Towards Principled, Practical Document Database Design

Michael Carey
UC Irvine and Couchbase, Inc.
mjcarey@ics.uci.edu

Wail Alkowaileet
Saudi National Center for AI (NCAI)
walkowaileet@nic.gov.sa

Nick DiGeronimo
UC Irvine
nick@nickdigeronimo.dev

Peeyush Gupta
Couchbase, Inc.
peeyush.gupta@couchbase.com

Sachin Smotra
Dataworkz, Inc.
sachin@dataworkz.io

Till Westmann
Couchbase, Inc.
till@couchbase.com

ABSTRACT

Relational database design is a well-understood process enabled by a combination of database theory (e.g., normal forms) as well as conceptual modeling (e.g., ER-based design). In contrast, database design for NoSQL databases, notably document databases, is often approached in a much more ad hoc manner. It is frequently driven by application details and physical considerations that muddy the design process in ways all too reminiscent of the pre-relational database era. In this paper, we argue for a return to sanity – for a logical, data-first, conceptually grounded approach to document database design. We explain how such an approach can work, yielding a clean, query-friendly document database design. We also highlight a collection of document (JSON) anti-patterns to avoid. The process and the anti-patterns both stem from the authors' experiences in current and past lives when dealing with a wide variety of JSON document data from commercial applications, government applications, and university research applications.

PVLDB Reference Format:
Michael Carey, Wail Alkowaileet, Nick DiGeronimo, Peeyush Gupta, Sachin Smotra, and Till Westmann. Towards Principled, Practical Document Database Design. PVLDB, 18(12): 4804-4816, 2025.
doi:10.14778/3750601.3750606

1 INTRODUCTION

As the popular saying from Spider-Man goes, "With great power comes great responsibility." Since their inception in the 1960s, database management systems have served as a powerful tool for storing, managing, and analyzing information, both for powering interactive applications as well as supporting downstream business intelligence activities. To effectively harness that power, the activity of database design has been a key problem in the database field from the outset. As described in modern database textbooks (e.g., [28, 33]) and summarized in [35], three key components of the database design process are: (1) conceptual design, where the information requirements for a given database are captured and modeled; (2) logical design, where the conceptual design is translated into a design expressed in terms of a data model (e.g., the relational model); and (3) physical design, where various physical design choices (such as indexing) are made based on the features provided by the target database management system.

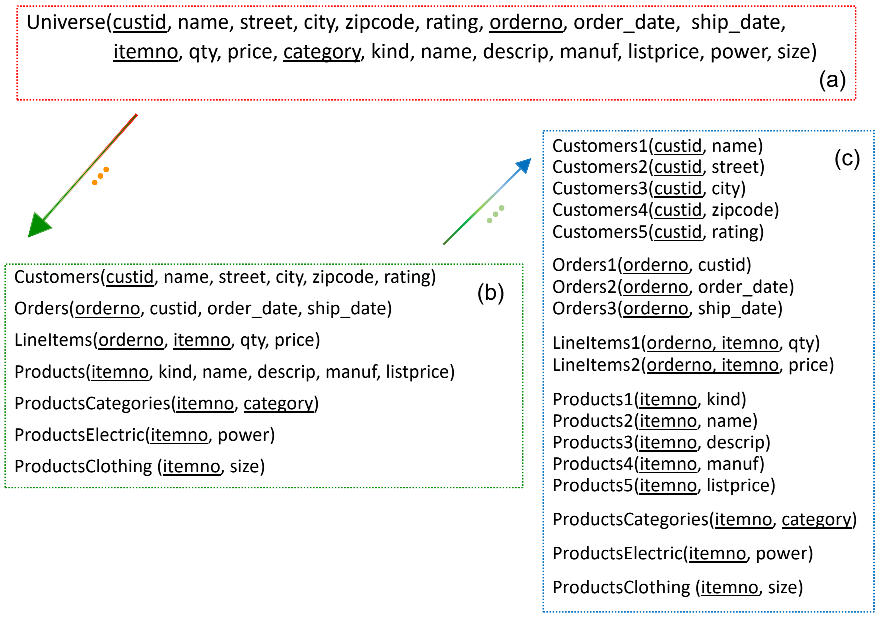
For the first two decades in database history, commercial database systems were based on data models that were quite physical in nature, e.g., the hierarchical model of IBM's IMS system or the network (CODASYL DBTG) model [22]. Their use required application developers and data analysts to write programs to carefully navigate through data structures and to perform look-ups in indexes to achieve acceptable performance. As a result, database design involved thoroughly understanding the physics of a given database's intended application(s), and adding to or changing an application required rethinking the database's design. This changed in a revolutionary manner when Ted Codd introduced the relational model, with its declarative, set-oriented query interface, thereby gifting us with *data independence* – the ability to change a database's physical structure without having to reprogram the applications that use it (and vice versa) – and with the ability to support unforseen ad hoc queries. Owing to the usability of languages like Quel and SQL and their resulting productivity gains, commercial relational database systems appeared and largely displaced their commercial predecessors for new applications over the course of the 1980s.

The database design process was similarly revolutionized by the arrival of relational database systems. Relational database systems enabled a data-first approach to database design in which the physical database design step is last and can be performed separately from the conceptual and logical design steps, serving to tune the database "under the hood" to the application(s) needs without affecting the logical structure of the database. Similarly, changes in an application's performance requirements (e.g., the need for new or different queries) can be addressed via such tuning (e.g., the creation of a new index) at the physical level alone. This is Ted Codd's legacy – thank you, Ted! – resulting in today's thriving, nearly $80 billion dollar relational database software market.

Five decades after Codd's revolution, the world is different. Relational database systems remain the dominant technology in the database market [34], but today's application requirements are much more demanding in terms of scale, performance, and schema flexibility. In addition, the kinds of data that now need to be managed, in terms of their variety and regularity (or lack thereof), is very different. These changes have led over the last two decades to the creation and deployment – in serious production use – of alternative, so-called NoSQL[1], database systems [20, 31, 34]. These include key-value stores, column-family database systems, document database systems, and graph database systems.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750606

[1]NoSQL initially stood for "no SQL", but today it is commonly said to mean "not only SQL", as many NoSQL systems now offer SQL-inspired query interfaces as well.



Universe(custid, name, street, city, zipcode, rating, orderno, order_date, ship_date, itemno, qty, price, category, kind, name, descrip, manuf, listprice, power, size)

(a)

Customers(custid, name, street, city, zipcode, rating)
Orders(orderno, custid, order_date, ship_date)
LineItems(orderno, itemno, qty, price)
Products(itemno, kind, name, descrip, manuf, listprice)
ProductsCategories(itemno, category)
ProductsElectric(itemno, power)
ProductsClothing (itemno, size)

(b)

Customers1(custid, name)
Customers2(custid, street)
Customers3(custid, city)
Customers4(custid, zipcode)
Customers5(custid, rating)
Orders1(orderno, custid)
Orders2(orderno, order_date)
Orders3(orderno, ship_date)
LineItems1(orderno, itemno, qty)
LineItems2(orderno, itemno, price)
Products1(itemno, kind)
Products2(itemno, name)
Products3(itemno, descrip)
Products4(itemno, manuf)
Products5(itemno, listprice)
ProductsCategories(itemno, category)
ProductsElectric(itemno, power)
ProductsClothing (itemno, size)

(c)

Figure 1: A Few Alternative Relational Designs.

Spanish philosopher George Santayana (in *The Life of Reason*, 1905) stated: "Those who cannot remember the past are condemned to repeat it." NoSQL database systems originated outside the relational database community, and Santayana's adage has unfortunately been borne out in the database design methodologies being commonly recommended by their vendors and consultants. NoSQL designers have been guided to return to design methods that are much more application-first and physical rather than being data-first in nature. (Sorry, Ted...!) One goal of this paper is to stem that tide by showing how a data-first approach can be used effectively, given the nature of today's document database systems. A second goal is to present a set of document design anti-patterns that should be avoided regardless of the design approach being followed.

The remainder of this paper is organized as follows: Section 2 reviews the relational database design process, briefly discussing both normalization theory and ER-based design thinking. Section 3 then delves into document database design, discussing flexible schemas and current design practices followed by recommending a methodology that flows from an ER model (similar to relational database design). Section 4 presents a set of document anti-patterns that are either dangerous or query-unfriendly or both; these patterns have been encountered in practice by one or more of the authors. Finally, Section 5 concludes the paper.

2 RELATIONAL DATABASE DESIGN

There are two main tools used in relational database design: normalization theory, and ER modeling followed by ER-to-relational schema translation. We review both here as an important precursor to proposing our recommendation for document database design in Section 3, as we will show how a number of these principles also apply there. Both tools lead to a logical schema, i.e., a set of relational tables. This is then followed by a physical design step in which decisions are made about the file organizations, index structures, and partitioning options for the tables.

