Design of Document Databases: What can we learn from Object-Relational Databases?

George Feuerlicht
Unicorn University
Prague, Czech Republic
george.feuerlicht@unicornuniversity.net

Marek Beranek
Unicorn University
Prague, Czech Republic
marek.beranek@unicornuniversity.net

Vladimir Kovar
Unicorn University
Prague, Czech Republic
vladimir.kovar@unicorn.eu

Abstract

The emergence of applications that manage very large volumes of semi-structured data is driving the popularity of document databases such as MongoDB. One of the main appeals of document databases is schema flexibility as this allows the schema to be biased towards a particular set of applications. This approach is favoured by many developers who regard document databases to be essentially *schema-less* and focus on performance and scalability issues rather than ensuring that the schema can be *gracefully* evolved as application requirements change. There is evidence that many developers use ad-hoc methods based on heuristics rather than well-established design principles, potentially creating designs that cause data modification anomalies that can lead to the loss of data consistency. In this paper we argue that the design of document databases is not an entirely new problem and that existing object-relational database design methods can be adapted for the design of document databases. We describe a design approach that is an adaptation of object-relational design and serves as a framework for making informed decisions about alternative design strategies for document databases.

Keywords: Document databases, Database design, MongoDB

1. Introduction

Document databases represent an important recent trend in data management driven by the need to manage large amounts of semi-structured data. The rising popularity of cloud-based document databases such as MongoDB [1] has created interest in suitable design methods that can provide guidelines for developers involved with the design of document databases. There is some evidence that developers regard document databases as quasi schema-less and tend to bias the database design towards a narrow set of application requirements. Document data structures are often *hard-coded* within individual applications, breaking the separation between the data structures and applications, and resulting in the loss of data independence [2]. Although databases such as MongoDB do not enforce any particular structure on documents in a collection, good design dictates that a collection should contain documents with identical, or closely related structure resulting in a polymorphic schema [3]. The hierarchical structures inherent in JSON that form the underlying data model for document databases tend to promote denormalization, and there is evidence that developers often use various heuristics (e.g. “data accessed together should be stored together”) rather than reliable design principles to produce embedded document

structures [4]. This is in sharp contrast with relational design methods that have been widely investigated in the literature and successfully applied in professional practice for a number of decades.

In this paper we argue the importance of schema design and demonstrate that object-relational design methods [5] can be adapted for the design of document databases and assist designers in making informed decisions that ensure long-term stability of the design. Object-relational databases [6], have emerged towards the end of the last century with the aim to address the need to manage complex data types used in modern applications, as well as the requirement for integration with object-oriented languages. SQL:1999 was the first SQL standard that incorporated support for object-oriented features that include user-defined types (UDTs), structured types, collections (arrays and multisets), object identifiers (OIDs) and references (REFs), as well as support for type inheritance and methods. These features were further enhanced with the release of SQL:2003 and the following versions of SQL standard that incorporate multivalued attributes and improved array and multiset semantics. SQL:1999 is a superset of the relational SQL standard (SQL:1992) ensuring backward compatibility with earlier versions of SQL. In this paper we use SQL:1999 to refer to object-relational features of SQL, noting that these features have been further enhanced most recently by incorporating support for XML and JSON data structures [7].

The main contribution of this paper is the adaptation of an object-relational design method for the design of document databases. We focus specifically on the process of converting the conceptual ERA (Entity-Relationship Attribute) model into MongoDB data structures. We illustrate our approach with an example of a Movie Database discussing the advantages and drawbacks of the various design alternatives. In the next section (section 2) we review the existing literature on the design of document databases and in the following section (section 3) we describe our approach to the design of document databases. Section 4 contains our conclusions.

