Model-Driven Design of Graph Databases

Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone

Dipartimento di Ingegneria
Università Roma Tre, Rome, Italy
{dvr,maccioni,torlone}@dia.uniroma3.it

Abstract. Graph Database Management Systems (GDBMS) are rapidly emerging as an effective and efficient solution to the management of very large data sets in scenarios where data are naturally represented as a graph and data accesses mainly rely on traversing this graph. Currently, the design of graph databases is based on best practices, usually suited only for a specific GDBMS. In this paper, we propose a model-driven, system-independent methodology for the design of graph databases. Starting from a conceptual representation of the domain of interest expressed in the Entity-Relationship model, we propose a strategy for devising a graph database in which the data accesses for answering queries are minimized. Intuitively, this is achieved by aggregating in the same node data that are likely to occur together in query results. Our methodology relies a logical model for graph databases, which makes the approach suitable for different GDBMSs. We also show, with a number of experimental results over different GDBMSs, the effectiveness of the proposed methodology.

1 Introduction

Social networks, Semantic Web, geographic applications, and bioinformatics are examples of a significant class of application domains in which data have a natural representation in terms of graphs, and queries mainly require to traverse those graphs. It has been observed that relational database technology is usually unsuited to manage such kind of data since they hardly capture their inherent graph structure. In addition, graph traversals over highly connected data involve complex join operations, which can make typical operations inefficient and applications hard to scale. This problem has been recently addressed by a new brand category of data management systems in which data are natively stored as graphs, nodes and edges are first class citizens, and queries are expressed in terms of graph traversal operations. These systems are usually called GDBMSs (Graph Database Management Systems) and allow applications to scale to very large graph-based data sets. In addition, since GDBMSs do not rely on a rigid schema, they provide a more flexible solution in scenarios where the organization of data evolves rapidly.

GDBMSs are usually considered as part of the NoSQL landscape, which includes non-relational solutions to the management of data characterized by elementary data models, schema-free databases, basic data access operations,

and eventually consistent transactions [8,13]. GDBMSs are considered however a world apart from the other NoSQL systems (e.g., key-value, document, and column stores) since their features are quite unique in both the data model they adopt and the data-access primitives they offer [18].

In this framework, it has been observed that, as it happens with traditional database systems [5], the availability of effective design methodologies would be very useful for database developers [3,8,13]. Indeed, also with NoSQL systems, design choices can have a significant impact on the performances and the scalability of the application under development [18]. Unfortunately however, database design for GDBMS is currently based only on best practices and guidelines, which are usually related to a specific system, and the adoption of traditional approaches is ineffective [4]. Moreover, design strategies for other NoSQL systems cannot be exploited for graph databases since the underlying data models and the systems used for their management are very different.

In this paper, we try to fill this gap by proposing a general, model-driven [19] methodology for the design of graph databases. The approach starts, as usual, with the construction of a conceptual representation of application data expressed in the ER model. This representation is translated into a special graph in which entities and relationships are suitably grouped according to the constraints defined on the ER schema. The aim is to try to minimize the number of access operations needed to retrieve related data of the application. This intermediate representation refers to an abstract, graph-based data model that captures the modeling features that are common to real GDBMSs. This makes the approach independent of the specific system that will be used to store and manage the final database. We also provides a number of experimental results showing the advantages of our proposal with respect to a naive approach in which a graph database is derived by simply mapping directly conceptual to physical objects.

The rest of the paper is organized as follows. In Section 2 we introduce an abstract model for graph database and discuss basic strategies for modeling data with a GDBMS. Section 3 illustrates in detail our design methodology whereas Section 4 illustrates the experimental results. In Section 5 we discuss related works and finally, in Section 6, we sketch conclusions and future work.

## 2   Modeling Graph Databases

2.1   Graph Databases

The spread of application domains involving graph-shaped data has arisen the interest on graph databases and GDBMSs. Unfortunately, due to diversity of the various systems and of the lack of theoretical studies on them, there is no widely accepted data model for GDBMSs and of the basic features they should provide. However, almost all the existing systems exhibit three main characteristics.

First of all, a GDBMS stores data by means of a multigraph[1], usually called *property graph* [17], where both nodes and edges are labelled with data in the form of key-value pairs.

---

[1] A multigraph is a graph where two nodes can be connected by more than one edge.