2.1 Normalization Theory

Given the information needed for a given application, the problem for the relational database designer is to come up with a "good" set of tables in which to store the required information. As a simple example, Figure 1 shows three alternative designs for a database involving customers, products, and the orders placed by customers for products. As a first option, Figure 1(a) sketches how the entire database's worth of information could be stored in one large spreadsheet-like "universal" table (with the table's primary key columns being underlined). As an alternative, Figure 1(b) shows what will likely strike most readers as a more natural design where the information has been decomposed into multiple tables that separate the information about customers, products, and orders. Figure 1(c) shows a third alternative design in which the tables of the second alternative have been further decomposed into an extreme design in which each piece of information (each "fact") lives in its very own table. In this last design the tables' common primary keys hold the separated values together logically.

All of these designs can hold the application's information – but which is a "good" design, and which might be viewed as the "best" design? Functional dependencies and normalization theory were invented to answer these questions by giving relational database designers the tools to formulate or improve on a given relational database design by decomposing its current table(s) into a set of tables that minimize redundancy – i.e., into a design that avoids storing a given fact, like that the name of customer "C13" is "T. Cody", in more than one row of a table in the database. Figure 1(a) is clearly not a good design, as it will store an instance of this fact over and over, once for each line item in each order placed by a

Table 1: Dependency-Driven Relational Database Design

Normal Form	Description
1NF	All fields must be atomic (*i.e.,* scalar)
2NF	Avoid partial dependencies on PK (using FDs)
3NF	Avoid transitive dependencies on PK (using FDs)
BCNF	Avoid "all" remaining redundancy (using FDs)
4NF, 5NF (PJNF)	Avoid "nesting" redundancies (using MVDs)

given customer. This redundancy will waste space, obviously, and it will also waste time since every row where "C13" appears as the customer id will have to be updated if the customer requests that their name be updated (e.g., to "Thomas Cody"). It also introduces room for inconsistent data if some rows are somehow missed by the application when a name change occurs.

Functional dependencies (FDs) provide a formal notation for capturing such facts. The FD $custid \longrightarrow name$ captures the fact that a given customer id should only have one name associated with it in the database, so wherever those two fields appear together in a table, the name known to be associated with a given customer id should be the same. This example FD can be read as "the customer id determines the name". Given a proposed database design and a set of FDs that properly model which fields determine which other fields, normalization theory provides a way to arrive at a design that minimizes redundancy. As the relational design problem was studied, researchers identified a series of normal forms – based on FDs – that avoid various ways in which redundancy can crop up in a given relational schema design.

Without going into great detail, Table 1 lists the major normal forms as we know them today along with the kinds of FD-related problems that they avoid [28, 33]. 1NF is different from the others and is simply the relational model's rule that each field in a table must be atomic (a scalar value). The other normal forms each aim to avoid potential redundancies. 2NF avoids situations where a table has a composite key, like a primary key (PK) of ($custid$, $itemno$), and has non-key fields that are involved in facts (FDs) that depend on some but not all of the PK fields. Such facts would be redundantly stored for each $custid$/$itemno$ pair. 3NF avoids problems that 2NF would miss where there is a transitive dependency on a PK that could also lead to redundant fact storage.[2] Lastly, in terms of FDs, BCNF avoids certain corner cases where redundancy can be hiding within the fields of a composite primary key.

The bottom row in Table 1 lists two additional normal forms, 4NF and 5NF (*a.k.a.* PJNF), that address potential redundancies due to dependencies involving what would be nested fields if the relational model didn't insist on all fields being atomic. In our example, associated with a given product is a *set* of categories, so wherever the product's id ($itemno$) field appears together in a table with a category field ($category$), a given id value should always be associated with the same *set* of category values across the rows of the table. This can be captured with the addition of the notion of a multivalued dependency (MVD), e.g., $itemno \longrightarrow\!\!\!\!\longrightarrow category$ in this example. (A good mental model is to think about the *set of categories* associated with a product being functionally dependent on the product's id, $itemno$.)

As it turns out, Figure 1(b) is the relational design that one would obtain by starting from the Universe table and then using functional dependency theory to decompose it (normalize it) into a nice BCNF equivalent. Figure 1(a) is only in 1NF. Figure 1(c) also turns out to be in BCNF and avoids redundant storage, but decomposing tables to that extent would lead to more difficult querying (lots of joins) and more expensive updates (lots of little updates).

2.2 ER Modeling

An alternative path to arriving at a "good" relational database design is based on the seminal work of Peter Chen from the mid-1970's on the Entity-Relationship (ER) model [23] and its subsequent extensions. This path involves (1) capturing the conceptual data model for an application (or set of applications) in the form of an extended ER model and then (2) translating the conceptual data model into a logical data model, in this case a relational one. The ER model for an application expresses the information that's needed in terms of *entities*, *relationships*, and *attributes*. Later extensions to Chen's initial ER model include support for modeling composite attributes, multivalued attributes, and inheritance [28, 33].

To illustrate step (1), Figure 2 shows an extended ER model for the simple commerce database that we have thus far been describing informally or relationally. This model was inspired by the sample data in Appendix A of Don Chamberlin's SQL++ book [21], but it includes some extensions to his example data that we will utilize in Section 3 when we explain how to translate from an extended ER design into a document database "schema". To explain the extended ER model in a nutshell, here is what Figure 2's model is saying:

- There is an entity set called Customers. A Customers entity has four attributes, $custid$ (which is its key), $name$, $address$, and $rating$. The attribute $address$ is composite (nested) and has sub-attributes $street$, $city$, and $zipcode$.
- There is an entity set called Orders with attributes $orderno$ (its key), $order\_date$, and $ship\_date$.
- There is a one-to-many relationship Place to relate each Customers entity to its associated Orders entities.
- There is a weak entity set called Items with attributes $lineno$ (its key), $quantity$, and $price$. (The double lines in the figure express the entity set's weak-ness.)
- Each entity in the Items entity set is existence-dependent on a parent entity in Orders. This is modeled by the weak

[2]E.g., if we wanted to record which continent a customer is from along with that continent's name (region), size, and population, the FDs $custid \longrightarrow region$ and $region \longrightarrow population$ would lead to the transitive dependency $custid \longrightarrow population$, warning us not to store region facts in the same table as customer facts lest we redundantly repeat region facts for each customer.

Table 2: ER-Driven Relational Database Design

ER Concept	Relational Artifact
Entity	Table with entity's attributes and PK
Relationship (M:N)	Table with relationship's attributes and FKs
Relationship (1:N)	Merge relationship table into N-side's entity table
Composite attribute	Use a flattened column naming convention
Multivalued attribute	Add separate side table with entity's PK as FK
Inheritance hierarchy	Delta tables or mashup table

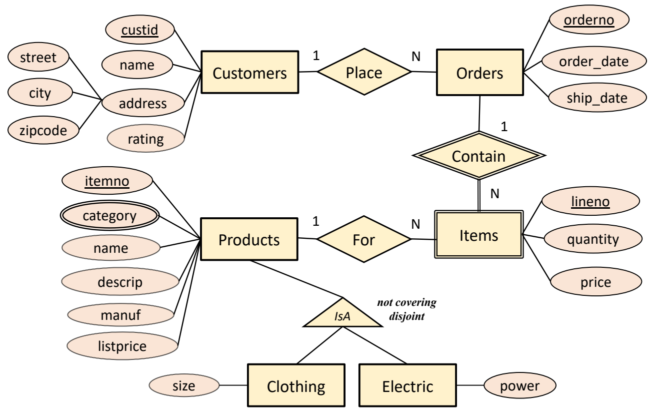


Figure 2: Extended ER Model for a Commerce Example.

1:N relationship, Contain, which relates each Items entity (a line item) of an order to its containing Orders entity.

- There is an entity set called Products with attributes *itemno* (its key), *category*, *name*), *descript*, *manuf*, and *listprice*. The attribute *categories* is multivalued, so a Products entity may have several category values.
- The Products entity has two entity subtypes, Clothing and Electric products, each having its own additional attributes (*size* for Clothing products, *power* for Electric products). A given Products entity instance can be a regular product or it can be one of the product subtypes.
- There is a 1:N relationship called For that relates each Items entity to the Products entity that that line item is for.

Given an ER diagram produced by step (1), like the example that we just walked through, step (2) involves translating the ER diagram into an appropriate relational schema. Table 2 lists how each feature in a given extended ER diagram is typically translated into tables. Figure 1(b), introduced earlier, shows what Table 2's translation process would produce when given Figure 2's ER diagram. Notice how in Figure 1(b) the entity sets have become tables with appropriate primary key fields and other fields. Had there been an M:N relationship, it would have become a table whose primary key is a composite of the primary keys from the two related tables. If a relationship has attributes, which is not the case here, they would appear as additional fields in the relationship table. 1:N relationships can be modeled in the same way, but their primary keys are just the key from the N side; as a result, their fields can be included in the N side's entity table to avoid having a separate relationship table and resulting unnecessary joins if so desired. An example of this can be seen in Figure 1(b), where the Orders entity table includes the field *custid* rather than introducing a separate table with (*custid*, *orderno*) pairs to relate customers and orders.

We turn now to translating the extensions of Chen's original model. Since the relational model does not allow data to be nested, composite attributes have been flattened[3] in the relational schema. Again due to no nesting, multivalued attributes result in the need for a side table (*ProductsCategories* in our example) where the multiple values can be held and associated with the entity table's key field value. Inheritance can be modeled in several ways, but the most common one, used here, is to have side tables for each subtype to hold its additional fields. Again, Figure 1(b) shows the complete relational translation result for Figure 2.

