CA ERwin® Data Modeler

Methods Guide

r7.3

CA Product References

This document references the following CA products:

- CA® ERwin® Data Modeler (CA ERwin DM)
- CA® Model Manager (CA ERwin MM)
- CA® Process Modeler (CA ERwin PM)

Contact CA

Contact Technical Support

For your convenience, CA provides one site where you can access the information you need for your Home Office, Small Business, and Enterprise CA products. At http://ca.com/support, you can access:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Provide Feedback

If you have comments or questions about CA product documentation, you can send a message to techpubs@ca.com.

If you would like to provide feedback about CA product documentation, please complete our short customer survey, which is also available on the CA support website, found at http://ca.com/support.

Contents

Chapter 1: Introduction

This section contains the following topics:

Data Modeling Concepts

While data modeling can be complex, this *Methods Guide* will help you understand data modeling and its uses. If you are just a beginner, you will understand enough to put the methods to work for you by the time you finish reading this guide.

Overall, this guide has the following purposes:

- To provide a basic level of understanding of the data modeling method used by CA ERwin DM that is sufficient to do real database design.
- To introduce some of the descriptive power and richness of the IDEF1X and IE modeling languages supported and to provide a foundation for future learning.
- Experienced IDEF1X or IE users can use this as a guide to the features of IDEF1X and IE supported, and the mapping between these methods.

Benefits of Data Modeling

Regardless of the type of DBMS you use or the types of data models you want to develop, modeling your database in CA ERwin DM has many benefits:

- Enables usage by database and application development staff to define system requirements and to communicate among themselves and with end-users.
- Provides a clear picture of referential integrity constraints. Maintaining referential integrity is essential in the relational model where relationships are encoded implicitly.
- Provides a logical RDBMS-independent picture of your database that can be used by automated tools to generate RDBMS-specific information. This way, you can use a single diagram to generate DB2 table schemas, as well as schemas for other relational DBMSs.
- Lets you produce a diagram summarizing the results of your data modeling efforts and generate a database schema from that model.

Methods

Two methods of data modeling are supported by CA ERwin DM, which include:

IDEF1X

The IDEF1X method was developed by the United States Air Force. It is now used in various governmental agencies, in the aerospace and financial industry, and in a wide variety of major corporations.

IE (Information Engineering)

The IE method was developed by James Martin, Clive Finkelstein, and other IE authorities and is widely deployed in a variety of industries.

Both methods are suited to environments where large scale, rigorous, enterprise-wide data modeling is essential.

Typographical Conventions

This guide uses special typographic conventions to identify the user interface controls and key terms that appear in the text. The following table describes these conventions:

Text Item	**Convention**	**Example**
Entity Name	All uppercase; followed by the word "entity" in lowercase	MOVIE COPY entity

Text Item	Convention	Example
Attribute Name	All lowercase in quotation marks; hyphen replaces embedded spaces	“movie-name”
Column Name	All lowercase	movie_name
Table Name	All uppercase	MOVIE_COPY
Verb Phrase	All lowercase in angle brackets	<is available for rental as>

Chapter 2: Information Systems, Databases, and Models

This section contains the following topics:

Introduction

Two different types of models are used:

Data modeling

Data modeling is the process of describing information structures and capturing business rules in order to specify information system requirements. A data model represents a balance between the specific needs of a particular RDBMS implementation project, and the general needs of the business area that requires it.

Process modeling

Process models are used to identify and document the portion of system requirements that relates to data. Structured system development approaches in general, and data-centered design approaches specifically, invest heavily in front-end planning and requirements analysis activities. Process models, such as data flow diagram sets, distribution models, and event/state models can be created in CA ERwin PM and other tools to document processing requirements. Different levels of these models are used during different development phases.

Data Modeling

When created with the full participation of business and systems professionals, the data model can provide many benefits. These benefits generally fall into the following two classes:

Effort

Those associated with the process of creating the model.

Product of the Effort

Those primarily associated with the model.

Examples of Product Benefits

- A data model is independent of implementation, so it does not require that the implementation is in any particular database or programming language.
- A data model is an unambiguous specification of what is wanted.
- The model is business user-driven. The content and structure of the model are controlled by the business client rather than the system developer. The emphasis is on requirements rather than constraints or solutions.
- The terms used in the model are stated in the language of the business, not that of the system development organization.
- The model provides a context to focus discussions on what is important to the business.

Examples of Process Benefits

- During early project phases, model development sessions bring together individuals from many parts of the business and provide a structured forum where business needs and policies are discussed. During these sessions, it is often the case that the business staff, for the first time, meets others in different parts of the organization who are concerned with the same needs.
- Sessions lead to development of a common business language with consistent and precise definitions of terms used. Communication among participants is greatly increased.
- Early phase sessions provide a mechanism for exchanging large amounts of information among business participants and transferring much business knowledge to the system developers. Later phase sessions continue that transfer of knowledge to the staff who will implement the solution.

