1 WHAT IS DATA MODELING?

Data modeling is the act of exploring data-oriented structures. Like other modeling artifacts data models can be used for a variety of purposes, from high-level conceptual models to physical data models. From the point of view of an object-oriented developer data modeling is conceptually similar to class modeling. With data modeling, you identify entity types whereas with class modeling you identify classes. Data attributes are assigned to entity types just as you would assign attributes and operations to classes. There are associations between entities, similar to the associations between classes – relationships, inheritance, composition, and aggregation are all applicable concepts in data modeling.

Traditional data modeling is different from class modeling because it focuses solely on data – class models allow you to explore both the behavior and data aspects of your domain, with a data model you can only explore data issues. Because of this focus data modelers tend to be much better at getting the data “right" than object modelers. However, some people will model database methods (stored procedures, stored functions, and triggers) when they are physical data modeling. It depends on the situation of course, but I personally think that this is a good idea and promote the concept in my UML data modeling profile (more on this later).

Although the focus of this article is data modeling, there are often alternatives to data-oriented artifacts (never forget Agile Modeling’s Multiple Models principle). For example, when it comes to conceptual modeling ORM diagrams aren’t your only option – In addition to LDMs it is quite common for people to create UML class diagrams and even Class Responsibility Collaborator (CRC) cards instead. In fact, my experience is that CRC cards are superior to ORM diagrams because it is very easy to get project stakeholders actively involved in the creation of the model. Instead of a traditional, analyst-led drawing session you can instead facilitate stakeholders through the creation of CRC cards.

2 HOW ARE DATA MODELS USED IN PRACTICE?

Although methodology issues are covered later, we need to discuss how data models can be used in practice to better understand them. You are likely to see three basic styles of data model:

- Conceptual data models. These models, sometimes called domain models, are typically used to explore domain concepts with project stakeholders. On Agile teams, high-level conceptual models are often created as part of your initial requirements envisioning efforts as they are used to explore the high-level static business structures and concepts. On traditional teams, conceptual data models are often created as the precursor to LDMs or as alternatives to LDMs.
- Logical data models (LDMs). LDMs are used to explore the domain concepts, and their relationships, of your problem domain. This could be done for the scope of a single project or for your entire enterprise. LDMs depict the logical entity types, typically referred to simply as entity types, the data attributes describing those entities, and the relationships between the entities. LDMs are rarely used on Agile projects although often are on traditional projects (where they rarely seem to add much value in practice).

- Physical data models (PDMs). PDMs are used to design the internal schema of a database, depicting the data tables, the data columns of those tables, and the relationships between the tables. PDMs often prove to be useful on both Agile and traditional projects and thus the focus of this article is on physical modeling.

The table below compares the different features:

Feature	*Conceptual*	*Logical*	*Physical*
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

Below we show the conceptual, logical, and physical versions of a single data model.

3 CONCEPTUAL DATA MODEL

A conceptual data model identifies the highest-level relationships between the different entities. Features of conceptual data model include:

- Includes the important entities and the relationships among them.
- No attribute is specified.
- No primary key is specified.

4 LOGICAL DATA MODEL

A logical data model describes the data in as much detail as possible, without regard to how they will be physical implemented in the database. Features of a logical data model include:

- Includes all entities and relationships among them.
- All attributes for each entity are specified.
- The primary key for each entity is specified.
- Foreign keys (keys identifying the relationship between different entities) are specified.
- Normalization occurs at this level.

The steps for designing the logical data model are as follows:

1. Specify primary keys for all entities.
2. Find the relationships between different entities.
3. Find all attributes for each entity.
4. Resolve many-to-many relationships.
5. Normalization.

