its attributes are aggregated according to the cuboid by the `cub` function. For instance, `cub` would compute $(*, *, Belgium)$ from $(3, English, Belgium)$ given the cuboid $(*, *, country)$. This aggregation is then searched in the Γ hash table. If it is not already there, the condensed vertex is created

```
Require: $\mathbb{N}$ is a multidimensional network
    $(V, E, A, W_V, W_E)$.
Require: $cub$ is a given cuboid function that, given a vertex,
    computes the identifier of the aggregate vertex.
Require: $aggregate$ is a function that aggregates a vertex
    given a cuboid
  procedure CUBOID($\mathbb{N}$ : Multidimensional Network)
      Initialize hash structure $\Gamma$
      Initialize hash structure $\Delta$
      for all $v \in V$ do
          if $\Gamma(cub(v)) = NULL$ then
              $aggregateVertex = aggregate(v, cub)$
              $W_{V'}(aggregateVertex) = W_V(v)$
              $\Gamma(cub(v)) = aggregateVertex$
          else
              $W_{V'}(cub(v)) = W_{V'}(cub(v)) + W_V(v)$
          end if
      end for
      for all $e \in E$ do
          if $\Delta((cub(e.v1), cub(e.v2)) = NULL$ then
              $aggrEdge = (\Gamma(cub(e.v1)), \Gamma(cub(e.v2)))$
              $W_{E'}(aggrEdge) = W_E(e)$
              $\Delta((cub(e.v1), cub(e.v2)) = aggrEdge$
          else
              $W_{E'}((cub(e.v1), cub(e.v2)) =$
                  $W_{E'}((cub(e.v1), cub(e.v2)) + W_E(e)$
          end if
      end for
      return The aggregate network with $V'$,
       the values of $\Gamma$ and $E'$, the values of $\Delta$
  end procedure
```

Figure 2. Centralized cuboid query

by the aggregate function and the entry is added to the structure. aggregate acts similarly to cub and it effectively creates the condensed vertex. Its weight corresponds to the weight of the original vertex. Otherwise, the weight of the condensed, stored in the hash table, is summed with the weight of the current vertex. A similar approach is adopted for aggregating edges.
Assuming that cub and aggregate have a time complexity of $O(1)$[2], a cuboid query executes in $\theta(\#V + \#E)$, $\#V$ and $\#E$ being respectively the number of vertices and edges in the input network.
This centralized approach has scalability issues due to the two limitations presented in the introduction: (1) the input is processed sequentially and (2) the algorithm relies on hash tables. If we combine the sequentiality of the cuboid query and its linear time complexity, we expect the execution time to linearly increase according to the multidimensional network size, which is an important issue for large graphs. Indeed, as the hash tables have a size proportional to the cuboid size, if the main memory is limited and the cuboid is large, swapping issues will occur. In order to resolve this problem we propose to process the graph in parallel with multiple workers in order to increase the performances and to avoid the use of this large centralized hash table. We describe this approach in the Section III.

C. The materialization challenge

It has been shown in [6] that an aggregate network is not larger and, in practice, often much smaller than the base multidimensional network. Moreover, a cuboid query can be executed from one of its descendant without loss of information. Therefore, a key optimization technique is to find the smallest materialized descendant in order to use it as input to the cuboid query. The time complexity remains in $\theta(\#V + \#E)$, but in the other hand, the size of the input aggregate network is expected to be much smaller than the size of the initial network.
We explained previously that there are $2^N - 1$ aggregate networks that can be materialized from a multidimensional graph, N being the number of dimensions. For a large number of dimensions, this exponential number is at the root of the materialization challenge. As a result, we need a materialization strategy that finds which aggregate networks need to be materialized in order to be used as descendant for maximizing the performances of cuboid queries. A greedy [8] strategy and the MinLevel techniques have been presented and assessed by Zhao and al. in Graph Cube. MinLevel consists in starting materializing aggregate networks at a determined dimension level (for instance 3 or 4)[3] and continuing to materialize all the cuboids at that level. Then, the dimension level is increased one by one and it continues until the desired number of aggregate networks have been materialized or the base cuboid has been reached. We chose to implement MinLevel based on the results presented in the Graph Cube paper showing that this materialization strategy outperforms the greedy one.