Definition 1 (Graph database). *A graph database, is a directed multigraph* $\mathbf{g} = (N, E)$ *where every node* $n \in N$ *and every edge* $e \in E$ *is associated with a set of pairs* $\langle key, value \rangle$ *called properties.*

A simple example of graph database is reported in Fig. 1: it represents a portion of a database storing information about blogs, having users as administrators and/or followers. Nodes n_1 and n_2 represent a user and a blog, respectively. They both have an id and a name as properties. The edges between n_1 and n_2 represent the relationships follower and admin, respectively (in our case, the user is both follower and admin of the blog), and are associated with properties that simply specify these relationships.
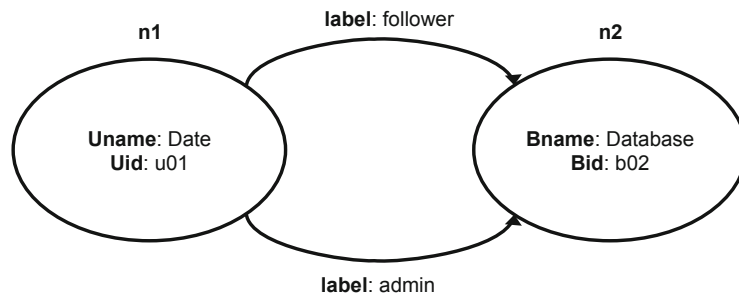
Fig. 1. An example of property graph

Note that this notion of graph database is very flexible since no further constraint is imposed on graphs and their topology. In particular, nodes representing objects of the same class (e.g., different users) can differ in the number of properties and in the data type of a specific property. This makes the data model very general and able to capture various graph-based data models, such as RDF and RDFS. Indeed, GDBMSs are often used in Semantic Web applications [1,14].

The second important feature of a GDBMS is the property of *index-free adjacency.*

Definition 2 (Index Free Adjacency). *We say that a graph database g satisfies the index-free adjacency if the existence of an edge between two nodes of g can be tested by visiting those nodes and does not require the existence of an external, global, index.*

In other words, each node carries the information about its neighbors and no global index of reachability between nodes exists. As a result, the traversal of an edge from a node is independent on the size of data and can be tested in constant time. This guarantees that local analysis can be performed very efficiently on GDBMS and this makes those systems suitable in scenarios where the size of data increases rapidly.

The third feature common to GDBMSs is the fact that data is queried using path traversal operations expressed in some graph-based query language, such as Gremlin[2]. We will not address specific query languages in this paper.

2.2   Modeling Strategies for Graph Databases

A main goal in the design of a graph database is the minimization of data access operations needed in graph traversals at query time. Intuitively, this can be achieved in two different ways: (i) by adding edges between nodes or (ii) by merging different nodes. We call these approaches *dense* and *compact* strategy, respectively.

Basically, the *dense* strategy heavily relies on adding as many edges as possible between nodes representing conceptual entities. This clearly reduces the length of paths between nodes and so the number of data access operations needed at run time. However, it requires to add edges that do not correspond to conceptual relationships in the application domain and such "semantic enrichment" demands an additional effort of the designer.

Conversely, the *compact* strategy relies on aggregating in the same node data that are related but are stored in different nodes. This clearly reduces the number of data accesses as well but, on the other hand, it asks the designer to deal with possible data inconsistencies. Consider for instance the case in which we decide to merge each user node with the blog nodes he follows in the database in Fig. 1 to make more efficient queries involving both users and blogs. If the users follow multiple blogs we have a conflict on the Bname property, which requires a suitable renaming of keys.

Actually, some modeling approaches for graph-shaped data (e.g., in the direct conversion of relational data into RDF graphs [20]) follow yet another strategy, which we call *sparse*. Basically, in the sparse strategy the properties of an object with n properties is decomposed into a set of n different nodes, with the goal of minimizing the number of edges incident to the nodes of the graph. This avoids potential conflicts between the properties of a node but it usually increases largely the number of nodes that need to be traversed during query execution. Moreover, it can make database updates inefficient since the simple insertion or deletion of an object requires multiple database accesses, one for each property of the object.

3   Graph Database Design

This section illustrates a design methodology for graph databases. Our solution supports the user in the design a graph database for the application on the basis of an automatic analysis of a conceptual representation of the domain of interest. In principle, any conceptual data model could be used and in this paper we will consider the Entity-Relationship (ER) model.

