

Modelling and Querying Star and Snowflake Warehouses Using Graph Databases

Alejandro Vaisman^(✉), Florencia Besteiro, and Maximiliano Valverde

Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina
{avaisman,mabestei,mvalverd}@itba.edu.ar

Abstract. In current “Big Data” scenarios, graph databases are increasingly being used. Online Analytical Processing (OLAP) operations can expand the possibilities of graph analysis beyond the traditional graph-based computation. This paper studies graph databases as an alternative to implement star and snowflake schemas, the typical choices for data warehouse design. For this, the MusicBrainz database is used. A data warehouse for this database is designed, and implemented over a Postgres relational database. This warehouse is also represented as a graph, and implemented over the Neo4j graph database. A collection of typical OLAP queries is used to compare both implementations. The results reported here show that in ten out of thirteen queries tested, the graph implementation outperforms the relational one, in ratios that go from 1.3 to 26 times faster, and performs similarly to the relational implementation in the three remaining cases.

1 Introduction

Online Analytical Processing (OLAP) [6,8] comprises a set of tools and algorithms that allow querying large data repositories called data warehouses (DW). At the conceptual level, these DWs are modelled as *data cubes*, following the multidimensional (MD) model. In such model, each cell of the cube contains one or more *measures* of interest, that quantify *facts*. Measure values can be aggregated along *dimensions*, organized as sets of hierarchies. The most popular OLAP operations on cube data are aggregation and disaggregation of measure values along the dimensions (called roll-up and drill-down, respectively); selection of a portion of the cube (dice); or projection of the data cube over a subset of its dimensions (slice operation).

Property Graphs [5,7] underlie the most popular graph database engines [1]. In addition to traditional graph analytics, it is also of interest of the data scientist to have the possibility of performing OLAP on graphs. This paper explores graph databases as an alternative for storing DWs or data marts (that is, DWs oriented to analyse focused particular problems, generally at departmental level). Relational OLAP models are typically of two kinds: star schema-based and snowflake

schema-based. Both of them are composed of a collection of fact and dimension tables. Fact tables contain the foreign keys of the dimension tables, and a set of measures quantifying the facts. Dimension tables contain dimensional data. In the star model, dimension tables are denormalized, whereas in the snowflake model, dimension tables are normalized. This paper studies graph databases as an alternative to implement the star and snowflake DW schemas. The main hypothesis here is that graph databases can be more efficient than relational ones to address the typical kinds of OLAP queries, which consist, basically, in sequences of selection-projection-join-aggregation operations (SPJA).

A case study is discussed throughout the paper, based on the MusicBrainz database (<http://musicbrainz.org/>), an open music encyclopedia that collects music metadata and makes it available to the public. In this work, MusicBrainz is considered the OLTP (Online Transactional Processing) database, from which a DW is designed following the snowflake schema. This design is implemented on a PostgreSQL database, populated with data from the OLTP database through an ETL process (not reported in this paper). The same logical design is performed following the property graph data model, and implemented over a Neo4j database, which is then populated from the relational data. A collection of SPJA queries is defined and executed on both implementations, and the results are reported and discussed. *The experiments showed (with some exceptions) a clear advantage of the graph alternative over the relational one.*

The remainder of this paper is organized as follows: In Sect. 2 related work is discussed. Section 3 presents the case study, and the corresponding relational and graph models. Section 4 presents the queries implemented in the paper, and Sect. 5 describes the experiments, and reports and discusses the results. Section 6 concludes the paper and suggest future research directions.

2 Related Work

There is an extensive bibliography on graph database models, comprehensively studied in [1, 3]. The interested reader is referred to this corpus of work for details. In real-world practice, two graph database models are used, namely (a) Models based on RDF (<https://www.w3.org/RDF/>) oriented to the Semantic Web; and (b) Models based on Property Graphs. Models of type (a) represent data as sets of triples where each triple consists of three elements that are referred to as the subject, the predicate, and the object of the triple. Informally, a collection of RDF triples is an RDF graph. In the *property graph* data model, [2] nodes and edges are labelled with a sequence of (attribute, value)-pairs. Extending traditional graph database models, Property Graphs are the usual choice in modern graph databases used in real-world practice.

