IDERA® | ER/Studio Whitepaper

MONGODB AND JSON DATA MODELING

BY STEVE HOBERMAN

TABLE OF CONTENTS

INTRODUCTION

Traditionally, data modeling produces a set of structures for a Relational Database Management System (RDBMS). First, we build the Conceptual Data Model (CDM) to capture the common business language for the initiative (e.g., “What’s a Customer?”). Next, we create the Logical Data Model (LDM) using the CDM’s common business language to precisely define the business requirements (e.g., “I need to see the customer’s name and address on this report.”). Finally, in the Physical Data Model (PDM), we design these business requirements specific for a particular technology such as Oracle, Teradata, or SQL Server (e.g. “Customer Last Name is a variable length not null field with a non-unique index…”). Our PDM represents the RDBMS design for an application. We then generate the Data Definition Language (DDL) from the PDM, which we can run within a RDBMS environment to create the set of tables that will store the application’s data. To summarize, we go from common business language to business requirements to design to tables.

Although the conceptual, logical, and physical data models have played a very important role in application development over the last 40 years, they will play an even more important role over the next 40 years. Regardless of the technology, data complexity, or breadth of requirements, there will always be a need for a diagram that captures the business language (conceptual), the business requirements (logical), and the design (physical).

These diagrams continue to prove useful in more and more types of projects. For example, here are four trends where I have used modeling over the last few years:

- **Greater focus on data strategy.** A good data strategy builds upon a common business language and, therefore, we might only create a conceptual and not a logical or physical. For example, I recently worked with a large financial institution to create a CDM for a global human resources initiative. All stakeholders on this project needed to speak the same business language before determining how to integrate several large Enterprise Resource Planning (ERP) applications. For example, stakeholders would need to have the same understanding of an Employee before discussing how to integrate employee information. Although using a CDM in itself and not as a stepping stone to a logical is becoming increasingly popular, this trend is outside the scope of this paper.
- **NoSQL is becoming mainstream.** NoSQL means "NoRDBMS". A NoSQL database stores data differently than a RDBMS. A RDBMS stores data in tables (sets) where primary and foreign keys drive data integrity and navigation. A NoSQL database stores data in files. Although these files can take various forms, such as Resource Description Framework (RDF) triples or Extensible Markup Language (XML), the most common form is JavaScript Object Notation (JSON). NoSQL requires both a tactical and strategic shift in how we approach modeling. The tactical shift involves needing to model types of structures non-existent in a RDBMS, such as nested arrays. The strategic shift involves approaching modeling primarily from a query instead of a set theory perspective. Although knowing the queries (requirements) helps us collect the necessary set of attributes for the application as well as steer us towards database-specific constructs such as indexes and views, in a RDBMS our primary focus is to correctly assign attributes to entities. Recall the normalization axiom, "every attribute depends upon the key, the whole key, and nothing but the key." We assign the customer attributes to Customer and the account attributes to Account based on their primary key relationships and not primarily because a user wants to see a set of attributes a certain way. In NoSQL, however, queries dictate our modeling approach. A requirement, for example, to query customer data before account data may lead to a different model than if querying account data first. NoSQL requires a different modeling approach which we will cover in this paper.
- **JSON is becoming the defacto interchange format.** We often exchange data between applications using JSON. As we move to more service-based architectures, data modelers will need to understand the requirements "between" the applications. That is, what does system A need from system B? Luckily, to model interfaces, the modeler can leverage the skill set from modeling NoSQL and specifically JSON. This trend is within the scope of this paper. In fact, the steps we will cover for modeling the document database, MongoDB, are identical for modeling JSON.
- **Hybrid databases.** We are seeing more traditional RDBMS platforms now allowing single columns to be typed as JSON. The overall structure of the database is now a hybrid of the two worlds which presents challenges to traditional data modeling.

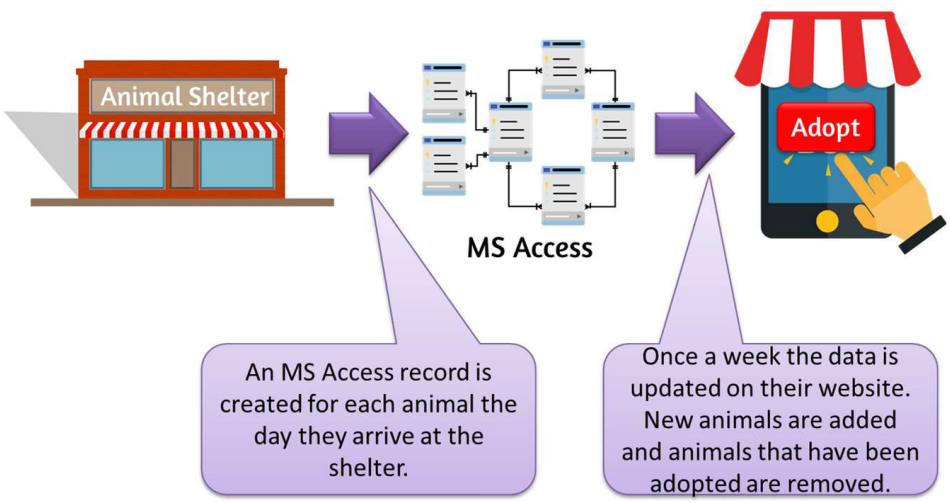
To make our conversations more meaningful, we will illustrate NoSQL modeling principles with MongoDB and interchange formats with JSON using an animal shelter case study. There are three topics that we will cover:

- **The project.** The best way to learn is by example. There is an animal shelter that needs our help. They must extract data from a traditional relational database in JSON format and load it into a MongoDB database. We will review the ER/Studio conceptual, logical, and physical relational models for the source application, and the queries that need to be answered via JSON and MongoDB.
- **The challenges.** We will cover the tactical and strategic modeling hurdles with modeling NoSQL (using MongoDB as an example) and interchange formats (using JSON as an example).
- **The solution.** We will go through the modeling approach for MongoDB and JSON. This approach will be general enough to apply to any NoSQL database or interchange format. We will apply this approach to our animal shelter project and build the conceptual, logical, and physical models for both JSON and MongoDB. We will use the ER/Studio 19.2 features to show two ways of modeling nested arrays.

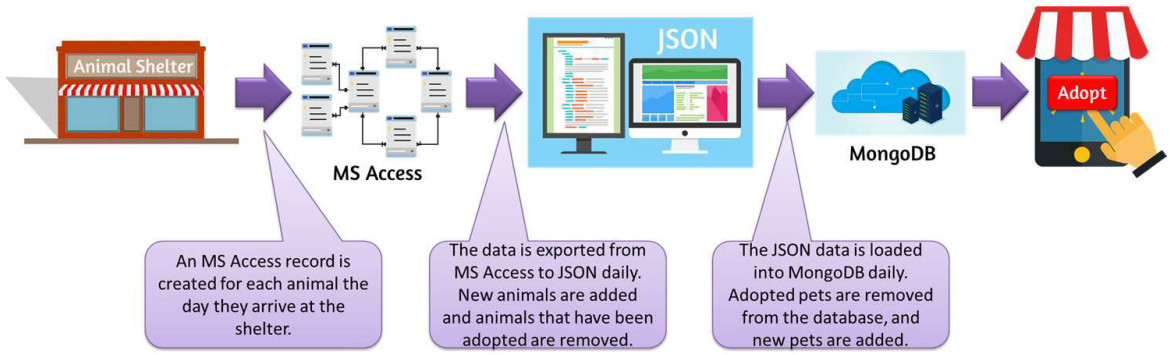
We will then conclude with a list of five key takeaways from this paper.