---

[2] `https://github.com/thinkaurelius/titan/wiki/Gremlin-Query-Language`

As in the compact strategy, our technique tries to reduce the number of data access required at runtime by aggregating objects occurring in the conceptual representation as much as possible. In addition, to preserve the semantics of the application domain, we also try to take advantage from the benefits of the sparse strategy discussed in Section 2.2. In fact, similarly to the sparse strategy, the aggregation technique avoids potential inconsistencies between properties of nodes. This is done by carefully analyzing the many-to-many relationships of the the ER diagram, which may introduce many connections between nodes and thus conflicting properties in their aggregation.

In our methodology human intervention is extremely reduced: in most cases the translation between the conceptual representation and the graph database is completely automatic and the designer does not need to introduce artificial concepts and relationships. Indeed, all the elements of the output database originate directly from concepts appearing in the input Entity-Relationship diagram. This is coherent with the NoSQL philosophy where the persistence layer is semantically close to the design layer.

In this paper, we refer to a basic version of the ER including entities, relationships, attributes and cardinalities. However such a choice does not introduce limitations on the generality of our approach.
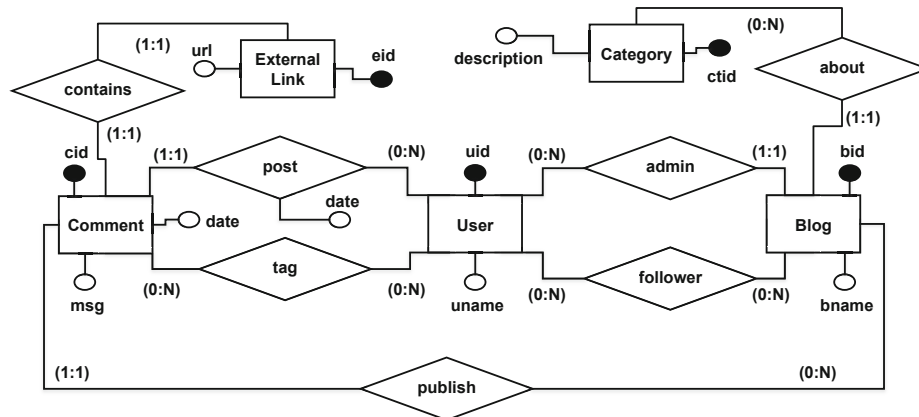

Fig. 2. An example of ER diagram

In the following, we will consider, as running example, an application domain of blogs represented in conceptual terms by the ER diagram in Fig. 2. Intuitively, our strategy aims at building a "template" of a graph database for this initial schema. This template describes how data have to be organized into nodes and how nodes have to be connected to each other. The design strategy is organized in three different phases: (i) generation of an oriented ER diagram, (ii) partitioning of the elements (entities and relationships) of the obtained diagram and (iii) definition of a template over the resulting partition.

3.1   Generation of an Oriented ER Diagram

In the first phase, we transform an ER diagram, which is an undirected and labelled graph, into a directed, labelled and weighted graph, called *Oriented ER* (O-ER) diagram. In O-ER diagrams, a special function w assigns a weight with each edge of the diagram.
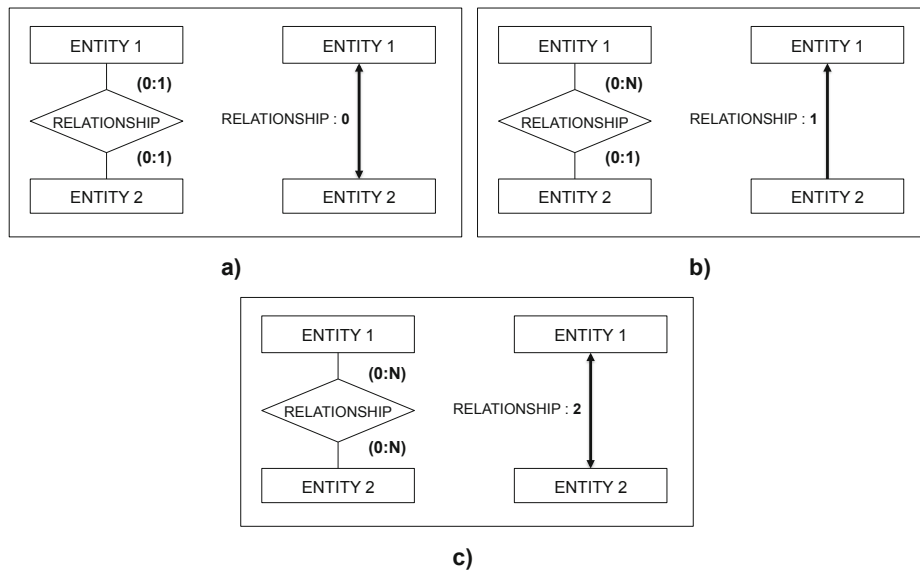