- Session participants are generally able to better see how their activities fit into a larger context. Also, parts of the project can be seen in the context of the whole. The emphasis is on cooperation rather than separation. Over time, this can lead to a shift in values, and the reinforcement of a cooperative philosophy.
- Sessions foster consensus and build teams.

Design of the data structures to support a business area is only one part of developing a system. *Function modeling*, the analysis of processes (function) is equally important. Function models describe how something is done. They can be presented as hierarchical decomposition charts, data flow diagrams, HIPO diagrams, and so on. You will find, in practice, that it is important to develop both your function models and data models at the same time. Discussion of the functions that the system will perform uncovers the data requirements. Discussion of the data normally uncovers additional function requirements. Function and data are the two sides of the system development coin.

Process Modeling

Direct support for process modeling is supported and can work well with many techniques. For example, CA also provides CA ERwin PM, a function modeling tool that supports IDEF0, IDEF3 workflow, and data flow diagram methods and can be used along with CA ERwin DM to complete an analysis of process during a data modeling project.

Data Modeling Sessions

Creating a data model involves not only construction of the model, but also numerous fact-finding sessions (meetings) that uncover the data and processes used by a business. Running good sessions, like running good meetings of any kind, depends on a lot of preparation and real-time facilitation techniques. In general, modeling sessions should include the right mix of business and technical experts and should be facilitated. This means that modeling sessions are scheduled well in advance, carefully planned to cover sets of focused material, and orchestrated in such a way that the desired results are achieved.

When possible, it is highly recommended that modeling of function and data be done at the same time. This is because functional models tend to validate a data model and uncover new data requirements. This approach also ensures that the data model supports function requirements. To create both a function model and a data model in a single modeling session, it is important to include a data modeler and a process modeler who are responsible for capturing the functions that are explored.

Session Roles

Formal, guided sessions, with defined roles for participants and agreed upon procedures and rules, are an absolute requirement. The following roles work well:

Facilitator

Acts as the session guide. This person is responsible for arranging the meetings and facilities, providing follow-up documentation, and intervening during sessions, as necessary, to keep sessions on track and to control the scope of the session.

Data Modeler

Leads the group through the process of developing and validating the model. The modeler develops the model, in real time if possible, in front of the group by asking pertinent questions that bring out the important details and recording the resulting structure for all to see. It is often possible (although somewhat difficult) for the same individual to play both facilitator and data modeler roles.

Data Analyst

Acts as the scribe for the session and records the definitions of all the entities and attributes that make up the model. Based on information from the business experts, the data analyst can also begin to package entities and attributes into subject areas, manageable and meaningful subsets of the complete data model.

Subject Matter Expert

Provides the business information needed to construct the model. You can have more than one subject matter experts. They are business experts, not systems experts.

Manager

Participates in the sessions in an assigned role (facilitator, subject matter expert, and so on) but has the additional responsibility of making decisions as needed to keep the process moving. The manager has the responsibility of “breaking ties” but only when absolutely necessary. The manager can be from either the systems or business community.

Sample IDEF1X Modeling Methodology

CA ERwin DM was developed to support the IDEF1X and IE modeling standards. The use of various levels of models within the IDEF1X method can be very helpful in developing a system. General model levels are outlined in the IDEF1X standard and are presented next. In practice, you may find it useful to expand or contract the number of levels to fit individual situations.

The model levels generally span from a very wide but not too detailed view of the major entities that are important to a business, down to a level of precision required to represent the database design in terms understandable by a particular DBMS. At the very lowest level of detail, models are technology dependent. For example, a model for an IMS database looks very different from a model for a DB2 database. At higher levels, models are technology independent and may even represent information, which is not stored in any automated system.

The modeling levels presented are well suited to a top-down system development life cycle approach, where successive levels of detail are created during each project phase.

The highest level models come in two forms:

Entity Relationship Diagram (ERD)

Identifies major business entities and their relationships.

Key-Based (KB)

Sets the scope of the business information requirement (all entities are included) and begins to expose the detail.

The lower level models also come in two forms:

Fully-Attributed (FA)

Represents a third normal form model which contains all of the detail for a particular implementation effort.