The present paper works with models based on Property Graphs, and it is based on the work of Gómez et al. [4], who propose a model for performing OLAP on hypergraphs, based on the notion of graphoids, which are defined as graphs aggregated at different levels of granularity, using a collection of OLAP dimension hierarchies. The proposal supports *heterogeneous graphs*, which is required for typical multidimensional problems. The authors show that this model captures the semantics of the star and snowflake schemas.

3 Data Model

The model adopted in this paper will be presented through a running example based, as mentioned above, on the MusicBrainz database. For clarity of presentation, and to make the analysis more comprehensive, only the portion of the database containing information about music track releases and musical events, is tackled here. The core data in the original database includes, for example: *Artists* (with, e.g., name, aliases, type, begin and end dates); *Releases* (title, artist credit, type, status, language, date, country, label, etc.); *Recordings* (title, artist credit, duration, etc.); *Labels* (name, aliases, country, type, code, begin and end dates). Based on the original MusicBrainz database, a relational DW is defined at a conceptual level. Then, an equivalent graph DW is created, following [4]. In the next two sections, the relational and graph representations of the DW are described and discussed.

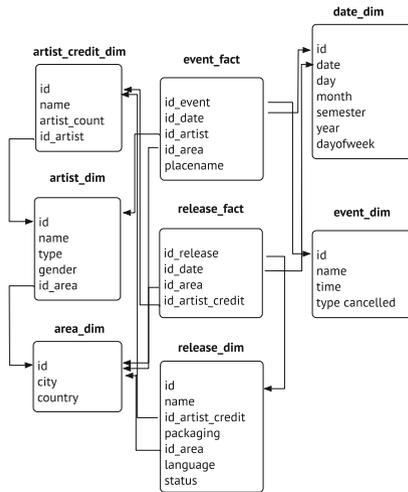


Fig. 1. ROLAP design for the musicbrainz database.

The MusicBrainz Relational Data Warehouse. Figure 1 depicts the snowflake schema for the MusicBrainz DW. There are two fact tables, namely *release_fact* and *event_fact*, representing the occurrence of the release of a music piece, or of an event, respectively. Each fact table refers to a dimension table called *release_dim* and *event_dim*. Artists in the *release_fact* fact table are described at the artist credit level (using the *id_artist_credit* attribute, which is a foreign key of the *artist_credit_dim* dimension table). An artist credit indicates a reunion of artists for a release, like, for example, David Bowie and Queen for the “Under Pressure” track. For the events, artists are described in the *artist_dim* dimension table, referred through the *id_artist* attribute. There is also an *area_dim* dimension table

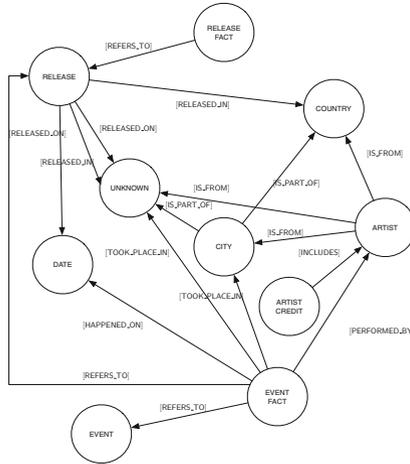


Fig. 2. Graph design for the musicbrainz database.

which includes the city \rightarrow country dimension hierarchy. Thus, actually the design is a mixture of a star and snowflake schemas, as it is usual in data warehousing design practice. Finally, there is a time dimension represented by a dimension table denoted `date.dim`. This DW was implemented on a PostgreSQL database, and part of the data exported using an ETL process (not described here).

The MusicBrainz Graph Datawarehouse. Figure 2 depicts the schema of the graph database model for the MusicBrainz DW represented in Fig. 1. Attributes are not included in the graph representation, for the sake of clarity, and only the node types are depicted. There is a node for each event fact and a node for each release fact. In addition, there is an “Unknown” node type, which indicates missing data. The references to the dimensions are materialized through links to nodes which represent the members of the dimensions. Therefore, there is one node for each dimension member linked to a fact, such that the descriptive attributes are properties of these nodes (e.g., a `Date` node, with `month` and `year` properties). Some dimension hierarchies are made explicit. For example, the `area.dim` dimension is transformed into a node hierarchy of the kind `city \rightarrow country`. Event and release facts are associated with nodes representing event and release data (e.g., the date of the event). In summary, with respect of the model introduced in [4], Fig. 2 represents the *base graphoid* (the graphoid at the finest level of granularity), and the background dimension hierarchies, one for each dimension in the relational schema, plus an `Unknown` dimension, with one level. This DW was implemented on a Neo4j graph database, which was populated using a script containing a sequence of Cypher statements.

