ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Logical design of multi-model data warehouses

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Sandro Bimonte, Enrico Gallinucci, Patrick Marcel, Stefano Rizzi (2023). Logical design of multi-model data warehouses. KNOWLEDGE AND INFORMATION SYSTEMS, 65, 1067-1103 [10.1007/s10115-022-01788-0].

Availability:
This version is available at: https://hdl.handle.net/11585/915602 since: 2023-02-15

Published:
DOI: http://doi.org/10.1007/s10115-022-01788-0

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2026

Under consideration for publication in Knowledge and Information Systems

Logical design of multi-model data warehouses

Sandro Bimonte[1], Enrico Gallinucci[2], Patrick Marcel[3] and Stefano Rizzi[2]

[1]INRAE - TSCF, University of Clermont Auvergne, Aubiere, France; [2]DISI, University of Bologna, Bologna, Italy; [3]LIFAT Laboratory, University of Tours, Tours, France

Abstract. Multi-model DBMSs, which support different data models with a fully integrated backend, have been shown to be beneficial to data warehouses and OLAP systems. Indeed, they can store data according to the multidimensional model and, at the same time, let each of its elements be represented through the most appropriate model. An open challenge in this context is the lack of methods for logical design. Indeed, in a multi-model context, several alternatives emerge for the logical representation of dimensions and facts. The goal of this paper is to devise a set of guidelines for the logical design of multi-model data warehouses so that the designer can achieve the best trade-off between features such as querying, storage, and ETL. To this end, for each model considered (relational, document-based, and graph-based) and for each type of multidimensional element (e.g., non-strict hierarchy) we propose some solutions and carry out a set of intra-model and inter-model comparisons. The resulting guidelines are then tested on a case study that shows all types of multidimensional elements.

Keywords: OLAP; Multi-Model Databases; Data Variety; Data Warehouse

1. Introduction

A *multi-model DBMS* (MMDBMS) is a data managing platform that supports different data models with a fully integrated backend, thus providing unified data governance, management, and access via a single query language, while still granting performance, scalability, and fault tolerance (Lu and Holubová, 2019). Using a single platform for multi-model data delivers several benefits to users besides that of providing a unified query interface; namely, it will reduce maintenance and data integration issues, speed up development, and eliminate

Received 13 Dec 2021
Revised 10 Aug 2022
Accepted 23 Oct 2022

migration problems (Tsunakawa, 2017; Lu and Holubová, 2019). Examples of MMDBMSs are PostgreSQL, ArangoDB, Cosmos DB, and CouchBase. Specifically, PostgreSQL (`www.postgresql.org/`) is a relational DBMS that supports the row-oriented, column-oriented, key-value, and document-oriented data models, offering XML, hstore, JSON/JSONB data types for storage.

MMDBMSs can effectively cope with the variety issues that characterize big data while preserving volume and velocity. Within modern architectures of information systems, this is particularly valuable to manage *data lakes*. Data lakes have been defined as central repository systems for storage, processing, and analysis of raw data, in which the data are kept in their original format and is processed to be queried only when needed; they can store a varied amount of formats in big data ecosystems, from unstructured, semi-structured, to structured data sources (Couto, Borges, Ruiz, Marczak and Prikladnicki, 2019). The support to variety, volume, and velocity ensured by MMDBMSs promises to be beneficial to data warehouses (DWs) and On-Line Analytical Processing (OLAP) systems too: in fact, warehoused data result from the integration of huge volumes of heterogeneous data, and OLAP requires very good querying performances (Bimonte, Hifdi, Maliari, Marcel and Rizzi, 2020). A *multi-model data warehouse* (MMDW) could store data according to the multidimensional model and, at the same time, let each of its elements be represented through the most appropriate model.

An investigation of the effectiveness and efficiency of MMDWs to store multidimensional data is done by Bimonte, Gallinucci, Marcel and Rizzi (2022). Specifically, starting from the UniBench multi-model benchmark (Zhang, Lu, Xu and Chen, 2018), a conceptual multidimensional schema is defined first, then three logical schemata are proposed to support it. The first one (called M^3D) extends the star schema by introducing semi-structured (JSON, XML, graph-based, and key-value) data in the multidimensional elements. The second one is a classical (full-relational) star schema. The third one corresponds to a data lake-like approach where data are not put in multidimensional form and maintain their source format. These three schemata are implemented using PostgreSQL (with the AgensGraph extension, `bitnine.net/agensgraph/`, to support graph data) and compared in terms of efficiency and effectiveness. Remarkably, it is shown that M^3D offers a valuable trade-off between querying performance, ETL costs, design flexibility, extensibility in presence of variable source schemata, and evolvability.

One of the research challenges left open by Bimonte et al. (2022) is the lack of best practices for logical design. Indeed, in a multi-model setting, several alternatives emerge for the logical representation of dimensions and facts (Ferrahi, Bimonte and Boukhalfa, 2017), and some of them may be better than others from one or more points of view. The goal of this paper is to devise a set of guidelines for the logical design of MMDWs so that the designer can achieve the best trade-off between features such as querying, storage, and ETL. To this end, for each model considered (relational, document-based, and graph-based) and for each type of multidimensional element (essentially related to how hierarchies are structured, e.g., shared hierarchy and non-strict hierarchy), we propose some solutions and carry out a set of intra-model and inter-model comparisons. The resulting guidelines are then tested on a case study that shows all types of multidimensional elements. The roadmap we follow to determine the guidelines is sketched in Figure 1.

The novel contributions we offer in this paper can be summarized as follows:

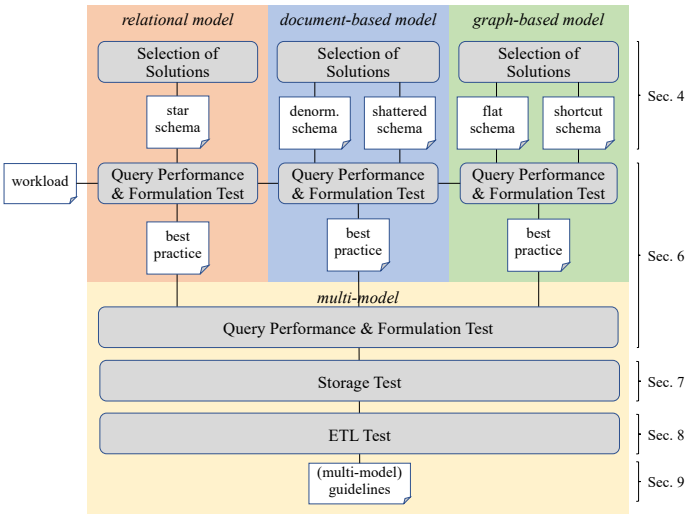

Fig. 1. Test roadmap (with corresponding paper sections)

– We discuss mono-model multidimensional design with reference to the relational, document-based, and graph-based models (Section 4). This leads to recognize five basic schemata: star schema (relational); denormalized and shattered schema (document-based); flat and shortcut schema (graph-based). For each type of multidimensional element, it is shown how it can be modeled in each of these schemata.
– We carry out a set of quantitative comparative tests (whose goal and setting are described in Section 5) aimed at comparing the different schemata at the intra-model and inter-model levels. The comparison is done in terms of querying performance and query formulation complexity (Section 5.1), storage (Section 5.2), and ETL formulation complexity and performance (Section 5.3).
– We address multi-model multidimensional design in Section 6 by first proposing a set of guidelines derived from the analysis carried out in the previous sections, then we apply them to our case study. Other features (flexibility, extensibility, and evolvability) are qualitatively considered here.

The paper is completed by Section 2, which discusses the related literature, Section 3, which describes a case study centered on orders, and Section 7, which draws the conclusion.

2. Related work

2.1. Polystores and multi-model databases

Traditional DBMSs were conceived for handling a specific type of data; for example, relational DBMSs for structured data, document-based DBMSs for semi-structured data, etc. For applications that require the integration of different types of data, two solutions are possible: (i) integrate all data into a single DBMS, or (ii) use two or more DBMSs together. The obvious drawback of the former

solution is that some types of data cannot be stored and analyzed (e.g., the pure relational model does not support the storage of XML, arrays, etc. (Shimura, Yoshikawa and Uemura, 1999)). The latter approach (known as *polyglot persistence* (Gadepally, Chen, Duggan, Elmore, Haynes, Kepner, Madden, Mattson and Stonebraker, 2016)) presents drawbacks as well, namely, a difficulty in technically managing more DBMSs, a steep learning curve for developers, inadequate performance optimization, complex logic in applications, data inconsistency, etc. (Tsunakawa, 2017).

MMDBMSs have been proposed to overcome these issues (see the survey by Lu and Holubová (2019)). MMDBMSs support different models using specific storage strategies; for example, PostgreSQL stores data using relational tables, text, or binary format, while ArangoDB uses a document storage technique. To enable queries on different data models, these DBMSs provide new query languages, namely, extended-SQL and AQL for PostgreSQL and ArangoDB, respectively. Besides, depending on the storage strategy, each DBMS implements a particular set of physical structures (indexes and partitions). The main cloud providers either support multi-model databases to some extent (e.g., Azure's Cosmos DB supports all non-relational models) or offer cross-querying functionalities over multiple database systems (e.g., Amazon's standard data warehousing solution RedShift can be used to query semi-structured and unstructured data on S3).

Among the topics currently investigated for MMDBMSs, we mention conceptual modeling of multi-model data (Holubová, Contos and Svoboda, 2021), inference of multi-model schemata, multi-model querying (Holubová, Svoboda and Lu, 2019) as well as evolution management (Holubová, Klettke and Störl, 2019). To the best of our knowledge, the only work dealing with multi-model logical design starting from a conceptual schema is the one by Svoboda, Contos and Holubová (2021), which uses category theory to formalize the transformation from elements of an Entity/Relationship diagram to relational tables, documents, and graphs in an MMDBMS. The approach is devised for operational databases rather than for multidimensional data; besides, it is more focused on providing a uniform modeling framework than on supporting the designer in deciding which model to use for each piece of data.

2.2. NoSQL OLAP

A DW is a repository of integrated data periodically fed by (possibly heterogeneous) data sources and interactively queried using the OLAP (On-Line Analytical Processing) paradigm. To facilitate OLAP querying, DWs are normally based on the multidimensional model, which introduces the concepts of facts, dimensions, and measures to analyze data. Thus, source data must be transformed to fit a multidimensional logical schema (*schema-on-write* approach). To this end, ROLAP architectures rely on a single, relational DBMS for storage, while MOLAP architectures store data in multidimensional arrays.

To offer better support to large volumes of data while maintaining velocity, some works propose the usage of NoSQL DBMSs. Chevalier, Malki, Kopliku, Teste and Tournier (2016*c*) propose three different logical models, using one or more document collections to store data in document-based DBMSs (Chevalier, Malki, Kopliku, Teste and Tournier, 2016*b*). The same authors also investigate how to handle complex hierarchies and summarizability issues with document-

based DWs (Chevalier, Malki, Kopliku, Teste and Tournier, 2016*a*). A logical model for column-based DWs has been proposed by Boussahoua, Boussaid and Bentayeb (2017) and Chevalier, Malki, Kopliku, Teste and Tournier (2015*a*) to address volume scalability. Sellami, Nabli and Gargouri (2018) propose to use transformation rules for DW implementation in graph-based DBMSs for better handling social network data. Some works also use XML DBMSs for warehousing XML data (Ouaret, Chalal and Boussaid, 2013). While this is a first effort towards native storage of semi-structured data, the querying performances do not scale well with size, and compression techniques must be adopted (Boukraâ, Bouchoukh and Boussaïd, 2015). Among all these proposals, it is hard to champion one logical and physical implementation for NoSQL and XML DWs, since no approach clearly outperforms the other. Moreover, although these single-model proposals offer interesting results in terms of volume and velocity, they have been mainly conceived and tested for structured data, without taking variety into account, neither do they address other issues related to warehousing big data, such as reducing the cost of ETL, evolution, and improving flexibility.

Adopting a schema-on-write approach is not always painless because of the schemaless nature of some source data. This, in some recent papers, OLAP queries are directly rewritten over schemaless data sources that are not organized according to the multidimensional model (*schema-on-read* approach). In this case the multidimensional schema is not devised at design time and forced in a DW, but decided at querying time. Chouder, Rizzi and Chalal (2019) describe a schema-on-read approach to automatically extract facts and hierarchies from document data stores and trigger OLAP queries. A similar approach is presented by Gallinucci, Golfarelli and Rizzi (2019); there, schema variety is explicitly taken into account by choosing not to design a single crisp schema where source fields are either included or absent, but rather to enable an OLAP experience on some sort of "soft" schema where each source field is present to some extent. In the same direction, Dehdouh (2016) proposes a MapReduce-based algorithm to compute OLAP cubes on column stores, while the work by Castelltort and Laurent (2014) aims at delivering the OLAP experience over a graph-based database.