[2]With a naive approach, those functions are executed in $O(N)$, N being the number of dimensions and can be considered as constant compared to the network size, so it is equivalent to $O(1)$.

```
Require: $cub$ is a given cuboid
Require: $aggregate$ is a function that aggregates a vertex
  or an edge given a cuboid

  procedure MAP(key : VertexOrEdge, value : Weight)
      $aggrVertexOrEdge = aggregate(key, cub)$
      $emit(aggrVertexOrEdge, value)$
  end procedure

  procedure REDUCE(key : VertexOrEdge, values : Iterator[Weight])
      $aggrWeight = 0$
      for all $weight \leftarrow values$ do
          $aggrWeight = aggrWeight + weight$
      end for
      $emit(key, aggrWeight)$
  end procedure
```

Figure 3. Cuboid query with Map-Reduce

III. A Distributed Cuboid Query Implementation

Scalability limitations of the centralized cuboid query were discussed in the previous section. A centralized approach with vertical scalability has the disadvantage to be eventually limited by the CPU and the RAM capacities.
We propose to process the cuboid query in parallel on several workers. A distributed cuboid query can be achieved with the following steps, assuming the SUM(.) aggregate function :

1) The input multidimensional network (or aggregate network) is splitted between the workers (it is usually automatically managed by a distributed file-system such as HDFS [10]).
2) Locally, every vertex and edge of the input chunks are aggregated and their original weight is kept.
3) Still locally, the aggregated elements with the same aggregated attributes are merged together by summing their weights.
4) The aggregated elements are sent over the network to a set of workers. The destination is defined by a partition function, such as $elemID\%n$. $elemID$ is an identifier representing the aggregated attributes of

[3]Analysts would hardly often analyze more than few dimensions at the same time because it is hard to interpret the results when mixing too many dimensions [9]

a graph element and n is the number of workers. It ensures that all the aggregated elements with the same attributes are received by an unique worker.

5) Step 3 is repeated.
6) The final output is written on a distributed file-system.

Note that steps 2 and 3 correspond to the centralized cuboid query algorithm. However, instead of having one central machine that processes the whole input sequentially, there are n units working in parallel on fractions of the multidimensional network. This approach provides a horizontal scalability where performances can be improved by adding more workers. As the local input is smaller, the size of the in-memory hash tables is also likely to be decreased[4].

The distributed algorithm for a cuboid query fits the Map-Reduce model [11]. Map-Reduce is a programming abstraction where a programmer specifies a `map` and a `reduce` function in order to perform distributed computations. A `map` takes as parameter a key-value pair and outputs zero, one or more intermediary key-value pairs. The `reduce` parameters are an intermediary key and an iterator of associated intermediary values. `Reduce` aggregates the values to typically produce zero or one final key-value pair that is written to the output file. A Map-Reduce framework handles automatically the parallelization of the `map` and the `reduce` tasks, the shuffling of the intermediary results and the management of the cluster.

A cuboid query can be expressed in terms of Map-Reduce as shown in Figure 3 in which we use the `aggregate` function[5] before emitting the output of the mapper. In our implementation, `aggregate` does not need to access the network.

After having evaluated Hadoop and Spark architectures and performances, we chose Spark[6]. The comparison between those two frameworks is out of the scope of this paper. Spark is a Scala library with a flexible API suited for real-time distributed querying, as we require to implement Distributed Graph Cube. Our implementation is available on Github as an open-source software[7]. Spark supports the Map-Reduce primitives and automatically sets `combiners` [12]. A combiner performs locally the `reduce` function before sending the intermediate key-value pairs over the network. Therefore, the execution of a cuboid query with Spark matches exactly the algorithm described at the beginning of this section. Indeed, HDFS handles steps 1 and 6. Step 2 corresponds to the `map`, and step 3 and 5 corresponds to the `reduce`. Finally, step 4 is handled by Spark in the `shuffle` phase.

When performing cuboid queries using Spark, persistence levels have to be defined for the data sets (in this case, the multidimensional network and the aggregate networks). Spark offers different alternatives such as keeping the data in the cache, possibly in a serialized form, or spilling it on the disk, with an optional replication factor for fault-tolerance. Our default policy is to persist aggregate networks computed by the `MinLevel` materialization strategy on disk. Indeed, we suppose that many aggregate networks are materialized and it would overload the RAM to keep them all in main memory. Otherwise, if the size of the distributed main memory is large enough, it is possible to change the persistence level to the main memory. When an OLAP query is performed on an aggregate network, such as roll-up, it is automatically cached in order to optimize the response time of subsequent queries on this aggregate network.

IV. Experimentation

Several empirical experiments were realized on the Spark-based implementation. We also implemented a centralized cuboid query using Java hash tables[8] in order to validate our distributed approach. This section presents the experimental setup, the testing protocols and the interpretation of the results of the various experiments.

First we compare the performances of executing cuboid queries with the centralized program and the distributed implementation. Going further, we evaluate the scalability of our distributed approach. Then, the cuboid query efficiency is studied and finally we conduct tests on materialized aggregate networks. The next paragraph introduces our experimental setup.

