New Approach based on Association Rules for Building and Optimizing OLAP Cubes on Graphs

Redouane LABZIOUI1, Khadija LETRACHE2, Mohammed RAMDANI3
Informatics Department, LIM Laboratory
Faculty of Sciences and Techniques of Mohammedia
University Hassan II, Casablanca, Morocco

Abstract—The expansion of data has prompted the creation of various NoSQL (Not only SQL) databases, including graph-oriented databases, which provide an understandable abstraction for modeling complex domains and managing highly connected data. However, to add graph data to existing decision support systems, new data warehouse systems that consider the special characteristics of graphs need to be developed. This work proposes a novel method for creating a data warehouse under a graph database and demonstrates how OLAP (Online Analytical Processing) structures created for reporting can be handled by graph databases. Additionally, the paper suggests using aggregation algorithms based association rules techniques to improve the efficiency of reporting and data analysis within a graph-based data warehouse. Finally, we provide a Cypher language implementation of the suggested approach to evaluate and validate our approach.

Keywords—NoSQL; graph-oriented databases; data warehouse; OLAP; aggregation algorithms; association rules; cypher language

I. INTRODUCTION

Modern databases have been considerably altered by the expansion of data. NoSQL databases have expanded as a result of these new demands and now come in a wide range of models [1], including key-value, document, column, and graph.

In particular, graph-oriented databases are one of the most-known various of NoSQL systems; they have attracted a lot of attention and popularity. Graph-oriented databases are a fundamental form that offer an understandable abstraction to model numerous complicated domains, manage highly connected data, and run sophisticated queries over them [2] [3].

Currently, many businesses and entrepreneurs are interested in developing business intelligence systems on graph databases to take advantage of its benefits. Enterprises are also interested in expanding their OLAP analysis to include the new forms of data because OLAP technology is currently widely used[4]. However, despite the growing interest in graph-based data warehouses and their potential benefits, there is a lack of research addressing the challenge of creating an optimized OLAP model under graphs, specifically focusing on selecting the most relevant OLAP aggregations to effectively meet the diverse user's needs. This gap calls for further investigation and exploration to bridge the divide between the advantages of graph-based data warehousing and the need for efficient and user-centric aggregation selection techniques. The aim of this study is to create new data warehouse under graph database systems that consider the special characteristics of graphs, this work also aims to optimize the efficiency of reporting and data analysis within this graph-based data warehouse by employing user-centric aggregation techniques that select the most relevant OLAP aggregations to meet the diverse needs of users effectively. To address this issue, it is possible to use aggregation algorithms that automatically select the most relevant aggregations for the cube. These aggregation algorithms are often based on machine learning and data mining techniques to identify the best possible aggregations based on raw data. The use of these algorithms can help reduce cube build time and improve the accuracy of analysis results. In this context, we demonstrate how OLAP structures created for reporting can be handled by graph databases, additionally, We provide new approach to optimize OLAP cube using the association rules algorithm.

The remainder of this paper is organized as follows. In Section II, we present some works of the literature reviews related on graph data warehouse. In Section III, we give a background overview of our approach. In Section IV, we describe the implementation of our approach as well as a case study to assess it. Section V concludes this paper and suggest future research directions.