THE PROJECT

A local animal shelter needs your help. They currently advertise their ready-to-adopt pets on their own website. They use a MS Access relational database to keep track of their animals, and they publish this data weekly on their website. Here is their current architecture:



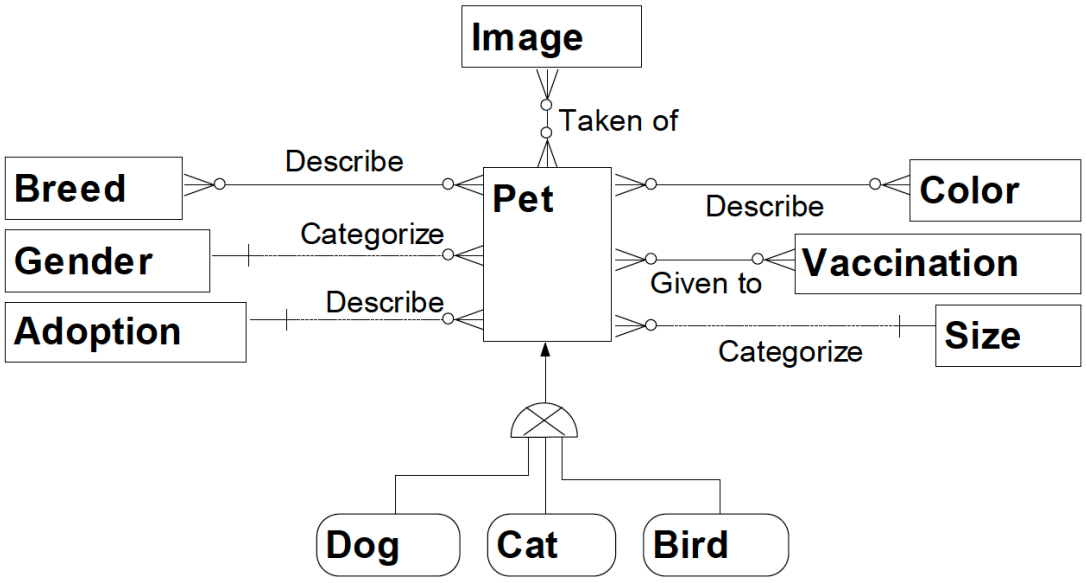
Not many people know about this shelter, and often animals remain unadopted for much longer than the national average. Therefore, they would like to partner with a group of animal shelters in a similar situation and publish their pet data together to create a much more popular site. Our shelter will need to extract data from their current MS Access database and send it to the consortium database in JSON format. The consortium will then load these JSON feeds into their MongoDB database with a web front end. The animal shelter's proposed architecture looks like this:



Let's now look at the shelter's current models.

Conceptual:

The animal shelter built the following Conceptual Data Model (CDM) to capture the common business language for the initiative (e.g., "What's a Customer?"):



There is more than one way to create a conceptual data model in ER/Studio. The animal shelter chose the approach of hiding the attributes (under “Diagram And Object Display Options”, select “Display Level” of “Entity”).

Note that each term on this model must have a precise definition. I can talk for an hour on how to write a clear, complete, and correct entity and attribute definition (and I do during my Data Modeling Master Class!). To save space, however, and also because the subject of this paper is not definitions, let’s just define one term. We’ll choose Adoption because this is the least intuitive of these terms, and could lead to some integration issues later on:

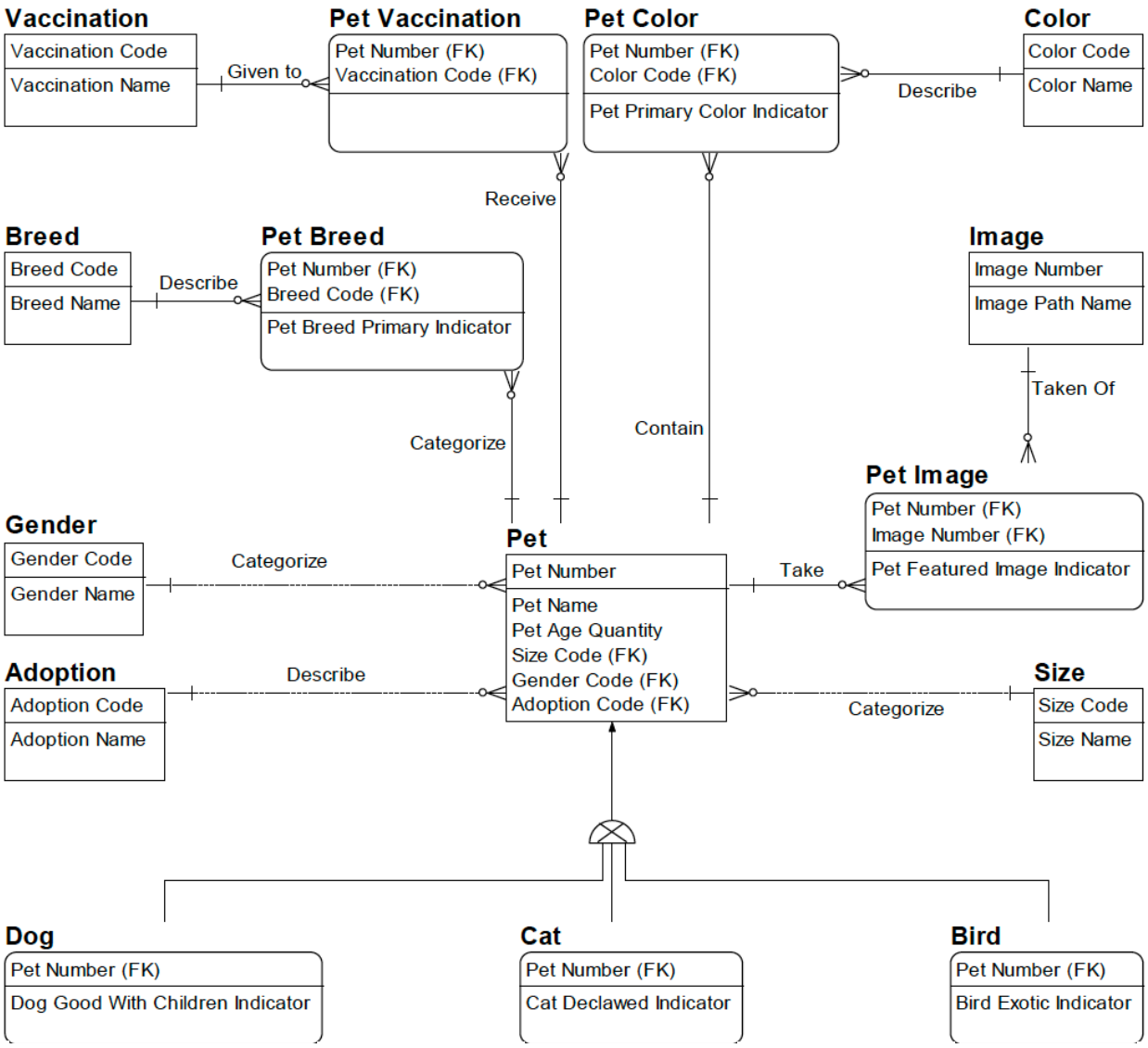
Adoption – Adoption captures the lifecycle of a pet within the context of our animal shelter. The current lifecycle stages are:

- *Received*
- *Approved for adoption by vet*
- *Approved for adoption by animal behavioral expert*
- *Posted for adoption*
- *Reserved for adoption*
- *Adopted*

Note that these are not industry-standard adoption statuses. They exist just within our animal shelter. Also, the above set of stages is currently our full set. That is, there are only six stages. We are a no-kill shelter.