A. Experimental setup

Experiments were conducted on a cluster of six virtual machines running on Linux Ubuntu. Each machine had a Intel Xeon X3440 quadri-core CPU (but only one core cadenced at 2.53 GHz was allocated to the virtual machine), 60 GB of stable storage and 512 MB of RAM were used by Distributed Graph Cube.

The multidimensional network was extracted from the IMDB data-set[9]. It has about 200 000 vertices and more than 30 millions edges. Vertices represent movies and are linked by an edge if they have at least one male actor in common. Movies are characterized by a 7-dimensional vector. The dimensions are the release year, the spoken language, the category (drama, adventure, romance, etc.), the IMDB rank (an integer from 1 to 10), the country location, the production company and the certification (all, 13+, unrated, etc.).

[4]The hash table size is ensured to be smaller than the central one if there are less elements in the input split than elements in the output aggregate network.

[5]The `aggregate` function is the same as the homonym function in the centralized algorithm.

[6]http://spark-project.org/

[7]https://github.com/taguan/SGraphCube

[8]https://github.com/taguan/CGraphCube

[9]http://www.imdb.com/interfaces

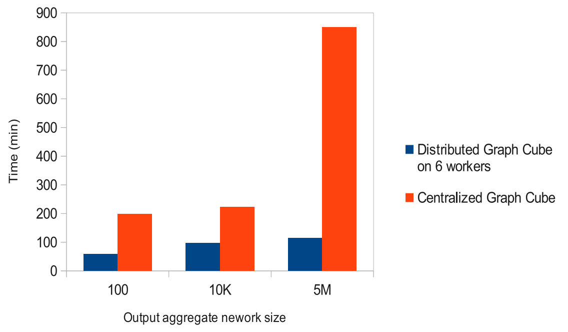


Figure 4. Centralized Graph Cube vs Distributed Graph Cube (Spark)

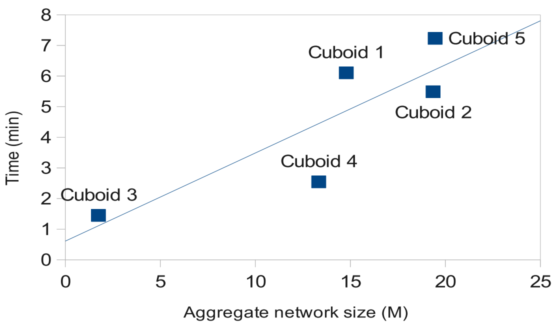


Figure 6. Effect of the size of the aggregate network to be materialized on the materialization time

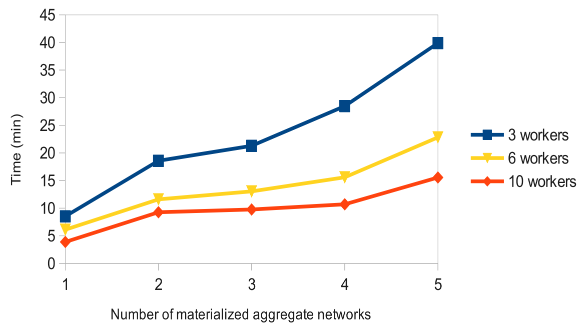


Figure 5. Distributed Graph Cube scalability

Note that the range's size of the possible values are quite different according to the dimension. For instance, the rating has only ten allowed values whereas there are many more than ten production companies. It will have an important impact on the aggregate network's sizes, as demonstrated in our experiment results.

B. *Distributed approach validation*

We listed in the introduction and in Section II the limitations of the centralized approach for computing cuboid queries. First, the centralized algorithm is sequential and its time complexity depends on the input size. This leads to scalability issues when the input multidimensional network becomes large. However, we propose to split the input between several workers and to process it in parallel. It provides a horizontal scalability allowing to decrease the querying time of large networks by adding more machines. Second, the centralized cuboid query relies on hash tables that contain one entry per condensed vertex and edge in the computed aggregate network. If the output is larger than the RAM, Centralized Graph Cube is expected to run into memory issues.
In order to validate our assumptions and the relevance of our solution, we executed some cuboid queries from the base network using both the centralized and the Spark-based distributed implementations. Results are presented in Figure 4. We can see that Distributed Graph Cube over six workers is faster than the centralized approach. This validates our first assumption, processing the input in parallel leads to better performances. When the size of the output aggregate network reaches a size limit (in our setup, around five millions entries), the performances of Centralized Graph Cube collapse. It is due to the internal Java hash tables becoming too large and that cannot be processed efficiently in the main memory. It validates the second assumption. Of course, it is possible to avoid this performance collapse by using a disk-based key-value store instead of an in-memory hash table. However, as accessing the disk is slower than accessing the main memory, this solution would increase the performance gap between the distributed and the centralized approach. Note that the cuboid query execution time of Distributed Graph Cube increases according to the output size. This phenomena is explained in the next subsection.
The next experiment aims at demonstrating the scalability of our application. The materialization times for computing incremental numbers of aggregate networks was measured with a varying number of Spark workers (one worker per virtual machine). Performances presented in Figure 5 show an expected horizontal scalability. The computation speed increases close to linearly with the number of workers. We measured an average speedup of $10,46\%$ when adding a worker for computing an aggregate network. This result demonstrates the relevance of our contribution : the distributed implementation overcomes the limitations of a centralized approach and provides a horizontal scalability.