II. RELATED WORK

Many approaches were proposed in the literature as a result of the growing interest in combining graph databases and business intelligence technology in recent years. In [5], The authors have proposed a new concept Graph Cube, a new data warehouse model that supports OLAP queries in large multidimensional networks, in the Graph Cube the dimensions are based on the attributes of the nodes, while the computed measures represent the aggregations of these node attributes. The Graph Cube approach have some limitations in analyzing dynamic or evolving graphs, where the structure of the graph changes over time. Moreover, the accuracy and reliability of the analysis results may be affected by data quality issues such as missing or erroneous data. In [6], The author introduce a GraphAware Framework for Neo4j, which enables the precalculation and storage of node information in graphs. For instance, the framework can compute the number of friends in a social network and store the result for efficient querying. The GraphAware Framework also supports the analysis of node degrees in the graph, which can be useful for identifying important or influential nodes. The GraphAware Framework is focused on precalculating and storing node information, which may be limited in scope and may not capture the
full complexity of the graph data. For example, some graph analytics tasks may require more sophisticated computations that go beyond simple node attributes, such as graph centrality measures or community detection. In [7], the authors proposed to use the graph structure as a basis for OLAP queries; this approach relies on using the performance and efficiency of the Neo4j graph database to store and query OLAP queries. In this model, dimensions and measures are transformed into nodes. The connection between dimensions and measures is done through arcs of the graph. For hierarchical dimensions are also stored in nodes and linked together by hierarchical relationships. The approach is limited to using only the snowflake model, which may not be suitable for all types of data. This may restrict the flexibility and adaptability of the approach. In [8], the authors' goal is to compare the execution times of the identical OLAP queries in the relational and graph databases by using a MusicBrainz database database warehouse in a PostgreSQL relational environment and then implementing the same decision model in Neo4j. The authors only tested their approach on a single dataset (the MusicBrainz database), which may limit the generalizability of their results. Additionally, they only considered a specific decision model and did not explore the potential impact of different OLAP queries or decision models on the performance of the two database types. Furthermore, while the results of the study showed that the graph database outperformed the relational database in terms of execution time, the authors did not provide a detailed analysis of the factors that contributed to these results. This lack of analysis makes it difficult to fully understand the advantages and disadvantages of each database type for OLAP queries. In [9], the authors define a set of transformation rules that can transform conceptual models into graph-oriented models. They have defined four transformation rules, namely: fact transformation, Name and Identifier of Dimension Transformation, Hierarchies Transformation, Transformation of the relation between Fact and Dimension. Still within the framework of Datawarehouse modelling in a NoSQL graph database, the authors propose a conceptual mapping between a multidimensional schema and a graph-oriented NoSQL model. The authors chose to concentrate on proving the viability of their approach rather than providing any experimental campaign to validate it. In [10], the authors propose to integrate NoSQL Graph-oriented Data into Data Warehouses as a solution to tackle Big Data challenges. The paper introduces a new approach called "Big-Parallel-ETL" that adapts the classical ETL (Extract-Transform-Load) process with Big Data technologies, leveraging the efficiency of the MapReduce concept for parallel processing. However, this work does not address OLAP aggregations. The authors in [11], suggest a set of guidelines for creating a graph database model from a multidimensional data model (MDM2G). Then the authors compare the performance of the two star and snowflake designs in the graphs and relational databases, in terms of dimensionality and size. After doing this comparison, the authors concluded that a graph implementation of a data warehouse with multiple tables is more effective than a relational implementation, and that a star model performs similarly to a snowflake model in graph databases. The study does not provide a detailed description of the conversion rules and does not present any experimental results or validation of the proposed approach. In [12], the authors proposed employing two alternative logical models, equivalent to the ROLAP (Relational Online Analytical Processing) and MOLAP (Multidimensional Online Analytical Processing) models, to create OLAP engines within a graph database. They specify a set of guidelines for mapping these models from the multidimensional model. Additionally, they suggest an aggregation technique for constructing the lattice of cuboids from a data warehouse. However, the choice of aggregations is random and imprecise, which makes the model unoptimized and burdens the graph with several unnecessary nodes. The authors in [13], suggest an approach founded on a multi-version evolutionary schema model. Data instances corresponding to various schema versions are stored in a graph data warehouse. A meta-model is utilized to manage these warehouse schema versions. Additionally, they introduce evolution functions at the schema level. To validate their approach, they implement a software prototype and conduct a case study that demonstrates queries on schema versions, cross-queries, and the runtime performance of their approach. However, the impact of the multi-version approach on OLAP aggregations is not addressed in this study.

The Table I provides a comprehensive summary of the literature review, highlighting the key findings and identified gaps in the research.

All of the cited works provide an important context for the implementation of decision systems using graph databases. However, most of these works focus on converting relational data warehouses into graph databases or applying traditional business intelligence methods, which can limit the advantages of using graph databases. Our proposed approach is different and relies on the properties of graphs to implement data warehouses. It highlights the importance of studying a model that optimizes the choice of OLAP aggregations to enhance the graph cube’s performance. By selecting the optimal set of aggregations for a graph cube, OLAP query performance can be significantly improved, resulting in faster query response times and more efficient use of system resources. This can enable users to analyze larger volumes of data more quickly and accurately, leading to more informed decision-making. Moreover, reducing the computational resources required to execute OLAP queries can result in cost savings for organizations that need to process large amounts of data.

III. OUR APPROACH AND BACKGROUND INFORMATION

A. Background Information

Graph Oriented Database: Store data entities as nodes and entity relationships as edges. A periphery always has a start node, an end node, a type, and a direction. A node can describe relationships, actions, parent-child ownership, etc. The number and type of relationships a node can have are unlimited.