Fig. 3. Rules for generating an O-ER diagram

As illustrated in Fig. 3, the O-ER diagram is generated by applying to an ER diagram the following transformation rules.

a) A one-to-one relationship becomes a double directed edge e such that $w(e) = 0$;

b) A one-to-many relationship becomes a single-directed edge e such that $w(e) = 1$ going from the entity with lower multiplicity to the entity with higher multiplicity;

c) A many-to-many relationship becomes a double-directed edge e with $w(e) = 2$.

All entities and attributes (including those of relationships) are kept in the output O-ER diagram.

For instance, given the ER diagram of Fig. 2, by applying the rules discussed above we obtain the O-ER diagram shown in Fig. 4 (in the figure, we have omitted the attributes for the sake of readability).

Note that our methodology can refer to other data modeling formalisms, such as UML, by just adapting this phase.
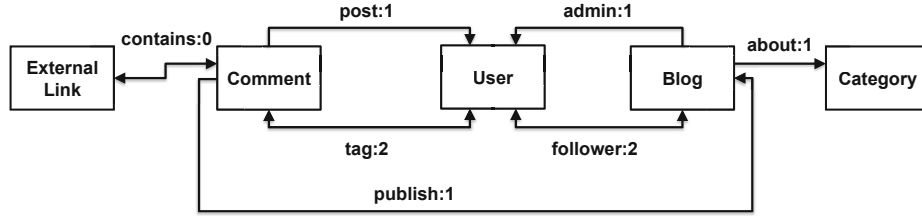
Fig. 4. An example of O-ER diagram

3.2   Partitioning of an Oriented ER Diagram

The second phase consists in the partitioning of the O-ER diagram we have obtained in the first phase. It is based on a set of rules for grouping together elements of the O-ER diagram so that every element of the diagram belongs to one and only one group. The aim of this step is to identify entities whose instances are likely to occur together in the same query results. Intuitively, this reduces the number of accesses to the database needed for query answering.

Let us consider an O-ER diagram in terms of a graph $\langle N, E, w \rangle$, where N is the set of nodes (entities), E is the set of edges, and w the weighting function. Then let $\mathsf{in}(n) = \{(m,n) \mid (m,n) \in E\}$ and $\mathsf{out}(n)\{(n,m) \mid (n,m) \in E\}$ be the sets of incoming and outcoming edges of a node n, respectively. Then consider the following weight functions for nodes of an O-ER diagram:

$$w^+(n) = \sum_{e \in out(n)} w(e)$$

$$w^-(n) = \sum_{e \in in(n)} w(e)$$

The functions $w^-(n)$ and $w^+(n)$ compute the sum of the weights of the incoming and outcoming edges of a node n, respectively. For instance, referring to Fig. 4, the weights associated with the node Comment are the following

$$w^+(\mathsf{Comment}) = w(\mathsf{post}) + w(\mathsf{tag}) + w(\mathsf{publish}) + w(\mathsf{contains}) = 1 + 2 + 1 + 0 = 4$$

$$w^-(\mathsf{Comment}) = w(\mathsf{tag}) + w(\mathsf{contains}) = 2 + 0 = 2$$

The partitioning is then based on the following rules for grouping nodes of an O-ER diagram.

– **Rule 1:** if a node n is disconnected then it forms a group by itself;
– **Rule 2:** if a node n has $w^-(n) > 1$ and $w^+(n) \geqslant 1$ then n forms a group by itself. Intuitively, in this case the node n represents an entity involved with high multiplicity in many-to-many relationships. Therefore we do not aggregate n with other nodes having a similar weight. This rule applies for example to the nodes User, Comment and Blog in the diagram of Fig. 4;

– **Rule 3:** if a node n has $w^-(n) \leqslant 1$ and $w^+(n) \leqslant 1$ then n is added to the group of a node m such that there exists the edge (m, n) in the O-ER diagram. In this case, the node n corresponds to an entity involved in a one-to-one relationship or in a one-to-many relationships in which n has the lower multiplicity. This rule applies for example to the nodes Category and External Link in the diagram of Fig. 4: Category is aggregated with Blog and External Link with Comment.