2. Related Work

Atzeni, et al. [2] argue that traditional notions of data modeling can be useful in the NoSQL context and propose the NoAM (NoSQL Abstract Model) model for NoSQL databases that exploits the commonalities between various NoSQL systems. The authors note that today's developers underestimate the importance of the separation between data and application programs and often hardcode data structures within individual applications. This leads to the loss of data independence and can result in excessive maintenance if the underlying data structures are modified. NoAM design methodology initially specifies a system-independent representation of the data that is then implemented in a target NoSQL database, taking into account its specific features. NoAM methodology defines a basic data access and distribution unit called *block* identified by a unique *block key*. Blocks represent "a maximal data unit for which atomic, efficient, and scalable access operations are provided". A NoAM database is a set of collections, and a collection is a set of blocks. The NoAM approach involves two main activities: 1) conceptual data modeling and 2) aggregate design. Aggregate design groups related entities into *aggregates*. This is followed by aggregate partitioning where the aggregates are partitioned into smaller data elements and then mapped to the NoAM intermediate data model. Finally, the intermediate data representation is mapped to the specific elements of the target NoSQL database. Decisions about aggregate boundaries are driven by data access patterns and by scalability and consistency considerations. However, NoAM provides insufficient details about how aggregates should be formed. Another weakness of NoAM approach is that it attempts to address the design of different types of NoSQL databases using a single method and this detracts from the effectiveness of this approach for document databases.

Hoberman [8] argues the importance of data modelling in the context of document database design as a process of understanding and precisely documenting the data requirements of applications. The author adopts a traditional three-layered design approach that starts with the development of the conceptual model, and then progresses to design logical and physical data models specifically targeting MongoDB. Starting from the conceptual model, the logical model involves detailed data analysis identifying all attributes flowed by data normalization. Decisions about denormalization of data structures

using document embedding and arrays are made during the physical design stage based on the frequency of query and update operations and the nature of the relationship between entities, i.e., considerations of relationship cardinality and the *strength*, differentiating between identifying and non-identifying relationships.

According to Mason [9] data modeling for document-oriented databases is similar to data modeling for traditional relational databases during the conceptual and logical modeling phases. The authors propose that in the *physical data model* entities can be combined (denormalized) by using document embedding and by replacing foreign keys by references. The paper illustrates how a relational model can be transformed into a design suitable for a MongoDB document database using a Rental Car company case study. The transformation is based on the heuristics proposed by Hoberman [8].

Another design approach for NoSQL document databases based on a conceptual schema and workload information is proposed in [10]. Workload information is used to determine an optimized logical schema, improving performance for applications. The document logical schema is composed of collections, blocks and attributes. Each collection has a root block, and all data access operations must use the root block to access data within the collection. The authors describe a conversion process and define conversion rules for mapping EER (Extended Entity-Relationship) constructs into equivalent NoSQL document structures. The conversion algorithm consists of two main steps: 1) conversion of generalization types and 2) conversion of relationship types. The conversion of generalization types follows a standard relational design methodology for mapping disjoint subtypes and produces three alternatives: a) single block based on the supertype that includes all subtype attributes, b) blocks based on the subtypes, and c) a block based on the supertype and block based on subtypes. The second step involves the conversion of relationship types based on cardinality. Conversion rules are specified for 1:1, 1:N and N:M relationships that produce either nested block design or independent blocks with references.

In [11] the authors propose a data modeling techniques that can be used for document-oriented databases to assist with data visualization. The proposed solution uses two basic concepts: Documents and Collections, and defines graphical modelling primitives for embedded and referenced documents and relationships with different cardinalities (1:1, 1:N, and N:M). The authors use illustrative examples based on MongoDB data structures to show that the proposed graphical notation produces compact and intuitive diagrams for document-orientated conceptual models.

According to Shin et al. [12], designers of NoSQL databases focus primarily on addressing non-functional application requirements to ensure good performance over large data clusters. The authors propose a NoSQL database design method that consists of three phases. The conceptual modelling phase uses the Entity-Relationship modelling approach or UML class diagrams to develop a conceptual data model that is independent of the target database platform. The logical phase is specific to the type of NoSQL database and produces a model that matches the data model of the target database (for example a document database). The final physical design phase focuses on performance and availability issues and describes the storage structures and data access methods taking advantage of a particular database platform (e.g., MongoDB).

In [3] the author discusses database design for MongoDB and considers the role of data normalization in the context of document databases, noting that resolving 1:N relationship by document embedding leads to improved data locality and consequently improved query performance. In the specific case of MongoDB, embedding documents avoids the need for join operations that are not directly supported by the database. Another advantage of document embedding is that it achieves atomicity and isolation for transactions, avoiding the complexities associated with distributed (multi-document) transactions in the context of data shards. The author considers alternatives for implementing N:M relationships and notes that using document references leads to "application-level joins" as well as a performance penalty, while embedding documents results in redundancy and the need to update the documents in multiple places. The author concludes that schema design for MongoDB tends to be more of an art than a science.