Transformation Model (TM)

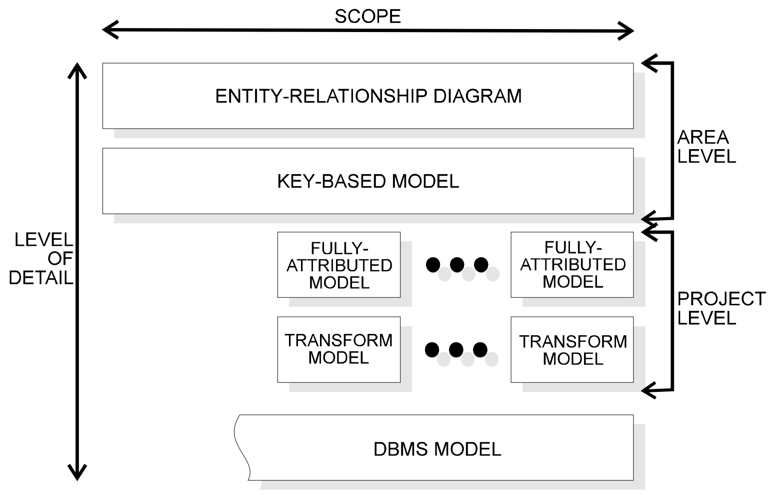
Represents a transformation of the relational model into a structure, which is appropriate to the DBMS chosen for implementation. The TM, in most cases, is no longer in third normal form. The structures are optimized based on the capabilities of the DBMS, the data volumes, and the expected access patterns and rates against the data. In a way, this is a picture of the eventual physical database design.

DBMS Model

The database design is contained in the DBMS Model for the system. Depending on the level of integration of the information systems of a business, the DBMS Model may be a project level model or an area level model for the entire integrated system.

Modeling Architecture

These five modeling levels are presented in the following figure. Notice that the DBMS Model can be at either an Area Level scope, or a Project Level scope. It is not uncommon to have single ERD and KB models for a business and multiple DBMS Models, one for each implementation environment, and then another set within that environment for projects that do not share databases. In an ideal situation, there are a set of Area Level scope DBMS Models, one for each environment, with complete data sharing across all projects in that environment.



These models fall into two categories:

- Logical
- Physical

Logical Models

There are three levels of logical models that are used to capture business information requirements: the Entity Relationship Diagram, the Key-Based Model, and the Fully-Attributed model. The Entity Relationship Diagram and the Key-Based models are also called *area data models* since they often cover a wide business area that is larger than the business chooses to address with a single automation project. In contrast, the Fully-Attributed model is a *project data model* since it typically describes a portion of an overall data structure intended for support by a single automation effort.

Entity Relationship Diagram

The Entity Relationship Diagram (ERD) is a high-level data model that shows the major entities and relationships, which support a wide business area. This is primarily a presentation or discussion model.

The objective of the ERD is to provide a view of business information requirements sufficient to satisfy the need for broad planning for development of its information system. These models are not very detailed (only major entities are included) and there is not much detail, if any, on attributes. Many-to-many (non-specific) relationships are allowed and keys are generally not included.

Key-Based Model

A Key-Based (KB) Model describes the major data structures, which support a wide business area. All entities and primary keys are included along with sample attributes.

The objective of the KB model is to provide a broad business view of data structures and keys needed to support the area. This model provides a context where detailed implementation level models can be constructed. The model covers the same scope as the Area ERD, but exposes more of the detail.

Fully-Attributed Model

A Fully-Attributed (FA) Model is a third normal form data model that includes all entities, attributes, and relationships needed by a single project. The model includes entity instance volumes, access paths and rates, and expected transaction access patterns across the data structure.

Physical Models

There are also two levels of physical models for an implementation project: the Transformation Model and the DBMS Model. The physical models capture all of the information that systems developers need to understand and implement a logical model as a database system. The Transformation Model is also a project *data model* that describes a portion of an overall data structure intended for support by a single automation effort. Individual projects within a business area are supported, allowing the modeler to separate a larger area model into submodels, called subject areas. Subject areas can be developed, reported on, and generated to the database in isolation from the area model and other subject areas in the model.

Transformation Model

The objectives of the Transformation Model are to provide the Database Administrator (DBA) with sufficient information to create an efficient physical database, to provide a context for the definition and recording of the data elements and records that form the database in the data dictionary, and to help the application team choose a physical structure for the programs that will access the data.

When it is appropriate for the development effort, the model can also provide the basis for comparing the physical database design against the original business information requirements to:

- Demonstrate that the physical database design adequately supports those requirements.
- Document physical design choices and their implications, such as what is satisfied, and what is not.
- Identify database extensibility capabilities and constraints.

DBMS Model

The Transformation Model directly translates into a DBMS model, which captures the physical database object definitions in the RDBMS schema or database catalog. The schema generation function directly supports this model. Primary keys become unique indices. Alternate keys and inversion entries also may become indices. Cardinality can be enforced either through the referential integrity capabilities of the DBMS, application logic, or "after the fact" detection and repair of violations.