Note that these rules can be applied either recursively or iteratively. An iterative procedure would analyze once every node of the O-ER diagram. By applying these rules to the diagram in Fig. 4, we obtain the partition shown in Fig. 5.
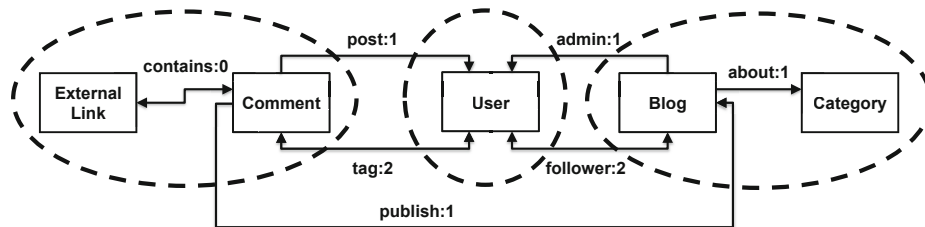

Fig. 5. An example of partitioning.

3.3   Definition of a Template over a Graph Database

Once the groups are formed, the third phase generates a *template* for the final graph database. While it is improper to speak of a schema for graph databases (as usually data do not strictly adhere to any schema), there are always similar nodes in a graph database, that is, nodes that share many attributes and are connected by the same kind of edges. Therefore we can say that homogeneous nodes identify a "data type". Basically, a *template* describes the data types occurring in a graph database and the ways they are connected. Thus, it represents a logical schema of the graph database that can be made transparent to the designer and to the user of the target database.

Indeed, the database instance is not forced to conform the template in a rigid way. Rather, it is the initial structure of the graph database that can be later extended or refined. In addition, it is a valid mean to address the impedance mismatch between the persistence layer and the application layer.

Then, a template works as a "schema" for a graph database where each node and each edge is equipped with the names (i.e., the keys) of the involved properties. The names of properties originate from the attributes of the entities occurring in the same group. Every attribute determines a property name of the instance of an entity. In a template, a property name is composed by the name of the entity concatenated to the name of the attribute it originates from (e.g., User.uname). In the same way, the attributes of the relationships between entities determine the names of the property of the edges connecting the corresponding
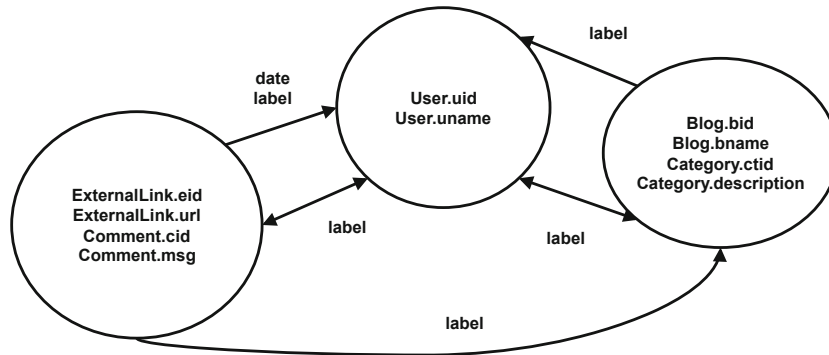
Fig. 6. An example of template.

nodes in the graph database (e.g., date for the relationship post). In addition, for each relationship r with label l we introduce the property name label that will be associated with the value l in the graph database.

As an example, the template produced from the partitioning shown in Fig. 5 is illustrated in Fig. 6. Note as the template adheres perfectly to the input domain without "artificial" concepts that are not present in the conceptual representation. Using the template, we can generate different instances. For example, Fig. 7 illustrates an instance of graph database conforming to the template in Fig. 6. In this case we have three instances of User, three instances of Blog (with the corresponding Category) and one instance of Comment (with the corresponding External Link).

4   Experimental Results

In order to evaluate the effectiveness of our methodology, we have implemented a tool that aggregates data using the technique illustrated in Section 3. In particular, we have extended a system, called R2G, that we have developed for migrating relational to graph databases [11].