5 PHYSICAL DATA MODEL

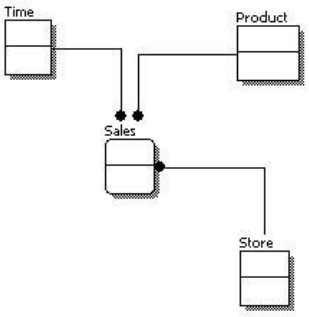
Physical data model represents how the model will be built in the database. A physical database model shows all table structures, including column name, column data type, column constraints, primary key, foreign key, and relationships between tables. Features of a physical data model include:

- Specification all tables and columns.
- Foreign keys are used to identify relationships between tables.
- Denormalization may occur based on user requirements.
- Physical considerations may cause the physical data model to be quite different from the logical data model.
- Physical data model will be different for different RDBMS. For example, data type for a column may be different between MySQL and SQL Server.

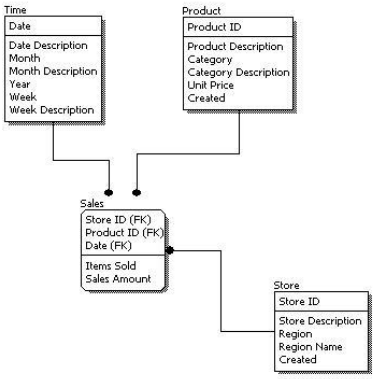
The steps for physical data model design are as follows:

1. Convert entities into tables.
2. Convert relationships into foreign keys.
3. Convert attributes into columns.
4. Modify the physical data model based on physical constraints / requirements.

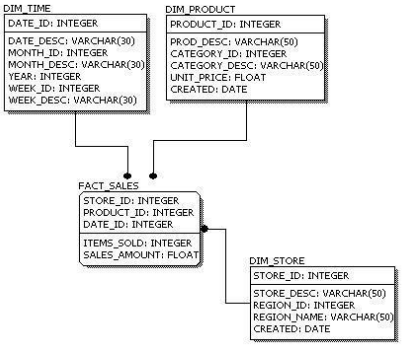
Conceptual Model Design



Logical Model Design



Physical Model Design



We can see that the complexity increases from conceptual to logical to physical. Therefore, we always first start with the conceptual data model (so we understand at high level what are the different entities in our data and how they relate to one another), then move on to the logical data model (so we understand the details of our data without worrying about how they will be implemented), and finally the physical data model (so we know exactly how to implement our data model in the database of choice). In a data warehousing project, sometimes the conceptual data model and the logical data model are considered as a single deliverable.

6 How to Model Data

It is critical for an application developer to have a grasp of the fundamentals of data modeling so they can not only read data models but also work effectively with Agile DBAs who are responsible for the data-oriented aspects of your project. Your goal reading this section is not to learn how to become a data modeler, instead it is simply to gain an appreciation of what is involved.

The following tasks are performed in an iterative manner:

1. Identify entity types
2. Identify attributes
3. Apply naming conventions
4. Identify relationships
5. Apply data model patterns
6. Assign keys
7. Normalize to reduce data redundancy
8. Denormalize to improve performance

6.1 Identify Entity Types

An entity type, also simply called entity (not exactly accurate terminology, but very common in practice), is similar conceptually to object-orientation's concept of a class – an entity type represents a collection of similar objects. An entity type could represent a collection of people, places, things, events, or concepts. Examples of entities in an order entry system would include Customer, Address, Order, Item, and Tax. If you were class modeling you would expect to discover classes with the exact same names. However, the difference between a class and an entity type is that classes have both data and behavior whereas entity types just have data.

Ideally an entity should be normal, the data modeling world's version of cohesive. A normal entity depicts one concept, just like a cohesive class models one concept. For example, customer and order are clearly two different concepts; therefore, it makes sense to model them as separate entities.

6.2 Identify Attributes

Each entity type will have one or more data attributes. For example, in Figure 1 you saw that the Customer entity has attributes such as First Name and Surname and in Figure 2 that the TCUSTOMER table had corresponding data columns CUST_FIRST_NAME and CUST_SURNAME (a column is the implementation of a data attribute within a relational database).