The approaches mentioned above rely on a single-model database. Conversely, Hamadou, Gallinucci and Golfarelli (2019) propose a pay-as-you-go approach which enables OLAP queries against a polystore supporting relational, document, and column data models by hiding heterogeneity behind a dataspace layer. Data integration is carried out on-the-fly using a set of mappings. Even this approach can be classified as schema-on-read; the focus is on query rewriting against heterogeneous databases and not on the performance of the approach.

Bimonte et al. (2022) investigate the effectiveness and efficiency of MMDWs to store multidimensional data. Though one multi-model solution coping with variety is proposed with reference to a case study, there is no discussion and evaluation of all the single- and multi-model solutions made available for the different types of multidimensional elements when relying on a MMDBMS.

3. Case study

UniBench is a benchmark for multi-model databases (Zhang et al., 2018; Zhang and Lu, 2021). It includes a retail dataset composed of relational, XML, JSON, key-value, and graph data, which makes it a good representative for variety. To investigate the pros and cons of MMDBMSs, in (Bimonte et al., 2022) a

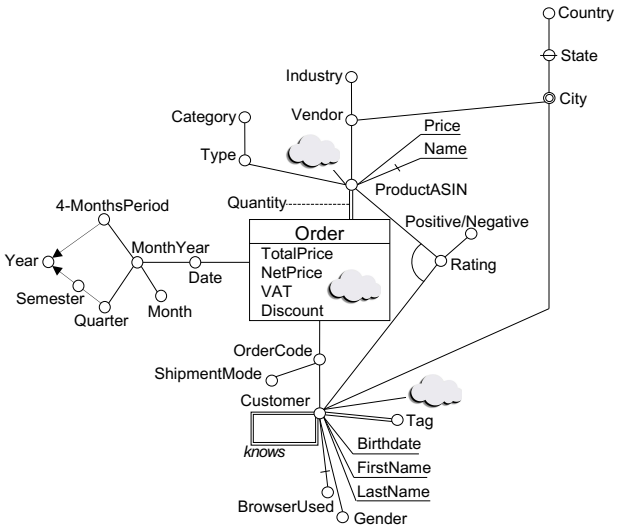

Fig. 2. Multidimensional schema for our case study, based on the DFM notation

multidimensional schema is derived from UniBench by adopting a classical data-driven approach based on the functional dependencies inferred from the data. Though that schema was good for a preliminary study, it is not sufficient in this paper because it is not representative of all the possible situations that may arise in multidimensional modeling. For this reason we had to extend it by adding new types of multidimensional elements, such as shared hierarchies and convergences (Golfarelli and Rizzi, 2009). The resulting extended schema is shown in Figure 2 using the DFM notation (Golfarelli and Rizzi, 2009), where the box represents a fact with its measures surrounded by its dimensions, hierarchy levels are shown as circles, while descriptive properties are underlined. The schema is focused on the Order fact and can be described as follows:

- The fact has four measures, namely, TotalPrice, NetPrice, VAT, and Discount; the cloud symbol inside the box denotes that additional measures that were not known at design time can be fed into the fact as a consequence of the evolution of the data sources.
- The OrderCode dimension describes each order by its shipment mode and customer. Customers can by grouped by their Gender and BrowserUsed (the latter is optional, i.e., it is known only for some customers, as shown in the DFM by a dash on the arc), and are described by some properties, e.g., LastName. Customers are also related to their City. Some unexpected levels and properties can be related to customers, as denoted by the cloud symbol. To model the graph of inter-customer acquaintances, a knows *non-onto hierarchy* (Pedersen, Jensen and Dyreson, 2001) (also called *recursive* (Golfarelli and Rizzi, 2009) or *unbalanced* (Niemi, Nummenmaa and Thanisch, 2001) hierarchy, represented in the DFM with a loop) is set on Customer. Finally, each customer expresses her interest for one or more tags (*non-strict hierarchy*, represented in the DFM with a double arc).
- The temporal dimension has several levels, ranging from Date to Year. The diamond shape denotes a *convergence*, i.e., MonthYear can be aggregated either

by 4-MonthPeriod or by Quarter/Semester, but in both cases one single Year is reached.
– The ProductASIN[1] dimension features, besides a couple of properties, a Vendor hierarchy. Even here, a product can have some additional levels and properties not specified at design time. Since an order is associated with many products, a non-strict hierarchy is set between the fact and the product dimension; each couple of order and product is described by a Quantity.
– A geographical hierarchy rooted in City is shared by vendors and customers (double circle in the DFM). Level State is optional, since not all countries have states (*non-covering hierarchy* (Pedersen et al., 2001), denoted in the DFM with a dash on the circle).
– Level Rating is *cross-dimensional*, i.e., its value is jointly determined by ProductASIN and Customer (a customer can rate several products). This is shown in the DFM with an arc touching two arcs.

4. Mono-model multidimensional design

Several possible alternatives arise when modeling a multidimensional fact within an MMDW. Indeed, the possibility of mixing different models into a single schema gives rise to a huge number of combinations, where two or more models are used even within the same hierarchy. In the direction of providing guidelines for designing these combinations, we proceed by first listing, in the following subsections, the main design alternatives for implementing the different types of multidimensional elements using each model. Then, in Sections from 5.1 to 5.3, we will compare these solutions from different points of view. The models we consider are: relational, document-based, and graph-based. We do not consider key-value because, in key-value data modeling, the value is a black-box that cannot be used for selections nor aggregations (Sadalage and Fowler, 2009), thus it makes little sense to adopt it in a data warehousing context. Document-based model implementations are considered only in JSON and not XML because, as discussed by Bimonte et al. (2022), XML does not add expressiveness to JSON (while yielding slightly worse performances).

4.1. Relational model

Multidimensional design for the relational model has been largely investigated, and a set of best practices is already available for all types of multidimensional elements (Golfarelli and Rizzi, 2009). In the following we briefly recap them:

– In a star schema[2], a dimension table is created for each dimension, storing a (normally surrogate) primary key and one attribute for each level of the corresponding hierarchy.
– Convergences (e.g., the one on Year), properties (e.g., LastName), optional arcs (e.g., the one to BrowserUsed), and non-covering hierarchies (e.g., State) do not require any adjustment to the rule above.

[1] ASIN stands for Amazon Standard Identification Number.
[2] For simplicity we do not consider snowflake schemata here, since the star vs. snowflake issues have already been studied (Golfarelli and Rizzi, 2009).

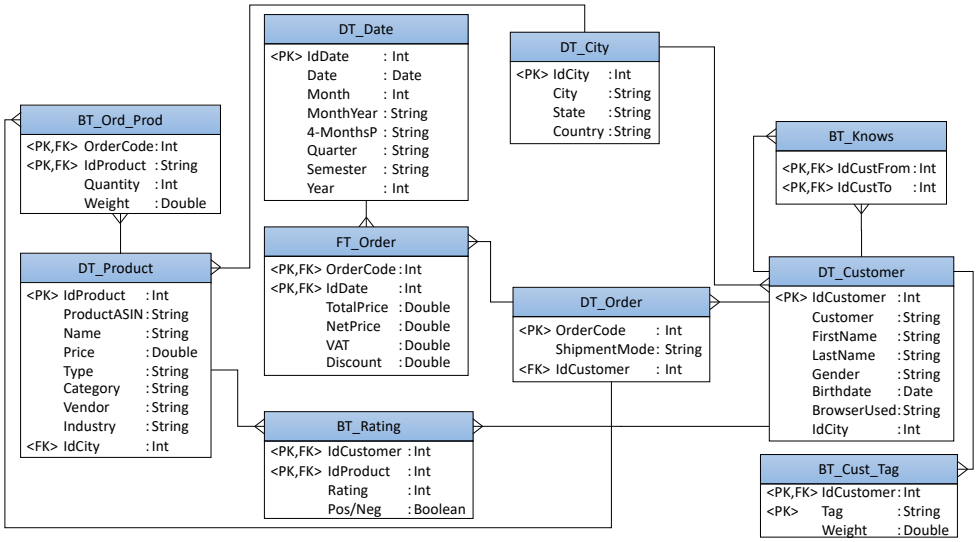

Fig. 3. Star schema for the Order fact

– Non-onto hierarchies (e.g., Knows), cross-dimensional levels (e.g., Rating), and non-strict hierarchies (e.g., the one to Tag) are designed using a bridge table, i.e., a table that establishes a many-to-many association between two dimension tables.
– Shared hierarchies (e.g., the one rooted in City) are designed using snowflaking, i.e., by partially normalizing the dimension table.
– Unexpected levels and measures are simply not dealt with.

The relational schema obtained by applying these guidelines to the Order fact is shown in Figure 3; prefixes FT, DT, and BT are used for fact, dimension, and bridge tables, respectively.

4.2. Document-based model

Multidimensional design for the document-based model has been investigated only to a limited extent, however some papers propose and compare different solutions.

Four different solutions are proposed by Chevalier et al. (2016*c*): a *denormalized flat schema* (where a fact is stored using a single collection of documents including all its measures and levels with no nesting); a *deco schema* (denormalized like the previous one, but the measures and the levels of each dimension are stored in separate subdocuments); a *shattered schema* (where each dimension is stored in a separate collection of documents and connected to the fact documents using a reference); and a *hybrid schema* (like a shattered schema, but with all documents stored within a single collection). These solutions are experimentally compared on MongoDB against the Star Schema Benchmark (O'Neil, O'Neil, Chen and Revilak, 2009) in terms of storage space, loading time, and querying performance, to find out that:

– Due to their redundancy, the first two schemata require about 4 times the space required by the other two, which leads to significantly higher loading times.
– Denormalized flat schemata and shattered schemata tend to have better querying performances; however, there is not a single winner between these two since the execution times largely depend on the query features (mostly, on the number of joins they require).

Chevalier et al. (2016*a*) also propose solutions to deal with irregular hierarchies; specifically, they suggest to adopt arrays to model non-strict hierarchies, and using a dummy value (such as 'other') to balance non-covering hierarchies.

Two solutions are proposed by Challal, Bala, Mokeddem, Boukhalfa, Boussaid and Benkhelifa (2019) and Yangui, Nabli and Gargouri (2016): a *simple schema* (where the fact and each dimension are stored in separate documents of the same collection, like in the hybrid schema mentioned above) and *hierarchical schema* (like a simple schema, but using separate documents for each dimension hierarchy, much like the shattered schema mentioned above). The experimental comparison, made on MongoDB against the TPC-DS benchmark, does not highlight significant differences in loading time and querying performance.

Based on the above, here we consider two solutions: *denormalized schema* and *shattered schema*. These solutions are exemplified in Figure 4, together with the variants we propose for them in presence of different types of multidimensional elements. Nested boxes represent arrays. A brief comment on the solutions:

– Figure 4.a: a fact with two simple hierarchies is modeled either including all levels and measures in the same document (denormalized), or creating three separate documents, one for the fact and one for each dimension (shattered).
– Figure 4.b: a non-onto hierarchy is modeled using a document that includes a reference to itself (denormalized and shattered).
– Figure 4.c: a non-onto and non-strict hierarchy is modeled using a document that includes an array of references to itself (denormalized and shattered).
– Figure 4.d: a convergence is modeled including all the levels in the same document (denormalized and shattered).
– Figure 4.e-f: a non-covering hierarchy and an optional arc are both modeled using optional fields (denormalized and shattered).
– Figure 4.g: a non-strict hierarchy is modeled using an array, either including all levels in the same document (denormalized) or including the children level (b) in a separate document (shattered).
– Figure 4.h: a shared hierarchy is modeled either by replicating the children levels (denormalized) or by including them in a separate document (shattered).
– Figure 4.i: a cross-dimensional level c is modeled either including c and its children in the same document (denormalized, it can be done because, in terms of functional dependencies, ab $\rightarrow$ cd) or including them in a separate document (shattered).

Note that, in practice, the modeling solutions taken in the source JSON documents may possibly differ from the ones considered here. For instance, the concepts included in a plain hierarchy could be modeled starting from the leaves rather than from the root (e.g., for the hierarchy rooted in a in Figure 4.a, there could be a document for each value of c, each including an array of values of a).