With this tool, we have compared the query performances of graph databases obtained with our strategy with those obtained with the sparse strategy, which is adopted by the most common GDBMSs: Neo4J [22], ArangoDB[3], Infinite-Graph[4], Oracle NoSQL[5], OrientDB[6] and Titan[7]. Our system makes use of the Blueprints framework[8], a general, open-source API for graph databases adopted by all GDBMS. Blueprints, as JDBC, allows developers to plug-and-play their

---

[3] https://www.arangodb.org/

[4] http://www.objectivity.com/infinitegraph

[5] http://www.oracle.com/technetwork/database/database-technologies/
nosqldb/overview/index.html

[6] http://www.orientechnologies.com/orientdb/

[7] http://thinkaurelius.github.io/titan/

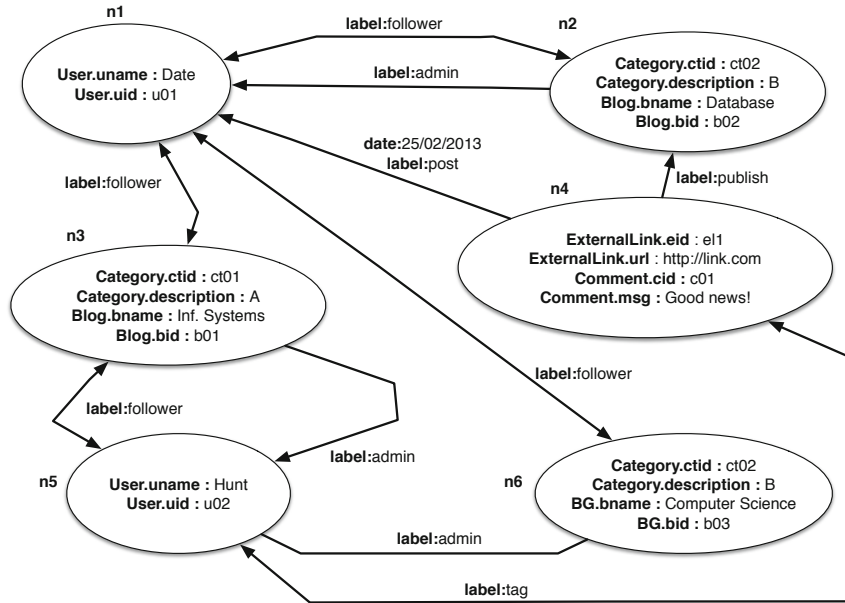[8] https://github.com/tinkerpop/blueprints/wiki

Fig. 7. The final graph database

graph database back-end. In this way, R2G is able to connect to each GDBMS and import data by using both the proprietary strategy (sparse-like) and our strategy.

Experiments were conducted on a dual core 2.66GHz Intel Xeon, running Linux RedHat, with 4 GB of memory and a 2-disk 1Tbyte striped RAID array.

Aggregate data sources. In a first experiment we considered, as in [11], aggregate datasets with different sizes. Such data sources present data highly correlated where the sparse strategy fits nicely (i.e. graph traversal operations are performed over quite short paths). We used a relational representation of MONDIAL (17.115 tuples and 28 relations) and of two ideal counterpoints (due to the larger size): IMDB (1.673.074 tuples in 6 relations) and WIKIPEDIA (200.000 tuples in 6 relations), as described in [9]. The authors in [9] defined a benchmark of 50 keyword search queries for each dataset. We used the tool in [7] to generate SQL queries from the keyword-based queries defined in [9]. The SQL queries are then mapped to the Gremlin language supported by the Blueprints framework.

Then, we evaluated the performance of query execution. For each dataset, we ran the 50 queries ten times and measured the average response time. We performed *cold-cache* experiments (i.e. by dropping all file-system caches before restarting the various systems and running the queries) and *warm-cache* experiments (i.e. without dropping the caches). Fig. 8 shows the performance for cold-cache experiments. Due to space constraints, in the figure we report times only on IMDB and WIKIPEDIA, since their much larger size poses more
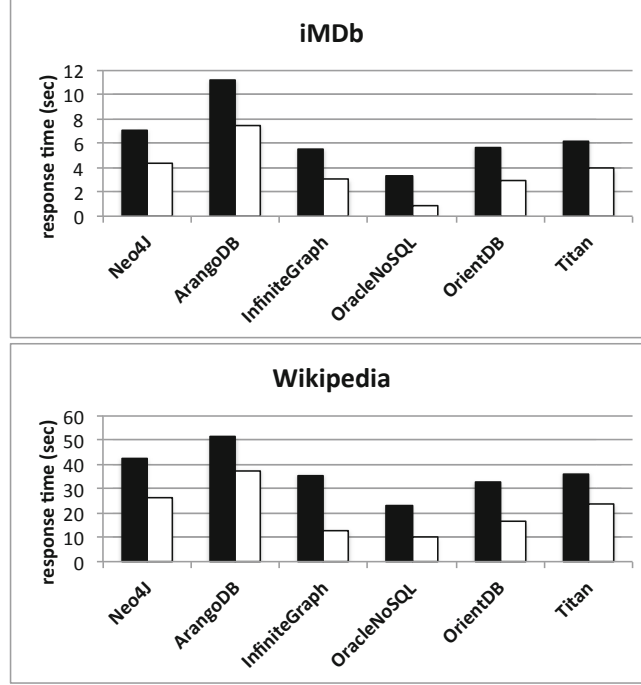
Fig. 8. Performance on aggregate data sources: black bars refer to the sparse strategy and white bars refer to our strategy