Another issue identified in the literature is the problem of NoSQL schema evolution and lack of tools for schema management [13]. The authors note that as an application

evolves, so does its schema, and that this also applies to applications with a polymorphic schema.

Numerous other approaches to the design of NoSQL databases have been documented in the literature [12, 14, 15]. A systematic review of such methods is provided in [16] with the authors concluding that the design of NoSQL databases is an active research area and that further work is required to demonstrate the applicability of these methods in real-world scenarios.

Most authors agree about the importance of the conceptual model as a starting point for the design of document databases, however there is a lack of agreement about the mapping from the conceptual to the logical to data model. Another weakness of several of the above approaches is that they attempt to address the design of different types of NoSQL databases using a single unified method. We argue that different types of NoSQL databases need a different design approach as NoSQL data models vary significantly according to the type of the database. For example, Document Databases and Graph Databases cannot be addressed using a unified design method as they have vastly different data models. Furthermore, many of the above methods introduce performance related considerations into the early stages of the design process and this can lead to sub-optimal schema design.

3. Design of Document Databases

While NoSQL is a relatively new approach to data management, in the specific case of document databases, there are important similarities with object-relational databases [6], making the design methods originally developed for object-relational databases applicable to document databases. There is no internationally recognized standard for document databases, but the data models of most existing systems are based on JSON or XML and have common characteristics. As indicated in Table 1, there are clear similarities between the data structures supported in SQL:1999 (e.g., Oracle DBMS) and the data structures used in document databases (e.g., MongoDB). From a design perspective, apart from user-defined types and type inheritance which are not supported in MongoDB, all other features are effectively identical. Design methods for object-relational databases have been extensively investigated and a number of approaches are documented in the literature [17-21].

Table 1. Comparison of Oracle Object-Relational SQL and MongoDB data structures.

SQL:1999 (Oracle)	Document Databases (MongoDB)
table	collection
table row	document
object identifier	object identifier (_id)
reference (REF)	reference
varray	array
structured type	embedded document
nested table	embedded array of documents
user-defined type	-
type inheritance	-

While there are important differences between object-relational and document databases, there are important universal principles that are independent of the underlying data model and apply to both types of databases. More specifically, the principle of separation of concerns that leads to splitting-up of the process of database design into three distinct phases: conceptual design, logical design and physical design, so that each phase can be carried out independently often by professional with specialized knowledge of the particular design area. The conceptual design phase focuses on precisely identifying user requirements and produces a data model, typically in the form of an Entity-Relationship Diagram or a UML Class Diagram. This modelling phase is entirely independent of the target DBMS (Database Management System) platform and should not consider any application specific requirements. The basic entity-relationship-attribute (ERA) constructs include strong and weak entity types (E), relationship types (R), attributes (A) and ISA (is-a) hierarchies. Most ERA approaches support only simple (atomic) attributes allowing

direct transformation of the conceptual model into a set of normalized relations. More recently, conceptual models were extended to support XML and JSON data [22].

The logical design phase is concerned with the transformation of the conceptual model into a logical model for the specific database type under consideration. In the case of relational databases, logical design is typically performed using design tools (e.g., Oracle Data Modeler [23]) that follow relational transformation rules (Table 2) and produce in a set of fully normalized tables with primary and foreign keys. This is often followed by considerations of denormalization in order to simplify the design by combining tables and avoiding *expensive* join operations. Table 2 does not include mapping of ISA hierarchies that involves design decisions about creating subtype and supertype tables and requires an input from the designer.

Table 2. Relational Mapping

Conceptual Model (ERA)	**Logical Model**
entity	table
unique identifier	primary key (PK)
attribute	table column
1:1 relationship	target table with a foreign key (FK)
1:N non-identifying relationship	target table with a foreign key (FK)
1:N identifying relationship	target table with a composite foreign key (FK)
N:M relationship	intersection table with a composite PK and FKs

The physical design phase involves mapping of the logical model to the physical model of the target DBMS (e.g., Oracle DBMS or MongoDB) taking advantage of the specific features of the target database platform and may include the design of index structures, clustering, deciding on replication strategies, and tuning the level of data consistency.

It can be argued that this well-established three-phase design process originally developed for relational databases is suitable for the design of document databases [8] and that object-relational design methods can be readily adapted for the logical design phase of this process [21]. In this paper we focus entirely on the logical design phase that involves the transformation of the conceptual ERA model into the JSON-based logical model as exemplified by MongoDB.