A property graph is defined as
TABLE I. LITERATURE REVIEW

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Findings</th>
<th>Gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Zhao et al.</td>
<td>Graph Cube: A new paradigm for Data Warehouse (DW) that supports OLAP queries in large multidimensional networks, with dimensions based on node attributes and computed measures representing aggregations.</td>
<td>Limitations in analyzing dynamic or evolving graphs and potential data quality issues.</td>
</tr>
<tr>
<td>2013</td>
<td>Bachman</td>
<td>GraphAware Framework for Neo4j enables pre-calculation and storage of node information for efficient querying, but may not capture the full complexity of graph data.</td>
<td>Limited in handling more sophisticated computations beyond simple node attributes.</td>
</tr>
<tr>
<td>2014</td>
<td>Castelltort et al.</td>
<td>Proposes using graph structure as a basis for OLAP queries, but limited to the snowflake model, reducing flexibility.</td>
<td>May not be suitable for all types of data.</td>
</tr>
<tr>
<td>2019</td>
<td>Vaisman et al.</td>
<td>Comparing the execution timings of identical OLAP queries in relational and graph databases reveals that the graph database provides superior performance.</td>
<td>Lack of detailed analysis of factors contributing to the performance difference.</td>
</tr>
<tr>
<td>2020</td>
<td>Sellami et al.</td>
<td>Define transformation rules to convert conceptual models into graph-oriented models.</td>
<td>No experimental campaign to validate the approach.</td>
</tr>
<tr>
<td>2021</td>
<td>Soussi</td>
<td>Propose parallel loading based integration of NoSQL graph-oriented data into data warehouses.</td>
<td>Doesn’t address OLAP aggregations.</td>
</tr>
<tr>
<td>2022</td>
<td>Akid et al.</td>
<td>Creating graph data models based on multidimensional data and comparing star and snowflake designs in graphs and relational databases.</td>
<td>Lack of detailed conversion rules and experimental validation.</td>
</tr>
<tr>
<td>2022</td>
<td>Khalil et al.</td>
<td>Propose alternative logical models for OLAP engines within a graph database and an aggregation technique for constructing the lattice of cubes from a data warehouse.</td>
<td>Unoptimized aggregations and unnecessary nodes in the graph.</td>
</tr>
<tr>
<td>2023</td>
<td>Benhissen et al.</td>
<td>Propose an approach based on a multi-version evolutionary schema model in a graph data warehouse.</td>
<td>Doesn’t address the impact of multi-version approach on OLAP aggregations.</td>
</tr>
</tbody>
</table>

A dimension consists of a set of attributes representing different levels of granularity on the data to be analyzed (measures).

A dimension, denoted $D_i \in D^S$, This is characterized by $(N^D, A^D, H^D)$ where:

- $N^D$ is the name of the dimension.
- $A^D = A_1, \ldots, A_n$ is a set of dimension attributes.
- $H^D = H_1, \ldots, H_n$ is a set of hierarchies, arranging the properties in accordance with the level of granularity that each one represents.

The Fig. 1 illustrates our multidimensional use case model:

![Fig. 1. The multi dimensional model.](https://www.ijacsa.thesai.org)
B. Our Approach

Our approach involves leveraging the advantages of graph databases by creating the OLAP cube in the graph and using user queries to identify frequently used dimensions in OLAP analyses. To optimize the aggregations to be created, we use the Apriori algorithm to extract the most frequently associated sets of dimensions in OLAP queries, and then apply a rule-based association algorithm to identify the most relevant aggregations. The resulting aggregations are created in the OLAP cube, leading to improved OLAP query performance.

The approach consists of four steps:

- Creation of the graph data warehouse.
- Extraction of the most frequently associated sets of dimensions in OLAP queries using the Apriori algorithm.
- Identification of the most relevant aggregations using a rule-based association algorithm.
- Creation of identified aggregations.

1) Graph Data Warehouse: When used with relational databases, a relational fact table is created for each fact in the multi-dimensional conceptual model. Measures are columns in the fact table. Additionally, each dimension is transformed into a normalized dimension table with columns for each attribute (including parameters and weak attributes). Fact and dimension tables each have a unique row for storing each instance [16]. Similarly, we provide our rules, which define a graph DW, using the definitions of multi-dimensional model and property graph ideas that were previously presented.

**Dimension** $D^S$ in our model is created in node format identified by $(L^N, P^N)$ where:

- $L^N$ represents the label of the node, a node can have zero to many labels.
- $P^N$ represents attributes of the dimension.