The denormalized and the shattered schemata obtained for the Order fact

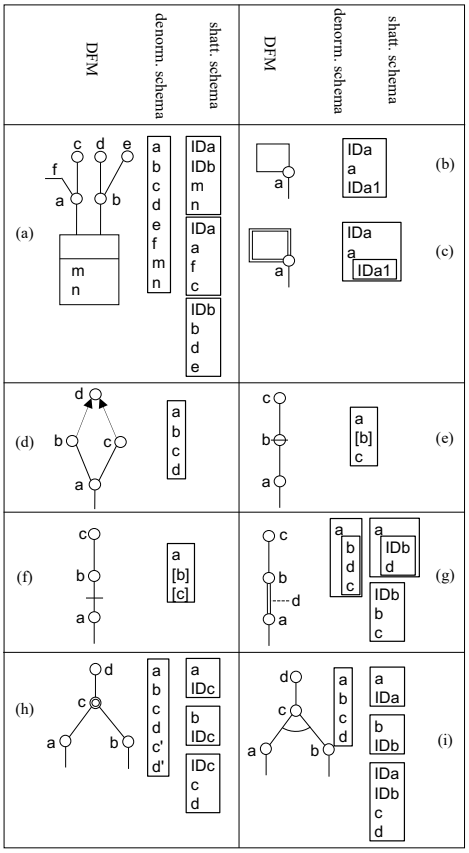

Fig. 4. Document-based design alternatives for each multidimensional element

are shown in Figures 5 and 6, respectively. Noticeably, in PostgreSQL document-based data are actually supported only in terms of column data typed JSON in relational tables; for this reason the schemata show each document as embedded in a relational table. In the denormalized schema all levels are included within a single document, the only exception being Customer where a non-onto hierarchy is rooted. Conversely, in the shattered schema separate documents are created for the fact, for each dimension, for the shared hierarchy in City, for the cross-dimensional level Rating, and for the non-onto hierarchy in Customer (there is no need to create a document for tag since if has no children).

4.3. Graph-based model

Though some papers propose extensions of the multidimensional model and of OLAP to deal with graph data (Chen, Yan, Zhu, Han and Yu, 2009; Beheshti, Benatallah, Nezhad and Allahbakhsh, 2012; Gómez, Kuijpers and Vaisman, 2020), to the best of our knowledge only a couple of approaches cope with the problem of implementing a multidimensional schema against the graph-based model. Sellami et al. (2018) propose two solutions. In the first one, the fact is stored in

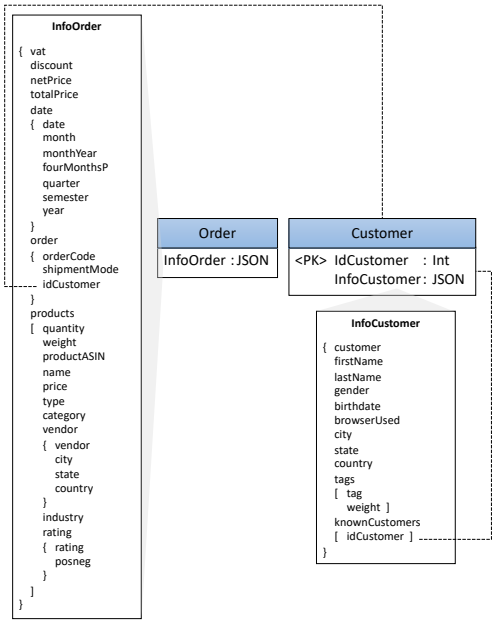

Fig. 5. Denormalized schema for the Order fact (dashed lines represent implicit inter-attribute relationships)

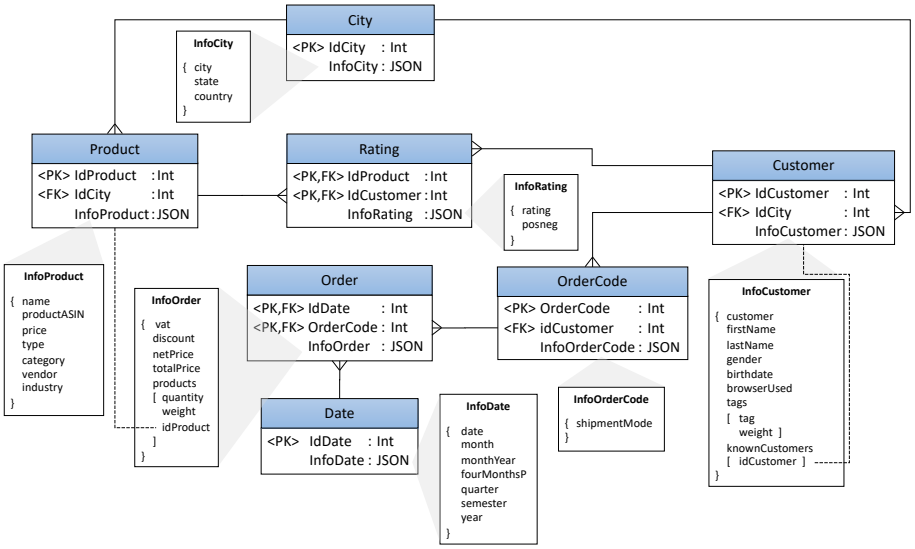

Fig. 6. Shattered schema for the Order fact

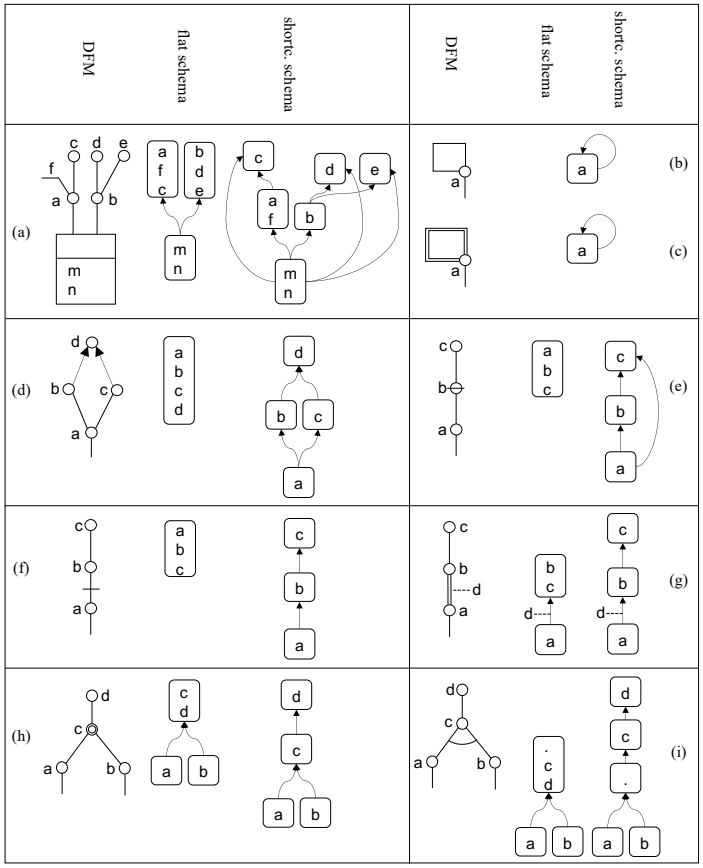

Fig. 7. Graph-based design alternatives for each multidimensional element

a graph node having measures as properties, and each level is stored in a node with its properties; the fact node points to the dimension nodes, which in turn point to the level nodes following the structure of the hierarchies. The second one is similar, except that the fact node points to a single node, which in turn points to each dimension node. A third solution is proposed by Sellami, Nabli and Gargouri (2020), where the fact node points to the dimension nodes, and each dimension node includes *all* the levels and properties of the corresponding hierarchy. Note that these three solutions are not experimentally compared in terms of efficiency.

Here we consider two solutions: *flat schema* (the third solution described above) and *shortcut schema* (like the first solution above, but extended with additional transitive arcs from the fact node to the other nodes to improve querying performance). These solutions are exemplified in Figure 7, together with their variants in presence of different types of multidimensional elements. A brief comment on the solutions:

– Figure 7.a: a fact with two simple hierarchies is modeled either including all

levels in each hierarchy in a single node (flat), or creating separate nodes for each level (shortcut). In both cases, measures are stored within a fact node.
- Figure 7.b-c: a non-onto hierarchy is modeled using a node pointing to itself (flat and shortcut).
- Figure 7.d: a convergence is modeled either including all the levels in the same node (flat) or using separate nodes for each level (shortcut).
- Figure 7.e: a non-covering hierarchy is modeled either using an optional field b (flat) or an additional arc that directly links a to c when b is missing (shortcut).
- Figure 7.f: an optional arc is modeled using either optional fields b and c (flat) or an optional arc from a to b (shortcut).
- Figure 7.g: a non-strict hierarchy is modeled using an arc attribute d to store the attribute (if any) connected to the many-to-many relationship (flat and shortcut).
- Figure 7.h: a shared hierarchy is modeled by having two nodes a and b pointing to the same node c (flat and shortcut).
- Figure 7.i: for cross-dimensional level c, one node must necessarily be created to store the couples of matching members of a and b —similarly to what done with the bridge table in the relational model. This node (represented with a dot in the figure) can either store also c and its children (flat), or point to a separate node storing c (shortcut).

The arc directions do not usually impact querying performances (Bitnine Global Inc., 2017; Marzi, 2020); conventionally, we directed all arcs from the hierarchy root towards its leaves.

The flat and the shortcut schemata obtained for the Order fact are shown in Figures 8 and 9, respectively. The shortcut schema also includes transitive arcs from the fact node to all other nodes, not shown in the figure for simplicity. Noticeably, in both solutions, the cross-dimensional level Rating could have been modeled as a property of an arc directly linking Product to Customer; this solution has not been considered because it is not applicable in the general case (i.e., when the cross-dimensional level has some children).

5. Comparative tests

As stated in the Introduction, the goal of this paper is to devise a set of guidelines for the logical design of MMDWs. To this end, we need to (i) evaluate the solutions outlined above from different points of view, namely, querying, storage, and ETL, and (ii) discuss if and how two or more solutions (possibly corresponding to different models) can be effectively mixed together within a single schema. To this end, for each of these aspects, in the following sections we carry out a set of comparative tests (please refer to Figure 1 for a roadmap).

We wish to emphasize that the idea behind these comparative tests (and the motivation to the whole paper) is not to investigate whether a relational DW is better/worse than a document-based or graph-based DW. Indeed, this kind of comparison has already been made in the literature to some extent, e.g., by Challal et al. (2019), Chevalier, Malki, Kopliku, Teste and Tournier (2015*b*), and Gómez et al. (2020). The motivation is to find guidelines for mixing different models when implementing a DW via a MMDBMS; this is the reason why all tests were made on a single MMDBMS (namely, PostgreSQL) rather than using

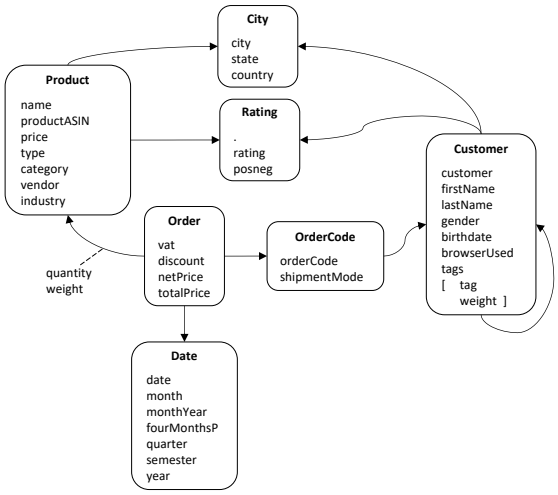

Fig. 8. Flat schema for the Order fact

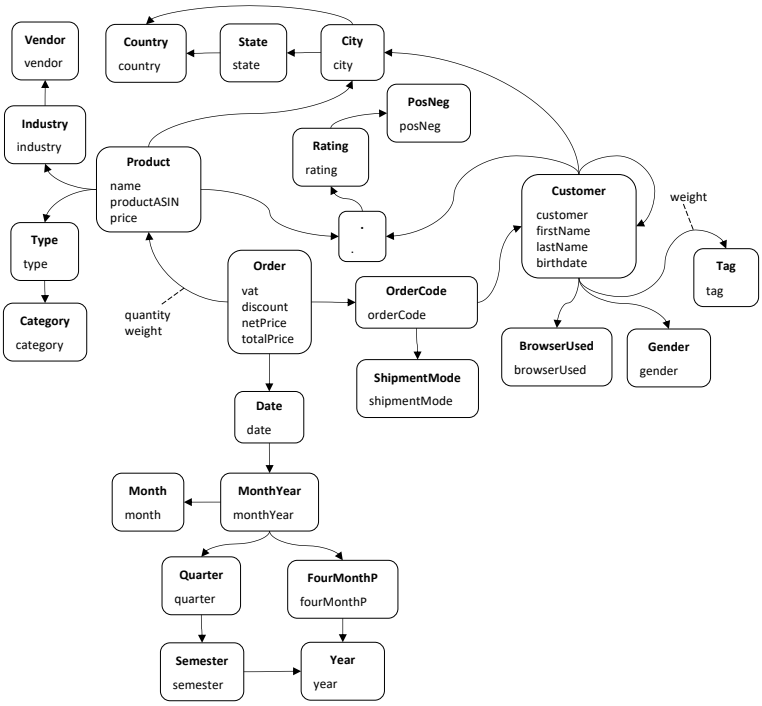

Fig. 9. Shortcut schema for the Order fact

different mono-model DBMSs (such as MongoDB and Neo4j). Clearly, the choice of which model to use for which part of multidimensional data is the result of a trade-off between different features, such as having better query performance, fewer transformations (which suggests maintaining the source data model in the DW as well), etc.