3.1. Logical design objectives and considerations

The logical design phase involves making informed decisions about the various design options that balance the following design objectives:

a) minimizing data redundancy and avoiding data modification anomalies
b) maximizing performance of database operations (CRUD operations)
c) minimizing the complexity of the design (*design elegance*)

In practice, such decisions are made in the context of information about data volumes and data volatility and will result in different designs for read-intensive and write-intensive applications; in this paper we restrict our consideration to data volatility. As already noted, the logical design phase involves mapping the ERA conceptual model into a set of corresponding data structures supported by the target database management system; in our case a JSON-based document database exemplified by MongoDB. Table 3 shows the mappings from the ERA model to the object-relational and document database structures. There are clear correspondences between the two mappings, and this makes it possible to adapt object-relational design methods for the logical design phase of document databases. Table 3 does not include mapping of ISA hierarchies that follows the same rules for relational and document database design and results either in a single collection that contains the supertype as well as all subtype documents, or alternatively separate collections for each subtype document. Schema flexibility that characterizes document databases makes the single collection solution preferable as the varying structure of subtype objects can be easily accommodated within a single collection.

Table 3. ERA mapping to Object-Relational and Document database structures

Conceptual Model (ERA)	**Object-Relational Database (Oracle)**	**Document Database (MongoDB)**

entity	(typed) table	collection
unique identifier	OID	identifier (_id)
attribute	table column	field
1:1 relationship	REF/structured type	reference/embedded document
1:N non-identifying relationship	REF/varray/nested table	reference/array/embedded document
1:N identifying relationship	REF/varray/nested table	reference/array/embedded document
N:M relationship	associative table with REFs	collection with references

Our proposal is to proceed in two separate phases:

1) Map the ERA conceptual model into the corresponding normalized relational model. The resulting relational model is fully normalized and contain no redundancies. Bypassing this step, as is the case with some NoSQL design methods (e.g., NoAM) can result in unintended introduction of redundancy into the design.

2) Map the relational model into the target document model with the aim to maximize performance and at the same time minimize data redundancy avoiding potential data modification anomalies. This step is analogous to denormalization in relational database design and involves the implementation of relationships using referencing (analogous to using REFs in SQL:1999) or via embedding objects or arrays of objects (analogous to arrays and multisets in SQL:1999). We discuss the trade-offs of different design strategies in the following sections.

3.2. Illustrative example

To illustrate our design approach, we use a simplified Movie Database scenario (Figure 1). The Movie Database ERA diagram uses Barker notation [24] with the red arrows indicating subtypes (Actor, Director and Crew are subtypes of Person); the identifying relationship between Movie and Review is indicated by a line across the relationship, and CreditOrder and Job are attributes of the N:M relationships between Actor and Movie and Crew and Movie, respectively. For the purpose of this discussion, we assume that the subtypes Actor, Director and Crew are complete and disjoint.

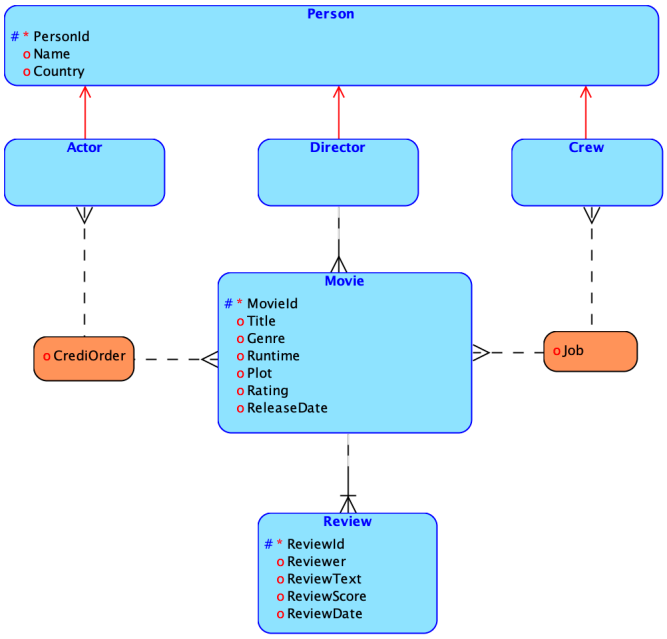

Fig. 1. Entity-Relationship Diagram of the Movie Database.