challenges. In the figure, for each GDBMS we consider the time to perform a query on the graph database generated by using the sparse strategy (i.e. black bar) and the time to perform the same query on the graph database generated by using our strategy (i.e. white bar).

Our methodology allows to each system to perform consistently better for most of the queries. This is due to our strategy reducing the space overhead and consequently the time complexity of the overall process w.r.t. the competitors strategy that spends much time traversing a large number of edges. Warm-cache experiments follow a similar trend.

A significant result is the *speed-up* between the two strategies. For each dataset D, we computed the speed-up for all systems P as the ratio between the average execution time over the graph database generated by the sparse strategy of P, and that of our strategy in R2G, or briefly $S_D = {}^{t_P}/t_{R2G}$: $S_{\mathrm{IMDB}} = 2,03$, $S_{\mathrm{WIKIPEDIA}} = 1,97$, and $S_{\mathrm{MONDIAL}} = 2,01$.

Disaggregate data sources. In a second experiment we used a different benchmark for path-traversal queries. In particular, we have considered disaggregate datasets with a large number of nodes sparsely connected and long paths between nodes. To this aim, we used the *graphdb-benchmarks project*[9] that involves

---

[9] `https://github.com/socialsensor/graphdb-benchmarks`
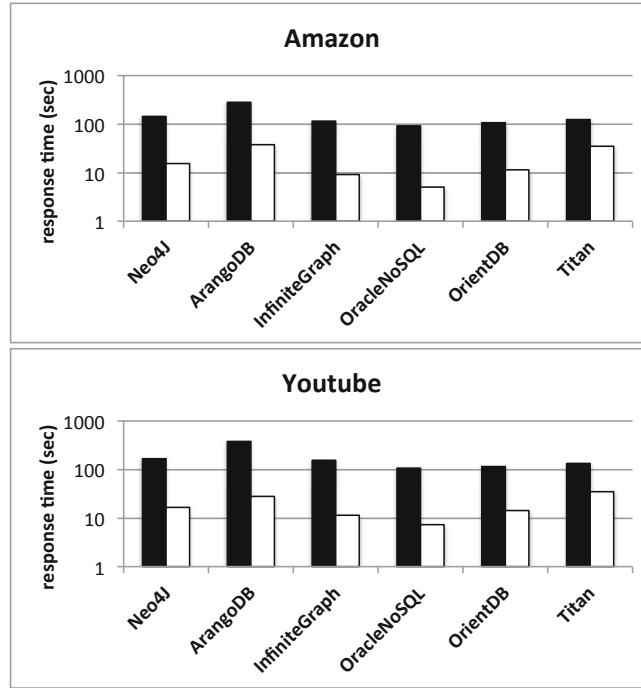
Fig. 9. Performance on disaggregate data sources: black bars refer to the sparse strategy and white bars refer to our strategy

social data from AMAZON, YOUTUBE and LIVEJOURNAL. This benchmark involves different path-traversal queries: (i) to find the neighbors of a node, (ii) to find the nodes of a edge and (iii) to find the shortest path between two nodes.

As in the first experiment, for each dataset we ran the queries ten times and measured the average response time. The final results are depicted in Fig. 9 (times are in seconds and the scale is logarithmic). Due to space constraints we omitted the results for LIVEJOURNAL since they are quite similar to the other datasets. In this case our strategy generates databases that perform significantly better in each system. In this context, the aggregation at the basis of our methodology reduces significantly the number of nodes to traverse, contrary to the sparse strategy, as shown by the following speed-up: $S_{\text{AMAZON}} = 9,97$, $S_{\text{YOUTUBE}} = 10,01$, $S_{\text{LIVEJOURNAL}} = 9,98$. In words, our strategy allows us to perform queries 10 times better than the proprietary strategy of each system.

5   Related Work

The idea of storing and managing graph-based data natively is quite old (see [2] for an extensive survey on this topic) and is recently re-born with the advent of the Semantic Web and other emerging application domains, such as social

networks and bioinformatics. This new interest has led to the development of a number of GDBMSs that are becoming quite popular in these scenarios.