Logical:

The shelter's Logical Data Model (LDM) uses the common business language from the CDM to precisely define the business requirements (e.g., "I need to see the customer's name and address on this report."). The LDM is fully-attributed, yet independent of technology. Here is the shelter's LDM:



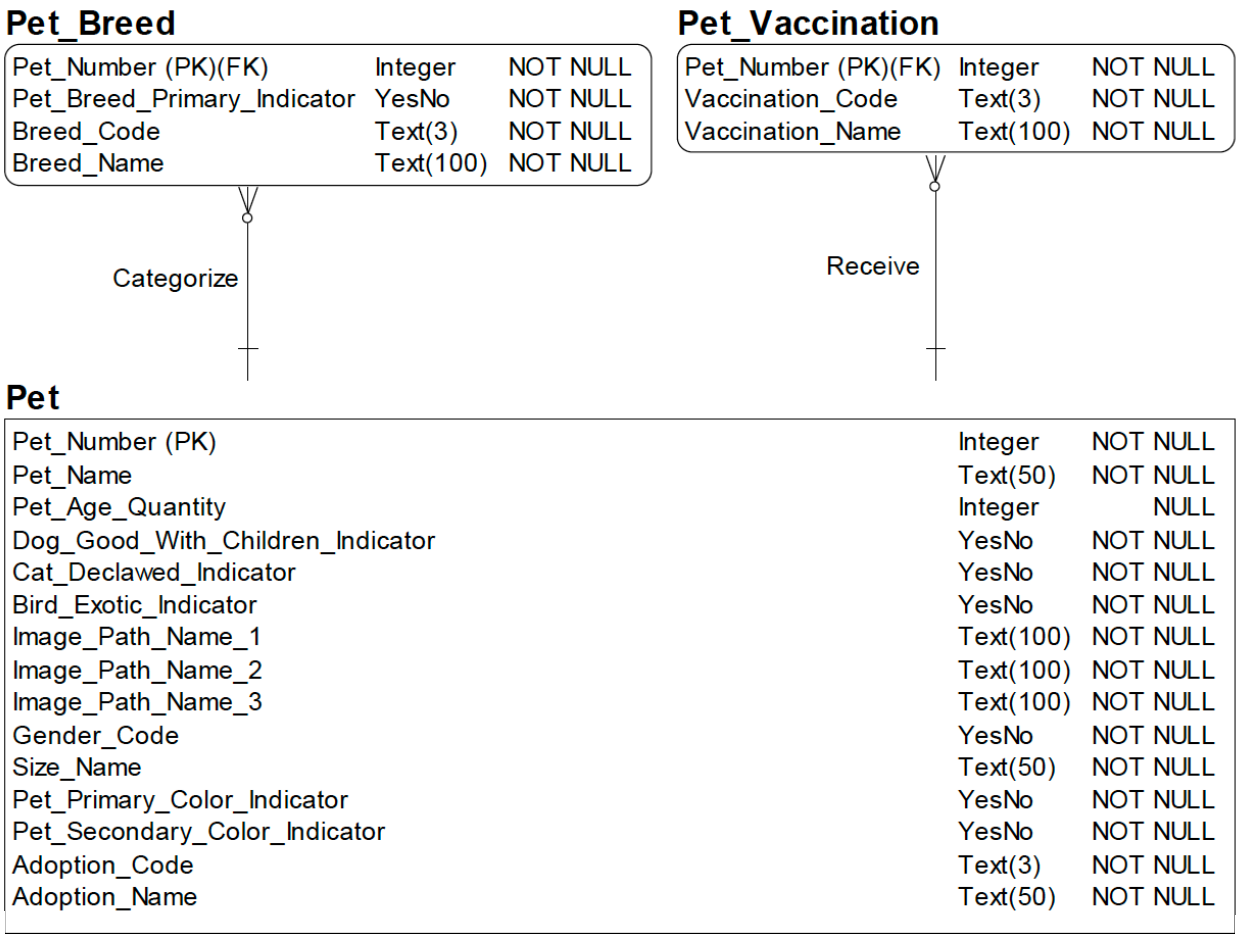
This model does not change based on query and, therefore, can be used as the starting point model for all queries. Let's briefly walk through the model. The shelter identifies each Pet with a Pet Number, which is a unique counter assigned to the Pet the day the Pet arrives. Also entered at this time is the Pet's name (Pet Name) and age (Pet Age Quantity). If the Pet does not have a name, it is given one by the shelter employee entering the Pet's information. If the age is unknown, it is estimated by the shelter employee entering the Pet's information. If the Pet is a Dog, the shelter employee entering the information performs a few assessments to determine whether the Dog is good with children (Dog Good With Children Indicator). If the Pet is a Cat, the shelter employee determines whether the Cat has been declawed (Cat Declawed Indicator). If the Pet is a Bird, the shelter employee enters whether it is an exotic bird such as a parrot (Bird Exotic Indicator).

Note that although data never lies, it might not always tell the truth either. There are opportunities for data quality issues at every stage when data is created or changed. For example, with the Pet's age, name, whether they are good with children, etc. But this is a topic for another paper...

Also, notice two patterns on the model. The first is the use of decode entities. Gender, Adoption, Size, Vaccination, Breed, and Color all provide a lookup structure with a code and decode (e.g. 'B' is for the color 'Brown' and 'L' is for the size 'Large'). The second is the many-to-many relationship pattern. A Pet can be assigned more than one Breed, be administered more than one Vaccination, contain more than one Color, and take more than one Image. The featured image for the Pet on the website is the image where Pet Featured Indicator equals 'Y'. The primary breed for the Pet is the assigned breed where Pet Breed Primary Indicator equals 'Y'. The primary color for the Pet is the assigned color where Pet Primary Color Indicator equals 'Y'.