The corresponding normalized relational model shown in Figure 2 was produced by applying standard relational mapping rules (Table 2). For simplicity, the Person subtype hierarchy is mapped into a single table Person. The N:M relationships between Actor and Movie and Crew and Movie generate intersection tables MovieActor and MovieCrew, the 1:N relationship between Director and Movie results in a foreign key (Director) in the

Movie entity, and the identifying 1:N relationship between Movie and Review generates a composite foreign key (Movie_MovieID, ReviewID) in the Review table.

We can now proceed with step 2 of the design process and explore the various options for mapping the conceptual model into the model of the target document database (MongoDB) using the mapping rules in Table 3.

MongoDB stores documents in collections (indicated in green in the diagrams) making the design of collections of fundamental importance. Insertion of documents into collections automatically generates identifiers (_id) for each new document; these identifiers act as surrogate keys and replace *artificial* identifiers that are not meaningful to applications, e.g., MovieID, PersonID, etc. MongoDB collections can contain documents of different type and structure, and this presents the designer with a range of design choices. Embedding documents into collections avoids joins (typically not fully supported in document databases) and consequently improves performance of applications but may cause data redundancy and data modification anomalies. It is therefore important to consider these design options carefully. Embedding documents into MongoDB collections is analogous to the process of denormalization in relational design. In situations where the data is never updated (e.g. historical data such as movie reviews), resolving 1:N relationships using an embedded array of documents as indicated in Figure 3 (Reviews: array of review objects in the Movie collection) is acceptable. Furthermore, the Review entity is modelled as a dependent entity (see Figure 1) and therefore deletions of Movie documents does not cause deletion anomalies as reviews have no meaning outside the scope of a movie. The N:M relationships between Person and Movie are implemented using references in separate collections MovieActor and MovieCrew, and the 1:N relationship between Director and Movie is implemented using ref_Director reference in the Movie collection.

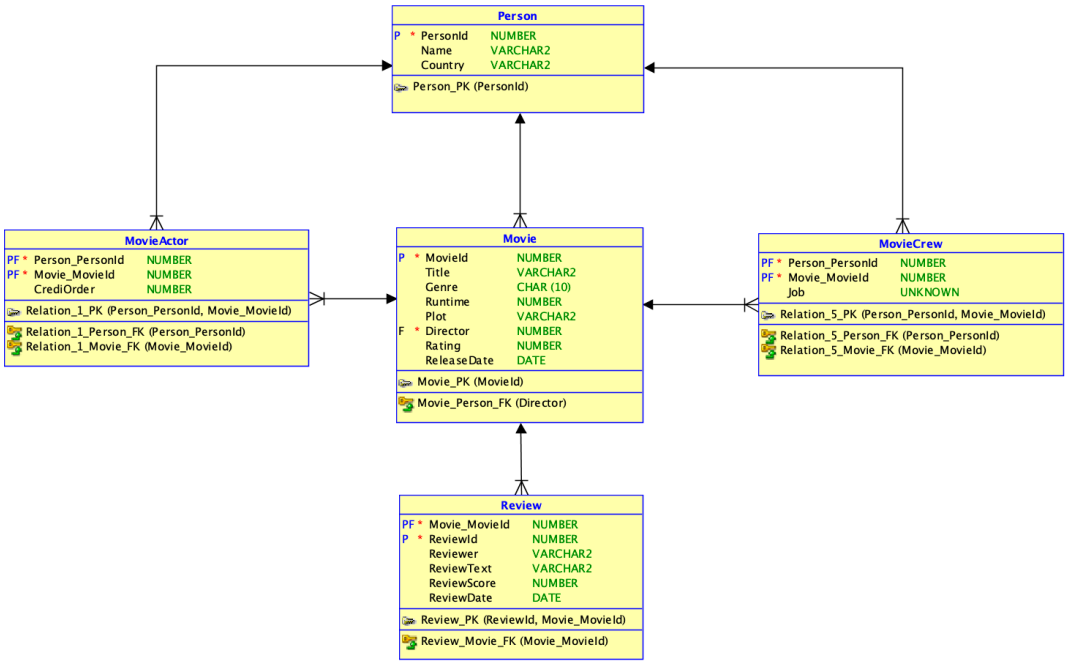

Fig. 2. Normalized relational model of the Movie Database.

This solution results in four separate collections (Person, MovieActor, MovieCrew and Movie), and while it fully avoids data redundancy, it produces a relatively complex design that is likely to suffer from poor performance.