C. *Cuboid query performance analysis*

It is visible from Figure 4 and Figure 5 that the execution time needed to materialize different aggregate networks varies. The aim of the Graph Cube strategy is to choose the smallest descendant in order to decrease the running time of

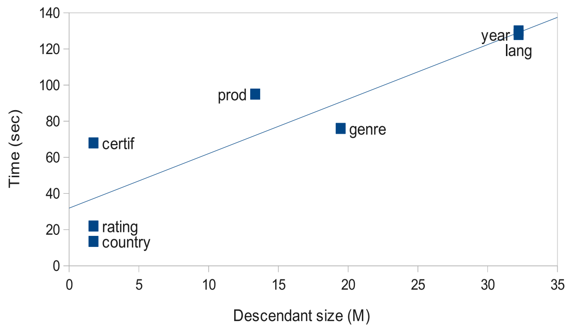


Figure 7. Descendant's size influence on the querying time

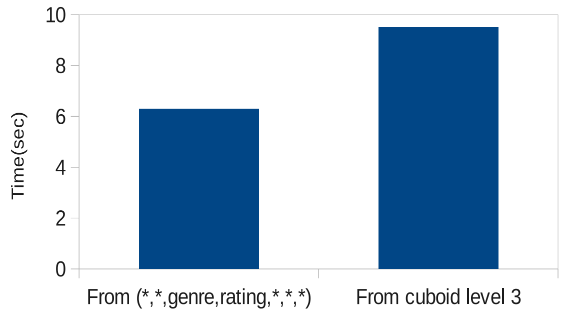


Figure 8. Roll-up execution time

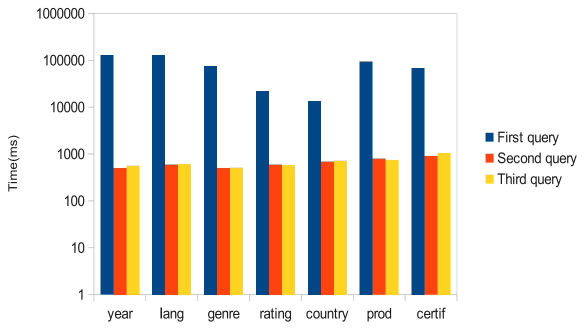


Figure 9. Performances of successive queries

a cuboid query which is in $\theta(n)$, n being the descendant's i.e., the sum of all the nodes and the edges. However, in the previous experiments, the base cuboid was systemically selected as the nearest descendant and n remained the same, therefore a nearby constant execution time would be expected. In this case, how can we explain that some aggregate networks are computed faster than others?
As mentioned earlier, in the IMDB network, some dimensions do aggregate much more than others. An explaination would be that computations resulting in the smallest aggregate networks take the shortest time to execute. Behind the scene, when performing a `reduceByKey` operation, Spark shuffles the output key-value pairs from the `map` stage before reducing the values with same keys. If the resulting aggregate network is small, there are less `reduce` procedures to launch which could result in a faster overall execution time. Also, `reduce` jobs are executed locally by a combiner in order to minimize the number of intermediary values sent over the network. If the multidimensional network is aggregated a lot, less data is transmitted, resulting in a shorter execution time. Finally, the time needed to sort the intermediary keys and to write the output on disk is proportional to the output aggregate network's size. Those assumptions are confirmed in Figure 6 where the time to materialize cuboids is shown in regard to the cuboid's size.
The next experiment consists in executing the seven uni-dimensional cuboid queries after having materialized the five tri-dimensional aggregate networks of the previous experiment. Figure 7 shows the measures of the queries execution time compared to the descendant's. The trend line confirms that cuboid queries can be computed much faster from a smaller descendant. This result was expected as the cuboid algorithm has a complexity in $\theta(n)$. The `year` and `language` aggregate networks have no materialized descendants and therefore were computed from the base multidimensional network. As predicted, they are computed from the most time expensive queries. Such a situation should be avoided as much as possible in a production environment by materializing more aggregate networks of superior level. On the other hand, `country` and `rating` are computed from the smallest materialized aggregate network and are processed in the best execution time. The `certificate` aggregate network is materialized from the same smallest descendant but is computed in much more time. The explanation is related to the conclusion of the previous experiment, this aggregate network is significantly larger than the `country` and the `rating` networks. Similarly, the `production` and the `genre` cuboid queries derive significantly from the trend line, because the earlier produces a very large network as opposite to the later.