Chapter 3: Logical Models

This section contains the following topics:

How to Construct a Logical Model

The first step in constructing a logical model is developing the *Entity Relationship Diagram* (ERD), a high-level data model of a wide business area. An ERD is made up of three main building blocks: entities, attributes, and relationships. If you view a diagram as a graphical language for expressing statements about your business, entities are the nouns, attributes are the adjectives or modifiers, and relationships are the verbs. Building a data model is simply a matter of putting together the right collection of nouns, verbs, and adjectives.

The objective of the ERD is to provide a broad view of business information requirements sufficient to plan for development of the business information system. These models are not very detailed (only major entities are included) and there is not much detail, if any, about attributes. Many-to-many (non-specific) relationships are allowed and keys are generally not included. This is primarily a presentation or discussion model.

An ERD may be divided into subject areas, which are used to define business views or specific areas of interest to individual business functions. Subject areas help reduce larger models into smaller, more manageable subsets of entities that can be more easily defined and maintained.

There are many methods available for developing the ERD. These range from formal modeling sessions to individual interviews with business managers who have responsibility for wide areas.

Entity Relationship Diagram

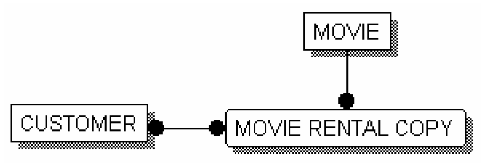
If you are familiar with a relational database structure, you know that the most fundamental component of a relational database is the table. Tables are used to organize and store information. A table is organized in columns and rows of data. Each row contains a set of facts called an instance of the table.

In a relational database, all data values must also be atomic, which means that each cell in the table can contain only a single fact. There is also a relationship between the tables in the database. Each relationship is represented in an RDBMS by sharing one or more columns in two tables.

Like the tables and columns that make up a physical model of a relational database, an ERD (and all other logical data models) includes equivalent components that let you model the data structures of the business, rather than the database management system. The logical equivalent to a table is an entity, and the logical equivalent to a column is an attribute.

In an ERD, an entity is represented by a box that contains the name of the entity. Entity names are always singular: CUSTOMER not CUSTOMERS, MOVIE not MOVIES, COUNTRY not COUNTRIES. By always using singular nouns, you gain the benefit of a consistent naming standard and facilitate reading the diagram as a set of declarative statements about entity instances.

The figure that follows is one created by a hypothetical video store that needs to track its customers, movies that can be rented or purchased, and rental copies of movies that are in stock in the store.



In an ERD, a relationship is represented by a line drawn between the entities in the model. A relationship between two entities also implies that facts in one entity refer to, or are associated with, facts in another entity. In the previous example, the video store needs to track information about CUSTOMERs and MOVIE RENTAL COPYs. The information in these two entities is related, and this relationship can be expressed in a statement: A CUSTOMER rents one or more MOVIE RENTAL COPYs.

Entities and Attributes Defined

An entity is any person, place, thing, event, or concept about which information is kept. More precisely, an entity is a set or collection of like individual objects called instances. An instance (row) is a single occurrence of a given entity. Each instance must have an identity distinct from all other instances.

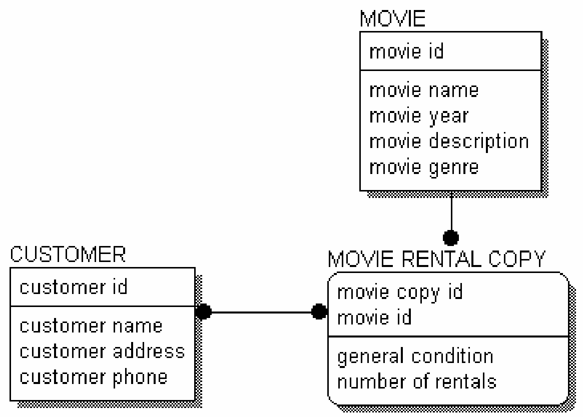
In the previous figure, the CUSTOMER entity represents the set of all of the possible customers of a business. Each instance of the CUSTOMER entity is a customer. You can list information for an entity in a sample instance table, as shown in the following table:

CUSTOMER

customer-id	customer-name	customer-address
10001	Ed Green	Princeton, NJ
10011	Margaret Henley	New Brunswick, NJ
10012	Tomas Perez	Berkeley, CA
17886	Jonathon Walters	New York, NY
10034	Greg Smith	Princeton, NJ

Each instance represents a set of facts about the related entity. In the previous table, each instance of the CUSTOMER entity includes information about the “customer-id,” “customer-name,” and “customer-address.” In a logical model, these properties are called the *attributes* of an entity. Each attribute captures a single piece of information about the entity.

You can include attributes in an ERD to describe the entities in the model more fully, as shown in the following figure:



Logical Relationships

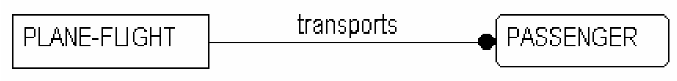
Relationships represent connections, links, or associations between entities. They are the *verbs* of a diagram that show how entities relate to each other. Easy-to-understand rules help business professionals validate data constraints and ultimately identify relationship cardinality.