**Hierarchies:** In a graph data warehouse, a hierarchy can be represented using nodes and edges. Each level of the hierarchy can be represented as a node, with edges connecting nodes at different levels to indicate parent-child relationships.

**Fact:** A fact node in a graph data warehouse can be represented as a node with edges connecting it to dimension nodes. The fact node can also have properties that represent the measures, such as the actual values or aggregate functions applied to them [17]. A fact node, is specified by $(N^F, M^F)$ where:

- $N^F$: represent the fact name.
- $M^F$: It comprises a collection of measures as node attributes, with each measure linked to an aggregation function.

**Relationship between fact and dimensions:** The relationship between fact and its associated dimensions are represented as edges connecting nodes in the graph, the link is defined by $(L^E, N^F, N^D, P^E)$, where:

- $L^E$ is the label of relationship.
- $N^F$ is the fact node.
- $N^D$ is a node that represents the dimension linked to the fact.
- $P^E$ represent the properties of the relationship, the properties are key-value pairs that are used for storing data on relationships.

2) Algorithm 1: Calculate Frequent Itemsets.: After creating our OLAP system, the second phase in our approach is to collect user queries from the OLAP system logs [18]. And after that, we determine common itemsets of predicates using the Apriori algorithm [19]. In the next phase we will use the generated itemsets as input to the second association rule algorithm to determine the most important aggregations to create. The algorithm starts by initializing an empty list to store frequent itemsets and another empty list to store previous frequent itemsets. Then, it loops on user queries predicates until there are no more frequent itemsets to explore. During each iteration of the loop, the algorithm generates candidates for new frequent itemsets by combining previous frequent itemsets. Then, it calculates the frequency of each candidate by scanning through all transactions. Candidates that have a frequency below a minimum support threshold are filtered out. Finally, the frequent itemsets are added to the list of frequent itemsets. The candidate generation and filtering functions are also defined in the algorithm.
Algorithm 1: Calculate frequent itemsets

```
// Initialization
frequent_itemsets ←
previous_frequent_itemsets ←
// Loop until there are no more frequent itemsets to explore
while (previous_frequent_itemsets is not empty) do
    // Generate candidates
    current_frequent_itemsets ← generate_candidates(previous_frequent_itemsets)
    // Calculate frequency of candidates
    for each aggregation in OLAP cube do
        for each candidate in current_frequent_itemsets do
            if candidate is used in aggregation then
                candidate.frequency ← candidate.frequency + 1
            end if
        end for
    // Filter candidates
    previous_frequent_itemsets ← current_frequent_itemsets
    current_frequent_itemsets ← filter_candidates(current_frequent_itemsets, min_sup)
    // Add frequent itemsets to the list
    frequent_itemsets ← frequent_itemsets ∪ current_frequent_itemsets
end while
```

FUNCTION generate_candidates(itemsets)

```
candidates ←
for each itemset1 in itemsets do
    for each itemset2 in itemsets do
        if itemset1 ≠ itemset2 and all elements of itemset1 except the last one are the same as the corresponding elements of itemset2 then
            candidate ← union of itemset1 and itemset2, keeping only the unique elements
            if candidate is not already in candidates then
                candidates ← candidates ∪ candidate
            end if
        end if
    end for
end for
return candidates
```

FUNCTION filter_candidates(candidates, min_sup)

```
frequent_candidates ←
for each candidate in candidates do
    if candidate.frequency ≥ min_sup then
        frequent_candidates ← frequent_candidates ∪ candidate
    end if
end for
return frequent_candidates
```

Algorithm 2: Generate Association Rules

```
association_rules ←
for each frequent_itemset in frequent_itemsets do
    subsets ← generate_subsets(frequent_itemset)
    for each subset in subsets do
        antecedent ← subset
        consequent ← (frequent_itemset) − (subset)
        association_rule ← {antecedent : consequent, consequent : confidence : 0}
        // Calculate confidence of the rule
        antecedent_frequency ← calculate_frequency(antecedent)
        consequent_frequency ← calculate_frequency(frequent_itemset)
        itemset_frequency_antecedent ←
        if confidence ≥ min_conf then
            association_rule.confidence ← confidence
            association_rules ← association_rules ∪ association_rule
        end if
    end for
end for
```

FUNCTION generate_subsets(itemset)

```
subsets ←
for i ← 1 to taille(itemset) do
    subset ←
        for j ← 1 to taille(itemset) do
            if j ≠ i then
                subset ← subset ∪ itemset[j]
            end if
        end for
    subsets ← subsets ∪ subset
end for
return subsets
```