4 Case Study and Discussion

This section shows how four kinds of OLAP queries can be expressed over both representations described in Sect. 3: (a) PJA queries, that means, queries that perform a projection after join operations, and finally aggregate the result; (b) SPJA queries, analogous to (a), but including a selection (a filter using a Boolean condition); (c) FPJA queries, that is, PJA queries that involve self joins of fact tables (in the snowflake model) of self references to the same kind of node (in the graph model); (d) PJA-DA queries, that is, PJA queries referring to different kinds of facts. For the sake of space, only some of Cypher expressions are shown in the queries below.

(a) PJA Queries. These typical OLAP queries, generally link facts and dimensions, and finally perform some kind of aggregation. The first query is a simple join, representing the climbing along the artist hierarchy starting from `artist_credit`, aggregating the releases.

Query 1. *Compute the number of releases per artist.*

In SQL, this is a join between the fact table containing the releases (`release_fact`, and the dimension tables `artist_credit_dim` and `artist_dim`. In Cypher:

```
MATCH (r:ReleaseFact)-[]->(a:ArtistCredit)-[]->(a1:Artist)
RETURN a1.name, count(r) ORDER BY a1.name ASC
```

Queries in Cypher are evaluated by means of pattern matching. In this case, the join is implemented through the matching of the `ReleaseFact` \rightarrow `ArtistCredit` \rightarrow `Artist` path, against the MusicBrainz graph. In this case, the relationship names of the path are not needed since the edges between nodes can be inferred from the node types. In the next query, aggregation is performed climbing along two dimensions, namely `Artist` and `Time`.

Query 2. *Compute the number of releases per artist and per year.*

```
MATCH (r:ReleaseFact)-[r1:RELEASED_BY]->(a:ArtistCredit)-[]->(a1:Artist),
(d:Date)<-[rd:RELEASED_ON]-(r)
RETURN a1.name, d.year, count(r) ORDER BY a1.name ASC,d.year ASC
```

Note that in Cypher, the aggregation `a1.name, d.year, count(r)` is a concise way to express a `GROUP BY` clause in SQL. Query 3 below, is similar to Query 1, but uses the event facts rather than the release facts.

Query 3. *Compute the number of events per artist.*

In the next query, aggregation of event facts is performed along the `Event` and `Artist` dimensions.

Query 4. *Compute the number of times the artist performed in each event.*

```
MATCH (e1:Event)<-[:REFERS_TO]-(e:EventFact)-[:PERFORMED_BY]->(a:Artist)
RETURN e1.name,a.name, count(*) ORDER BY e1.name ASC, a.name ASC
```

It can be seen that the kind of event is given in the event type of node (of the event_dim dimension in the relational model). The last PJA query aggregates data along three dimensions (Event, Artist, and Time).

Query 5. *For each (event, artist, year) triple, compute the number of times the artist performed in an event on an year.*

(b) *SPJA Queries.* These queries add, to the join condition between facts and dimensions, a selection (Boolean) condition. The first query to be analysed operates over event facts.

Query 6. *Same as Query 5, for artists in the United Kingdom and events occurred after year 2006.*

```
MATCH (e1:Event)<-[:r1:REFERS_TO]-(e:EventFact)-[:r:HAPPENED_ON]->(d:Date)
WHERE d.year > 2006
WITH e,d,e1
MATCH (e)-[:p:PERFORMED_BY]->(a:Artist)-
[:IS_FROM]->(c:Country{name:'United Kingdom'})
RETURN e1.name, d.year,a.name, count(*)
ORDER BY e1.name asc, d.year asc, a.name asc
```

In the query above, the WITH expression passes the variables on to the next step of the computation. The next query operates over releases.