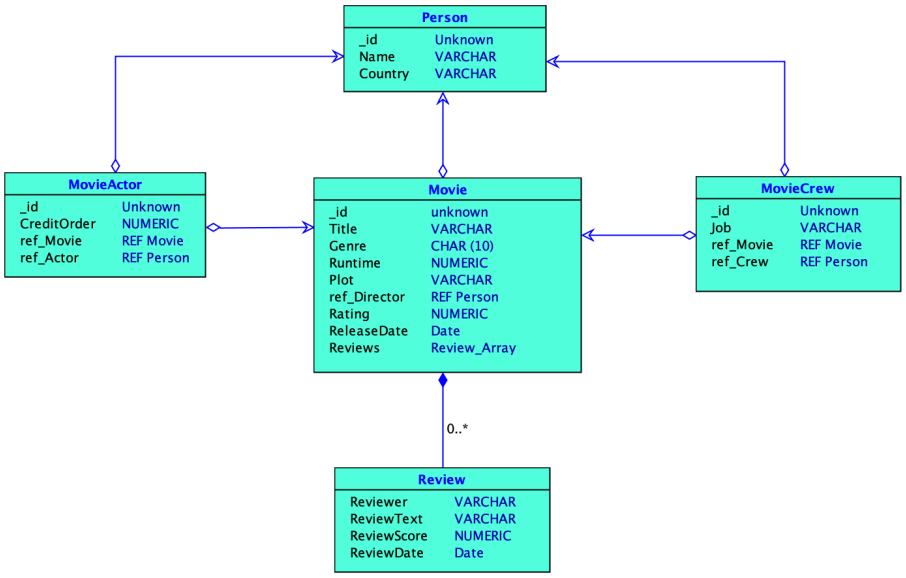

Fig. 3. Movie database design using references to implement relationships.

An alternative solution that simplifies the design by embedding the Director object into the Movie collection, and by eliminating the collections MovieActor and MovieCrew and implementing the N:M relationships using embedded arrays of references (array_ref_actor and array_ref_crew) is illustrated in Figure 4. This reduces the number of collections to two (Person and Movie) but results in more complex queries that now need to navigate array structures with corresponding impact on performance. For example, queries that need to find movies for a particular actor now need to scan the entire Movie collection and reference the Person collection via the array of references (array_ref_actor) without the benefit of indexing as the Movie collection does not contain the actor's name field. Introducing arrays of Movie references into the Person collection can alleviate this problem, but this complicates the design and introduces redundances with the potential for update anomalies.

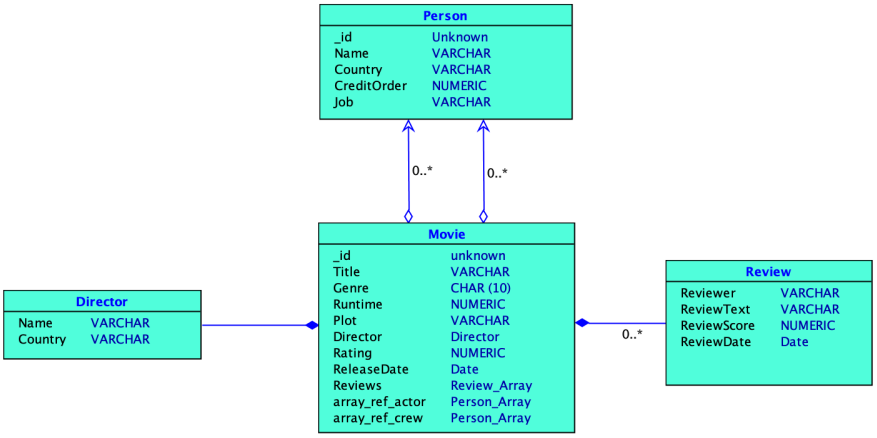

Fig. 4. Movie database design using embedded arrays of references to implement relationships.

Another design alternative is illustrated in Figure 5. Here both Actor and Crew subtypes are implemented as embedded arrays of objects, reducing the schema to a single Movie collection. This is clearly the simplest solution, but what is the impact on data redundancy? There is a general agreement that array structures violate 1NF, although it can be argued that explicit support for array data types by the database platform allows arrays to be treated as atomic attributes. In this specific case, data redundancy arises as a result of recording the Country attribute every time a given actor or member of the crew is stored. This is caused by the partial dependency of the Country attribute on the key (_id, Name) that violates 2NF. We note that for the purpose of normalization the Movie identifier (_id) is

not the key of the Movie *relation* once the embedded arrays for Actor and Crew are introduced. This redundancy and the potential loss of data consistency may be acceptable, in particular if it is assumed that the actor and crew data is rarely updated.