FUNCTION calculate_frequency(itemset)

```
frequency ← 0
for transaction in transactions do
    if transaction contains all elements of itemset then
        frequency ← frequency + 1
    end if
end for
return frequency
```

4) Graph Aggregation: The two optimization algorithms previously defined enable finding the best combinations of aggregations. For improving query performance and maximising data retrieval in a graph, OLAP aggregations must be created. Aggregations act as quick-query summaries of data that have

www.ijacsa.thesai.org 1001 | Page
The dimensions used in the model are as follows:

- Product dimension with the TypeProduct hierarchy.
- Customer dimension with the cityCustomer and RegionCustomer hierarchy.
- Supplier dimension with the citySupplier and RegionSupplier hierarchy.
- Year dimension with the Month hierarchy.

The script in Listing 1 is used to import data from a CSV file (“File.csv”) into Neo4j database by creating nodes to represent different entities related to the (CUSTOMER) as well as the hierarchies associated with these entities (CityCustomer and RegionCustomer). The script uses the APOC (Awesome Procedures on Cypher) library’s apoc.periodic.iterate procedure to efficiently manage the data import from the CSV file. It performs the import by iterating over batches of data, allowing it to process large amounts of data while avoiding overwhelming the system. The function ‘LOAD csv WITH HEADERS FROM “file:///File.csv” as row FIELDTERMINATOR “;” RETURN row’ loads the CSV data with headers into the “row” variable and uses the semicolon (;) as the field delimiter. The first function MERGE (creates or updates) a node with the “CUSTOMER” label having properties “CUSTOMER-ID” and “CUSTOMER-NAME” extracted from the corresponding columns in the CSV file. The options batchSize:10000, allow the data to be processed in batches of 10,000 rows in parallel for better performance. The second use of the MERGE function in the script creates or merges nodes with the “CityCust” label having properties “CUSTOMER-CITYID” and “CUSTOMER-CITY” extracted from the CSV file. Similarly, the third use of the MERGE function is also used for creating or matching nodes in the graph database for the “RegionCust” hierarchy.

### Listing 1: The Customer Dimension

```sql
// Dimension Customer
CALL apoc.periodic.iterate(
  'LOAD csv WITH HEADERS FROM "file:///File.csv" as row FIELDTERMINATOR ";" RETURN row',
  'MERGE (C:Customer [CUSTOMER_ID : row.CUSTOMER_ID, CUSTOMER_NAME : row.CUSTOMER_NAME])',
  [batchSize:10000, parallel:true]);

// Hierarchy CityCustomer
CALL apoc.periodic.iterate(
  'LOAD csv WITH HEADERS FROM "file:///File.csv" as row FIELDTERMINATOR ";" RETURN row',
  'MERGE (CC:CityCust [CUSTOMER_CITYID : row.CUSTOMER_CITY, CUSTOMER_CITY : row.CUSTOMER_CITY])',
  [batchSize:10000, parallel:true]);

// Hierarchy RegionCustomer
CALL apoc.periodic.iterate(
  'LOAD csv WITH HEADERS FROM "file:///File.csv" as row FIELDTERMINATOR ";" RETURN row',
  'MERGE (RC:RegionCust [CUSTOMER_REGIONID : row.CUSTOMER_REGION, CUSTOMER_REGION : row.CUSTOMER_REGION])',
  [batchSize:10000, parallel:true]);
```

matching “CUSTOMER-CITYID” property using the MATCH clauses. The First MERGE clause creates a relationship of type CITY-CUSTOMER between the matched “CUSTOMER” and “CityCust” nodes. The second use of the MERGE clause creates a relationship of type REGION-CUSTOMER between the matched “CityCust” and “RegionCust” nodes.

Listing 2: The relationship between the Customer dimension and its hierarchies

```cypher
// Relationship between Customer and CityCustomer
CALL apoc.periodic.iterate('LOAD CSV WITH HEADERS FROM ''file:///File.csv'' as row FIELDTERMINATOR '';
RETURN row',
'MATCH (C:CUSTOMER [CUSTOMER_ID: row.CUSTOMER_ID])
MATCH (CC:CityCust [CUSTOMER_CITYID: row.CUSTOMER_CITYID])
MERGE (C)-[:CITY_CUSTOMER]->(CC)
',
[batchSize:2000, iteratelist:true]);

// Relationship between CityCustomer and RegionCustomer
CALL apoc.periodic.iterate('LOAD CSV WITH HEADERS FROM ''file:///File.csv'' as row FIELDTERMINATOR '';
RETURN row',
'MATCH (CC:CityCust [CUSTOMER_CITYID: row.CUSTOMER_CITYID])
MATCH (RC:RegionCust [CUSTOMER_REGIONID: row.CUSTOMER_REGIONID])
MERGE (CC)-[:REGION_CUSTOMER]->(RC)
',
[batchSize:2000, iteratelist:true]);
```

To create the dimensions “PRODUCT”, “SUPPLIER”, and “Time”, we use a similar approach as employed for the “CUSTOMER” dimension.