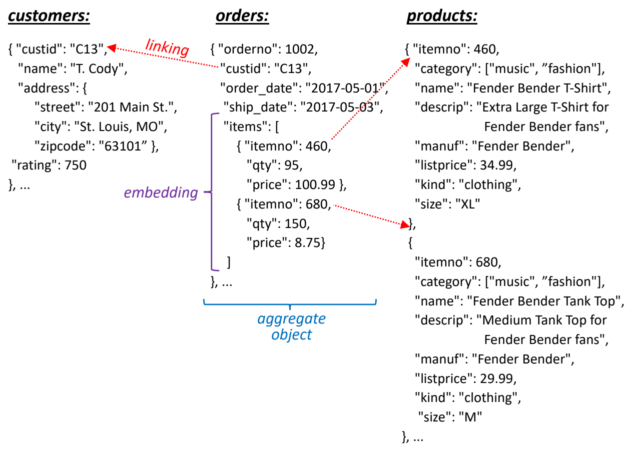
2.3 Relational Design Today

As mentioned above, both the normalization approach and the ER approach lead to the same final result in this case. So how are the two aforementioned approaches used in practice today? If one is given a set of tables to improve, with no accompanying ER model, normalization theory can help. It is more common nowadays, however, for a relational schema to be designed via the ER path since it is more intuitive and it nicely documents the data model. Moreover, done right, an ER model's relational translation will yield a 3NF or BCNF design; problem solved. Lastly, given a relational schema resulting from either approach, the final step would be physical design (*a.k.a.* tuning), including the selection of indexes to be created to accelerate the application's queries.

3 DOCUMENT DATABASE DESIGN

We have now seen, or been reminded of, how database design is handled in the relational world based on Ted Codd's gift of data independence. In this section we explain how that gift can be applied in the document database world. We first discuss "schemas" in that world and review common current practices and/or recommendations for document database design. We then explain and make the case for a data-first approach based on ER modeling and data independence. We conclude by briefly discussing some physical considerations that might feed back into the design process.

[3]One minor detail in our example is that since *address* is a composite attribute, a purely mechanical translation would likely name the tables fields for the nested ER attributes *address_street*, *address_city*, and *address_zipcode*.



```
{ "custid": "C13",
  "name": "T. Cody",
  "address": {
     "street": "201 Main St.",
     "city": "St. Louis, MO",
     "zipcode": "63101" },
 "rating": 750
}, ...
```

```
{ "orderno": 1002,
  "custid": "C13",
  "order_date": "2017-05-01",
  "ship_date": "2017-05-03",
  "items": [
    { "itemno": 460,
      "qty": 95,
      "price": 100.99 },
    { "itemno": 680,
      "qty": 150,
      "price": 8.75}
  ]
}, ...
```

```
{ "itemno": 460,
  "category": ["music", "fashion"],
  "name": "Fender Bender T-Shirt",
  "descrip": "Extra Large T-Shirt for
                Fender Bender fans",
  "manuf": "Fender Bender",
  "listprice": 34.99,
  "kind": "clothing",
  "size": "XL"
},
 {
  "itemno": 680,
  "category": ["music", "fashion"],
  "name": "Fender Bender Tank Top",
  "descrip": "Medium Tank Top for
                Fender Bender fans",
  "manuf": "Fender Bender",
  "listprice": 29.99,
  "kind": "clothing",
   "size": "M"
}, ...
```

Figure 3: Document Database Design Concepts.

3.1 Document Schemas

Document databases are known for their schema flexibility, i.e., for supporting collections of semistructured data that can be nested, heterogeneous, and/or schemaless [31]. Most document databases today have adopted self-describing data models based on JSON [10]. Examples, in alphabetical order, include: Apache AsterixDB [3, 19], CosmosDB [14], Couchbase [2, 7, 17, 26], DynamoDB [1], and last but certainly not least, MongoDB [11]. Cassandra [5] is another well-known NoSQL database system, but it is different in that while it supports nesting, it is less flexible because it expects collections (flat or nested tables) to have predefined schemas expressed using a user model provided for data definition purposes.[4]

As explained nicely in [31], the notion of a schema is still present in document databases. Applications that use document databases still have to be written based on the objects in the database and the information that they contain – it's just that the knowledge of the schema is embedded within the application code rather than being explicitly defined within and managed by the database. There are advantages and disadvantages to such schemalessness, but the ability for the data in the database to evolve quickly, i.e., to have schema fluidity as applications evolve, is one of several reasons that document databases have gained favor in a growing number of application areas.

Figure 3, which we will revisit again later, shows an example of the collections of documents that a document database system might store to support our commerce-inspired running example. Rather than the seven "flat" tables of Figure 1(b), here we have three collections, one for customers, one for orders, and one for products. Each object, at least conceptually, is self-describing; it comes with both field names and field values. Fields can be nested, and different objects might have more, fewer, or different fields than other objects. For example, customers outside the US might have different address information; orders that haven't shipped yet might not have a *ship_date* field at all; different kinds of products might have different fields.

[4]Cassandra is commonly referred to as being a "wide-column store", not a document store, and its data model is not JSON-based [6].

Even in the absence of a predefined schema, an order management application, for example, will be written based on expectations about what it may encounter in the database's collections. The program would be "surprised" if one of the objects in the orders collection is a customer, or if one of the order objects is missing an expected *custid* field indicating which customer placed it. As a result, when storing and manipulating document data, it is still the case that the schema(s) of the database's collections must be well-designed. Indeed, schema design for document databases is every bit as important as relational database design – perhaps more so, in fact, as document database systems have richer data models and provide fewer "guardrails".

3.2 Current "Best" Practices

As discussed earlier, database design for the pre-relational database systems of the B.C. ("Before Codd") era was application-first in nature. One had to think physically to ensure a proper data organization that could support efficient navigation by the application. One also had to rethink things when new requirements cropped up, potentially modifying both the data organization and the existing application to account for the additional requirements.

Unfortunately, the current database design processes being recommended for NoSQL database systems, including document database systems, have regressed to being much more physical in nature. Figure 3 shows that the data relationships in document databases can be represented either via *embedding* (nesting) or *linking* (relational-style value matching). Embedding is commonly pushed in favor of linking due to system limitations. CosmosDB [14], DynamoDB [1], and Cassandra [5] all lack support for joining data across collections, offering only single-collection query capabilities. Thus, data that needs to be accessed together needs to be stored together in these systems, or else the application programmer will be forced to deal with the additional complexity of writing application-level joins. In contrast, for document database systems whose query languages do support joins, like MongoDB [11], Apache AsterixDB [3, 19], and Couchbase [2, 7, 17, 26], the database designer has a wider range of choices in terms of when and why to favor linking over embedding or vice versa – to join or not to join – as we will discuss further in the next section.

Another challenge for database designers targeting DynamoDB, Cassandra, or CosmosDB is that they prominently expose their distributed data partitioning scheme(s) and per-partition storage organizations as part of the logical database design process. A collection's primary key is a carefully crafted composite of fields that serve to first guide partition formation and then guide the storage order of data within a partition, which is much more than the simple notion of a primary key being just a unique identifier in the ER or relational models. Careful key design is thus a critical aspect of database design for those systems, and getting it wrong can lead to something between poor performance and terrible performance (full scans of all data in all partitions) or the inability to execute certain queries at all. (E.g., Apache Cassandra won't filter on non-key columns by default, requiring `ALLOW FILTERING` to explicitly be specified in a query.) Applications that need to efficiently access data from several angles, e.g., with different predicates or different combinations of fields, have to maintain several copies of the same data with appropriately supportive keys.

Other document database systems have much less restrictive query capabilities, including language support for multi-collection queries like joins and subqueries. Such systems include MongoDB, Couchbase, and Apache AsterixDB. Historically, however, MongoDB and Couchbase started their lives as distributed key-value JSON document stores [13, 17]. Aggregation, joins, and multi-object transactions were not initially available in MongoDB, and similarly in Couchbase Server, so one still finds remnants of "physical thinking" and a preference for embedding over linking in some of their current materials on recommended JSON database design practices.

3.3 From ER Designs to Documents

As we described in Section 2, the first step in modern best practices for relational database design is to use ER (or similarly UML [31]) modeling to capture the conceptual model for the desired database. ER modeling is also recommended by some as a good initial step in the document (and other NoSQL) database design process [8, 24], while others quickly move into recommending product-specific models [6] or focus the bulk of their recommendations on the question "if you would have had these tables in the SQL world, how would you model their data in the document world instead?" [8, 12]. There have also been a few proposals for new conceptual modeling frameworks for the NoSQL world, e.g., NoAM [16]. The goal of this section is to show how one can methodically start with the familiar and time-tested ER model and end up with a "good" document database design consisting of collections of JSON objects.