PHYSICAL:

And here is the Physical Data Model (PDM) representing the animal shelter's Microsoft Access database design:



Note that the PDM includes formatting and nullability. Also, this model is heavily denormalized. For example:

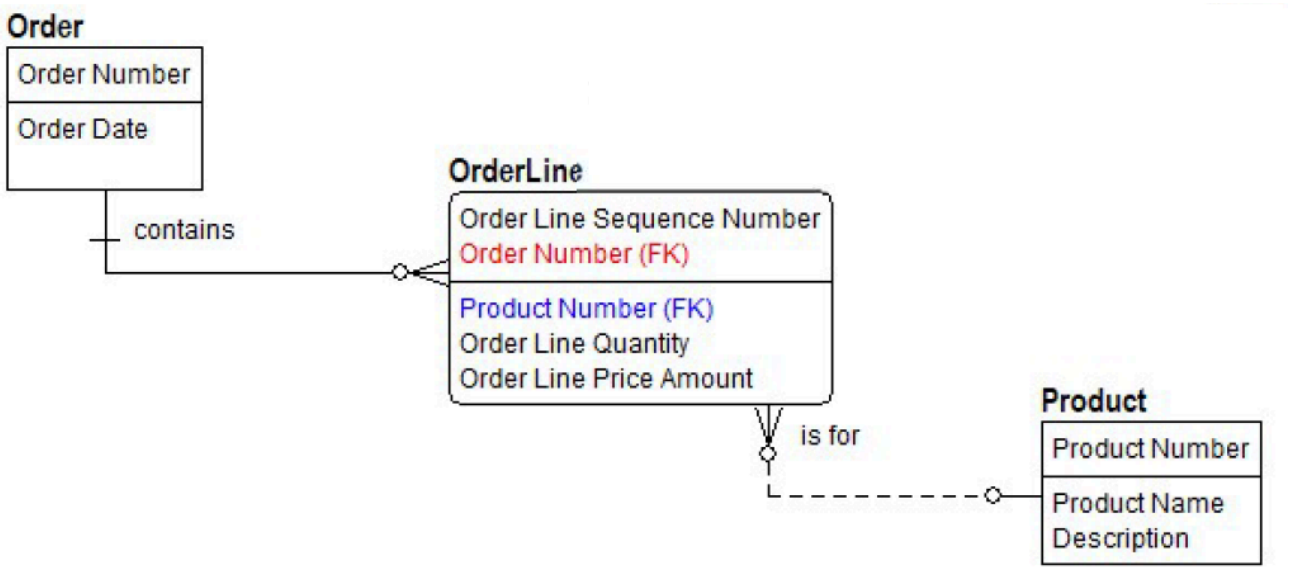
- Although the logical communicates that a Pet can have any number of images, their design only allows up to three images for each Pet. The shelter uses Image_Path_Name_1 for the featured image.
- Notice how the decode entities from the logical have been addressed. The one-to-many relationships is denormalized into Pet. Gender_Name is not needed because everyone knows the codes. People are not familiar with Size_Code so only Size_Name is stored. Both Adoption_Code and Adoption_Name appear because the users could not agree on using just one. Breed has been denormalized into Pet_Breed. It is common for decode entities to be modeled in different ways on the physical, depending on the requirements.
- Vaccination has been denormalized into Pet_Vaccination.

THE CHALLENGES

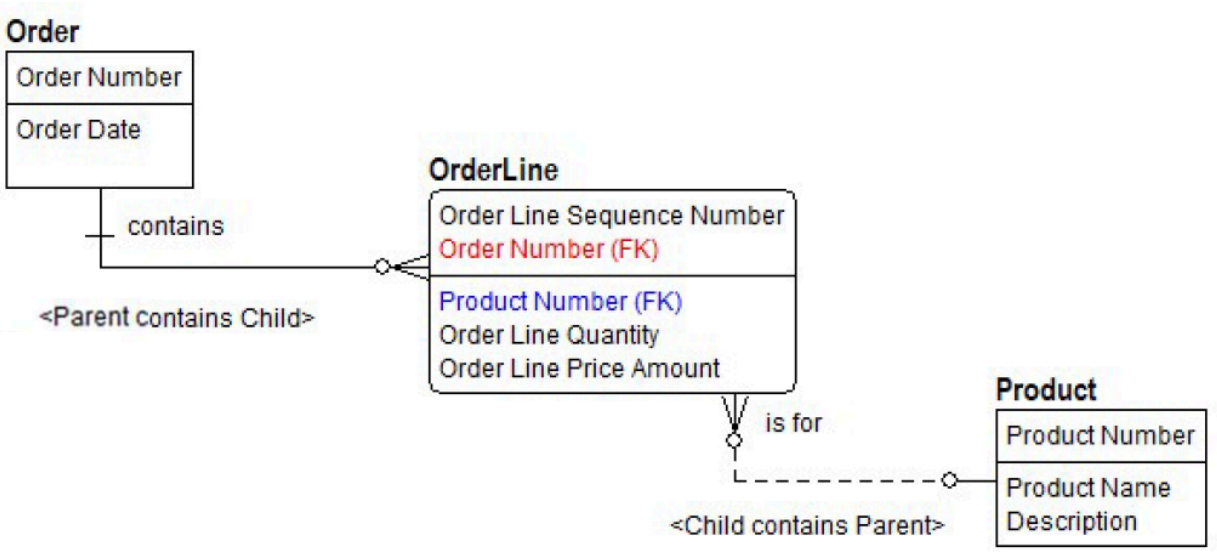
NoSQL requires both a tactical and strategic shift in how we approach modeling.

TACTICAL CHALLENGES

The tactical shift is that MongoDB and JSON allows nesting of structures. Relational databases are fully normalized with foreign key relationships allowing tables to be joined. With MongoDB, performance is key with JOINs between Collections slow. Thus entities deemed as substructures of an important entity can be nested inside the parent object and retrieved as a batch. In the following example, order lines will usually be added and retrieved with the parent order. In MongoDB, we would nest them inside the parent order for maximum performance. In practice, whenever we have an identifying relationship in a logical model, we may want to consider the child entity as a candidate for nesting in MongoDB.

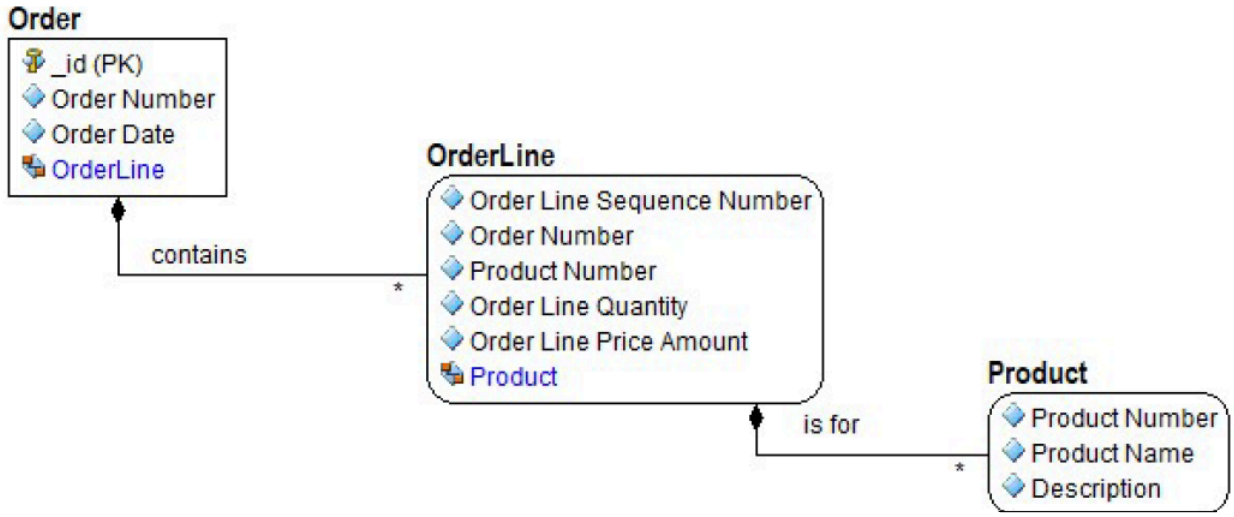


With ER/Studio, we can take any logical model and make another pass over it to look at which entities might be nested within others when a hierarchical physical model is generated.