After creating all the dimension nodes and their hierarchies in the same way, the next step is to create the fact node that contains the measures, and relationships between the fact and dimension nodes. The following script in Listing 3, demonstrates the creation of the fact node in Neo4j and the relationships between the “FACT” nodes and the corresponding nodes in the “PRODUCT”, “SUPPLIER”, and “TIME” dimensions in the Neo4j graph database.

Listing 3: The Fact Node

```cypher
// FACT NODE
CALL apoc.periodic.iterate(
'LOAD CSV WITH HEADERS FROM ''file:///File.csv'' AS row FIELDTERMINATOR '';
RETURN row',
',
[batchSize:10000, parallel:true]);

// Relationship FACT/CUSTOMER
CALL apoc.periodic.iterate(
'LOAD CSV WITH HEADERS FROM ''file:///File.csv'' AS row FIELDTERMINATOR '';
RETURN row',
'MATCH (C:CUSTOMER [CUSTOMER_ID: row.CUSTOMER_ID])
MATCH (FCT:FACT [ID: row.INTEGRATION_ID])
MERGE (FCT)-[:FACT_CUSTOMER]->(C)
',
[batchSize:20000, iteratelist:true]);
```

In the same way, we create relationships between the fact node and the other dimensions (PRODUCT, SUPPLIER, TIME). Using also the apoc.periodic.iterate procedure along with MATCH and MERGE statements to efficiently import data from the CSV file and establish the relationships between the “FACT” nodes and the corresponding nodes in the “PRODUCT”, “SUPPLIER”, and “TIME” dimensions in the Neo4j graph database.

The Fig. 2 represents the implementation of graph warehouse in Neo4j.

In the same way, we create relationships between the fact node and the other dimensions (PRODUCT, SUPPLIER, TIME). Using also the apoc.periodic.iterate procedure along with MATCH and MERGE statements to efficiently import data from the CSV file and establish the relationships between the “FACT” nodes and the corresponding nodes in the “PRODUCT”, “SUPPLIER”, and “TIME” dimensions in the Neo4j graph database.

B. OLAP Operators

Slice

The slice operator in OLAP enables the selection of slices from the data based on a condition on the dimension values [7].

In Listing 4 the slice operator is applied to the Customer Region dimension using the filter condition “AFRICA”, which allows for selecting the data related to the AFRICA.
**Listing 4: Selecting Price and Quantity Results from Africa.**

```cypher
MATCH (RC:RegionCust [CUSTOMER_REGION: 'AFRICA'])<-[*3]-(m:FACT)
RETURN RC.CUSTOMER_REGION, sum(tofloat(m.Price)), sum(tofloat(m.QUANTITY))
```

**Dice**

The Dice operator is used in OLAP to select a subset of data based on two or more conditions on dimensions. It is similar to the Slice operator, but allows for finer selection by applying multiple criteria on dimensions simultaneously.

In Listing 5 the Dice operator is applied to the Customer Region and Year dimensions using the filter condition “AFRICA” and “1997”.

**Listing 5: Dice-Selecting Quantity Results with a Dice Operation.**

```cypher
MATCH (RC:RegionCust [CUSTOMER_REGION: 'AFRICA'])<-[*3]-(m:FACT)
MATCH (Y:YEAR [YEAR : 1994])<-[*2]-(m:FACT)
RETURN RC.CUSTOMER_REGION, Y.YEAR, sum(tofloat(m.QUANTITY)) as QUANTITY
```

**Roll Up**

In OLAP,[23] the Roll-Up operation is used to aggregate data at a higher level of hierarchy than the current level.[24] It involves moving from a detailed level to a higher-level concept [25]. The Roll-Up operation is performed by grouping the data based on the dimensions and then performing the aggregation function on the measures. The result is a summarized view of the data at a higher level of abstraction. In Listing 6, the Roll-Up operation is carried out by moving up the Product dimension’s concept hierarchy (Product → Product Type) and the hierarchy of dimension Time(Month → Year). This query creates a relationship between the two dimensions that contains the aggregated measures.