Before proceeding down the ER path, it is worth taking a quick historical detour to make two points. First, nested database design is not a new problem. Significant research attention was paid in the 1980's to the idea of generalizing SQL's flat relational model to support nested tables and building systems to support such a model (e.g., [30, 32]); the line items of an order could then be stored more naturally as a nested *items* table field in each row of the orders table, for example. This is referred to as the non-first normal form (NF^2) relational model. Second, it is common today, but actually wrong, for nested document schemas to be referred to as "denormalized". While such schemas are indeed not in 1NF, it is still possible for nested designs to be normalized to avoid redundancy in a "one fact, one place" sense that the other relational normal forms do, and the late 1980's saw the use of MVDs to guide the design of normal forms for nested tables [27, 29]. However, for the same reason that ER modeling has largely displaced functional dependency-based design in the relational world, here we will follow the ER path rather than a path based on nested normalization theory. We will reflect later on how proper ER-model-to-JSON-schema translation will indeed lead to a non-redundant document database design.

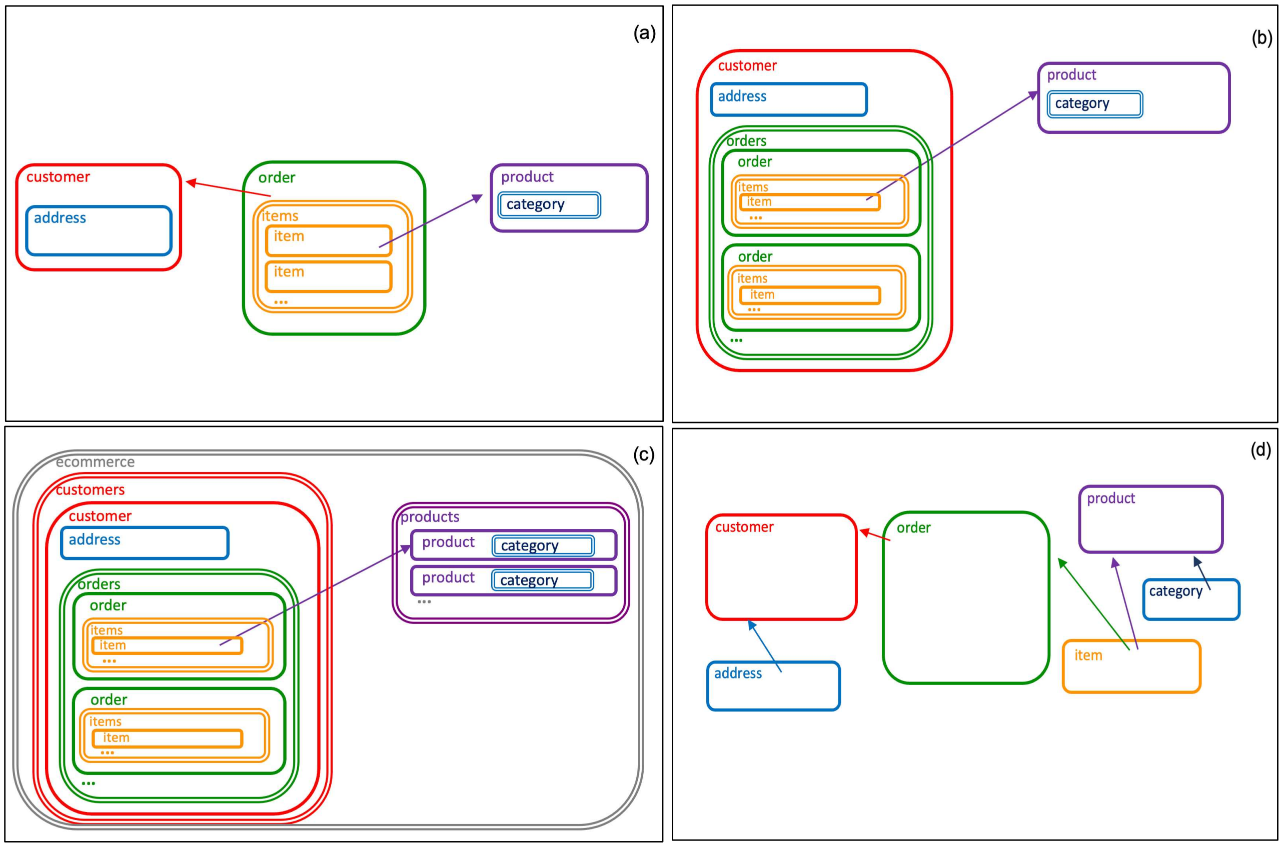


Figure 4: Some Alternative Nested Database Designs.

Table 3: ER-Driven Document Database Design

ER Concept	Document Artifact
Entity	Collection with entity's attributes and PK
Relationship (M:N)	Collection with relationship's attributes and FKs (linking)
Relationship (1:N)	Merge relationship collection into N-side's entity collection
Composite attribute	Use nested object (embedding)
Weak entity	Use nested object (embedding)
Multivalued attribute	Use nested array (embedding)
Inheritance hierarchy	Use entity collection with subtype indicator field(s)

We return now to our main path, the sharing of our ER-driven data-first document database design methodology for full-featured document database systems. We have seen that one of the central design decisions, hinted at in Figure 3, is the question of when and why to use linking versus embedding to tie data together – in other words, how to determine the aggregate objects for an application. Figure 4 sketches some of the different choices that one might make, at least in principle. Figure 4(a) shows the choice that would lead to the three collections of Figure 3. Figure 4(b) shows a more deeply nested alternative where each customer's orders are nested inside the customer object. Figure 4(c) shows a truly extreme nested alternative where the whole database is a *single* JSON object; obviously undesirable, yet not impossible. Last but not least, Figure 4(d) shows an alternative where the document schema is flat and similar to a BCNF relational database design. So which of these might be characterized as a "good" design, and why, and how might one methodically arrive at that design?

It is important to point out that the choice between targeting a relational or document database for a given application should not affect its conceptual data model. Thus, the decisions that identify the entities, relationships, and attributes for a document database application can follow the usual textbook approaches to ER design [28, 33]. The pertinent question for ER-driven document database design, then, is how to take an ER model as input and produce a good document database schema. Table 3 summarizes what we believe is a principled, practical, and easily mastered approach to creating JSON database designs from ER models. This approach has been tested with success over a period of over five years in a university setting. It has been used in a UC Irvine undergraduate database course, *COMPSCI 122A: Introduction to Data Management*, to teach several thousand students – in 1-2 weeks – how to design JSON schemas for Apache AsterixDB after learning the principles of ER modeling and relational database design and SQL. It has also been used by hundreds of students in a follow-on elective, *COMPSCI 122D: Beyond SQL Data Management*, to design and deploy databases for MongoDB, Couchbase Server, and Couchbase Analytics starting from an ER model.

Taking a closer look at the translation approach laid out in Table 3, we initially see similarities with the relational mappings of Table 2. Entity sets here become collections of JSON objects whose fields correspond to the entity's attributes. M:N relationships, again

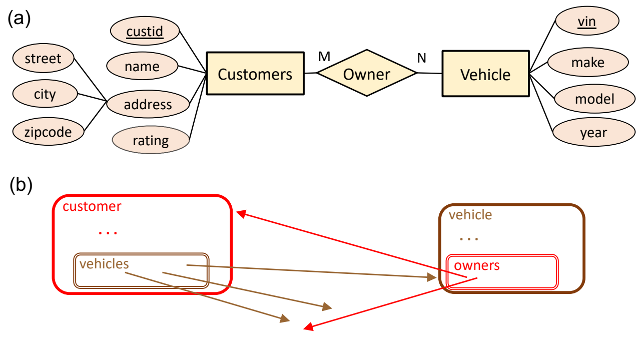


Figure 5: A Dual Array Option for Small M:N Relationships.

very similar to the relational case, become collections of JSON objects that use linking to model the entity-to-entity relationships.[5] And also as in the relational case, the relationship information (link and relationship attributes) for a 1:N relationship can optionally be placed in the N-side's entity collection to minimize the number of collections. Where the document versus relational differences emerge is when mapping the ER model's extended and more advanced features. Since JSON allows nesting, composite attributes in the ER model can be mapped to nested objects in JSON. Similarly, since JSON supports arrays, multivalued attributes (whether scalars or objects) can be mapped to array-valued fields in JSON. Weak entities are semantically similar to multivalued composite attributes and can be similarly mapped in JSON. Last but not least, since JSON schemas are flexible regarding which fields are present or absent, the objects in an inheritance hierarchy can be kept in a single collection whose objects are variations on the entity's theme (along with a field or fields that serve to indicate which subtype(s) a given object belongs to).

To illustrate the results of these ER to JSON feature mapping guidelines, it turns out (not coincidentally) that Figure 3 from earlier turns out to show the sample document structures that Table 3's mappings produce when applied to the running commerce ER model example from Figure 2. Some details to notice are that *address* is a nested, object-valued field of customers, *items* is a nested array of objects in orders, and *category* is a nested array

[5]One possible modeling exception for the M:N case would be to use nested arrays of links on each side to "cut out the middleman" if the arrays will be small on both sides. Figure 5 illustrates this option, showing how a relationship between Customers and Vehicles could utilize it if the commerce ER model were extended to keep track of customers' vehicles.