All solutions are implemented in AgensGraph 2.2, an open-source extension of PostgreSQL 10.4 including support to graph storage. Differently from *pure* graph-based DBMSs like Neo4j, the support given to graphs in AgensGraph is not native. In fact, AgensGraph relies on relational structures to store nodes and edges: several tables are created, one for each class; dynamic node properties are supported by modeling each node as a JSON object, and B-tree indexes are automatically computed to support efficient querying; ultimately, Cypher queries are supported and mapped to SQL queries on such structures (Bitnine Global Inc., 2017). We remark that, although other MMDBMSs (namely, Oracle DB and SQL Server (Lu and Holubová, 2019)) support all the data models considered in this work, they all use relational tables to store graphs, so the support they give is not native as well.

For all three models, B+trees have been used to index (i) primary and foreign keys in relational tables, (ii) attributes used in selection predicates, and (iii) identifiers (and attributes referencing them) in JSON/graph-based data. Also, GIN indexes have been used in document-based solutions to index the content of array attributes.

A custom application has been written in Java to generate data for all solutions. Although synthetic, the generation process complies with basic realistic assumptions (e.g., ratings are generated only for products that a customer has actually bought) and satisfies all the functional dependencies among levels. Specifically, we have 1M orders made by 100K customers on 10K products over 365 dates. The number of cities is 5K; the products bought in each order (expressed as average $\pm$ standard deviation) are 5.5 ± 2.9 (minimum 1, maximum 10), the number of customers known by each customer is 7.3 ± 2.7 (minimum 1, maximum 21), and the number of tags per customer is 2 ± 1 (minimum 1, maximum 5). Finally, we assume that ratings are given for each product ordered by each customer.

All tests have been run on a Core i7 with 8 CPUs @3.6GHz server with 32 GB RAM running Ubuntu. PostgreSQL memory parameters have been set as follows:

- *shared_buffers* (i.e., the number of shared memory buffers used by the server) is set to its default, 128MB; this avoids the entire database being stored in memory;
- *effective_cache_size* (i.e., the estimate that the query planner makes of how much memory is available for disk caching by the operating system and within the database itself) is set to 4GB;
- *work_mem* (i.e., the amount of memory actually used by PostgreSQL for each user query) is set to 80MB; this setting enables 100 concurrent connections.

The three solutions and the workload queries are all publicly available at `https://github.com/big-unibo/m3d-guidelines`.

Table 1. Workload for performance test (**Hops** is the number of hops on the non-onto hierarchy; **Card.** is the number of tuples in the query result)

MD element	Query	Group-by set	Selection	Hops	Card.
Plain	q1	Month	–	–	12
	q2	Month	Quarter	–	3
	q3	ShipmentMode	–	–	4
	q4	Gender	–	–	2
	q5	Gender	BrowserUsed	–	2
	q6	Gender	City	–	1
	q7	Date	–	–	365
	q8	Date	Quarter	–	90
	q9	Date	Month	–	30
	q10	Date	Date	–	3
Non-onto	q11	Customer	–	1	10^5
	q12	Gender	–	1	2
	q13	ShipmentMode	BrowserUsed	1	4
	q14	ShipmentMode	BrowserUsed	2	4
	q15	ShipmentMode	BrowserUsed	3	4
	q16	ShipmentMode	Customer	1	4
	q17	ShipmentMode	Customer	2	4
	q18	ShipmentMode	Customer	3	4
	q19	ShipmentMode	Customer, BrowserUsed	1	4
	q20	ShipmentMode	Customer, BrowserUsed	2	4
	q21	ShipmentMode	Customer, BrowserUsed	3	4
	q22	Tag	–	1	5
	q23	City	–	1	5000
	q24	Customer	Customer	var.	4
Convergence	q25	Year	–	–	1
Non-covering	q26	State	–	–	10^2
Opt. arc	q27	BrowserUsed	–	–	6
Non-strict	q28	ProductASIN	–	–	10^4
	q29	ProductASIN	Industry	–	62
	q30	ProductASIN	City	–	16
	q31	Vendor	–	–	500
	q32	Gender	Industry	–	2
	q33	Gender	City	–	2
	q34	Tag	–	–	4
	q35	City	–	–	469
Shared hier.	q36	City	–	–	5000
	q37	Country	–	–	28
Cross-dim- lev.	q38	Rating	–	–	5

5.1. Querying

In this section we compare the different solutions outlined above from the point of view of querying performance and formulation complexity; intra-model comparisons are done first, followed by an inter-model comparison.

The workload we use for these comparisons includes 38 queries, whose features are are summarized in Table 1, classified based on the main multidimensional element they involve: (i) plain (from q1 to q10, only regular hierarchies involved with different group-by sets and selection predicates), (ii) non-onto hierarchy (from q11 to q24, involving level Customer with different group-by sets, selection predicates, and number of hops)[3], (iii) convergence (q25, involving level Year),

[3] The Order example only shows one non-onto hierarchy based on a many-to-many association between customers. A non-onto hierarchy based on a many-to-one association (like the one

(iv) non-covering hierarchy (q26, involving level State), (v) optional arc (q27, involving level BrowserUsed), (vi) non-strict hierarchy (from q28 to q35, involving levels ProductASIN and Tag with different group-by sets and selection predicates), (vii) shared hierarchy (q36 and q37, involving level City with different group-by sets), and (viii) cross-dimensional level (q38, involving level Rating). As done by Zhang and Lu (2021), the queries were designed to cover the main challenges of multi-model query processing, such as graph traversal and shortest path-finding, string matching, joins, and aggregation, all from an OLAP point of view. For some types of multidimensional element we defined more queries than for other types, because the number of possibly interesting combinations is larger. Specifically, for plain hierarchies we tried different combinations of levels in the group-by set and in the selection predicate, to obtain different result cardinalities and encourage the adoption of different execution plans in the DBMS. For non-onto hierarchies, we also required different numbers of hops (from 1 to 3) to be executed; in particular, query q24 shows a variable number of hops because it computes the shortest connection between two customers along the knows non-onto hierarchy. Even for non-strict hierarchies we tried different combinations, to involve either Tag or ProductASIN either in the group-by set or in the selection predicate.

Queries are formulated in the extended query language provided by PostgreSQL and Agensgraph to query JSON and graph data, respectively. In particular, as mentioned at the beginning of this section, AgensGraph supports Cypher to formulate queries over graph data and then rewrites them to SQL operations over the relational structures that implement the graph. The Cypher query language is integrated with SQL and hybrid queries (e.g., over relational, JSON, and graph data) are allowed. A sample hybrid query (corresponding to an implementation of q22) is shown in Listing 1.

Listing 1: A sample hybrid query over relational, JSON, and graph data in AgensGraph

```
select tag, round(sum((ft.info->>'totalprice')::numeric),5),
       round(avg((ft.info->>'discount')::numeric),5)
from doc1t4_ft ft, doc1_dt_order o, rel_bt_customer_tag ct,
     ( match (c:dt_customer)-[:knows]->(c1:dt_customer)
       return c.id, c1.id as parentid ) c
where ft.idorder = o.idorder
      and o.idcustomer = c.id::text::int
      and c.parentid::text::int = ct.idcustomer
group by tag;
```

Execution times are obtained by running a PostgreSQL procedure that runs each query of the workload in random order; the reported execution times are the average of five workload runs[4].

To evaluate the query formulation complexity in terms of the cognitive load on the user during query authoring, we compute for each schema the main indicator proposed by Jain, Moritz, Halperin, Howe and Lazowska (2016), i.e., the character length of each query as a string, a proxy for the effort it takes to craft the query.

defining a company organization chart, case (b) in Figures 4 and 7) can be seen as a particular case of this.

[4] PostgreSQL does not provide any functionality to clear the cache; as a remedy, we run the queries multiple times in random order.

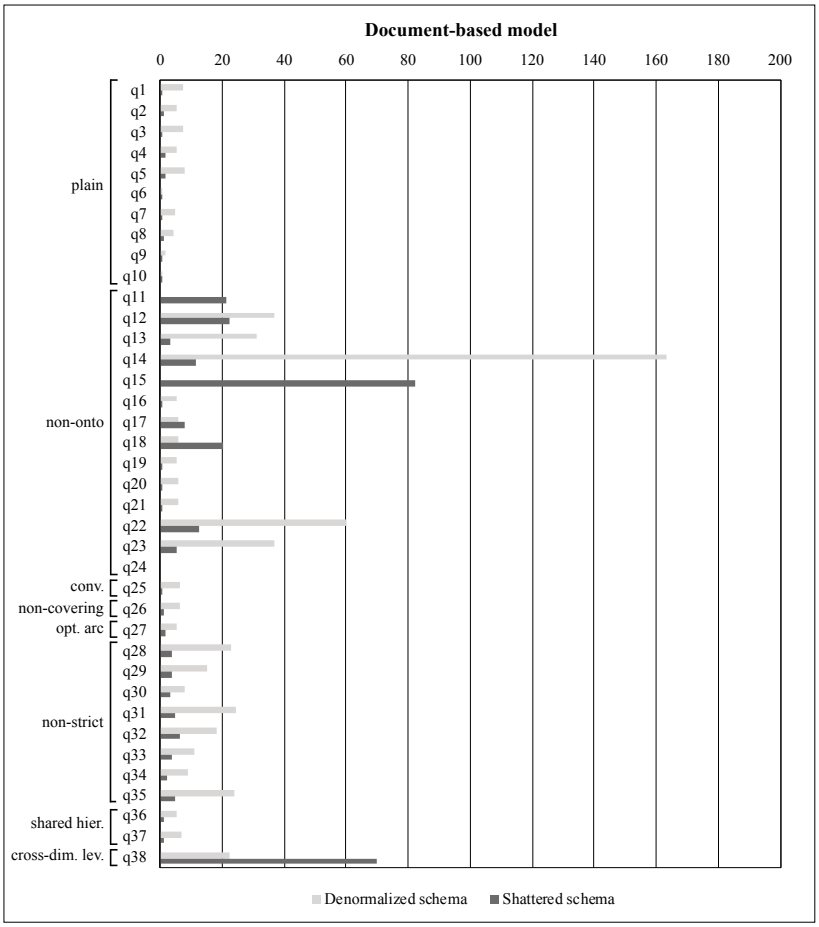

Fig. 10. Query performance (in seconds) for the document-based model

5.1.1. Document-based model

The results for the performance tests made on document-based solutions are shown in Figure 10. The shattered schema appears to perform better than the denormalized one in almost all situations. The main reason is that the fact table is significantly larger in the denormalized schema (1 KB per row on average, against the 670 B in the shattered schema), thus requiring longer execution times to operate on the data; for instance, a simple sequential scan of the fact table takes less than 100 ms in the shattered schema and almost 2 seconds in the denormalized one. A careful evaluation of the execution plans showed that JSON objects are carried out along the execution plans until the end (i.e., fields are not efficiently projected out of JSON objects). This substantially increases the footprint of queries, putting much pressure not only on the main memory, but also on the disk (that is used when an external sort on a large amount of data is carried out). More in detail:

(i) With both schemata, the query execution times fall below 10 seconds for all queries on plain hierarchies.

(ii) For queries on the non-onto hierarchy, times are obviously longer, significantly depending on the number of hops in the query and on the cardinality of the grouping level. The shattered schema is clearly better, also considering that for two queries (q11 and q15) the execution on the denormalized schema failed due to a timeout error. However, the shortest-path query q24 failed on both schemata (out-of-memory).

(iii) For all other multidimensional constructs, the performance of the shattered schema is fully compatible with the one required by an interactive analysis session, and significantly better than the denormalized schema.

(iv) The only exception is for query q38, which groups tuples by cross-dimensional level Rating. Here, the execution time on the shattered schema is almost 70 seconds, more than double the one on the denormalized schema. As querying the cross-dimensional level in the shattered schema creates a join path with a loop, the query optimizer struggles to produce an efficient query plan. In particular, the optimizer estimates that the number of records obtained by closing the loop is an order of magnitude lower than the actual one, and it relies on a Nested Loops join strategy (which becomes inefficient in presence of large numbers of records)[5].