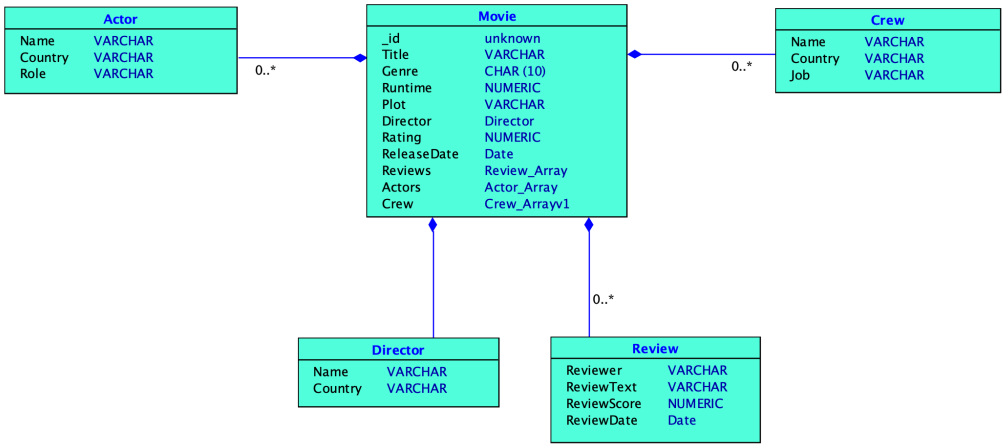

Fig. 5. Movie database design using embedded arrays of objects.

Other design solutions, such implementing all Person subtypes as a single embedded array are possible but will not be considered here. Each design alternative needs to be carefully evaluated considering the impact of data redundancy on CRUD (Create, Read, Update and Delete) operations. Clearly, read-intensive applications can tolerate higher levels of data redundancy than write-intensive applications. The objective of the design is not to minimize the number of separate collections, but to create a design that is *elegant* (i.e., simple and easy to evolve), avoids maintenance anomalies and supports the required read and write throughput.

3.3. Summary of design guidelines

In this section we summarize the design guidelines illustrated in section 3 above:

1) Map the ERA model into the corresponding set of normalized relations (relational model) using the mapping rules in Table 2.
2) Map the relational model into the document model of the target database (MongoDB):
    a. replace unnecessary (artificial) primary keys by object identifiers
    b. consider implementing 1:1 relationships by embedding objects into one of the collections – evaluate the impact of data redundancy on CRUD operations
    c. consider implementing 1:N identifying relationships as arrays of objects embedded into the parent collection - evaluate the impact of data redundancy on CRUD operations
    d. consider implementing 1:N non-identifying relationships using references, alternatively consider embedding arrays of objects into the parent collection - evaluate the impact of data redundancy on CRUD operations
    e. consider implementing N:M relationships using a separate collection of references, alternatively consider embedding arrays of references into the parent collection, or embedding arrays of objects into the parent collection - evaluate the impact of data redundancy on CRUD operations

Note that 2) above is analogous to the process of denormalization; each step that involves embedding objects or arrays of objects into a collection should be followed by considerations of redundancy that may introduce and the potential that CRUD operations may result in the loss of data consistency. In situations where the data is never or rarely updated, embedding is a good design option as it avoids having to deal with combining data from different collections. This applies in particular to historical data as is the case with the Reviews in our Movie Database example.

4. Conclusions

Document databases represent an important type of NoSQL that has gained widespread popularity with some organizations adopting databases such as MongoDB as a replacement for their existing relational DBMS. This has generated interest in design methods that support the design of document databases. There is evidence that many developers use ad-hoc methods for document database design based on heuristics rather than of well-established design principles. In this paper we argue that the design of document databases is not an entirely new problem and that object-relational design methods can be readily adapted for the design of document schemas. The design guidelines described in section 3 represent a framework for making informed decisions about alternative design strategies and help to avoid maintenance issues that may arise as the database evolves in response to future application requirements. Finally, we note that universal database principles such as data independence and data normalization apply equally to document databases and cannot be disregarded in the new era of NoSQL.