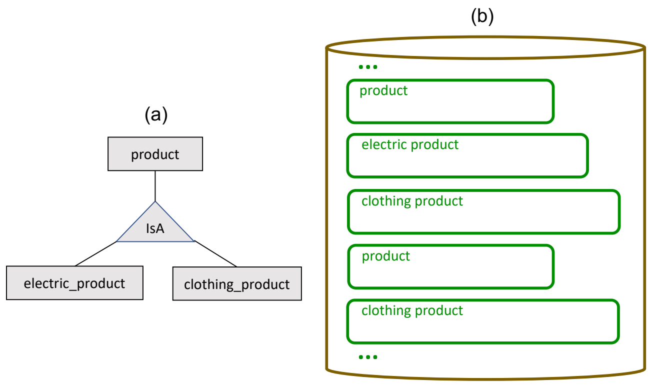


Figure 6: Modeling an IsA Relationship.

```
...
{  "itemno": 375,
   "category": ["music"],
   "name": "Stratuscaster Guitar",
   "manuf": "Fender Bender",
   "listprice": 149.99
},
{  "itemno": 780,
   "category": ["essentials", "kitchen"],
   "name": "Automatic Beer Opener",
   "descrip": "Robotic beer bottle opener",
   "manuf": "Robo Brew",
   "listprice": 29.95,
   "kind": "electric",
   "power": "AA batteries"
},
{  "itemno": 460,
   "category": ["music", "fashion"],
   "name": "Fender Bender T-Shirt",
   "descrip": "Extra Large T-Shirt for Fender Bender fans",
   "manuf": "Fender Bender",
   "listprice": 34.99,
   "kind": "clothing",
   "size": "XL"
},
...
```

Figure 7: JSON Objects from an IsA Hierarchy.

of strings in products. Figure 6 zooms in on the handling of *IsA* hierarchies, with Figure 6(a) showing the entity type hierarchy for product objects and Figure 6(b) showing them neatly co-existing in a single products collection. Figure 7 shows a sample JSON object of each type with a *kind* field[6] indicating their particular subtype.

3.4 Other Design Considerations

The ER to JSON schema mapping methodology in the preceding section targets document database systems that are fully query-capable, like MongoDB, Couchbase, and Apache AsterixDB. MongoDB's MQL query language (with its *find* and *aggregate* interfaces) provides support for aggregation, joins, and subqueries, and queries in Couchbase and Apache AsterixDB are based on SQL++ [18], a JSON-oriented generalization of SQL. Decisions about data partitioning and indexing are treatable in those systems as physical layer decisions that can be made after the logical design step is finished

[6]If a single product could belong to multiple subtypes, i.e., if the *IsA* relationship was not disjoint, then a pair of boolean fields *is_clothing* and *is_electric* would have been needed as a subtype indicator instead.

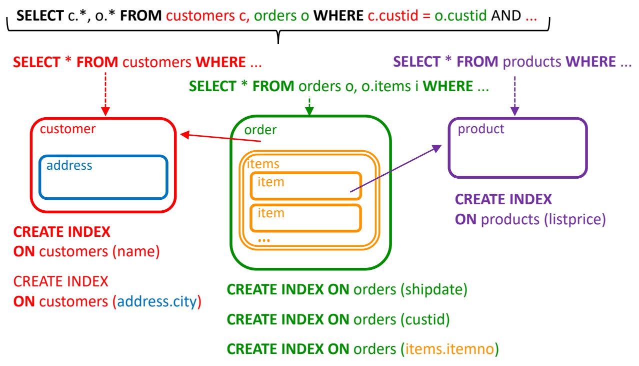

SELECT c.*, o.* FROM customers c, orders o WHERE c.custid = o.custid AND ...
SELECT * FROM customers WHERE ...
SELECT * FROM orders o, o.items i WHERE ...
SELECT * FROM products WHERE ...
customer
address
order
items
item
item
...
product
CREATE INDEX ON customers (name)
CREATE INDEX ON customers (address.city)
CREATE INDEX ON orders (shipdate)
CREATE INDEX ON orders (custid)
CREATE INDEX ON orders (items.itemno)
CREATE INDEX ON products (listprice)

Figure 8: Query Entry Points and Indexing Considerations.

and that can be modified later, as needed, without affecting the application's query statements. As a result, those systems provide physical data independence similar to the relational model.

Figure 8 sketches how the document mapping guidelines, based on following the ER model's choices about entity sets, will provide target applications with a convenient set of entry points for querying purposes. Queries that are just focused on customers can target just that one collection, and likewise for queries that focus on just orders or products. Queries that need to combine information from several of the resulting collections, e.g., to ask about customers and their orders, can use joins or subqueries to follow the inter-collection links. To accelerate an application's queries, indexes can be created at the physical level, and more indexes can be added as the application's lookup needs evolve.

We would be remiss if we totally dismissed physical considerations (e.g., I/O) as being thought-worthy in the logical design phase. For example, Figure 9 shows how nesting choices can affect the read and/or write performance of an application. Figure 9(a) shows the amount of data that might be read when querying orders in a design with nested items, while Figure 9(b) shows how having orders and items as separate JSON objects in separate collections could save I/O for an application in which most orders queries don't require data about line items. Updates to orders could involve a similar I/O consideration, as most systems don't write partial JSON objects. These considerations are not unique to JSON; similar tradeoffs exist even in the relational world and sometimes lead designers to vertically partition a table into multiple parallel tables. It should also be mentioned that this I/O consideration may be less of a factor for document database systems that employ a columnar JSON storage format under the hood [2, 15].

Lastly, one other factor worth considering towards the end of the JSON document design process is an application's possible need for fast access to information whose inclusion at a certain place in the schema would lead to a degree of redundancy (i.e., to a "one fact, one place" rule violation, likely violating 2NF or 3NF in relational theory parlance). Again, this consideration is not unique to JSON; e.g., Chapter 20 of [28] has a similar discussion related to relational database design tuning. There is also a related JSON discussion in [4] referred to as an "Extended Reference", and a discussion involving nesting of "recent" JSON objects in [9].

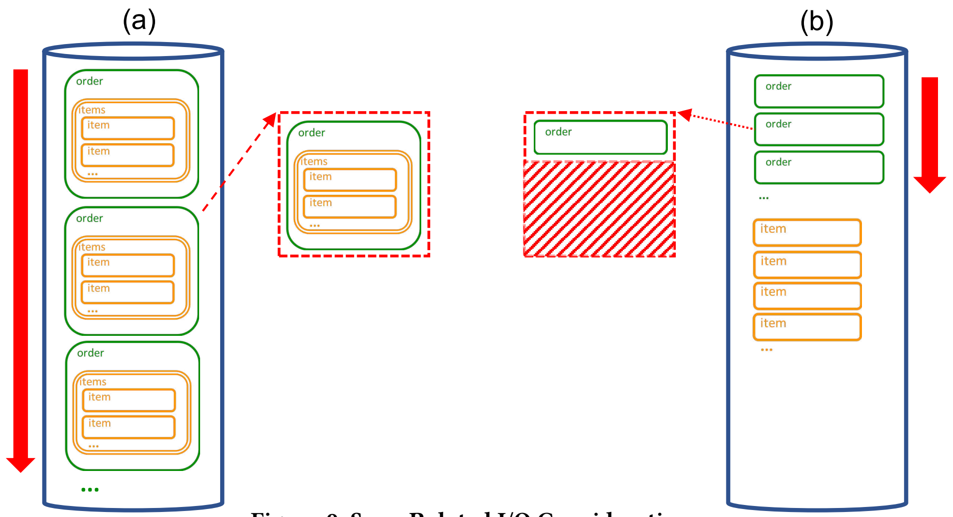


Figure 9: Scan-Related I/O Considerations.

As a concrete example, a web page to display an order in our commerce example would be more user-friendly if its list of line items included the product name of each item in addition to its *itemo*. Retrieving this information in our "good" design requires a join (albeit a small one). Product names will change rarely, while order web pages will frequently be displayed, so one could opt to add a *product_name* field to the objects in *items* and manage the resulting redundancy by propagating name changes to places like this where the *itemno-name* association is redundantly represented. To informally quantify this trade-off, we had ChatGPT help us generate a scaled-up version of our example data with 100,000 products, 500,000 customers, and 1,000,000 orders with 1 to 5 line items each. We loaded this data into the free tier of a document database service, indexed orders on *orderno* and products on *itemno*, and then ran 40 executions of a query to retrieve the line items of a randomly selected order. Without redundancy, the average server time for the query was 5.1 msec and its roundtrip time from a laptop was 69 msec. With redundancy, the times were 2.4 msec (half the previous server time) and 68 msec (roughly the same end-to-end latency).

4 PATTERNS AND ANTI-PATTERNS

The previous section offered a data-first design methodology for document databases – in short, it presented a set of do's for database designers who are targeting applications at document database systems that aren't limited to single-collection queries. This section offers a set of don'ts based on various anti-patterns that the authors have encountered over the years in real-world JSON document designs in both research and industrial settings. It is important to point out that the don'ts presented here apply not just to full-featured document database systems but also to NoSQL database systems with more limited query capabilities. For each anti-pattern – i.e., each don't – we provide an example of the anti-pattern, explain why it should be avoided, and then suggest a cure in the form of a pattern with the same (or greater) information capacity that avoids its problems. As we will see, the overall theme will be to ensure that the chosen document design is query-friendly, both from a user level perspective as well as being amenable to subsequent optimization through indexing at the physical design level.