Examples of one-to-many relationships:

- A TEAM <has> many PLAYERs.
- A PLANE-FLIGHT <transports> many PASSENGERs.
- A DOUBLES-TENNIS-MATCH <requires> exactly 4 PLAYERs.
- A HOUSE <is owned by> one or more OWNERs.
- A SALESPERSON <sells> many PRODUCTs.

In all of these cases, the relationships are chosen so that the connection between the two entities is what is known as one-to-many. This means that one (and only one instance) of the first entity is related or connected to many instances of the second entity. The entity on the *one-end* is called the parent entity. The entity on the *many-end* is called the child entity.

Relationships are displayed as a line connecting two entities, with a dot on one end, and a verb phrase written along the line. In the previous examples, the verb phrases are the words inside the brackets, such as <sells>. The following figure shows the relationship between PLANE-FLIGHTs and PASSENGERs on that flight:

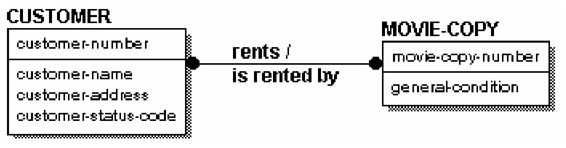


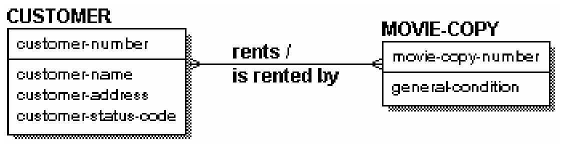
Many-to-Many Relationships

A many-to-many relationship, also called a non-specific relationship, represents a situation where an instance in one entity relates to one or more instances in a second entity and an instance in the second entity also relates to one or more instances in the first entity. In the video store example, a many-to-many relationship occurs between a CUSTOMER and a MOVIE COPY. From a conceptual point of view, this many-to-many relationship indicates that:

- A CUSTOMER <rents> many MOVIE COPYs
- A MOVIE COPY <is rented by> many CUSTOMERs

You typically use many-to-many relationships in a preliminary stage of diagram development, such as in an ERD, and are represented in IDEF1X as a solid line with dots on both ends.





Since a many-to-many relationship can hide other business rules or constraints, they should be fully explored at a later point in the modeling process. For example, sometimes a many-to-many relationship identified in early modeling stages is mislabeled and is actually two one-to-many relationships between related entities. Or, the business must keep additional facts about the many-to-many relationship, such as dates or comments, and the result is that the many-to-many relationship must be replaced by an additional entity to keep these facts. You need to fully discuss all many-to-many relationships during later modeling stages to ensure that the relationship is correctly modeled.

Logical Model Design Validation

Since a data model exposes many of the business rules that describe the area being modeled, reading the relationships helps you validate that the design of the logical model is correct. Verb phrases provide a brief summary of the business rules embodied by relationships. Although they do not precisely describe the rules, verb phrases do provide an initial sense of how the entities are connected.

If you choose your verb phrases correctly, you should be able to read a relationship from the parent to the child using an *active* verb phrase.

Example:

A PLANE FLIGHT <transports> many PASSENGERs.

Verb phrases can also be read from the perspective of the child entity. You can often read from the child entity perspective using *passive* verb phrases.

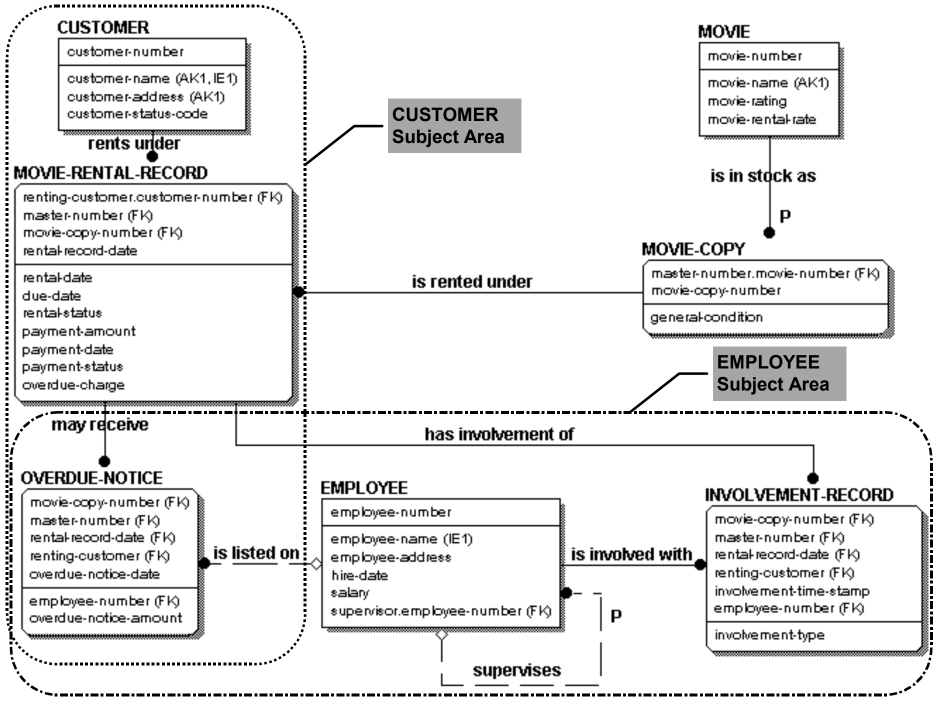
Example:

Many PASSENGERs <are transported by> a PLANE FLIGHT.

It is a good practice to make sure that each verb phrase in the model results in valid statements. Reading your model back to the business analysts and subject matter experts is one of the primary methods of verifying that it correctly captures the business rules.

Data Model Example

The following model of a database was constructed for a hypothetical video store and appears in the following figure:



The data model of the video store, along with definitions of the objects presented on it, makes the following assertions:

- A MOVIE is in stock as one or more MOVIE-COPYs. Information recorded about a MOVIE includes its name, a rating, and a rental rate. The general condition of each MOVIE-COPY is recorded.
- The store's CUSTOMERs rent the MOVIE-COPYs. A MOVIE-RENTAL-RECORD records the particulars of the rental of a MOVIE-COPY by a CUSTOMER. The same MOVIE-COPY may, over time, be rented to many CUSTOMERs.
- Each MOVIE-RENTAL-RECORD also records a due date for the movie and a status indicating whether or not it is overdue. Depending on a CUSTOMER's previous relationship with the store, a CUSTOMER is assigned a credit status code which indicates whether the store should accept checks or credit cards for payment, or accept only cash.
- The store's EMPLOYEEs are involved with many MOVIE-RENTAL-RECORDs, as specified by an involvement type. There must be at least one EMPLOYEE involved with each record. Since the same EMPLOYEE might be involved with the same rental record several times on the same day, involvements are further distinguished by a time stamp.

- An overdue charge is sometimes collected on a rental of a MOVIE-COPY. OVERDUE-NOTICEs are sometimes needed to remind a CUSTOMER that a movie needs to be returned. An EMPLOYEE is sometimes listed on an OVERDUE-NOTICE.
- The store keeps salary and address information about each EMPLOYEE. It sometimes needs to look up CUSTOMERs, EMPLOYEEs, and MOVIEs by name, rather than by number.

This is a relatively small model, but it says a lot about the video rental store. From it, you get an idea of what a database for the business should look like, and you get a good picture of the business. There are several different types of graphical objects in this diagram. The entities, attributes, and relationships, along with the other symbols, describe our business rules. In the following chapters, you will learn more about what the different graphical objects mean and how to use CA ERwin DM to create your own logical and physical data models.

Chapter 4: The Key-Based Data Model

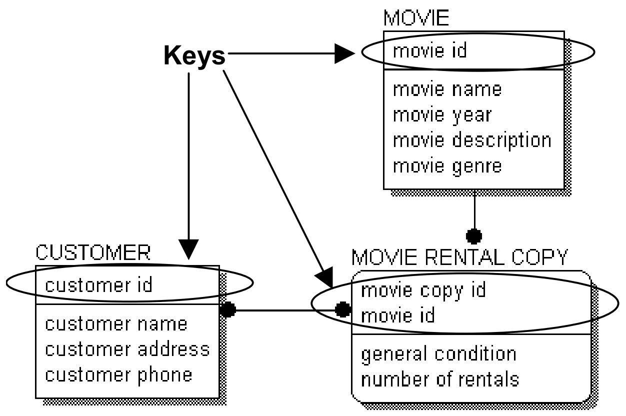
This section contains the following topics:

Key Based Data Model

A Key-Based (KB) Model is a data model that fully describes all of the major data structures that support a wide business area. The goal of a KB model is to include all entities and attributes that are of interest to the business.

As its name suggests, a KB model also includes keys. In a logical model, a *key* identifies unique instances within an entity. When implemented in a physical model, a key provides easy access to the underlying data.

Basically, the key-based model covers the same scope as the Entity Relationship Diagram (ERD) but exposes more of the detail, including the context where detailed implementation level models can be constructed.



Key Types

Whenever you create an entity in your data model, one of the most important questions you need to ask is: “How can a unique instance be identified?” In order to develop a correct logical data model, you must be able to uniquely identify each instance in an entity.