In spite of this trend, the approach presented in this paper is, to our knowledge, the first general methodology for the design of graph databases and so the related bibliography is very limited. Batini et al. [6] introduce a logical design for the Network model [21] that follows a sparse-like strategy for mapping an ER schema into a Network schema. Current approaches mainly rely on best practices and guidelines based on typical design patterns, published by practitioners in blogs [15] or only suited for specific systems [16]. In [12], the author gathers different design patterns for various NoSQL data stores, including one for graph databases called *application side joins*. This design pattern is based on the join operations that need to be performed over the database. Conversely, we do not make any assumption on the way in which the database under development is accessed and our approach relies only on the knowledge of conceptual constraints that can be defined with the ER model. Moreover, it provides a system-independent intermediate representation that makes it suitable for any GDBMS.

In earlier works [10,11], we have designed and developed a tool for migrating data from a relational to a graph database management system. In this work, we consider a different scenario where the database needs to be built from scratch.

6   Conclusion and Future Work

In this paper we have presented a design methodology for graph databases. Our approach involves a preliminary conceptual design phase followed by a strategy for translating the conceptual representation into a intermediate representation that is still independent of the specific target system. The goal is to try to keep together data that are likely to occur together in query results while keeping separate independent concepts. An evaluation study shows that our methodology provides considerable advantages in terms of query performance with respect to naive approaches.

In the future work we will consider more aspects for driving the design process such as transaction requirements and query operation loads for the application at hand. This information can improve the effectiveness of the methodology by helping to disambiguate between different, possible decisions. We also intend to verify if a similar approach is also possible for other NoSQL data stores, in particular to key-value and document stores.

References

1. Angles, R., Gutierrez, C.: Querying RDF data from a graph database perspective. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 346–360. Springer, Heidelberg (2005)
2. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1), 1–39 (2008)

3. Atzeni, P., Jensen, C.S., Orsi, G., Ram, S., Tanca, L., Torlone, R.: The relational model is dead, SQL is dead, and i don't feel so good myself. SIGMOD Record 42(2), 64–68 (2013)
4. Badia, A., Lemire, D.: A call to arms: revisiting database design. SIGMOD Record 40(3), 61–69 (2011)
5. Batini, C., Ceri, S., Navathe, S.B.: Conceptual Database Design: An Entity-Relationship Approach. Benjamin/Cummings (1992)
6. Batini, C., Ceri, S., Navathe, S.B.: Conceptual Database Design: An Entity-Relationship Approach. Benjamin/Cummings (1992)
7. Bergamaschi, S., Domnori, E., Guerra, F., Lado, R.T., Velegrakis, Y.: Keyword search over relational databases: A metadata approach. In: SIGMOD Conference, pp. 565–576 (2011)
8. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Record 39(4), 12–27 (2010)
9. Coffman, J., Weaver, A.C.: An empirical performance evaluation of relational keyword search techniques. TKDE 26(1), 30–42 (2014)
10. De Virgilio, R., Maccioni, A., Torlone, R.: Converting relational to graph databases. In: SIGMOD Workshops - GRADES (2013)
11. De Virgilio, R., Maccioni, A., Torlone, R.: R2G: A tool for migrating relations to graphs. In: EDBT (2014)
12. Katsov, I.: NoSQL data modeling techniques (2012), `http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques`
13. Mohan, C.: History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In: EDBT, pp. 11–16 (2013)
14. Ovelgönne, M., Park, N., Subrahmanian, V.S., Bowman, E.K., Ogaard, K.A.: Personalized best answer computation in graph databases. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 478–493. Springer, Heidelberg (2013)
15. Parastatidis, S.: On graph data model design (2013), `http://savas.me/2013/03/on-graph-data-model-design-relationships/`
16. Robinson, I.: Designing and building a graph database application with neo4j. In: Graph Connect (2013)
17. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. CoRR abs/1006.2361 (2010)
18. Sadalage, R.J., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional (2012)
19. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
20. Sequeda, J., Arenas, M., Miranker, D.P.: On directly mapping relational databases to RDF and OWL. In: WWW, pp. 649–658 (2012)
21. Taylor, R.W., Frank, R.L.: Codasyl data-base management systems. ACM Comput. Surv. 8(1), 67–103 (1976)
22. Webber, J.: A programmatic introduction to neo4j. In: SPLASH, pp. 217–218 (2012)