Before presenting our anti-patterns, i.e., our list of don'ts, we should point out that our list is undoubtedly incomplete. For example, we would also encourage interested readers to check out the set of patterns (both do's and don'ts) that MongoDB has assembled to guide their JSON schema designers [4]. While less top-down and less prescriptive than our ER-based approach, having more of an "if your JSON data currently looks like this, you might consider doing that instead" flavor, their pattern set contains some well-explained and valuable document design suggestions.

4.1 Anti-Pattern 1: Unbounded Nesting

Figure 10 illustrates our first anti-pattern, which is to use embedding instead of linking in situations where the nested data may grow without bound. Nesting line items within orders is a sensible decision for the reasons covered earlier; a given order will likely contain a reasonable number of line items. Nesting the entire order history for a customer within a customer object, on the other hand, is a recipe for disaster. In addition to negative I/O implications, over-nesting can lead to the exhaustion of a system's internal limits on the size permitted for a single JSON object or a field thereof.

There is another, less drastic, but also inadvisable, variant of the unbounded nesting anti-pattern where the orders objects are stored in a separate collection while keeping an array of order references within each customer. We have seen this variant in practice as well, and while it postpones the eventual exhaustion problem, it is still a form of unbounded nesting and will cause customer objects and customer-order queries to become unruly over time.

The best cure for anti-pattern 1 is simply to follow the design guidelines of Table 3 in the previous section.[7] In this example the

[7]We caution against attempting to work around this anti-pattern by "chunking" a nested array into large blocks that, when exceeded, lead to the addition of another copy of the upper-level object to hold the next array chunk. Unbounded nested arrays should be treated as a call to use linking rather than embedding. However, on a tangentially related note, MongoDB's "Bucket Pattern" [4] is an interesting idea for making intentional use of nesting and chunking in use cases like time series data.

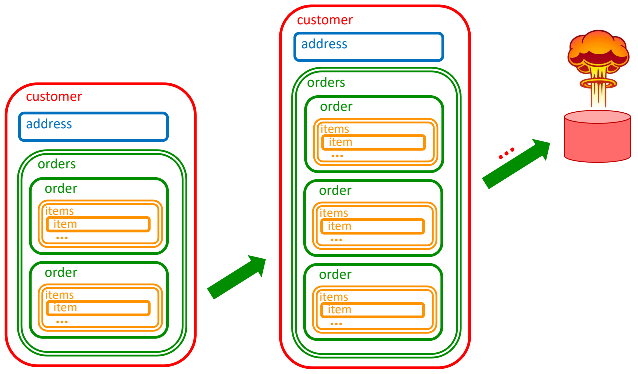
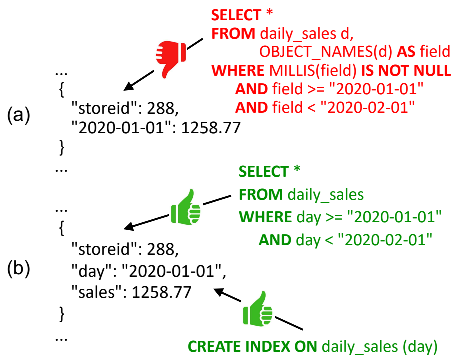


Figure 10: Over-Nesting Doesn't End Well.



```
SELECT *
FROM daily_sales d,
        OBJECT_NAMES(d) AS field
WHERE MILLIS(field) IS NOT NULL
    AND field >= "2020-01-01"
    AND field < "2020-02-01"
```

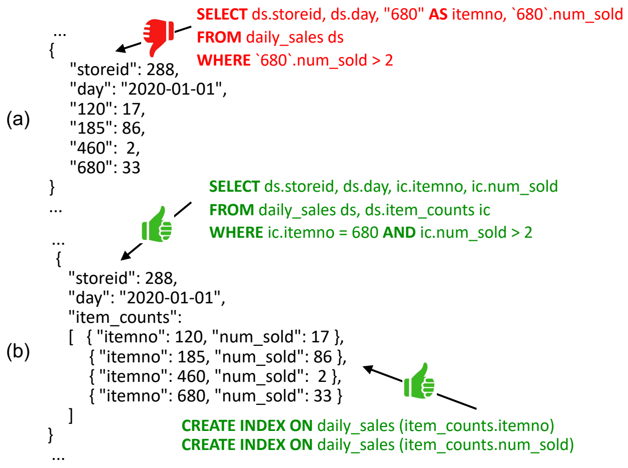
```
(a)
 ...
 {
   "storeid": 288,
   "2020-01-01": 1258.77
 }
 ...
```

```
SELECT *
FROM daily_sales
WHERE day >= "2020-01-01"
    AND day < "2020-02-01"
```

```
(b)
 ...
 {
   "storeid": 288,
   "day": "2020-01-01",
   "sales": 1258.77
 }
 ...
```

```
CREATE INDEX ON daily_sales (day)
```

Figure 11: Don't Confuse Metadata and Data.



```
SELECT ds.storeid, ds.day, "680" AS itemno, `680`.num_sold
FROM daily_sales ds
WHERE `680`.num_sold > 2
```

```
(a)
 ...
 {
   "storeid": 288,
   "day": "2020-01-01",
   "120": 17,
   "185": 86,
   "460":  2,
   "680": 33
 }
 ...
```

```
SELECT ds.storeid, ds.day, ic.itemno, ic.num_sold
FROM daily_sales ds, ds.item_counts ic
WHERE ic.itemno = 680 AND ic.num_sold > 2
```

```
(b)
 ...
 {
   "storeid": 288,
   "day": "2020-01-01",
   "item_counts":
   [ { "itemno": 120, "num_sold": 17 },
     { "itemno": 185, "num_sold": 86 },
     { "itemno": 460, "num_sold":  2 },
     { "itemno": 680, "num_sold": 33 }
   ]
 }
 ...
```

```
CREATE INDEX ON daily_sales (item_counts.itemno)
CREATE INDEX ON daily_sales (item_counts.num_sold)
```

Figure 12: Don't Confuse Fields with Arrays.

result of following the guidelines will be as shown in Figure 3, where customers and orders are stored separately and each order is linked back to its associated customer.

4.2 Anti-Pattern 2: Data as Metadata

Figure 11 shows the next anti-pattern, which might be described from a database point of view as confusing metadata and data. In the relational world (of SQL tables), the system catalogs store the metadata (field names and types) and the rows of a table store its fields' values; their query languages (usually SQL) are oriented in this way. Query languages for document databases have their roots in relational languages, albeit generalized to work against JSON data. Figure 11 is an example of a collection that captures the daily sales history from a company's stores. In Figure 11(a) the designer has structured the data to have a field named after the date and whose value holds the sales volume for that date. Notice how this decision makes it very awkward to write date range queries like the one in the figure if the application needs to do so, either now or in the future – and note that this design is not amenable to the creation of a secondary index to speed up such queries.

The cure for anti-pattern 2, as shown in Figure 11(b), is to use a more "databasey" design that avoids using data values as field names – i.e., to separate what would be schema from what would be data even though JSON databases are schemaless and make the anti-pattern possible. (It is worth mentioning that JSON exhibiting this anti-pattern sometimes originates from map-oriented Javascript libraries and their associated persistent stores [25].)

We have also encountered a more complex, nested variant of anti-pattern 2 in practice (which we can probably thank Javascript for as well). An example based on monthly sales data for stores might have objects of the following form:

```
{"storeid": 288,
 "data": {"2020": {"Q1": {"Jan": {"sales": 14999.99}}}}}
```

Applying the cure, based on "schema thinking", leads to the following much more query- and index-friendly alternative that doesn't confuse data and metadata:

```
{"storeid": 288,
 "year": 2020,
 "quarter": "Q1".
 "month": "Jan",
 "sales": 14999.99}
```

4.3 Anti-Pattern 3: Arrays as Fields

The third anti-pattern is shown in Figure 12. This is a cousin of anti-pattern 2, but for arrays. In this example, the goal is to track, for each store on each day, the number of each product that was sold there on that day. In Figure 12(a) the designer has used the products' ids (*itemno*) as field names and their associated counts as field values. As indicated in the figure, this design is query-unfriendly; it is awkward to search for sales records based on combinations of their product ids and associated counts. As shown, such a query is messy to formulate against the anti-pattern. Other queries could be even worse, like a range query to identify the stores and days where more than 100 units of any product were sold and then return the ids of such products. Another problem with this anti-pattern is that it can cause objects to have many, and highly variant, sets of fields. (This could be especially problematic when columnar JSON storage formats are in use internally).

The cure for anti-pattern 3, as shown in Figure 12(b), is to avoid confusing data with schemas and/or confusing fields with arrays.

```
(a)  ... { "itemno": 347,
        "name": "Beer Cooler Backpack",
        "colors": "black",
        "listprice": 29.95 },
      { "itemno": 375,
        "name": "Stratuscaster Guitar",
        "colors": ["sunburst", "black", "cherry"],
        "listprice": 1499.99 }, ...
                     vs.
(b)  ... { "itemno": 347,
        "name": "Beer Cooler Backpack",
        "colors": ["black"],
        "listprice": 29.95 },
      { "itemno": 375,
        "name": "Stratuscaster Guitar",
        "colors": ["sunburst", "black", "cherry"],
        "listprice": 1499.99 }, ...
```

Figure 13: Avoid Array Heterogeneity.