**Listing 6: Roll Up- Price and Quantity summarized by Product Type and Year.**

```cypher
MATCH (TP:TYPE_PRODUCT)<-[*2]-(FCT:FACT)
MATCH (Y:YEAR )<-[*2]-(FCT:FACT)
WITH distinct TP,Y, sum(tofloat(FCT.Price)) as Price, sum(tofloat(FCT.QUANTITY)) as QUANTITY
create (TP)<-[TYPRD_YEAR_AGG [Price: Price, QUANTITY: QUANTITY]]-(Y)
```

**C. Graph Aggregations**

After generating the most frequently user queries, the Apriori Algorithm was then used to determine the most common dimensions, and then we executed the second algorithm to generate the most commonly used combinations to create the aggregations. We set the support to 0.4 and the confidence to 0.7. The Fig. 3 shows the combinations of the most commonly used dimensions.

![Fig. 3. The combinations of the most commonly used dimensions.](image)

To create aggregations in Neo4j, we use the following script in Listing 7 that stores the aggregations in relationships:

**Listing 7: Aggregation Customer-Year**

```cypher
CALL apoc.periodic.iterate(
'LOAD CSV WITH HEADERS FROM ''file:///File.csv" AS row FIELDTERMINATOR '';
RETURN row',
'MATCH (C:CUSTOMER)<-[ ]-(FCT:FACT)
MATCH (Y:YEAR )<-[*2]-(FCT:FACT)
CREATE (C)<-[CUST_YEAR_AGG [Price: toFloat(row.O_TOTALPRICE),QUANTITY: toFloat(row.L_QUANTITY)]]-[A]',
[batchSize:10000, parallel:true]);
```

In Listing 8, the script is used to create aggregations that are stored in nodes, not in relationships.

**Listing 8: Aggregation Customer-Product-Supplier**

```cypher
CALL apoc.periodic.iterate(
'LOAD CSV WITH HEADERS FROM ''file:///File.csv" AS row FIELDTERMINATOR '';;
RETURN row',
'MATCH (P:PRODUCT)<-[ ]-(FCT:FACT)
MATCH (C:CUSTOMER)<-[ ]-(FCT:FACT)
MATCH (S:SUPPLIER) <-[* ]-(FCT:FACT)
create (PCS:PROD_CUST_SUPP [ PRICE: toFloat(row.O_TOTALPRICE),QUANTITY: toFloat(row.L_QUANTITY)]))
create (P)<-[PROD_3]-(PCS)
create(C)<-[CUST_3]-(PCS)
create (S)<-[SUPP_3]-(PCS')
[batchSize:10000, parallel:true]);
```
The Fig. 4 shows the creation of aggregations using in approach.

D. Experimental Results and Evaluation

To validate our approach and measure the effectiveness of optimizing the OLAP cube in the graph, we conducted a series of experiments in which we evaluated performance before and after using our optimization approach. We measured the query execution time before and after adding optimized aggregations, and compared the execution times to determine if adding the optimized aggregations led to a significant improvement in performance. We conducted our test on an i7 processor machine with 16GB of RAM and 1TB of storage memory. Additionally, we used the TPC-H database with a scale factor of SF1 (1GB). We use in Table II, Cypher queries before and after optimization.