With ER/Studio, when we forward engineer this logical model into any hierarchical model such as MongoDB, JSON, or Google BigQuery, our Order Line object would be nested inside Order as a Nested Object. ER/Studio allows you to visualize this in two ways:

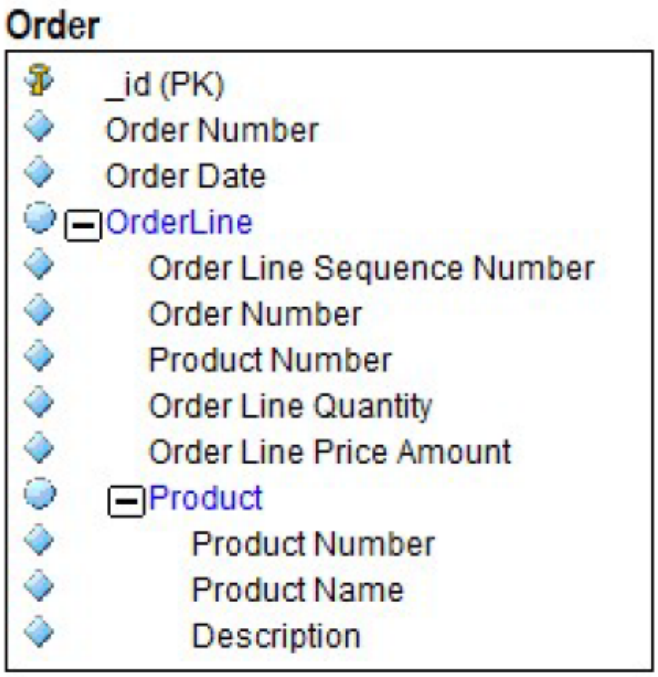
A PSEUDO RELATIONAL VIEW



Notice the containment line with the black diamond. This shows that OrderLine is contained in Order, and Product is contained in OrderLine. The tool has created an array field with the same name as the object OrderLine. This view is quite useful to see how nested objects repeated in multiple parents are used.

ROLLED UP VIEW

If you select Diagram/Object Display on the Diagram ribbon and in the Table tab select 'Roll Up Contained Objects', you get a view more aligned with the final structure of the MongoDB or JSON model.



You can click on the '-' symbol to hide or expand the nested array.

A data model is a communication tool, so choose the option that best resonates with your audience. I prefer the 'Rolled Up' option when communicating with a more technical audience (e.g., developers).

The corresponding JSON structure would look like this:

```
{
    orderNumber : NumberInt(0),
    orderDate : new Date(),
    OrderLine : [
    {
        orderLineSequenceNumber : 0,
        orderLineQuantity : 0,
        orderLinePriceAmount : 0,
        productNumber : “”
    } ]
}
```

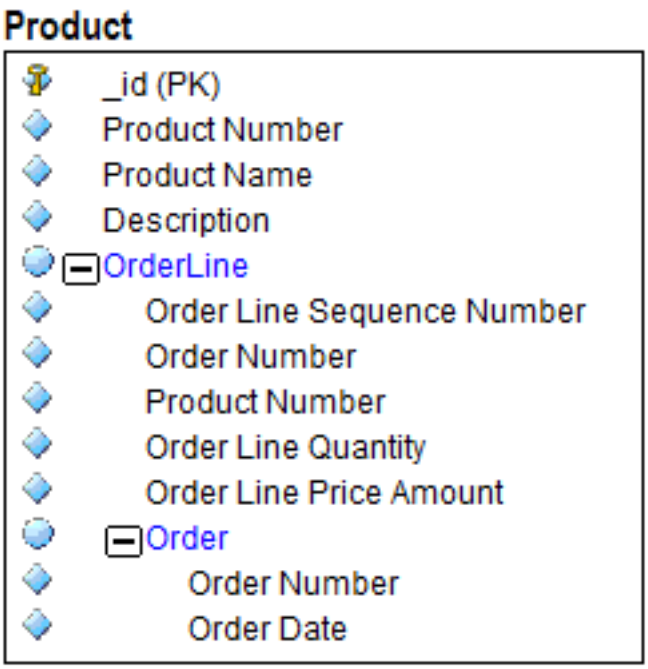
In addition to nested arrays, different NoSQL databases contain specific data types. For example, in MongoDB, there are traditional data types like Integer and Date. But also some which are specific to document databases, such as Object (for embedded documents like OrderLine above) and Arrays.

STRATEGIC CHALLENGES

The strategic shift involves approaching modeling primarily from a query instead of a set theory perspective. Recall from earlier, that although knowing the queries (requirements) helps us collect the necessary set of attributes for the application as well as add database-specific constructs such as indexes and views, in a RDBMS our primary focus is to correctly assign attributes to entities. In NoSQL, however, queries (also called data access) dictate our modeling approach.

The RDBMS model uses primary keys to communicate what attributes uniquely identify each entity instance. We assign the customer attributes to Customer and the account attributes to Account based on their primary key relationships and not primarily because a user wants to see a set of attributes a certain way. In NoSQL, however, queries dictate our modeling approach. For document databases like MongoDB and key-value databases like Dynamo, we organize our entities, attributes, and relationships by queries. We change our approach from “What is the dependency between this attribute and its primary key?” to “How does a user want to see this data?”

It's interesting how focusing more on usage than content also leads to more complex key structures. Instead of a key being solely used to identify a record, that key can also (or instead of) be used for sorting and searching. Therefore, we often use composite keys, where multiple pieces of information are stored in the same attribute. If someone is often searching by country and then postal code, for example, we might have a composite key containing both country and postal code. For instance in the previous example the perspective has been from the order. We wish to rapidly get orders and find out what they are orders for. We can refactor the same basic information from the perspective of the products. Now we can quickly query products and find out what orders we have for each.



Note that we cannot ignore primary keys in a MongoDB model and, more generally, in any NoSQL model. Even when all of our attributes are in a single structure, we need to know how to identify each set of attributes to retain data quality and provide meaningful query results. We cannot, for example, retrieve a particular order without knowing the Order Number or communicate among business experts on a particular Product Name without knowing the Product Number. In modeling for NoSQL, we cannot abandon the primary keys from our RDBMS model! Thus, having a logical data model with primary and alternate keys well-documented will really help with utlizing the MongoDB or JSON products.

COMMON GROUND

The common ground between modeling for a RDBMS and modeling for a NoSQL database is that both tie directly back to a business process.

The RDBMS either shows the business rules that constrain a business process in the case of operational data (OLTP), or the business questions when analyzing a business process in the case of analytics data (OLAP). For NoSQL, we model the access paths providing insights into a business process. OLTP modeling is relational, OLAP modeling is dimensional, and NoSQL modeling is query.

For example, take the process of managing a bank account. The relational model would capture the rule that an account must be owned by one and only one customer. The dimensional would answer the question, "How much in fees did we generate by account type, customer type, month, and bank branch?" followed by drilling up and down the dimensional levels, such as drilling up from month to year. The NoSQL model would assist with the insight, "Which customers own a checking account that generated over $1,000 in fees this year and live within 500 miles of New York City?"