Again, the original design is problematic for querying and would make effective indexing impossible.

4.4 Anti-Pattern 4: Non-Uniform Nesting

The fourth anti-pattern, shown in Figure 13(a), might best be described as non-uniform nesting. In the example, some products are available in a set of colors, while others have only one color option. Here a JSON document designer has chosen to exploit JSON's schema flexibility – specifically, its willingness to support heterogeneous field types and values. If there is only one color choice, *color* is a string, while its value is an array of strings if the product has multiple color options. This is again a query-unfriendly design (though not as unfriendly as the previous examples), as queries may need to test the field's type (string versus array) when they are searching for products that are available in a particular color. Indexing such non-uniform data may also turn out to be problematic in some systems.

The cure for this anti-pattern is to uniformly model a potentially multivalued field as an array, as shown in Figure 13(b), even if the array will have just one entry in some of the records. (It is worth mentioning that JSON of this kind sometimes originates from data conversion tools, such as tools that convert XML data into JSON.)

4.5 Anti-Pattern 5: Scalar Heterogeneity

The fifth anti-pattern, which is shown in Figure 14(a), is having a non-uniform representation of a scalar-valued field. Here, the illustrated use case involves dealing with monetary values expressed in different currencies. If no currency is specified here, the presumption is US dollars, and the records have a mix of string and numeric values for the price. If the currency is different, then an object representation is used. JSON is fine with such heterogeneity, of course, due to its flexibility, but exploiting that flexibility in this way makes both querying and indexing messier, requiring both to deal with the defaults as well as the more general case.

Not unlike the cure for the previous anti-pattern, the recommended cure for this anti-pattern would be to always model such a potentially variant field's value as an object and to use a numeric type for the price to support meaningful comparisons[8], as shown in Figure 14(b). Another specific variant that we have seen of this anti-pattern in the wild involves using different date formats (e.g., Unix timestamp, date string, datetime string) either across or within the fields of objects in a given collection.

```
(a)   ... { "itemno": 185,
         "category": ["black ", "silver"],
         "name": "Stapler",
         "listprice": 21.99 },
       { "itemno": 347,
         "name": "Beer Cooler Backpack",
         "colors": ["black"],
         "listprice": "29.95" },
       { "itemno": 375,
         "name": "Stratuscaster Guitar",
         "colors": ["sunburst", "black", "cherry"],
         "listprice": {"price": 227436.62, "currency": "Yen"} }, ...
                              vs.
(b)   ... { "itemno": 185,
         "category": ["black ", "silver"],
         "name": "Stapler",
         "listprice": {"price": 21.99, "currency": ”USD"} },
       { "itemno": 347,
         "name": "Beer Cooler Backpack",
         "colors": ["black"],
         "listprice": {"price": 29.95, "currency": ”USD"} },
       { "itemno": 375,
         "name": "Stratuscaster Guitar",
         "colors": ["sunburst", "black", "cherry"],
         "listprice": {"price": 227436.62, "currency": "Yen"} }, ...
```

Figure 14: Avoid Scalar Heterogeneity.

4.6 Anti-Pattern 6: A Mix of Anti-Patterns

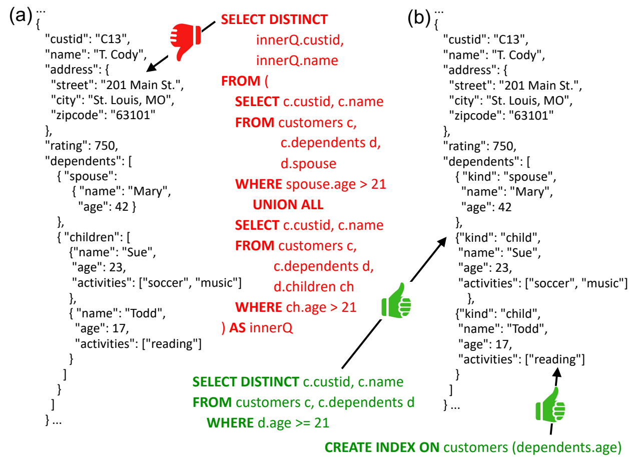
The sixth anti-pattern, shown in Figure 15, shows how a given design can suffer from multiple query-related challenges. Here we see an example where the desire is to store some additional information about customers, in this case adding a field containing information about their dependents (perhaps for insurance or marketing use). In Figure 15(a) the information about a customer's spouse is stored separately, as a nested object-valued field, with the information about children being stored as a nested array of objects. This design suffers from non-uniformity, making it both query- and index-unfriendly if the application needs to ask dependents-related questions such as the example shown in the figure.

The cure for this anti-pattern, as shown in Figure 15(b), is to adopt a more uniform approach to modeling the desired information. In the cure, all dependents are now modeled as objects in an array of dependents, which is fine despite the fact that more information (*activities*) is being maintained for the dependents that are children. (It is interesting to note that the design shown in this anti-pattern is a document schema that came from someone with a Ph.D. from a top 5 school and decades of SQL experience.)

4.7 Anti-Pattern 7: Failure to Embrace Diversity

The poet Maya Angelou said: "We all should know that diversity makes for a rich tapestry, and we must understand that all the threads of the tapestry are equal in value no matter what their

[8]It would also likely be desirable to use a user-defined function in queries to convert such prices into some chosen common currency, presumably based on the current exchange rate at query time.



```
...
{
  "custid": "C13",
  "name": "T. Cody",
  "address": {
   "street": "201 Main St.",
   "city": "St. Louis, MO",
   "zipcode": "63101"
  },
  "rating": 750,
  "dependents": [
    { "spouse":
        { "name": "Mary",
          "age": 42 }
    },
    { "children": [
      {"name": "Sue",
       "age": 23,
       "activities": ["soccer", "music"]
      },
      { "name": "Todd",
        "age": 17,
        "activities": ["reading"]
      }
     ]
    }
  ]
} ...
```

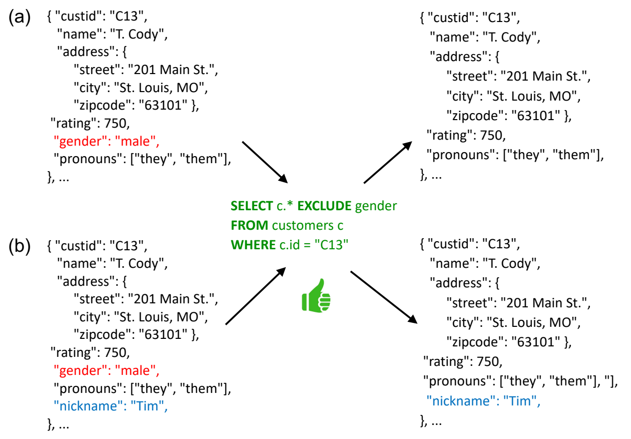
```
SELECT DISTINCT
      innerQ.custid,
      innerQ.name
FROM (
   SELECT c.custid, c.name
   FROM customers c,
          c.dependents d,
          d.spouse
   WHERE spouse.age > 21
      UNION ALL
   SELECT c.custid, c.name
   FROM customers c,
         c.dependents d,
         d.children ch
   WHERE ch.age > 21
) AS innerQ
```

```
SELECT DISTINCT c.custid, c.name
FROM customers c, c.dependents d
   WHERE d.age >= 21
```

```
...
{
  "custid": "C13",
  "name": "T. Cody",
  "address": {
   "street": "201 Main St.",
   "city": "St. Louis, MO",
   "zipcode": "63101"
  },
  "rating": 750,
  "dependents": [
    { "kind": "spouse",
      "name": "Mary",
      "age": 42
    },
    {"kind": "child",
     "name": "Sue",
     "age": 23,
     "activities": ["soccer", "music"]
       },
    {"kind": "child",
     "name": "Todd",
     "age": 17,
     "activities": ["reading"]
    }
  ]
} ...
```

```
CREATE INDEX ON customers (dependents.age)
```

Figure 15: A Mix of Design Problems.



```
{ "custid": "C13",
  "name": "T. Cody",
  "address": {
     "street": "201 Main St.",
     "city": "St. Louis, MO",
     "zipcode": "63101" },
  "rating": 750,
  "gender": "male",
  "pronouns": ["they", "them"],
}, ...
```

```
{ "custid": "C13",
  "name": "T. Cody",
  "address": {
     "street": "201 Main St.",
     "city": "St. Louis, MO",
     "zipcode": "63101" },
  "rating": 750,
  "pronouns": ["they", "them"],
}, ...
```

```
SELECT c.* EXCLUDE gender
FROM customers c
WHERE c.id = "C13"
```

```
{ "custid": "C13",
  "name": "T. Cody",
  "address": {
     "street": "201 Main St.",
     "city": "St. Louis, MO",
     "zipcode": "63101" },
  "rating": 750,
  "gender": "male",
  "pronouns": ["they", "them"],
  "nickname": "Tim",
}, ...
```

```
{ "custid": "C13",
  "name": "T. Cody",
  "address": {
     "street": "201 Main St.",
     "city": "St. Louis, MO",
     "zipcode": "63101" },
  "rating": 750,
  "pronouns": ["they", "them"], "],
  "nickname": "Tim",
}, ...
```

Figure 16: Benefiting from Diversity.

color." Having covered a series of anti-patterns that suffer partly from non-uniformity issues, our final don't is actually a do stated in double negative form: *Don't not* reap the benefits of diversity! It may seem, based on our ER-based data-first design methodology and on our recommendations about steering clear of various forms of heterogeneity, that we are arguing to move the document database world back five decades to a more rigid (albeit nested) world. That is not the case. A major benefit of document (JSON) database systems is their ability to accommodate the kinds of diversity that modern applications need in order to manage and fluidly evolve their underlying data.