In each entity in a data model, a horizontal line separates the attributes into two groups, key areas and non-key areas. The area above the line is called the key area, and the area below the line is called the non-key area or data area. The key area of CUSTOMER contains “customer-id” and the data area contains “customer-name,” “customer-address,” and “customer-phone.”

Entity and Non-Key Areas

The key area contains the primary key for the entity. The primary key is a set of attributes used to identify unique instances of an entity. The primary key may be comprised of one or more primary key attributes, as long as the chosen attributes form a unique identifier for each instance in an entity.

An entity usually has many non-key attributes, which appear below the horizontal line. A non-key attribute does not uniquely identify an instance of an entity. For example, a database may have multiple instances of the same customer name, which means that “customer-name” is not unique and would probably be a non-key attribute.

Primary Key Selection

Choosing the primary key of an entity is an important step that requires some serious consideration. Before you actually select a primary key, you may need to consider several attributes, which are referred to as candidate key attributes. Typically, the business user who has knowledge of the business and business data can help identify candidate keys.

For example, to correctly use the EMPLOYEE entity in a data model (and later in a database), you must be able to uniquely identify instances. In the customer table, you could choose from several potential key attributes including: the employee name, a unique employee number assigned to each instance of EMPLOYEE, or a group of attributes, such as name and birth date.

The rules that you use to select a primary key from the list of all candidate keys are stringent and can be consistently applied across all types of databases and information. The rules state that the attribute or attribute group must:

- Uniquely identify an instance.
- Never include a NULL value.
- Not change over time. An instance takes its identity from the key. If the key changes, it is a different instance.
- Be as short as possible, to facilitate indexing and retrieval. If you need to use a key that is a combination of keys from other entities, make sure that each part of the key adheres to the other rules.

Example:

Consider which attribute you would select as a primary key from the following list of candidate keys for an EMPLOYEE entity:

- employee-number
- employee-name
- employee-social-security number
- employee-birth-date
- employee-bonus-amount

If you use the rules previously listed to find candidate keys for EMPLOYEE, you might compose the following analysis of each attribute:

- "employee-number" is a candidate key since it is unique for all EMPLOYEEs
- "employee-name" is probably not a good candidate since multiple employees may have the same name, such as Mary Jones.
- "employee-social-security-number" is unique in most instances, but every EMPLOYEE may not have one.
- The combination of "employee-name" and "employee-birth-date" may work, unless there is more than one John Smith born on the same date and employed by our company. This could be a candidate key.
- Only some EMPLOYEEs of our company are eligible for annual bonuses. Therefore, "employee-bonus-amount" can be expected to be NULL in many cases. As a result, it cannot be part of any candidate key.

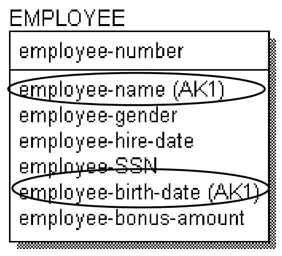
After analysis, there are two candidate keys-one is "employee-number" and the other is the group of attributes containing "employee-name" and "employee-birth-date." "employee-number" is selected as the primary key since it is the shortest and ensures uniqueness of instances.

When choosing the primary key for an entity, modelers often assign a surrogate key, an arbitrary number that is assigned to an instance to uniquely identify it within an entity. "employee-number" is an example of a surrogate key. A surrogate key is often the best choice for a primary key since it is short, can be accessed the fastest, and ensures unique identification of each instance. Further, a surrogate key can be automatically generated by the system so that numbering is sequential and does not include any gaps.

A primary key chosen for the logical model may not be the primary key needed to efficiently access the table in a physical model. The primary key can be changed to suit the needs and requirements of the physical model and database at any point.

Alternate Key Attributes

After you select a primary key from a list of candidate keys, you can designate some or all of the remaining candidate keys as alternate keys. Alternate keys are often used to identify the different indexes, which are used to quickly access the data. In a data model, an alternate key is designated by the symbol (AK*n*), where *n* is a number that is placed after the attributes that form the alternate key group. In the EMPLOYEE entity, "employee-name" and "employee-birth-date" are members of the alternate key group.

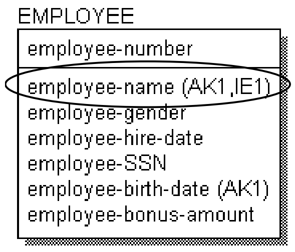


Inversion Entry Attributes

Unlike a primary key or an alternate key, an inversion entry is an attribute or set of attributes that are commonly used to access an entity, but which may not result in finding exactly one instance of an entity. In a data model, the symbol IE*n* is placed after the attribute.