D. *Caching strategy assessment*

Figure 8 shows the performance's improvement due to our strategy of reusing previously queried aggregate networks as descendant for subsequent cuboid queries. In this example, a user would have analyzed the level two aggregate network where the `genre` and the `rating` dimensions are expended. At the same time, Distributed Graph Cube caches this network for future utilization. Then, the user would want to perform a query similar to the

OLAP query called `roll-up`, navigating to the coarser-grained uni-dimensional network where `genre` is the only non aggregated dimension. As shown in Figure 8, there is a significant speedup resulting from having persisted the previous aggregate network and from using it as descendant instead of the higher level materialized aggregate networks. It is due to the fact that aggregate networks are never larger and often a lot smaller than their descendants, making in this case the two-dimensional network a minimal descendant.
Figure 9 shows the time needed to perform three consecutive `count`[10] queries on each uni-dimensional aggregate network, just after the materialization step. Note the logarithmic scale. The first query is up to 100 times slower than the second and third ones. This is expected as the first query does not only perform the `count` but also the cuboid query in order to compute the aggregate network. Again, this result shows the importance of persisting a resulting network in order to support fast real-time querying. It also demonstrates that by using Spark, even with our limited experimental cluster, Distributed Graph Cube can achieve sub-second response times.

V. Conclusion

In this paper, we proposed a distributed approach for graph-based OLAP cubes computation and aggregation. Our work extends state of the art approach limited to centralized architecture. Our approach is designed for distributed environments and solves issues related to the growing size of the data by providing a horizontal scalability.
We designed a distributed algorithm for the cuboid query based on the well-known Map-Reduce model. We have implemented it on top of Spark. We demonstrated the effectiveness of our distributed approach compared to a centralized implementation. We also validated the choice of Spark as our middleware framework by studying its scalability and its efficiency compared to Hadoop. Finally, we analyzed the behavior of distributed cuboid queries and our materialization strategy.
In the current paper, we took as an assumption that graphs are homogeneous and static. As future work we plan to extend this framework to support heterogeneous graph data, and to enable analysis on evolving networks. Moreover, extending the Graph Cube model to support hierarchical aggregation should be considered. Also, we only implemented the `MinLevel` materialization strategy and it would be interesting to assess more strategies.

References

[1] S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology," in *SIGMOD*, vol. 26, 1997, pp. 65–74.

[2] E. Zimnyi and S. Skhiri dit Gabouje, "Semantic visualization of biochemical databases," in *Semantics for GRID Databases, Proceedings of the International Conference on Semantics for a Networked World, ICSNW2004*, ser. Lecture Notes in Computer Science. Paris, France: Springer-Verlag, June 2004, no. 3226, pp. 199–214.

[3] K. Dasgupta, R. Singh, B. Viswanathan, D. Chakraborty, S. Mukherjea, A. A. Nanavati, and A. Joshi, "Social ties and their relevance to churn in mobile telecom networks," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, ser. EDBT '08. New York, NY, USA: ACM, 2008, pp. 668–677.

[4] M. Gomez Rodriguez, J. Leskovec, and A. Krause, "Inferring networks of diffusion and influence," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '10. New York, NY, USA: ACM, 2010, pp. 1019–1028.

[5] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, Nov. 2008.

[6] P. Zhao, X. Li, D. Xin, and J. Han, "Graph Cube: On Warehousing and OLAP Multidimensional Networks," in *SIGMOD '11 Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 853–864.

[7] C. S.Jensen, T. B. Pedersen, and C. Thomsen, *Multidimensional Databases and Data Warehousing*. Morgan and Claypool, 2010.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 205–216.

[9] X. Li, J. Han, and H. Gonzalez, "High-dimensional OLAP: a minimal cubing approach," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 528–539.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th anniversary*, vol. 51, pp. 107–113, January 2008.

[12] T. Das, M. Zaharia, M. Chowdhury, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for inmemory cluster computing," in *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[10]A `count` sums the number of vertices and edges of a network.