Attributes should also be cohesive from the point of view of your domain, something that is often a judgment call. – in Figure 1 we decided that we wanted to model the fact that people had both first and last names instead of just a name (e.g. "Scott" and "Ambler" vs. "Scott Ambler") whereas we did not distinguish between the sections of an American zip code (e.g. 90210-1234-5678). Getting the level of detail right can have a significant impact on your development and maintenance efforts. Refactoring a single data column into several columns can be difficult, database refactoring is described in detail in Database Refactoring, although over-specifying an attribute (e.g. having three attributes for zip code when you only needed one) can result in overbuilding your system and hence you incur greater development and maintenance costs than you actually needed.

6.3 Apply Data Naming Conventions

Your organization should have standards and guidelines applicable to data modeling, something you should be able to obtain from your enterprise administrators (if they don't exist you should lobby to have some put in place). These guidelines should include naming conventions for both logical and physical modeling, the logical naming conventions should be focused on human readability whereas the physical naming conventions will reflect technical considerations. You can clearly see that different naming conventions were applied in Figures 1 and 2.

As you saw in Introduction to Agile Modeling, AM includes the Apply Modeling Standards practice. The basic idea is that developers should agree to and follow a common set of modeling standards on a software project. Just like there is value in following common coding conventions, clean code that follows your chosen coding guidelines is easier to understand and evolve than code that doesn't, there is similar value in following common modeling conventions.

6.4 Identify Relationships

In the real-world entities have relationships with other entities. For example, customers PLACE orders, customers LIVE AT addresses, and line items ARE PART OF orders. Place, live at, and are part of are all terms that define relationships between entities. The relationships between entities are conceptually identical to the relationships (associations) between objects.

You also need to identify the cardinality and optionality of a relationship (the UML combines the concepts of optionality and cardinality into the single concept of multiplicity). Cardinality represents the concept of “how many" whereas optionality represents the concept of “whether you must have something." For example, it is not enough to know that customers place orders. How many orders can a customer place? None, one, or several? Furthermore, relationships are two-way streets: not only do customers place orders, but orders are placed by customers. This leads to questions like: how many customers can be enrolled in any given order and is it possible to have an order with no customer involved? Figure 5 shows that customers place zero or more orders and that any given order is placed by one customer and one customer only. It also shows that a customer lives at one or more addresses and that any given address has zero or more customers living at it.

6.5 Apply Data Model Patterns

Some data modelers will apply common data model patterns. Data model patterns are conceptually closest to analysis patterns because they describe solutions to common domain issues.

6.6 Assign Keys

There are two fundamental strategies for assigning keys to tables. First, you could assign a natural key which is one or more existing data attributes that are unique to the business

concept. Second, you could introduce a new column, called a surrogate key, which is a key that has no business meaning

6.7 Normalize to Reduce Data Redundancy

Data normalization is a process in which data attributes within a data model are organized to increase the cohesion of entity types. In other words, the goal of data normalization is to reduce and even eliminate data redundancy, an important consideration for application developers because it is incredibly difficult to stores objects in a relational database that maintains the same information in several places. With respect to terminology, a data schema is considered to be at the level of normalization of its least normalized entity type. For example, if all your entity types are at second normal form (2NF) or higher then we say that your data schema is at 2NF.

Why data normalization? The advantage of having a highly-normalized data schema is that information is stored in one place and one place only, reducing the possibility of inconsistent data. Furthermore, highly-normalized data schemas in general are closer conceptually to object-oriented schemas because the object-oriented goals of promoting high cohesion and loose coupling between classes results in similar solutions (at least from a data point of view). This generally makes it easier to map your objects to your data schema. Unfortunately, normalization usually comes at a performance cost.

6.8 Denormalize to Improve Performance

Normalized data schemas, when put into production, often suffer from performance problems. This makes sense – the rules of data normalization focus on reducing data redundancy, not on improving performance of data access. An important part of data modeling is to denormalize portions of your data schema to improve database access times.