<table>
<thead>
<tr>
<th>Query</th>
<th>Before Optimization</th>
<th>After Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q 1</td>
<td>MATCH (RS:RegionSupp) &lt;-[*3]-(FCT:FACT) return SUM(tofloat(FCT.Price)) AS Price, RS.SUPPLIER_REGION</td>
<td>MATCH (RS:RegionSupp) &lt;-[:REGION_SUPP_AGG]-(RA:Region_Supp_AGG) return RA.Price, RS.SUPPLIER_REGION</td>
</tr>
<tr>
<td>Q 2</td>
<td>MATCH (Y:YEAR) &lt;-[*2]-(m:FACT) MATCH (TP:TYPE_PRODUCT) &lt;-[*2]-(m:FACT) return Y.YEAR, TP.BRAND, sum(tofloat(m.Price)) as Price</td>
<td>MATCH (TP:TYPE_PRODUCT) &lt;-[:TYPE_PRD_YEAR_AGG]-(Y:YEAR) return Y.YEAR, TP.BRAND, sum(tofloat(r.Price)) as Price</td>
</tr>
<tr>
<td>Q 3</td>
<td>MATCH (C:CUSTOMER) &lt;-[]-(FCT:FACT) MATCH (P:PRODUCT) &lt;-[]-(FCT:FACT) return sum(tofloat(FCT.Price)) as Price, C.CUSTOMER_NAME, P.PRODUCT_NAME</td>
<td>MATCH (P:PRODUCT) &lt;-[:AGG_CUST_PROD]-(C:CUSTOMER) return r.Price, C.CUSTOMER_NAME, P.PRODUCT_NAME</td>
</tr>
<tr>
<td>Q 4</td>
<td>MATCH (TP:TYPE_PRODUCT) &lt;-[*2]-(FCT:FACT) return SUM(tofloat(FCT.QUANTITY)) AS QUANTITY, TP.BRAND</td>
<td>MATCH (TP:TYPE_PRODUCT) &lt;-[:TYPE_PRD_AGG]-(TPAGG:TYPE_PROD_AGG) return TPAGG.QUANTITY, TP.BRAND</td>
</tr>
<tr>
<td>Q 5</td>
<td>MATCH (P:PRODUCT) &lt;-[:FACT_PRODUCT]-(FCT:FACT) MATCH (C:CUSTOMER) &lt;-[:FACT_CUSTOMER]-(FCT:FACT) MATCH (S:SUPPLIER) &lt;-[:FACT_SUPPLIER]-(FCT:FACT) return SUM(tofloat(FCT.Price)) AS PRICE, P.PRODUCT_NAME, C.CUSTOMER_NAME, S.SUPPLIER_NAME LIMIT 43</td>
<td>MATCH (P:PRODUCT) &lt;-[:PROD_3]-(PCS:PROD_CUST_SUPP) MATCH (C:CUSTOMER) &lt;-[:CUST_3]-(PCS:PROD_CUST_SUPP) MATCH (S:SUPPLIER) &lt;-[:SUPP_3]-(PCS:PROD_CUST_SUPP) return PCS.PRICE, P.PRODUCT_NAME, C.CUSTOMER_NAME, S.SUPPLIER_NAME LIMIT 43</td>
</tr>
</tbody>
</table>
The Fig. 5 shows the query execution time before and after the optimization of the model.

![Query Execution Time](image)

The results demonstrate that the execution time of the queries decreased after the optimization and usage of OLAP aggregations, although the execution time may vary depending on the complexity of the query, with complex queries requiring traversal of numerous relationships taking more time, while simple queries involving only a few relationships having relatively short execution times.

For instance, in the first query, the execution time before optimization was approximately 14266 milliseconds, and after optimization, it reduced to 3 milliseconds. This translates to an impressive percentage improvement of approximately 99%. Furthermore, in the query 4, we achieved considerable improvements as well, the execution time decreased from 3,978 milliseconds to 2 milliseconds, substantial gains in performance clearly demonstrate the effectiveness of our approach in making the system nearly 2,000 times faster than its previous state.

The significant reduction in execution time showcases how this approach can make the system multiple times faster than its previous state, enhancing the efficiency of reporting and data analysis within the graph-based data warehouse.

We also compared our model implemented in the graph and the ROLAP model, comparing the execution time of the same multidimensional queries in Neo4j and Oracle.
The Figure 6 shows the query execution time in the Graph OLAP and ROLAP.

![Query execution time in Oracle and Neo4j.](image)

The results also show that the Graph cube delivers performance levels that are better than those of the ROLAP model. We also notice that Graph databases provide a great degree of flexibility for searching through data by conducting more complex pathways or by following direct linkages. While SQL queries that use joins to mix data from various tables provide the basis for data traversal in a relational architecture, while joins can be effective, they can also be less flexible and intuitive when navigating complex relationships.

V. CONCLUSION

Graph-Oriented Databases offers a clear abstraction for managing heavily connected data and modelling complicated domains. In this paper, we present our contribution for developing a data warehouse under a graph database, our approach relies on the properties of graphs to implement graph data warehouse. To enhance the graph cube’s performance, we provide a new technique that optimizes the choice of OLAP aggregations by using the association rules algorithm.

To validate our approach and measure the effectiveness of OLAP cube optimization in the graph, we conducted a series of experiments in which we evaluated the performance before and after the optimization, we also compared our model with the relational model in terms of query performance. The experiment’s findings demonstrate the benefits of creating OLAP cubes on graphs. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pages 853–864, Athens Greece, June 2011. ACM.


[5] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and OLAP multidimensional networks. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pages 853–864, Athens Greece, June 2011. ACM.