Figure 16 shows an example of how document databases and their query facilities can be exploited to enable application fluidity. Figure 16(a) shows an example of customer data that has been extended with a private field, *gender*, which should be hidden when a customer service web page retrieves a customer's data. As the application evolves, new fields may be added, as shown in Figure 16(b), where a *nickname* field is now also being maintained. The query in the middle of the figure shows how an application can be prepared for the addition of new fields without having to be changed when they appear. In particular, the query excludes personal information while retrieving any/all other fields that the retrieved object happens to have at the time. (The particular syntax shown is for SQL++, but MongoDB's MQL APIs offer a similar exclusionary option.) Assuming that the application renders customer objects based on inspecting the returned content, the customer service web page will now show customer service agents the customer's nickname as well. In contrast, adding a new field in a relational version of the application would require updating the customer table's schema (via ALTER TABLE) and probably updating the calling web page's query code as well, making the application far less fluid in nature.

5 CONCLUSION

Relational database design is a well-understood process today, and it is almost always approached in a data-first manner owing to the data independence provided by the relational model. In contrast, the art of database design for NoSQL databases, including document databases, is in a much less mature state at present. NoSQL database design is often approached in an ad hoc manner that mixes logical and physical considerations in ways that are all too reminiscent of the pre-relational database era.

In this paper, we have made the case for a return to sanity by presenting a logical, data-first, conceptually driven approach to document database design. We have explained how such an approach can start from an ER model and achieve a clean, query-friendly document database design even though it is not necessary to declare the resulting schema to the DBMS. We have also highlighted a set of document (JSON) anti-patterns that should be avoided in order to facilitate document queries and enable performance tuning via secondary indexing.

Both our design methodology and anti-patterns stem from the authors' experiences, in current and past lives, with a wide variety of JSON document data in commercial applications, government applications, and university research applications. We have found the proposed approach to be effective given the state of today's document database systems, as a number of these systems are now sufficiently rich in their query processing and indexing capabilities to be able to support data independence in the JSON data world. Our hope is that others will find these lessons and the resulting do's and don'ts helpful in their work as well.

ACKNOWLEDGMENTS

This work was supported in part by NSF award CNS-1925610, an industrial gift from Google, and funding from the Donald Bren Foundation (via a Bren Chair at UC Irvine).

REFERENCES

[1] Amazon Web Services 2024. *Amazon DynamoDB.* Amazon Web Services. Retrieved November 26, 2024 from https://aws.amazon.com/dynamodb/
[2] Couchbase, Inc. 2025. *About Capella Columnar.* Couchbase, Inc. Retrieved February 2, 2025 from https://docs.couchbase.com/columnar/intro/intro.html
[3] Apache Software Foundation 2025. *Apache AsterixDB.* Apache Software Foundation. Retrieved January 25, 2025 from https://asterixdb.apache.org/index.html
[4] MongoDB, Inc. 2025. *Building With Patterns: A Summary.* MongoDB, Inc. Retrieved February 9, 2025 from https://www.mongodb.com/blog/post/building-with-patterns-a-summary
[5] Apache Software Foundation 2025. *Cassandra Basics.* Apache Software Foundation. Retrieved January 25, 2025 from https://cassandra.apache.org/_/cassandra-basics.html

[6] Apache Software Foundation 2025. *Cassandra Data Modeling (Introduction)*. Apache Software Foundation. Retrieved January 30, 2025 from https://cassandra.apache.org/doc/stable/cassandra/data_modeling/intro.html
[7] Couchbase, Inc. 2025. *Couchbase Server*. Couchbase, Inc. Retrieved January 25, 2025 from https://www.couchbase.com/products/server/
[8] MongoDB, Inc. 2025. *Data Modeling*. MongoDB, Inc. Retrieved February 1, 2025 from https://www.mongodb.com/docs/manual/data-modeling/
[9] MongoDB, Inc. 2025. *Data Modeling*. MongoDB, Inc. Retrieved February 9, 2025 from https://www.mongodb.com/docs/manual/data-modeling/
[10] 2025. *Introducing JSON*. Retrieved January 25, 2025 from https://www.json.org/json-en.html
[11] MongoDB, Inc. 2025. *Introduction to MongoDB*. MongoDB, Inc. Retrieved January 25, 2025 from https://www.mongodb.com/docs/manual/introduction/#introduction-to-mongodb
[12] Microsoft, Inc. 2025. *Modeling Data in Azure Cosmos DB*. Microsoft, Inc. Retrieved February 1, 2025 from https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/modeling-data
[13] MongoDB, Inc. 2025. *MongoDB Evolved – Version History*. MongoDB, Inc. Retrieved February 1, 2025 from https://www.mongodb.com/resources/products/mongodb-version-history
[14] Microsoft, Inc. 2025. *Queries in Azure Cosmos DB for NoSQL*. Microsoft, Inc. Retrieved January 25, 2025 from https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/query/
[15] Wail Y. Alkowaileet and Michael J. Carey. 2022. Columnar Formats for Schemaless LSM-based Document Stores. *Proc. VLDB Endow.* 15, 10 (2022), 2085–2097. https://doi.org/10.14778/3547305.3547314
[16] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. 2020. Data Modeling in the NoSQL World. *Computer Standards & Interfaces* 67 (2020), 103149. https://doi.org/10.1016/j.csi.2016.10.003
[17] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have Your Data and Query It Too: From Key-Value Caching to Big Data Management. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 239–251. https://doi.org/10.1145/2882903.2904443
[18] Michael Carey, Don Chamberlin, Almann Goo, Kian Win Ong, Yannis Papakonstantinou, Chris Suver, Sitaram Vemulapalli, and Till Westmann. 2024. SQL++: We Can Finally Relax!. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5501–5510. https://doi.org/10.1109/ICDE60146.2024.00438
[19] M. J. Carey. 2019. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/ICDE.2019.00008
[20] Rick Cattell. 2011. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* 39, 4 (May 2011), 12–27. https://doi.org/10.1145/1978915.1978919
[21] Don Chamberlin. 2018. *SQL++ For SQL Users: A Tutorial*. Couchbase, Inc. https://www.couchbase.com/content/analytics/sql-book
[22] Donald Chamberlin. 2024. 50 Years of Queries. *Commun. ACM* 67, 8 (Aug. 2024), 110–121. https://doi.org/10.1145/3649887
[23] Peter Pin-Shan Chen. 1976. The Entity-Relationship Model—toward a Unified View of Data. *ACM Trans. Database Syst.* 1, 1 (mar 1976), 9–36. https://doi.org/10.1145/320434.320440
[24] Amol Deshpande. 2024. Beyond Relations: A Case for Elevating to the Entity-Relationship Abstraction [Unpublished Manuscript]. (2024).
[25] Firebase Documentation. 2025. *Structure Your Database*. Google, Inc. Retrieved February 9, 2025 from https://firebase.google.com/docs/database/web/structure-data
[26] Murtadha Al Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. 2019. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2275–2286. https://doi.org/10.14778/3352063.3352143
[27] Z. Meral Ozsoyoglu and Li-Yan Yuan. 1987. A New Normal Form for Nested Relations. *ACM Trans. Database Syst.* 12, 1 (mar 1987), 111–136. https://doi.org/10.1145/12047.13676
[28] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., USA.
[29] Mark A. Roth and Henry F. Korth. 1987. The Design of ¬ 1NF Relational Databases into Nested Normal Form. *SIGMOD Rec.* 16, 3 (dec 1987), 143–159. https://doi.org/10.1145/38714.38733
[30] Mark A. Roth, Herry F. Korth, and Abraham Silberschatz. 1988. Extended Algebra and Calculus for Nested Relational Databases. *ACM Trans. Database Syst.* 13, 4 (oct 1988), 389–417. https://doi.org/10.1145/49346.49347
[31] Pramod Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistnce*. Addison-Wesley, USA.
[32] H.J. Schek and M.H. Scholl. 1986. The Relational Model with Relation-Valued Attributes. *Inf. Syst.* 11, 2 (apr 1986), 137–147. https://doi.org/10.1016/0306-4379(86)90003-7
[33] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. https://www.db-book.com/
[34] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Rec.* 53, 2 (July 2024), 21–37. https://doi.org/10.1145/3685980.3685984
[35] Wikipedia contributors. 2025. Database design — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Database_design&oldid=1269437778 [Online; accessed 24-January-2025].