THE SOLUTION

NoSQL does not mean NoDataModeling! We still need to create conceptual, logical, and physical data models. The definitions for each level mentioned in the introduction to this paper are still accurate in a NoSQL world. However, with NoSQL, even though the conceptual still captures the common business language for the initiative, the logical the business requirements, and the physical the design, the models look different because of NoSQL's primary focus on the query. In addition, the names conceptual, logical, and physical, are deeply rooted on the RDBMS side. Therefore, I created a more encompassing name for each level to accommodate both RDBMS and NoSQL for all three levels.

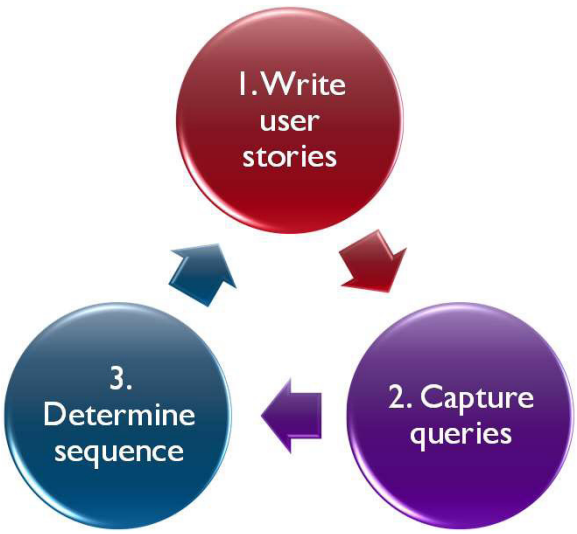
The conceptual level is Align, the logical the Refine, and the physical Design. Align, Refine, and Design – easy to remember and it even rhymes!

For NoSQL's focus on the query, the model produced during alignment is the Query Alignment Model, the model produced during refinement is the Query Refinement Model, and the model produced during design is the Query Design Model. Renaming the conceptual data model to query alignment model, the logical to query refinement, and the physical to query design, retains the important purpose of each level yet also acknowledges the focus on the query.

Our animal shelter knows their world well and has built fairly solid models. Recall they will send a subset of their data to a consortium via JSON, which the consortium's MongoDB database will receive and load for display on their website. Let's go through the align, refine, and design approach for the consortium, and then work on the JSON structure required to move the shelter's data from MS Access to MongoDB.

ALIGN

The align stage is all about developing the common business vocabulary for the initiative. The end goal is the query alignment model, which captures the queries and dependencies for the initiative, along with definitions. The steps to complete are:



We first focus on the user stories, then determine the detailed queries for each story, and finally sequence these queries in the order they occur. It can be iterative. For example, we might identify the sequence between two queries and realize that a query in the middle is missing that will require modifying or adding a user story. Let's go through each of these three steps.

1. Write user stories

User stories have been around for a long time and are extremely useful for NoSQL modeling. Wikipedia defines a user story as: ...an informal, natural language description of features of a software system.

The user story provides the scope and overview for the query alignment model. A query alignment model accommodates one or more user stories. The purpose of a user story is to capture at a very high level how an initiative will deliver business value. User stories take the structure of:

Template	Covers
As a [stakeholder]	Who?
I want to [requirement]	What?
So that [motivation]	Why?

Here are some examples of user stories from tech.gsa.gov:

- As a Content Owner, I want to be able to create product content so that I can provide information and market to customers.
- As an Editor, I want to review content before it is published so that I can assure it is optimized with correct grammar and tone.
- As a HR Manager, I need to view a candidate's status so that I can manage their application process throughout the recruiting phases.
- As a Marketing Data Analyst, I need to run the Salesforce & Google analytics reports so that I can build the monthly media campaign plans.

To keep our animal shelter example relatively simple, assume our animal shelter and others that are part of the consortium met and determined these are the most popular user stories:

1. As a potential dog adopter, I want to find a particular breed or breed mix, so that I get the type of dog I am looking for that will be good with my children.
2. As a potential bird adopter, I want to find an exotic bird, so that I get the bird I am looking for.
3. As a potential cat adopter, I want to find a particular gender, so that I get the type of cat I am looking for.

2. Capture queries

Next, we capture the queries for the one or more user stories within our initiative's scope. Note that it is ok to have just a single user story that drives a NoSQL application. A query starts off with a "verb" and is an action to do something. Some NoSQL database vendors use the phrase "access pattern" instead of query. Here we use the term "query" to also encompass "access pattern".

Here are the queries that satisfy our three user stories:

Q1: Only show pets available for adoption.

Q2: Only show dogs that are good with children.

Q3: Search available dogs by breed.

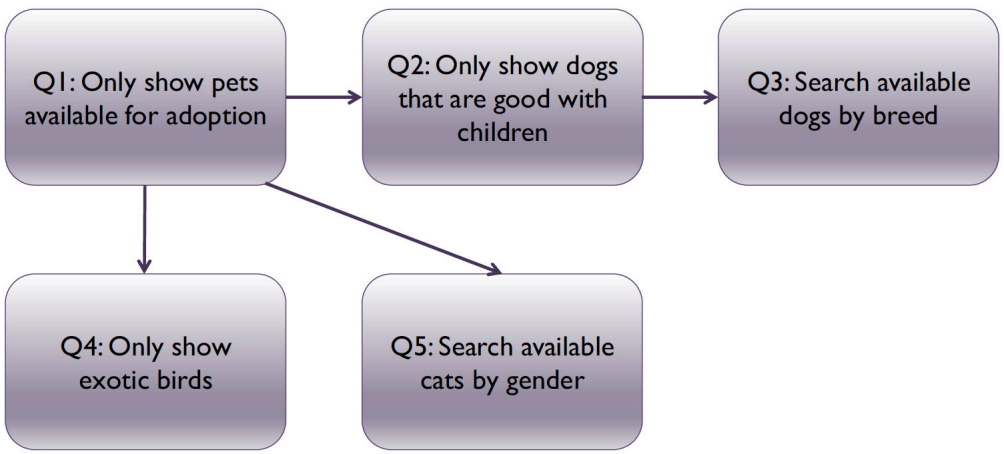
Q4: Only show exotic birds.

Q5: Search available cats by gender.

3. Determine sequence

Next, we need to determine the order someone would run the queries.

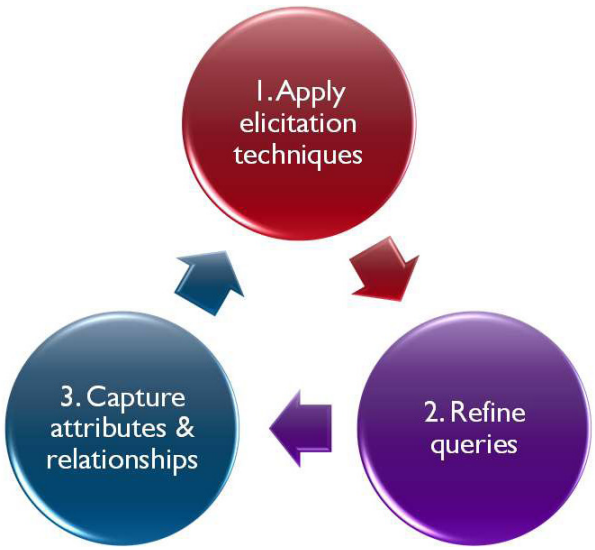
Graphing the sequence of queries leads to the query alignment model. The query alignment model is a numbered list of all queries necessary to deliver the user stories within the initiative's scope. The model also shows a sequence or dependency among the queries. The query alignment model for our five queries would look like:



All of the queries depend on the first query. That is, we first need to filter by animal type.