In general, in presence of a cross-dimensional level c, the denormalized and shattered schemata can be mixed by moving c within the fact collection. In the Order fact, if the shattered schema in Figure 6 is modified by removing table Rating and including levels rating and posneg within the products array of document InfoOrder, the execution time of q38 drops down to 2.6 seconds. Note that this mixed solution cannot be taken when cross-dimensional level c is preceded by a non-strict hierarchy. In this case, the granularity of c is finer than the one of the fact table, thus, the denormalized solution for c is not feasible. However, in the specific situation in which the non-strict hierarchy is directly attached to the fact (as in the Order fact), the shattered design of the fact table allows to include data at a finer level of granularity.

As to query formulation complexity, Figure 11 shows that (except for a slightly lower complexity for queries q26-q38 for the denormalized schema) there is not relevant difference between the denormalized and the shattered schema from this point of view. In fact, the average complexity turns out to be about 640 and 660 characters for the denormalized and shattered schemata, respectively, which means a 3% relative difference.

Following these results, we can conclude that, from the querying point of view, the best practice for document-based modeling of multidimensional data is to use a shattered schema for all constructs except for cross-dimensional levels, for which a local denormalized schema should be adopted.

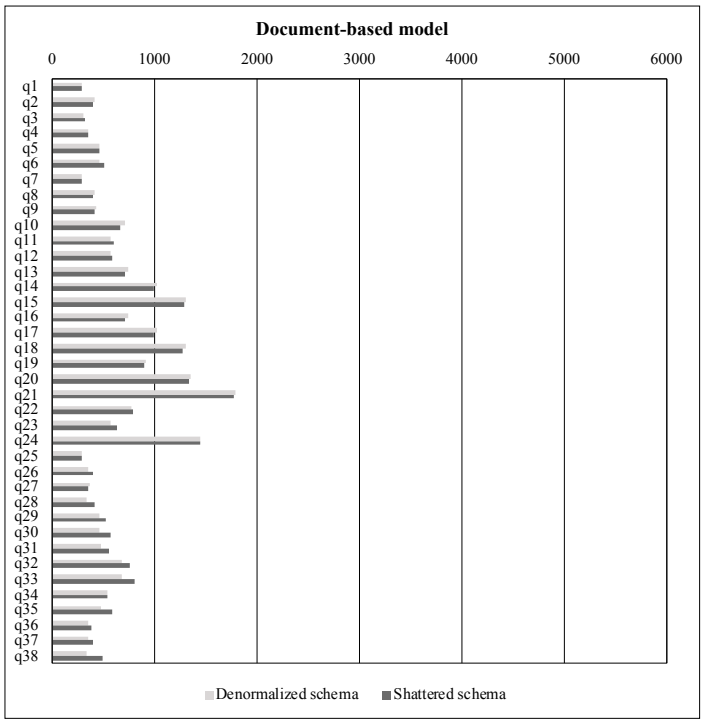

Fig. 11. Query formulation complexity (in characters) for the document-based model

5.1.2. Graph-based model

The results for the performance tests made on graph-based solutions are shown in Figure 12. The shortcut schema appears to perform better than the flat one in all situations. This is mainly because queries on the shortcut schema are quite simpler, and fewer edges must be usually navigated to join the fact nodes with the other nodes of interest.

Specifically:

(i) With both schemata, the query execution times fall below 3 seconds for all queries on plain hierarchies.

(ii) Like in the document-based case, recursive queries on the non-onto hierarchy are more challenging. The times here are mostly below 20 seconds, the only exception being q15 (group-by ShipmentMode, selection on BrowserUsed, 3 hops) which takes 114 seconds with the shortcut schema due to the high number of hops and low selectivity; q18 and q21 also require three hops, but their selectivity is much higher. Noticeably, the shortest-

[5] PostgreSQL gives the possibility to turn off Nested Loops joins by issuing the command `set enable_nestloop = off`. However, as stated in the documentation, "It is impossible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available"; indeed, turning it off does not change the execution plan (`https://www.postgresql.org/docs/current/runtime-config-query.html`).

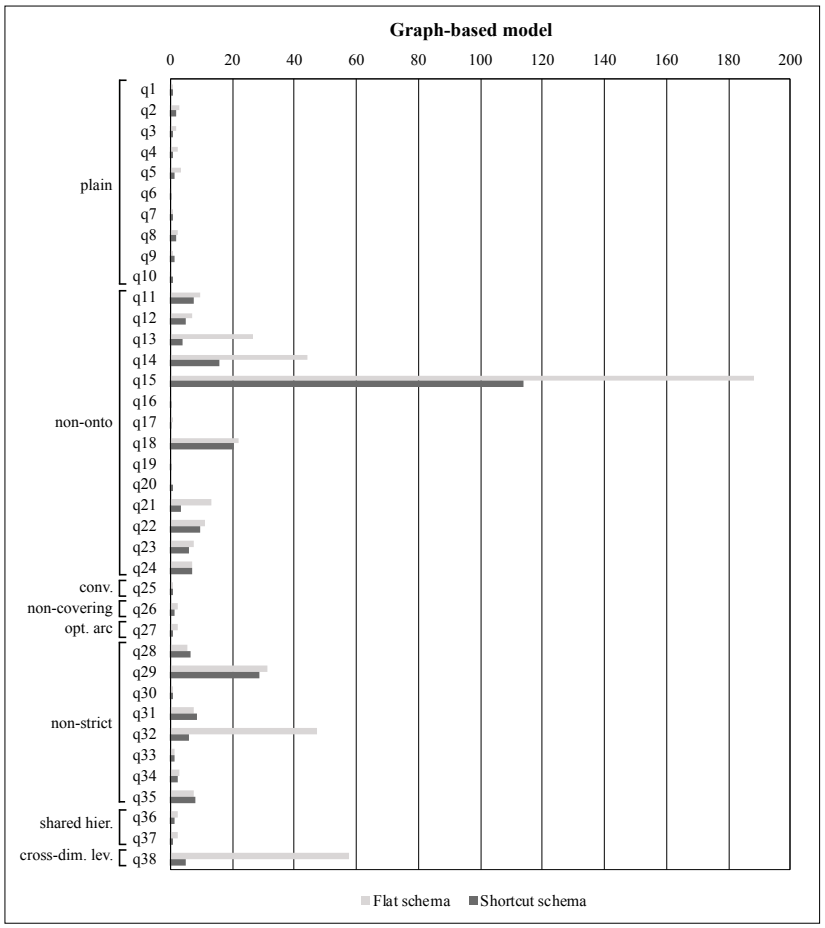

Fig. 12. Query performance (in seconds) for the graph-based model

path query q24 (which failed on document-based solutions) performs very well (about 7 seconds) on both schemata.

(iii) The high gap between the flat and shortcut schema in q13 to q15 is due to the very large number of records that must be aggregated by the values of ShipmentMode. As ShipmentMode is modeled as a single node in the shortcut schema, the query planner is able to aggregate the data efficiently by using a Hashed Aggregate strategy; conversely, in the flat schema, the query planner must first sort records by ShipmentMode and (as already mentioned for the document-based model) this operation is particularly expensive, as the disk is also involved.

(iv) The gap in q21 is due to mistakenly low estimates in the number of records by the optimizer, leading to favoring (unexpectedly slower) Nested Loops joins over (actually faster) Hash Joins.

(v) Queries on non-strict hierarchies are demanding as well, with q29 (group-by ProductASIN, selection on Industry) taking about 30 seconds with both

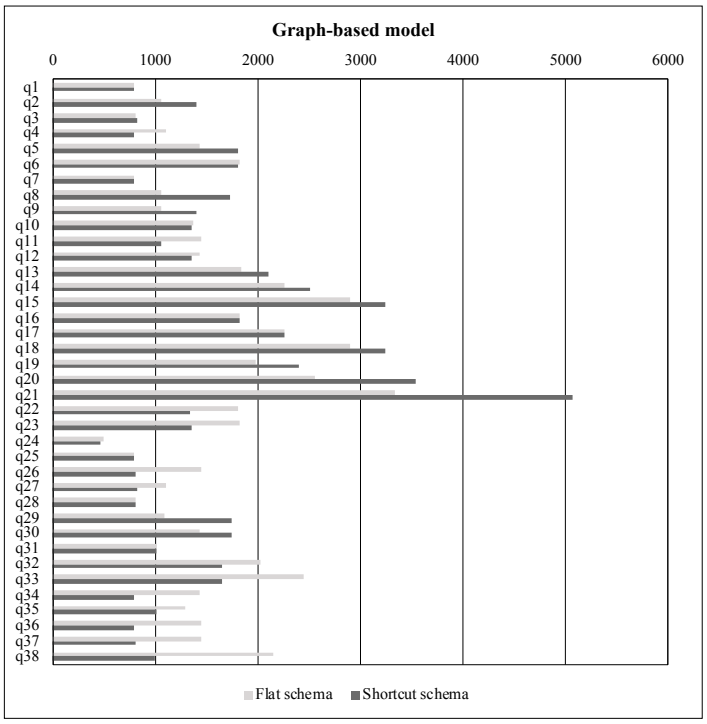

Fig. 13. Query formulation complexity (in characters) for the graph-based model

schemata. In particular, in queries with no selection predicates (i.e., q28, q31, q34, and q35), a large number of records is collected and most of the time is spent in the final aggregation; although some variability occurs, execution times are almost equivalent in these cases. Conversely, when selection predicates are present (i.e., in q29, q30, q32, and q33), the optimizer often favors Nested Loops joins on the flat schema; as already mentioned, when estimates are mistakenly low, the execution times are heavily affected. This behavior is not evident in q30 and q33 due to the very high selectivity of the selection predicates.

(vi) Query q38 on cross-dimensional level Rating performs well on the shortcut schema due to the direct edge from fact nodes to rating ones; conversely, it is slow on the flat schema as several edges must be navigated.

As to query formulation complexity, Figure 13 shows that there is no clear winner between the two schemata; the average complexity is 1577 for the flat schema and 1572 for the shortcut schema, so even for the graph-based model there is no relevant difference between the two schemata from this point of view.

Following these results we can conclude that, from the querying point of view, the best practice for graph-based modeling of multidimensional data is the shortcut schema.

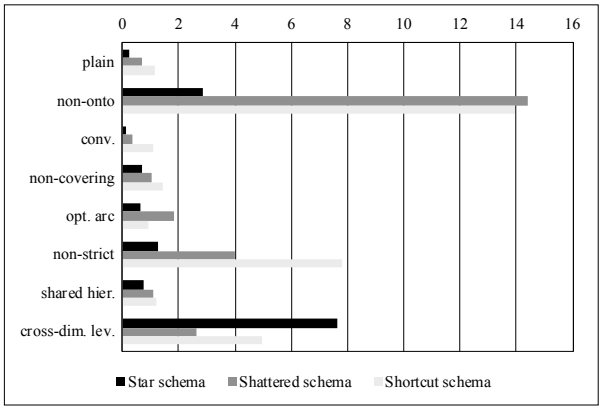

Fig. 14. Query performance (in seconds) for the relational, document-based, and graph-based models

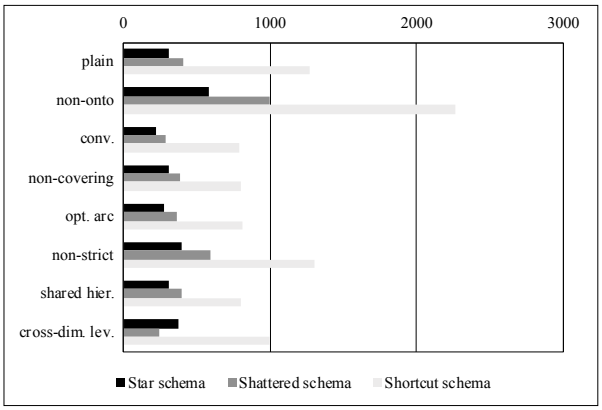

Fig. 15. Query formulation complexity (in characters) for the relational, document-based, and graph-based models

5.1.3. Inter-model comparison

Figures 14 and 15 compare the querying execution times and formulation complexity, averaged by query class, for the relational model (star schema), the document-based model (shattered schema + denormalized schema for Rating), and the graph-based model (shortcut schema). Clearly, the average is computed, for each model, only on the queries that did not fail (which excludes q24 for the relational and document-based models). The standard deviation (not shown in the chart for simplicity) is very similar across the different models, and is about ±61% for queries on plain and non-strict hierarchies; ±185% for queries on non-onto hierarchies (this is because the related queries are quite different from each other, in terms of both the number of hops on the recursive arc and the number of aggregated tuples); ±7% for queries on shared hierarchies; 0 for queries on other hierarchy types (as there is only 1 query per type). The figure can be commented as follows:

– Expectedly, the figure shows that the relational model outperforms both the

document-based and the graph-based ones for almost all types of queries. For queries on a cross-dimensional level, the shortcut schema outperforms the relational one thanks to the direct links from the fact nodes to the Rating nodes, through which rating data can be obtained with fewer join operations (2 instead of 3). The top performance of the document-based model for these queries is actually unexpected, as the execution times for q38 are higher for both the shattered and denormalized schemata (as shown in Figure 10). The improvement in performance is achieved by combining the benefit of denormalizing the rating (i.e., saving join operations) with the benefit of using a shattered schema (i.e., having lighter JSON objects in the fact table).