Query 7. *Compute the number of releases, per language, in the UK.*

The next query requires joining the event facts with themselves, once. Queries 9 and 10, require two and three self joins, respectively.

Query 8. *Compute, for each pair of artists, the number of times they have performed together at least twice in an event.*

```
MATCH (a1:Artist)<-[:]-[:(e:EventFact)-[:]->(a2:Artist) WHERE a1.id < a2.id
WITH a1, a2, COLLECT(e) AS events WHERE SIZE(events) > 1
RETURN a1.name, a2.name, SIZE(events) ORDER BY SIZE(events) desc
```

The COLLECT statement builds a list with all the events for each pair of artists. The SIZE function computes the length of this list.

(c) *FPJA Queries But Involving More Than One Self Fact Table Joins.* This kind of queries joins several fact nodes with other ones of the same type. The next query requires two of such joins.

Query 9. *Compute the triples of artists, and the number of times they have performed together in an event, if this number is at least 3.*

```

MATCH (a1:Artist)-<[]-(e:EventFact)->[]-(a2:Artist) WHERE a1.id < a2.id
WITH a1,a2,COLLECT(e) AS events WHERE SIZE(events) > 2
MATCH (a1:Artist)-<[]-(e1:EventFact)->[]-(a2:Artist)
MATCH (a3:Artist)-<[]-(e1) WHERE a2.id < a3.id
WITH a1.name as name1, a2.name as name2,a3.name as name3 ,
COUNT(e1.idEvent) as nbrTimes WHERE nbrTimes > 2
RETURN name1,name2,name3, nbrTimes ORDER BY nbrTimes DESC

```

Query 10. *Compute the quadruples of artists, and the number of times they have performed together in an event, if this number is at least 3.*

(d) *PJA-DA Queries.* Finally, queries involving events and releases are evaluated. In OLAP this is called a drill-across operation between event and release facts.

Query 11. *Compute the pairs of artists that have performed together in at least two events and that have worked together in at least one release, returning the number of events and releases together.*

Query 12. *List the artists who released a record and performed in at least an event, and the year(s) this happened.*

Table 1. Dataset sizes for the relational representation (left); Dataset sizes for the graph representation (center); Results of the experiments (right).

Table	# tuples
date_dim	90,033
artist_dim	1,151,920
artist_credit_dim	1,871,875
area_dim	74,211
release_dim	1,715,636
event_dim	19,441
event_fact	55,281
release_fact	1,724,365

Element(node/edge type)	# nodes	# edges
Artist	1,151,920	
ArtistCredit	1,871,875	
City	73,955	
Country	255	
Event	19,441	
Release	1,715,636	
ReleaseFact	1,724,365	
EventFact	19,457	
Unknown	1	
HAPPENED_ON		19,348
IS_FROM		1,151,920
INCLUDES		4,098,689
IS_PART_OF		73,955
PERFORMED_BY		52,599
RELEASED_BY		1,913,494
REFERS_TO		1,743,822
RELEASED_IN		1,755,598
RELEASED_BY		1,913,494
RELEASED_ON		1,407,504
TOOK_PLACE_IN		19,475

Query	PostgreSQL	Neo4j	Pg/Neo4j
query 1	22	11	2
query 2	30	12.5	2.4
query 3	0.8	0.12	6.66
query 4	1.5	0.2	7.5
query 5	2.4	0.4	6
query 6	0.2	0.2	1
query 7	0.2	0.2	1
query 8	2	1.5	1.33
query 9	24	5	4.8
query 10	1110	43	25.8
query 11	9	1.8	5
query 12	3	3	1
query 13	2	1	2

In Cypher, the query reads:

```

MATCH (a:Artist)-<[PERFORMED_BY]->[]-(e:EventFact)-[:HAPPENED_ON]->[]-(d:Date)
WITH distinct a,d.year as year
MATCH (r:ReleaseFact)-[r1:RELEASED_BY]->[]-(a2:ArtistCredit)->[]-(a),
(r)-[r2:RELEASED_ON]->[]-(d1:Date) WHERE d1.year=year
RETURN DISTINCT a.name, year