References

1. MongoDB. *MongoDB*. 2021 [cited 2021 15.03.2021]; Available from: https://www.mongodb.com/3.
2. Atzeni, P., et al., *Data modeling in the NoSQL world.* Computer Standards & Interfaces, 2016.
3. Copeland, R., *MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database*. 2013: " O'Reilly Media, Inc.".
4. MongoDB. *Data Modeling Introduction — MongoDB Manual*. https://github.com/mongodb/docs-bi-connector/blob/DOCSP-3279/source/index.txt 2021 [cited 2021 04.04.2021]; Available from: https://docs.mongodb.com/manual/core/data-modeling-introduction/.
5. Feuerlicht, G., J. Pokorný, and K. Richta, *Object-Relational Database Design: Can Your Application Benefit from SQL:2003?*, in *Information Systems Development*, C. Barry, et al., Editors. 2009, Springer US. p. 975-987.
6. Melton, J., *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features.* 2002, Elsevier Science Inc.: New York, NY, USA. p. 592
7. ISO. *ISO/IEC 9075-1:2016*. 2021 [cited 2021 04.04.2021]; Available from: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/35/63555.html.
8. Hoberman, S., *Data Modeling for MongoDB: Building Well-Designed and Supportable MongoDB Databases*. 2014: Technics Publications.
9. Mason, R.T. *NoSQL databases and data modeling techniques for a document-oriented NoSQL database*. in *Proceedings of informing science and IT education conference (InSITE)*. 2015.
10. de Lima, C. and R. dos Santos Mello. *A workload-driven logical design approach for NoSQL document databases*. in *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*. 2015. ACM.
11. Vera, H., et al. *Data modeling for NoSQL document-oriented databases*. in *CEUR Workshop Proceedings*. 2015.
12. Shin, K., C. Hwang, and H. Jung, *NoSQL database design using UML conceptual data model based on Peter Chen's framework.* International Journal of Applied Engineering Research, 2017. **12**(5): p. 632-636.
13. Scherzinger, S., M. Klettke, and U. Störl, *Managing schema evolution in nosql data stores.* arXiv preprint arXiv:1308.0514, 2013.
14. Varga, V., K. Jánosi, and B. Kálmán, *Conceptual design of document NoSQL database with formal concept analysis.* Acta Polytech. Hungarica, 2016. **13**(2): p. 229-248.
15. Benmakhlouf, A., *NOSQL implementation of a conceptual data model: UML class diagram to a document-oriented model.* International Journal of Database Management Systems (IJDMS), 2018. **10**(2).
16. Roy-Hubara, N. and A. Sturm, *Design methods for the new database era: a systematic literature review.* Software and Systems Modeling, 2020. **19**(2): p. 297-312.
17. Marcos, E., B. Vela, and J.M. Cavero. *Extending UML for object-relational database design*. in *International Conference on the Unified Modeling Language*. 2001. Springer.
18. Marcos, E., B. Vela, and J.M. Cavero, *A methodological approach for object-relational database design using UML.* Software and Systems Modeling, 2003. **2**(1): p. 59-72.

19. Marcos, E., et al. *Aggregation and composition in object-relational database design*. in *proceedings of the 5th east-european conference on Advances in Databases and Information Systems*. 2001.
20. Soutou, C., *Modeling relationships in object-relational databases.* Data & Knowledge Engineering, 2001. **36**(1): p. 79-107.
21. Feuerlicht, G., J. Pokorny, and K. Richta, *Object-Relational Database Design: Can Your Application Benefit from SQL:2003?* Information Systems Development: Challenges in Practice, Theory and Education, Vols 1and 2, ed. C. Barry, et al. 2009. 975-987.
22. Ganguly, R. and A. Sarkar, *Evaluations of conceptual models for semi-structured database system.* International Journal of Computer Applications, 2012. **50**(18).
23. SAAD, N.A.M. and M. MUNIANDI, *The Reflections on the using of Oracle Data Modeler in Creating Entity Relationship Diagram (ERD).* The Reflections on the using of Oracle Data Modeler in Creating Entity Relationship Diagram (ERD), 2020. **66**(1): p. 7-7.
24. Barker, R., *CASE method: entity relationship modelling*. 1990: Addison-Wesley Longman Publishing Co., Inc.