- For queries on plain hierarchies, by far the most common, the performance of the three models is substantially the same (around 1 second in the average).
- Queries on non-onto hierarchies are apparently the most challenging ones for the document-based and graph-based models, from both points of view of performance and formulation complexity. However, we remark that the average values shown on the chart do not consider query q24, which failed on both the relational and the document-based models.
- In general, the shattered schema suffers the need to deal with JSON data, which require more storage and are more challenging to be analyzed efficiently (as discussed in the single-model comparison). Although aggregate data modeling allows to save explicit join operations (instead, the `jsonb_array_elements` function is used to "enter" arrays), PostgreSQL implements the unnesting of arrays as a Nested Loop between the Sequential Scan of the table and the Function Scan of the contents of the array. Eventually, in our workload, the shattered schema performs a total of 113 joins (32 of which are due to the unnesting of arrays) against the 107 of the star schema. All these factors contribute to making the shattered schema generally less performing than the star schema.
- The shortcut schema suffers from issues similar to the shattered one in terms of JSON data. Additionally, we remark that the Agensgraph's implementation of graph data relies on the relational data model, i.e., all nodes of a certain class are stored in a different table, and a table is created for each type of edge (independently of the edge representing a one-to-one, one-to-many, or many-to-many relationship). This means that, against our expectations, the shortcut schema requires a larger number of joins than the star schema (184 against 107), because a join to an "edge table" is required even in presence of a one-to-* relationship (which does not require and extra table in the relational model, as it is simply implemented through a foreign key).
- The formulation complexity on the star schema is not very different from the one on the shattered schema, being significantly smaller only for queries on non-onto hierarchies.
- The formulation complexity on the shortcut schema is, for all query types, about double the one on the shattered schema.

5.2. Storage

In this section we compare the different solutions from the point of view of the storage space they require. Table 2 shows the storage size for the different schemata. As already noted by Bimonte et al. (2022), storing data in relational

Table 2. Storage size for the different schemata; the total size includes not only data but also indexes

Model	Schema	Data size	Total size
Relational	Star	660 MB	1557 MB
Document-based	Denormalized	2105 MB	2494 MB
	Shattered	1251 MB	1765 MB
Graph-based	Flat	2109 MB	3476 MB
	Shortcut	6629 MB	12321 MB

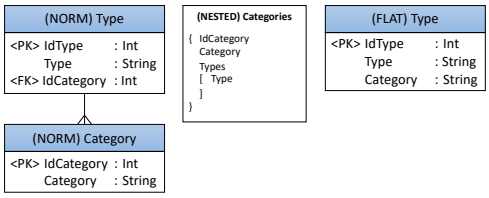

Fig. 16. Modeling the many-to-one relationship between Category and Type in a normalized (left), nested (center), and flat (right) schema

form is the cheapest solution. For document-based data, the shattered schema uses about half the space of the denormalized schema. For graph-based data, the shortcut schema uses three times the space of the flat schema, due to the presence of many more arcs (not only those between hierarchy levels, but also the transitive ones from the fact nodes to the levels).

5.3. ETL

In this section we firstly compare the different solutions outlined above from the point of view of the cost for designing ETL procedures. This is indeed quite relevant, since the design and maintenance of ETL procedures are recognized to make up for up to 60% of the resources spent in a DW project (Papastefanatos, Vassiliadis, Simitsis and Vassiliou, 2012). Then, we extend the comparison to the performance of ETL procedures.

To characterize the complexity of ETL formulation, we use the same indicator employed in Section 5.1 to assess the query formulation complexity. We assume that ETL procedures are written in terms of SQL statements to enable a better characterization of complexity and be independent of the specific features of ETL tools. Besides, we consider static ETL (which is performed when a DW is loaded for the first time), not incremental ETL (periodically performed to extract, transform, and load the data inserted/updated in the sources since the last run of the ETL).

An ETL query reads a piece of data from the data source and transforms it so that it can be loaded onto the MMDW. Hence, its precise formulation depends on the model and schema of the data source, on the multidimensional schema, and on the solution adopted for logical design; thus, a huge number of combinations arise. Precisely counting the number of characters of each possible ETL query —as suggested by Jain et al. (2016)— would require to write all of them. To

Table 3. Estimation of ETL formulation complexity

		Multidimensional element				
Source	**Target**	**Plain/Conv./Opt.**	**Non-onto**	**Non-strict**	**Shared**	**Cross-dim.**
Norm.	Star	60	56	79	79	79
	Denorm.	60	85	124	122	101
	Shatt.	60	85	114	79	114
	Flat	60	56	79	125	125
	Short.	198	56	178	263	224
Nested	Star	67	103	180	189	228
	Denorm.	67	27	27	182	76
	Shatt.	67	27	104	189	228
	Flat	67	103	180	301	235
	Short.	265	103	283	393	364
Flat	Star	27	66	107	204	107
	Denorm.	27	58	44	130	33
	Shatt.	27	58	81	204	107
	Flat	27	66	107	250	165
	Short.	147	66	168	386	226

avoid this, we introduce some rough approximation. Specifically, we proceed as follows:

(i) We classify data sources, based on their type of schema, into *normalized* (e.g., a relational database), *nested* (e.g., a JSON collection), and *flat* (e.g., a wide-column database); Figure 16 shows how the many-to-one relationship between Category and Type would be modeled according to the three schema types, respectively. The reason for not explicitly considering the model of the data source is that the formulation complexity of an ETL query is mainly driven by the need to nest/un-nest and normalize/denormalize the data.

(ii) For each type of multidimensional element, each type of source schema, and each target logical design solution, we write a sample ETL query. Plain hierarchies are considered together with convergences, optional arcs, and non-covering hierarchies, since the corresponding query does not change significantly.

(iii) We count the number of characters in each sample ETL query. We claim that the complexity is actually driven by the *structure* of the query, so we do not count the attribute names.

The results are shown in Table 3.

To give an empirical validation of these estimates, we applied it to the case study discussed by Bimonte et al. (2022), where the exact number of characters of specific ETL queries is counted with reference to a data source including JSON, XML, key-value, and graph data, and to two target logical solutions: a classical (full relational, FR) star schema and a multi-model star schema (MM) where relational, JSON, and graph data are mixed. The actual total formulation complexity for ETL (removing attribute names) turns out to be 412 for FR and 276 for MM. The estimates obtained using Table 3 (considering cost zero for the pieces of data whose model is preserved so that no transformation is required) yield 597 for FR and 315 for MM. Both estimates are satisfying for our purposes; indeed, they are higher than the real values because they do not consider the possibility of merging some queries together.

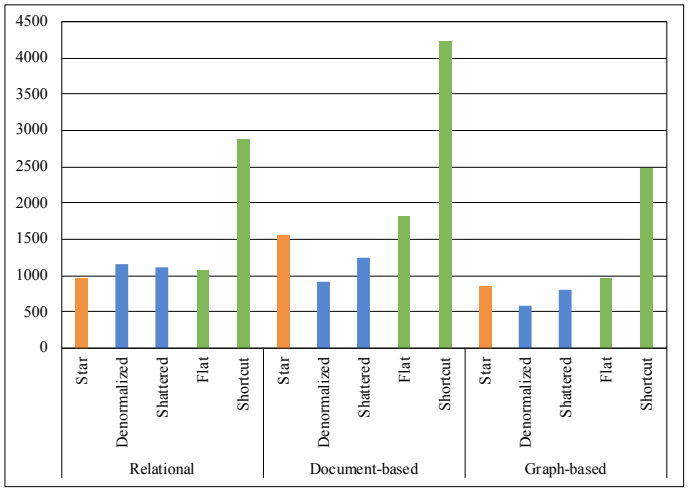

Fig. 17. ETL formulation complexity (in characters) for the different source data models and target logical solutions

To discuss the complexity of ETL formulation for our Order fact we use Table 3 with reference to three different situations, in which the source data are stored in (i) a (normalized) relational database, (ii) a (nested) document-based database, and (iii) a (flat) graph-based database[6]. The results are shown in Figure 17 for each source model and target solution. The figure can be commented as follows:

- For each source model, the ETL formulation complexity for the shortcut schema is more than double the one of the other schemata. Indeed, an extra effort is required to create different nodes for each level and to create transitive arcs from the fact nodes.
- The star and flat schemata are similar from the point of view of denormalization, thus their costs are mostly the same —except for shared hierarchies and cross-dimension levels, where more arcs need be created.
- When the source data are normalized and relational, using star as the target schema is the best choice; for the denormalized, shattered, and flat schemata the costs are roughly the same. The larger effort here is in creating the JSON objects by joining and aggregating the data.
- When the source is a nested document-based collection, using a document-based solution for logical design as well is the best option as data is already aggregated (indeed, most queries are very simple). Specifically, a denormalized schema is the cheapest solution, followed by a shattered (where some degree of normalization is required), a star, and a flat schema.
- The same ordering holds when the source is a flat graph-based database. In this case, adopting the graph-based model for the target as well is not the best choice, since using nodes and edges to model complex multidimensional elements can be seen as some sort of normalization —like the one carried out

[6] Though a flat schema type is not the most frequent choice in a graph-based database, it is technically possible.

Table 4. ETL performance (in seconds)

Source	Target	Multidimensional element	
		Plain/Conv./Opt.	**Non-strict**
Norm.	Star	1.6	2.2
	Denorm.	5.3	16.6
	Shatt.	3.8	4.4
	Flat	17.7	24.6
	Short.	28.3	71.0
Nested	Star	2.3	8.1
	Denorm.	4.5	8.6
	Shatt.	3.3	10.7
	Flat	19.6	32.8
	Short.	28.6	74.0
Flat	Star	1.7	2.1
	Denorm.	4.8	13.3
	Shatt.	2.8	4.2
	Flat	21.0	26.7
	Short.	32.8	71.0

when the target is a star schema. Document-based solutions are cheaper as flat data are significantly denormalized, thus building a document-based solution is just a matter of creating the JSON objects, while the star and flat schemata require some more aggregation and normalization steps.

To compare the solutions in terms of ETL performance we use the same approximation used for formulation complexity: (i) we classify data sources into normalized, nested, and flat; and (ii) we write a sample ETL query for each type of multidimensional element, each type of source schema, and each target logical design solution. To reduce the resulting number of queries, we restrict to consider two types of multidimensional elements: plain hierarchies (because they are the most frequent ones) and non-strict hierarchies (because Table 3 shows that their formulation complexity changes significantly from one solution to another). For all sample queries, the reference is a simple fact with a single (either plain or non-strict) hierarchy; the fact cardinality is 1M. The resulting execution times are shown in Table 4. Figure 18 plots the relationship between ETL performance and storage space taken by the target logical solutions; clearly, the non-strict hierarchy is the one requiring most storage space in every logical solution. It appears that, as we could expect, the ETL time is roughly proportional to the target storage for each source model. The most notable exception holds for the denormalized solution, especially in the case of a non-strict hierarchy; indeed, the ETL cost is lower than expected in the denormalized solution since it is the only one with a procedure composed by a single query (as only one table must be created), requiring very simple computations (especially in the case of a nested or flat source, where a single source table exists; this is also reflected in Table 3, where the complexity of such queries is very low).

6. Multi-model multidimensional design

In this section, we show how the mono-model solutions considered in the previous sections can be mixed to create a Multi-Model MultiDimensional (in short, M^3D)

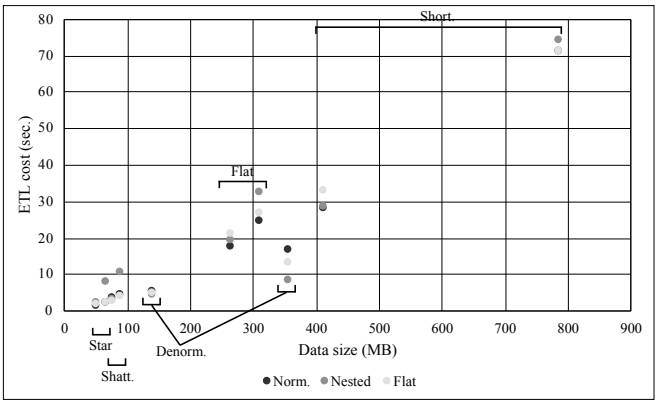

Fig. 18. Relationship between ETL performance and storage for the different source data models and target logical solutions

logical schema, using again as a working example the Order fact introduced previously. The only multi-model solution considered by Bimonte et al. (2022) was a classical star schema with fact and dimension tables, extended with semi-structured data in JSON, XML, key-value, and graph-based form. While this has some advantages (e.g., that performance optimization of star schema has been long studied and practiced), it may not exploit the flexibility enabled by an MMDBMS. Hence, in this work we freely explore all the design alternatives, without requiring that a star schema lies at the core.