```

Query 13. *Artists who released a record and performed in at least an event, and the year(s) this happened, for events and releases occurred since 2007.*

5 Experiments

The thirteen queries in Sect. 4 were run over relational and graph databases designed following the models described in Sect. 3. The solution based on the graph model is compared against the relational alternative containing exactly the same data. For the relational representation, the DW was implemented as a snowflake schema and stored in a PostgreSQL database. All tables were fully indexed for the workload described in the previous section. The left-hand side of Table 1 shows the sizes of the tables. For the graph representation, the number of nodes and edges in the graph, stored in a Neo4j database, community version 3.5.3, are depicted in the center part of the figure. As for the relational alternative, all the required indexes were defined. The right-hand side of Table 1 shows the results of the experiments. The queries introduced in Sect. 4 were run on machine with a i7-6700 processor and 32 GB of RAM, a 1 TB hard disk, and a 256 MB SSD disk. The execution times are depicted as the averages of five runs of each experiment, expressed in seconds. The best time for each query is highlighted in bold font. The ratio between the execution times in PostgreSQL and Neo4j are indicated in the fourth column.

The results of the experiments show that only in three out of thirteen queries the execution times were the same. In the remaining eleven queries, Neo4j clearly outperformed the relational alternative, ranging from 1.33 to almost 26 times faster. For all *PJA queries* (Queries 1 through 5), the graph alternative clearly outperforms the relational one. Note that this occurs for queries addressing both, events (the smaller fact table, Queries 1 and 2) and releases (Queries 3, 4, and 5). The probable reason for this is that neighbours are found very fast in Neo4j, due to its internal graph representation mechanism. Once the neighbours are found, aggregation is performed very efficiently. In the case of SQL, the join must be computed, and this turns out, in general, to be more expensive than finding the neighbours of a node. For *SPJA queries* (Queries 6 through 8), execution times are similar for both alternatives, except for Query 8, which requires a self join of the event facts, and the graph implementation behaves better than the relational one. The reason here is that selections in RDBMSs are very efficiently performed thanks to the indexing of the filtering attributes, and this compensates the cost of the joins. Probably the most surprising results are obtained for self-join queries (the *FPJA queries*), Queries 9 and 10, where the differences are clearly in favour of the graph alternative. Finally, for the *PJA-DA (drill-across) queries* (Queries 12 and 13), execution times seem to depend highly on the selectivity of the filtering attributes, but nevertheless, Neo4j behaves at least the same than SQL (like in Query 12, which does not include a selection).

6 Conclusion and Open Problems

This paper studied the plausibility of graph databases, and the property graph data model, for representing and implementing data warehouses modelled as star and snowflake schemas, using the MusicBrainz database. The results in

most situations, the graph representation was clearly faster, up to one order of magnitude. Building on the results reported in this paper, future work includes looking for new case studies, that would lead to building a benchmark to evaluate graph databases for OLAP queries.

Acknowledgments. Alejandro Vaisman was partially supported by project PICT-2017-1054, from the Argentinian Scientific Agency.

References

1. Angles, R.: A comparison of current graph database models. In: Proceedings of ICDE Workshops, Arlington, VA, USA, pp. 171–177 (2012)
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. *ACM Comput. Surv.* **50**(5), 68:1–68:40 (2017)
3. Angles, R., Gutierrez, C.: Survey of graph database models. *ACM Comput. Surv.* **40**(1), 1:1–1:39 (2008)
4. Gómez, L.I., Kuijpers, B., Vaisman, A.A.: Performing OLAP over graph data: query language, implementation, and a case study. In: Proceedings of BIRTE, Munich, Germany, 28 August 2017, pp. 6:1–6:8 (2017)
5. Hartig, O.: Reconciliation of RDF* and property graphs. *CoRR*, abs/1409.3288 (2014)
6. Kimball, R.: *The Data Warehouse Toolkit*. Wiley, New York (1996)
7. Robinson, I., Webber, J., Eifré, E.: *Graph Databases*. O’Reilly Media, Sebastopol (2013)
8. Vaisman, A., Zimányi, E.: *Data Warehouse Systems: Design and Implementation*. Springer, Heidelberg (2014). <https://doi.org/10.1007/978-3-642-54655-6>