REFINE

The refine stage is all about determining the business requirements for the initiative. The end goal is the query refinement model, which captures the attributes and relationships needed to answer the queries. The steps to complete are:



Similar to determining the more detailed structures in a traditional logical data model, we determine the more detailed structures needed to deliver the queries during the refinement stage. The query refinement model is all about discovery and captures the answers to the queries that reveal insights into a business process.

1. Apply elicitation techniques

This is where we interact with the business stakeholders to identify the attributes and relationships needed to answer the queries. We keep refining, usually until we run out of time. Techniques we can use include interviewing, artifact analysis (studying existing or proposed business or technical documents), job shadowing (watching someone work), and prototyping. You can use any combination of these techniques to obtain the attributes and relationships to answer the queries. Often these techniques are used within an Agile framework. You choose which techniques to use based on your starting point and the needs of the stakeholders. For example, if a stakeholder says, “I don’t know what I want, but I’ll know when I see it,” building a prototype would be the best approach. Frequently, and especially for graph databases, in situations where we don’t know what questions are even possible to answer, we start with artifact analysis by first studying an existing data set.

2. Refine queries

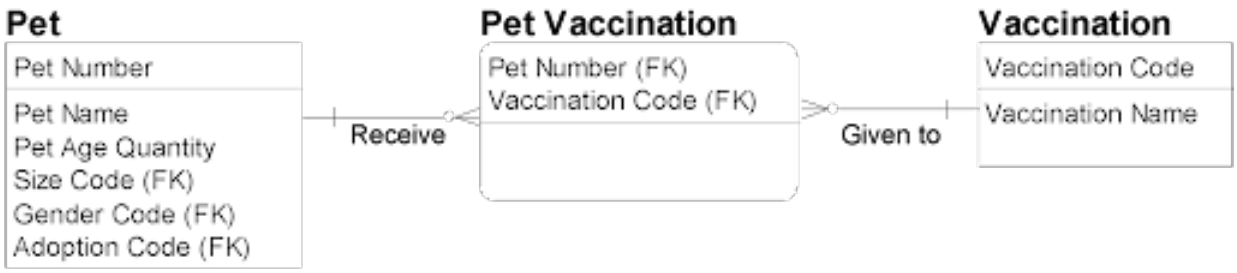
The refinement process is iterative, and we keep refining, again, usually until we run out of time.

3. Capture attributes & relationships

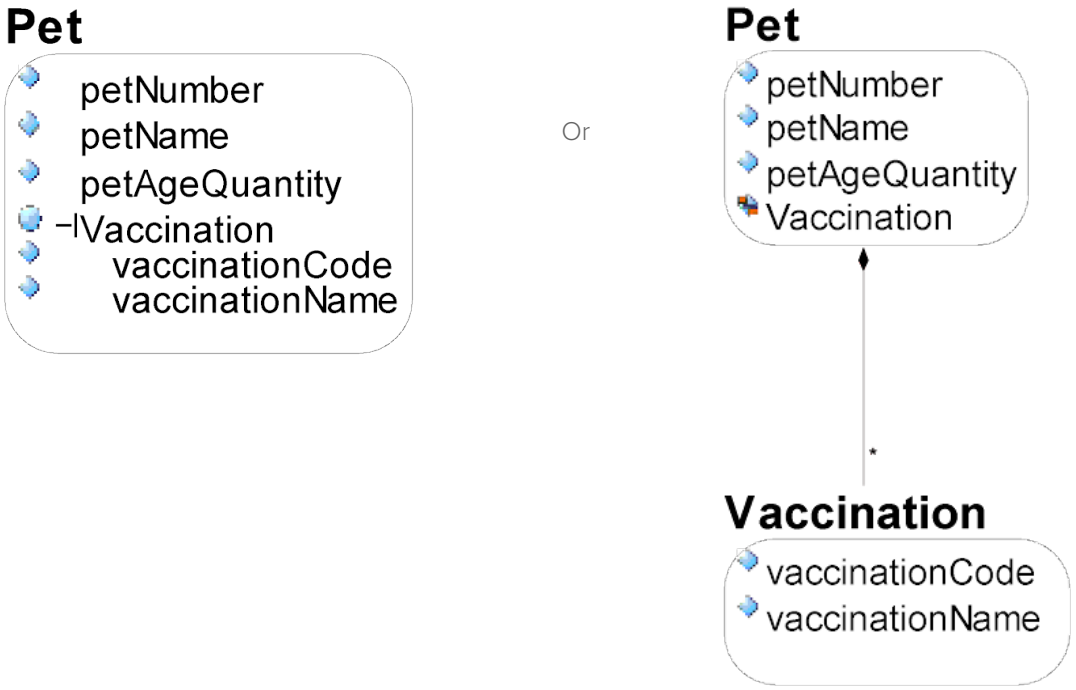
Ideally, because of the hierarchical nature of document (and also key-value) databases, we should strive to answer one or more queries with a single structure. I know this sounds weird (it is even weird for me to type this as I see the world through normalization glasses), but one structure organized to a particular query is much faster and simpler than connecting multiple structures. The query refinement model contains the attributes and related structures needed for each of the queries identified in the query refinement model.

Using artifact analysis, we can start with the animal shelter's logical, and use this model as a good way to capture the attributes and relationships within our scope. Based on the queries, quite a few of our concepts are not directly needed for search or filtering, and so they can become additional descriptive attributes on the Pet entity.

For example, no queries involved vaccinations. Therefore, we can simplify this model subset:

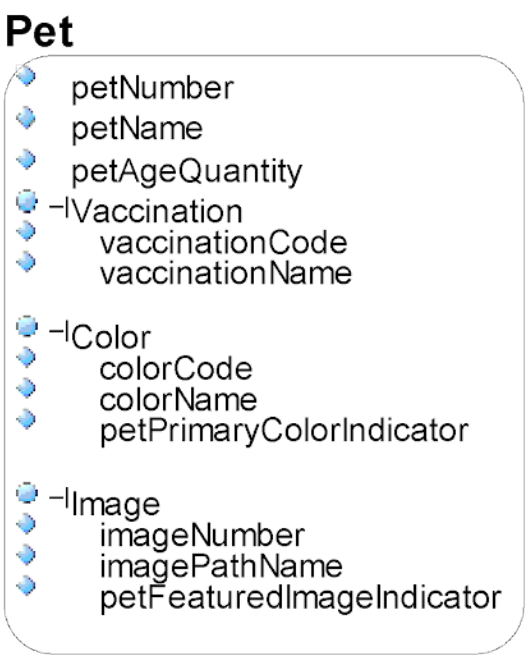


To this model:

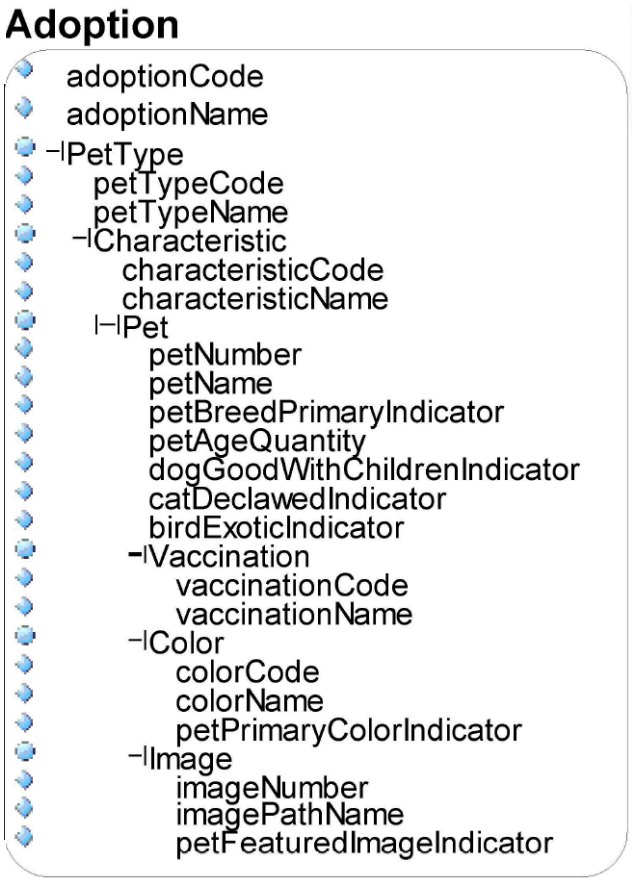


This example illustrates how traditional RDBMS models differ from NoSQL. On our original logical model, it was important to communication that a Pet can receive many Vaccinations and a Vaccination can be given to many Pets. In NoSQL, however, since there were no queries needing to filter or search by vaccination, the vaccination attributes just become other descriptive attributes of Pet. The vaccinationCode and vaccinationName attributes are now a nested array within Pet. So, for example, if Spot the Dog had five vaccinations, they would all be listed within Spot's record (or document to use MongoDB terminology).

Following this same logic, the pet's colors and images also become nested arrays:



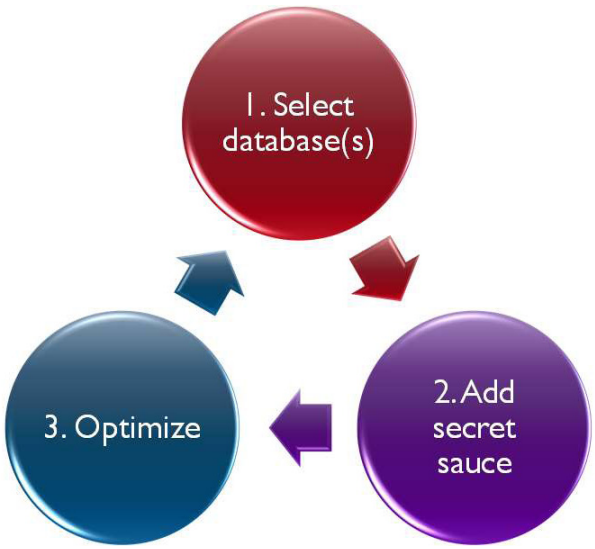
In addition, to help with querying, we need to create a PetType structure instead of the subtypes, Dog, Cat, and Bird. After determining the available pets for adoption, we need to distinguish whether the Pet is a Dog, Cat, or Bird. Our model would now look like this:



Notice that we used a bit of abstraction on this model. So the Breed, Gender, Size, and any other characteristic that comes up in the future has been abstracted into the Characteristic structure. This model answers all five queries. We start off only displaying Pets available for adoption (the highest level). This satisfies Q1 (Only show pets available for adoption). Next, we allow for filtering on the type of Pet (e.g., Dog, Cat, or Bird). Characteristic is essential for this model, as it contains all the different ways of filtering on pets relevant to the queries. For example, characteristics include whether the dog is good with children, whether the cat is a male or female, and whether the bird is exotic. Once the potential adopter chooses their desired characteristics, the pets that have these characteristics will appear. Therefore, the characteristics structure addresses Q2 (Only show dogs that are good with children), Q3 (Search available dogs by breed), and Q4 (Only show exotic birds).

DESIGN

The design stage is all about developing the database-specific design for the initiative. The end goal is the query design model, which captures in this case, the MongoDB design and JSON interchange format for the initiative. The steps to complete are:



1. Select database(s)

We now know enough to decide which database would be ideal for the application. Sometimes we might choose more than one database if we feel it would be the best architecture for the application. We know in the consortium's case that they are using JSON for transport and MongoDB for storage.

2. Add secret sauce

Although the NoSQL databases are very similar, each database has something special to consider during design. For example, Cassandra's view and non-unique index features, DynamoDB's stream feature, and Neo4j's geographic capabilities. For MongoDB, we would consider where to use their secret sauce, such as MongoDB-specific functions like find, aggregate, and transform.

3. Optimize

Similar to indexing, denormalizing, partitioning, and adding views to a RDBMS physical model, we would add database-specific features to the query refinement model to produce the query design model.

TAKEAWAYS

- We model JSON the same way we model MongoDB. There is no magic here, the steps to model data interchange in JSON are the same steps for modeling MongoDB. The query refinement model above would also capture the JSON interchange format. An important difference though is that JSON is always one structure (i.e., one entity or file containing arrays), whereas a document database like MongoDB can be just one structure but can also be multiple structures with references (common attributes) linking the structures.
- Regardless of the technology, data complexity, or breadth of requirements, there will always be a need for a tool that captures the business language (conceptual), the business requirements (logical), and the design (physical). Give each level a name that resonates best with all stakeholders, including developers. I prefer Align, Refine, and Design!
- The RDBMS either shows the business rules that constrain a business process in the case of operational (OLTP), or the business questions when analyzing a business process in the case of analytics (OLAP). For NoSQL, we model the access paths providing insights into a business process. OLTP modeling is relational, OLAP modeling is dimensional, and NoSQL modeling is query.
- NoSQL forces us to consider the queries first and most important. The relationships inherent in the data are important but secondary from a modeling perspective.
- There are a few items that we did not cover in this paper, that are still important to consider when modeling for NoSQL:
    - Ensure clear, complete, and correct definitions for each entity and attribute.
    - The conceptual model can be built on its own without logical or physical, just for the purpose of creating a common business vocabulary.
    - Integration challenges exist regardless of technology. For example, the animal shelter's adoption codes might be a very different set than those used by the consortium. This is one small example of hundreds we can encounter during the integration process.

ABOUT THE AUTHOR

Steve Hoberman's first word was "data". He has been a data modeler for over 30 years, and thousands of business and data professionals have completed his Data Modeling Master Class. His latest version of the class includes sections on modeling for NoSQL. Steve is the author of nine books on data modeling, including The Rosedata Stone and Data Modeling Made Simple. Steve is also the author of Blockchainopoly. One of Steve's frequent data modeling consulting assignments is to review data models using his Data Model Scorecard® technique. He is the founder of the Design Challenges group, creator of the Data Modeling Institute's Data Modeling Certification exam, Conference Chair of the Data Modeling Zone conferences, director of Technics Publications, lecturer at Columbia University, and recipient of the Data Administration Management Association (DAMA) International Professional Achievement Award.