Bimonte et al. (2022) argue that three more features are relevant —besides query performance, query formulation, storage, and ETL— when comparing different target models in MMDW design, namely, flexibility, extensibility, and evolvability. To make our guidelines more comprehensive, in this section we briefly recall the main findings described by Bimonte et al. (2022).

As to *flexibility*, we observe that an M^{3}D schema can preserve the data variety existing in the data sources to a greater extent than a mono-model schema. Besides, mixing different models in an MMDW enables the achievement of higher flexibility in the modeling solutions taken so as to adapt the target schema to the workload.

Extensibility is mainly related to variable source schemata. In case of schemaless sources, some levels not considered at design time may be occasionally present in some source documents. Clearly, these levels can be queried in an M^{3}D schema, while they cannot in classical (fully relational) star schema. This allows adopting sophisticated querying approaches capable of coping with variable schemata and structural forms within a collection of documents, such as approximate OLAP (Gallinucci et al., 2019).

Finally, as to *evolvability* we observe that, in a multi-model context, evolution issues are crucial because extensions and changes of the data schema in one model can cause changes in other models, too (Holubová, Klettke and Störl, 2019). M^{3}D schemata are partially schemaless, so they transparently support evolution to some extent by reducing the impact on tables and ETL. This is confirmed by Bimonte et al. (2022) by showing that the evolution effort for a classical star schema is about 10% higher that the one for an M^{3}D schema. In particular,

Table 5. Guidelines by design goals and multidimensional element type (R=relational, D=document-based, G=graph-based); the $\sim$ and $\succ$ symbols denote, respectively, that two models are equivalent or that the first one is better than the second one)

MD element	Query perf.	Query form.	Storage/ ETL perf.	ETL form.	Flex./Ext./ Evolv.
Plain/Conv./Opt.	R$\sim$D$\sim$G	R$\sim$D$\succ$G			
Non-onto	G$\succ$R$\succ$D	R$\succ$D$\succ$G			
Non-strict	R$\succ$D$\succ$G	R$\sim$D$\succ$G	R$\succ$D$\succ$G	R$\sim$D$\sim$G	D$\sim$G$\succ$R
Shared	R$\sim$D$\sim$G	R$\sim$D$\succ$G			
Cross-dim.	D$\succ$G$\succ$R	R$\sim$D$\succ$G			

the higher effort for evolving workload queries in an M^{3}D schema is largely compensated by the lower complexity of the schema and by the absence of ETL.

6.1. Design guidelines

In Table 5 we summarize the results of the comparative tests described in Sections 5.1–5.3, in the form of guidelines by main design goal (querying, storage, ETL, flexibility, extensibility, and evolvability) and multidimensional element type. We treat plain hierarchies, convergences, non-covering hierarchies, and optional arcs together since they essentially exhibit the same behaviour from all points of view.

Clearly, to complete the guidelines, we need to discuss when and how different models can be mixed together within the same schema (some examples will be given with reference to Figures 19 and 20):

– Different hierarchies can use different models.
– In general, mixing different models within the same hierarchy can be done as follows:
  · the connection from a fact/dimension/bridge table T to a document D is established by creating a new table N, adding to T a foreign key referencing N, and adding an attribute of type JSON with value D to N (e.g., see the connection from FT_Order to OrderCode in Figure 19);
  · the connection from a fact/dimension/bridge table T to a graph node G is established by adding to T an attribute whose values match with the identifier of G (e.g., see the connection from BT_Rating to Customer in Figure 19);
  · the connection from a document D to a dimension table T is established by adding to D a field whose values match with the primary key of T (e.g., see the connection from InfoOrder to DT_Product in Figure 20);
  · the connection from a graph node G to a dimension table T is established by adding to G a field whose values match with the primary key of T (e.g., see the connection from Customer to DT_City in Figure 20);
  · the connection from a graph node G to a document D embedded in table N is established by adding to G a field whose values match with the primary key of N;
  · the connection from a document D to a graph node G is established by adding to D a field whose values match with the identifier of G (e.g., see the connection from OrderCode to Customer in Figure 19).

– A denormalized schema can be used for a cross-dimensional level only if the fact is modeled as a document (e.g., as done for Rating in Figure 5).
– A shortcut schema can be locally adopted only if the fact is modeled as a graph node (since only in that case can the transitive arcs be added).

6.2. Case study

In this section we show how the guidelines in Table 5 can be concretely applied, using again the Order fact as a case study. We assume that (i) the source data are stored within a nested JSON collection, except for geographical data which are stored in a relational table; (ii) the main variety issues lie in the product and customer dimensions, as well as in the measures (consistently with the cloud symbols shown in Figure 2); and (iii) the expected workload is the one shown in Table 1. Our (often conflicting) design goal can be summarized as follows:

♯1 Yield good querying performance and low query formulation complexity.
♯2 Reduce storage space.
♯3 Yield low ETL formulation complexity and costs.
♯4 Encourage flexibility in presence of variety and evolvability.

To propose an M^{3}D schema in this setting we proceed as follows:

– Since geographical data are natively stored in the relational model, for the City shared hierarchy we adopt a relational solution —which reduces the storage space (goal ♯2) and the ETL complexity/cost (goal ♯3), while ensuring good query performance and formulation complexity (goal ♯1).
– A relational solution is chosen for the temporal (plain) hierarchy as well. Indeed, the source data only include a simple date attribute (the other levels must be derived during ETL), and the relational solution is the best one in terms of storage (goal ♯2, no need to support variety in this case since the temporal hierarchy has a fixed structure).
– To encourage flexibility in presence of variety and better evolvability (goal ♯4) while ensuring good query performance and formulation (goal ♯1), for the product and order hierarchies (for the latter, limitedly to levels OrderCode and ShipmentMode) we adopt a shattered schema (since source data are in JSON form).
– For the Customer hierarchy (which includes the knows non-onto hierarchy) we adopt the graph-based model, specifically, a flat schema. There are three reasons for this: (i) a schemaless model is required to cope with customer variety (goal ♯4); (ii) graph-based solutions are the only ones for which none of the queries fail (goal ♯1), and (iii) a shortcut schema is feasible only when the fact is modeled as a graph node, which is not the case here.
– Dealing with measure variety requires to store the fact according to the document-based model, which also ensures better evolvability (goal ♯4).
– To model the non-strict hierarchy on Tag we adopt the model that yields the best query performance and formulation, i.e., the relational one (goal ♯1).
– The Rating cross-dimensional level and the non-strict hierarchy to Product both lean on the Product level, so they should be modeled coherently, either in the relational or in the document-based model. We have a conflict here, since

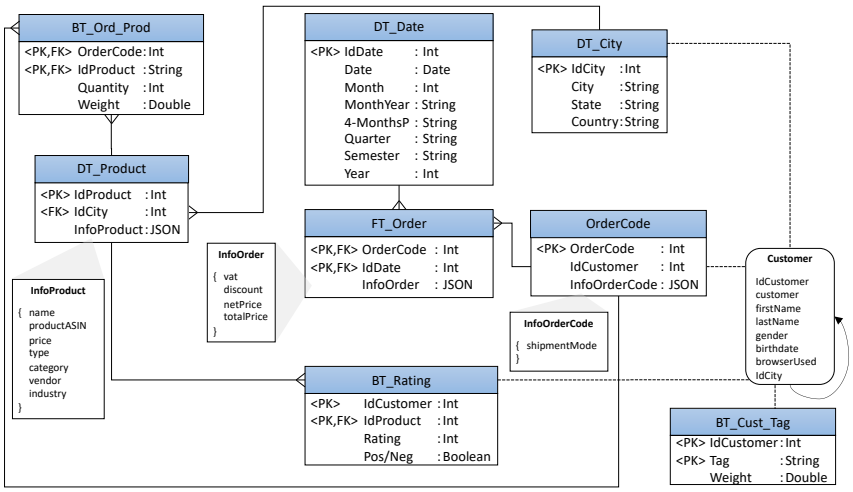

Fig. 19. M^{3}D-R schema for the Order fact; here, the relational model is used for both the Rating cross-dimensional level and the non-strict hierarchy to Product

Table 6. Total query performance, query formulation complexity, storage size, and estimated ETL formulation complexity for the different schemata; the out-of-memory (OOM) column refers to failure of query q24

Model	Schema	Query perf.	OOM	Query form.	Storage	ETL
Multi	M^{3}D-R	76.0 ± 3.2 sec		30531	1737 MB	1452
	M^{3}D-D	230.1 ± 12.5 sec		31430	1291 MB	1147
Relational	Star	60.3 ± 3.8 sec	✓	16218	1557 MB	1543
Doc.-based	Shattered	301.3 ± 17.3 sec	✓	25190	1765 MB	1002
Graph-based	Shortcut	279.9 ± 18.5 sec		59747	12321 MB	1602

cross-dimensional levels and non-strict hierarchies yield better query performance and formulation when modeled, respectively, in a shattered and in a star schema (goal ♯1). To explore this trade-off we consider two different solutions: the first one (called M^{3}D-R from here on) uses the relational model for both multidimensional elements, the second one (M^{3}D-D) uses the document-based model.

The two resulting schemata are depicted in Figures 19 and 20.

Table 6 shows an overall comparison between the two M^{3}D schemata and the three mono-model schemata proposed in the previous sections. The bad performance of M^{3}D-D is due to the higher storage space occupied by the InfoOrder JSON field in the fact table, which has a negative impact on performances in queries with high cardinalities —especially those on the knows non-onto hierarchy, namely, q11, q12, and q15. This is consistent with the considerations made in Section 5.1.1 about PostgreSQL's inefficiencies when aggregating JSON data. M^{3}D-D outperforms M^{3}D-R only for queries involving the Rating cross-dimensional attribute; as demonstrated in Section 6, such queries are more efficient when the multidimensional element is denormalized. However, in our workload, the relative impact of queries on Rating is low, thus the overall performances steer in favor of M^{3}D-R.

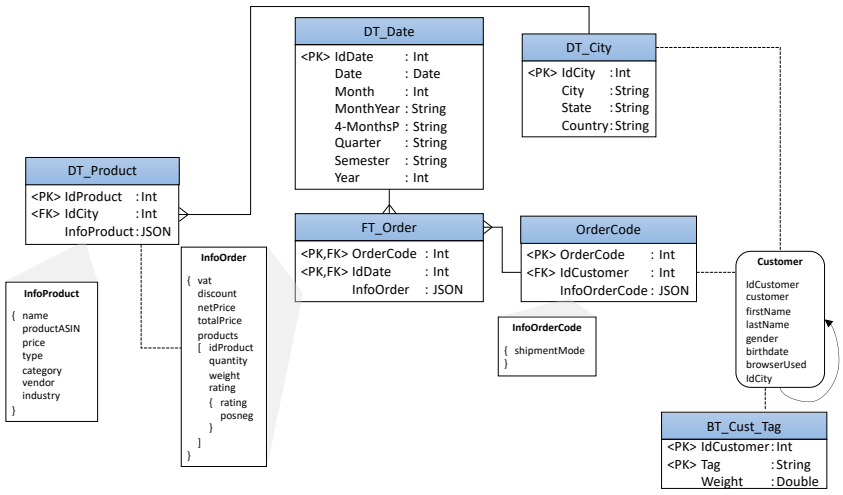

Fig. 20. M^{3}D-D schema for the Order fact; here, the document-based model is used for both the Rating cross-dimensional level and the non-strict hierarchy to Product

7. Conclusion

Multi-model data warehouses were recently proposed to store data according to the multidimensional model and, at the same time, let each of its elements be represented through the most appropriate model. To help designers in understanding how to mix different models when implementing a DW via an MMDBMS, we proposed a set of guidelines for multi-model multidimensional design based on a set of intra-model and inter-model comparisons, and showcased the guidelines on a case study. The design goals we took into account are querying performance and formulation complexity, storage, ETL formulation complexity, flexibility, extensibility, and evolvability. We studied various alternatives for the modeling of multidimensional elements, including complex hierarchies, according to three models: relational, document-based, and graph-based.

The main lessons learned are that, depending on the DW designer's will to balance design goals, different models can be mixed together within the same schema. In particular, hierarchies can use different models, and models can be mixed within the same hierarchy considering connections between fact/dimension/bridge tables and documents or graph nodes. Specific schemata (denormalized, shortcut) can be used depending on the fact model. Overall, it appears that the graph-based model is only convenient for storing non-onto hierarchies, especially in presence of a workload that includes recursive queries. While this may be due to the fact that PostgreSQL uses relational tables to store graphs, we remark that none of the MMDBMSs supporting all three models used in this paper stores graphs natively. As to the relational and document-based models, the former is preferred to reduce the storage space and improve the ETL performance, while the latter ensures better flexibility, extensibility, and evolvability.

MMDWs open many research avenues, at the conceptual level (e.g., how to extend the existing conceptual models to cope with schemaless data (Holubová, Svoboda and Lu, 2019; Holubová et al., 2021)), at the logical level (e.g., how to select and use materialized views in MMDWs), and at the physical level (e.g.,

what ad-hoc indexing strategies to adopt for MMDWs, or what is the benefit of using native graphs in an MMDWs). Our future work will be mainly placed at the conceptual level; specifically, we will investigate how to extend the UML profiles usually adopted for conceptual design of multidimensional data to cope with variety issues.

Acknowledgment

This work was partially supported by the French ANR Project ANR-20-PCPA-0002 "Building epidemiological surveillance & prophylaxis with observations near & distant" (BEYOND).

Declarations

Funding

The authors did not receive support from any organization for the submitted work.

Data availability

The datasets generated during the current study are available at `https://github.com/big-unibo/m3d-guidelines`.

Competing interests

All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Authors' contributions

All authors contributed to the study conception and design. Material preparation and data collection were performed by Enrico Gallinucci. Data analysis was performed by Enrico Gallinucci and Stefano Rizzi. The first draft of the manuscript was written by Stefano Rizzi and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

References

Beheshti, S., Benatallah, B., Nezhad, H. R. M. and Allahbakhsh, M. (2012), A framework and a language for on-line analytical processing on graphs, *in* 'Proc. WISE', pp. 213–227.

Bimonte, S., Gallinucci, E., Marcel, P. and Rizzi, S. (2022), 'Data variety, come as you are in multi-model data warehouses', *IS* **104**(101734).

Bimonte, S., Hifdi, Y., Maliari, M., Marcel, P. and Rizzi, S. (2020), To each his own: Accommodating data variety by a multimodel star schema, *in* 'Proc. DOLAP@EDBT/ICDT', Copenhagen, Denmark, pp. 66–73.

Bitnine Global Inc. (2017), 'Architecture of AgensGraph', https://bitnine.net/blog-agens-solution/architecture-of-agensgraph/. [Online; accessed 07-Sep-2021].

Boukraâ, D., Bouchoukh, M. A. and Boussaïd, O. (2015), 'Efficient compression and storage of XML OLAP cubes', *IJDWM* **11**(3), 1–25.

Boussahoua, M., Boussaid, O. and Bentayeb, F. (2017), Logical schema for data warehouse on column-oriented NoSQL databases, *in* 'Proc. DEXA', Lyon, France, pp. 247–256.

Castelltort, A. and Laurent, A. (2014), NoSQL graph-based OLAP analysis, *in* 'Proc. KDIR', Rome, Italy, pp. 217–224.

Challal, Z., Bala, W., Mokeddem, H., Boukhalfa, K., Boussaid, O. and Benkhelifa, E. (2019), Document-oriented versus column-oriented data storage for social graph data warehouse, *in* 'Proc. SNAMS', Granada, Spain, pp. 242–247.

Chen, C., Yan, X., Zhu, F., Han, J. and Yu, P. S. (2009), 'Graph OLAP: a multi-dimensional framework for graph data analysis', *Knowl. Inf. Syst.* **21**(1), 41–63.

Chevalier, M., Malki, M. E., Kopliku, A., Teste, O. and Tournier, R. (2015*a*), Implementation of multidimensional databases in column-oriented NoSQL systems, *in* 'Proc. ADBIS', Poitiers, France, pp. 79–91.

Chevalier, M., Malki, M. E., Kopliku, A., Teste, O. and Tournier, R. (2015*b*), Implementing multidimensional data warehouses into NoSQL, *in* 'Proc. ICEIS', Barcelona, Spain, pp. 172–183.

Chevalier, M., Malki, M. E., Kopliku, A., Teste, O. and Tournier, R. (2016*a*), Document-oriented data warehouses: Complex hierarchies and summarizability, *in* 'Proc. UNet', Casablanca, Morocco, pp. 671–683.

Chevalier, M., Malki, M. E., Kopliku, A., Teste, O. and Tournier, R. (2016*b*), Document-oriented data warehouses: Models and extended cuboids, extended cuboids in oriented document, *in* 'Proc. RCIS', Grenoble, France, pp. 1–11.

Chevalier, M., Malki, M. E., Kopliku, A., Teste, O. and Tournier, R. (2016*c*), Document-oriented models for data warehouses - NoSQL document-oriented for data warehouses, *in* 'Proc. ICEIS', Rome, Italy, pp. 142–149.

Chouder, M. L., Rizzi, S. and Chalal, R. (2019), 'EXODuS: Exploratory OLAP over document stores', *Inf. Syst.* **79**, 44–57.

Couto, J., Borges, O. T., Ruiz, D. D., Marczak, S. and Prikladnicki, R. (2019), A mapping study about data lakes: An improved definition and possible architectures, *in* 'Proc. SEKE', Lisbon, Portugal, pp. 453–578.

Dehdouh, K. (2016), Building OLAP cubes from columnar NoSQL data warehouses, *in* 'Proc. MEDI', Almería, Spain, pp. 166–179.

Ferrahi, I., Bimonte, S. and Boukhalfa, K. (2017), A model & DBMS independent benchmark for data warehouses, *in* 'Proc. EDA', Lyon, France, pp. 101–110.

Gadepally, V., Chen, P., Duggan, J., Elmore, A. J., Haynes, B., Kepner, J., Madden, S., Mattson, T. and Stonebraker, M. (2016), The BigDAWG polystore system and architecture, *in* 'Proc. HPEC', Waltham, MA, USA, pp. 1–6.

Gallinucci, E., Golfarelli, M. and Rizzi, S. (2019), 'Approximate OLAP of document-oriented databases: A variety-aware approach', *Inf. Syst.* **85**, 114–130.

Golfarelli, M. and Rizzi, S. (2009), *Data Warehouse Design: Modern Principles and Methodologies*, McGraw-Hill, Inc., New York, NY, USA.

Gómez, L. I., Kuijpers, B. and Vaisman, A. A. (2020), 'Online analytical processsing on graph data', *Intell. Data Anal.* **24**(3), 515–541.

Hamadou, H. B., Gallinucci, E. and Golfarelli, M. (2019), Answering GPSJ queries in a polystore: a dataspace-based approach, *in* 'Proc. ER', Salvador de Bahia, Brazil, pp. 189–203.

Holubová, I., Contos, P. and Svoboda, M. (2021), Multi-model data modeling and representation: State of the art and research challenges, *in* 'Proc. IDEAS', Montreal, QC, Canada, pp. 242–251.

Holubová, I., Klettke, M. and Störl, U. (2019), Evolution management of multi-model data - (position paper), *in* 'Proc. Poly/DMAH', Los Angeles, CA, USA, pp. 139–153.

Holubová, I., Svoboda, M. and Lu, J. (2019), Unified management of multi-model data - (vision paper), *in* 'Proc. ER', Salvador, Brazil, pp. 439–447.

Jain, S., Moritz, D., Halperin, D., Howe, B. and Lazowska, E. (2016), SQLShare: Results from a multi-year SQL-as-a-Service experiment, *in* 'Proc. SIGMOD', San Francisco, CA, USA, pp. 281–293.

Lu, J. and Holubová, I. (2019), 'Multi-model databases: A new journey to handle the variety of data', *ACM Comput. Surv.* **52**(3), 55:1–55:38.

Marzi, M. D. (2020), 'The secret sauce of Neo4j: Modeling and querying graphs', https:

//neo4j.com/blog/secret-sauce-neo4j-modeling-graphconnect/. [Online; accessed 07-Sep-2021].
Niemi, T., Nummenmaa, J. and Thanisch, P. (2001), Logical multidimensional database design for ragged and unbalanced aggregation, *in* 'Proc. DMDW', p. 7.
O'Neil, P. E., O'Neil, E. J., Chen, X. and Revilak, S. (2009), The star schema benchmark and augmented fact table indexing, *in* 'Proc. TPCTC', Lyon, France, pp. 237–252.
Ouaret, Z., Chalal, R. and Boussaid, O. (2013), 'An overview of XML warehouse design approaches and techniques', *IJICoT* **2**(2/3), 140–170.
Papastefanatos, G., Vassiliadis, P., Simitsis, A. and Vassiliou, Y. (2012), 'Metrics for the prediction of evolution impact in ETL ecosystems: A case study', *J. Data Semantics* **1**(2), 75–97.
Pedersen, T. B., Jensen, C. S. and Dyreson, C. E. (2001), 'A foundation for capturing and querying complex multidimensional data', *Inf. Syst.* **26**(5), 383–423.
Sadalage, P. J. and Fowler, M. (2009), *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*, Addison-Wesley Professional, Boston, US.
Sellami, A., Nabli, A. and Gargouri, F. (2018), Transformation of data warehouse schema to NoSQL graph data base, *in* 'Proc. ISDA', Vellore, India, pp. 410–420.
Sellami, A., Nabli, A. and Gargouri, F. (2020), Graph NoSQL data warehouse creation, *in* 'Proc. iiWAS', Chiang Mai, Thailand, pp. 34–38.
Shimura, T., Yoshikawa, M. and Uemura, S. (1999), Storage and retrieval of XML documents using object-relational databases, *in* 'Proc. DEXA', Florence, Italy, pp. 206–217.
Svoboda, M., Contos, P. and Holubová, I. (2021), Categorical modeling of multi-model data: One model to rule them all, *in* J. C. Attiogbé and S. B. Yahia, eds, 'Proc. MEDI', Tallinn, Estonia, pp. 190–198.
Tsunakawa, T. (2017), Road to a multi-model database – making PostgreSQL the most popular and versatile database. Presented at PGConf.ASIA, Tokyo, Japan.
URL: *https://www.pgconf.asia/EN/2017/day-1/#B2*
Yangui, R., Nabli, A. and Gargouri, F. (2016), Automatic transformation of data warehouse schema to NoSQL data base: Comparative study, *in* 'Proc. KES', York, UK, pp. 255–264.
Zhang, C. and Lu, J. (2021), 'Holistic evaluation in multi-model databases benchmarking', *Distributed Parallel Databases* **39**(1), 1–33.
Zhang, C., Lu, J., Xu, P. and Chen, Y. (2018), UniBench: A benchmark for multi-model database management systems, *in* 'Proc. TPCTC', Rio de Janeiro, Brazil, pp. 7–23.

Author Biographies

Enrico Gallinucci is junior assistant professor at the University of Bologna, where he received his Ph.D. in Computer Science and Engineering and teaches Business Intelligence and Big Data. His research interests currently focus on big data analytics, NoSQL and multimodel database systems, data democratization, and precision agriculture. He is associate editor for the DKE journal.

Sandro Bimonte is a researcher at INRAE–TSCF. He received his Ph.D. from INSA-Lyon, France, in 2007. From 2007 to 2008 he carried out research at IMAG, France. He is an editorial board member of the International Journal of Decision Support System Technology and of the International Journal of Data Mining, Modeling and Management, as well as a member of the Commission on GeoVisualization of the International Cartographic Association. His research activities concern spatial data warehouses and spatial OLAP, visual languages, geographic information systems, spatiotemporal databases, and geovisualization.

Patrick Marcel is an Associate Professor at the University of Tours, France. His current research focuses on database, OLAP and data warehousing, personalization, recommender systems, exploratory data analysis and data narration. He authored numerous publications in international conferences and journals on these subjects. He served as program committee member in top tier international conferences, including ER, VLDB, EDBT. He is a member of the steering committee of DOLAP and a member of the regular editorial board of DKE.

Stefano Rizzi is a Full Professor at the University of Bologna, Italy. He has authored more than 150 papers in international journals and conferences mainly in the fields of data warehousing, business intelligence, and pattern recognition. He is member of the steering committee of DOLAP and of the editorial board of DKE, and has been a member of the steering committee of the ER Conference. His research interests include data warehouse design and business intelligence, in particular OLAP on NoSQL data, social business intelligence, and analysis services for big data.

Correspondence and offprint requests to: Stefano Rizzi, DISI, University of Bologna, Bologna, 40136, Italy. Email: stefano.rizzi@unibo.it