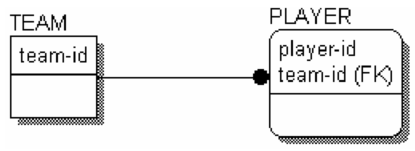
For example, in addition to locating information in an employee database using an employee's identification number, a business may want to search by employee name. Often, a name search results in multiple records, which requires an additional step to find the exact record. By assigning an attribute to an inversion entry group, a non-unique index is created in the database.

Note: An attribute can belong to an alternate key group as well as an inversion entry group.



Relationships and Foreign Key Attributes

A foreign key is the set of attributes that define the primary key in the parent entity and that migrate through a relationship from the parent to the child entity. In a data model, a foreign key is designated by the symbol (FK) after the attribute name. Notice the (FK) next to “team-id” in the following figure:

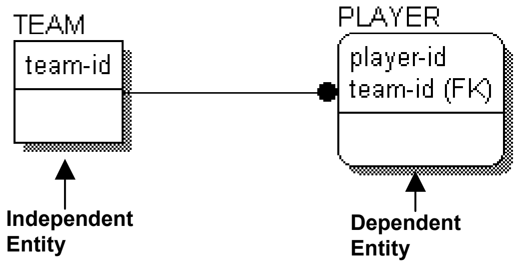


Dependent and Independent Entities

As you develop your data model, you may discover certain entities that depend upon the value of the foreign key attribute for uniqueness. For these entities, the foreign key must be a part of the primary key of the child entity (above the line) in order to uniquely define each entity.

In relational terms, a child entity that depends on the foreign key attribute for uniqueness is called a dependent entity. In IDEF1X notation, dependent entities are represented as round-cornered boxes.

Entities that do not depend on any other entity in the model for identification are called independent entities. In IE and IDEF1X, independent entities are represented as square-cornered boxes.

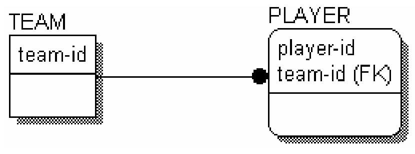


Dependent entities are further classified as existence dependent, which means the dependent entity cannot exist unless its parent does, and identification dependent, which means that the dependent entity cannot be identified without using the key of the parent. The PLAYER entity is identification dependent but not existence dependent, since PLAYERs can exist if they are not on a TEAM.

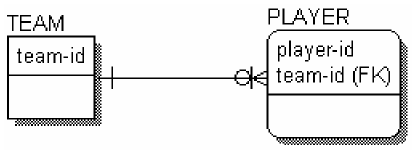
In contrast, there are situations where an entity is existence dependent on another entity. Consider two entities: ORDER, which a business uses to track customer orders, and LINE ITEM, which tracks individual items in an ORDER. The relationship between these two entities can be expressed as An ORDER <contains> one or more LINE ITEMS. In this case, LINE ITEM is existence dependent on ORDER, since it makes no sense in the business context to track LINE ITEMS unless there is a related ORDER.

Identifying Relationships

In IDEF1X notation, the type of the relationship that connects two entities enforces the concept of dependent and independent entities. If you want a foreign key to migrate to the key area of the child entity (and create a dependent entity as a result), you can create an identifying relationship between the parent and child entities. A solid line connecting the entities indicates an identifying relationship. In IDEF1X notation, the line includes a dot on the end nearest to the child entity, as shown in the following figure:



In IE notation, the line includes a *crow's foot* at the end of the relationship nearest to the child entity:



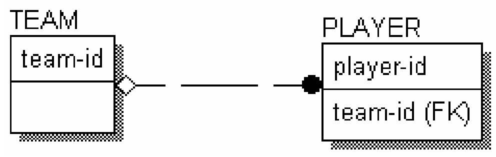
Note: Standard IE notation does not include rounded corners on entities. This is an IDEF1X symbol that is included in IE notation to ensure compatibility between methods.

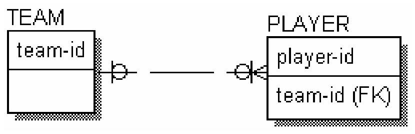
As you may find, there are advantages to contributing keys to a child entity through identifying relationships in that it tends to make some physical system queries more straightforward, but there are also many disadvantages. Some advanced relational theory suggests that contribution of keys should not occur in this way. Instead, each entity should be identified not only by its own primary key, but also by a logical handle or surrogate key, never to be seen by the user of the system. There is a strong argument for this in theory and those who are interested are urged to review the work of E. F. Codd and C. J. Date in this area.

Non-Identifying Relationships

A non-identifying relationship also connects a parent entity to a child entity. But, when a non-identifying relationship connects two entities, the foreign key migrates to the non-key area of the child entity (below the line).

A dashed line connecting the entities indicates a non-identifying relationship. If you connect the TEAM and PLAYER entities in a non-identifying relationship, the "team-id" migrates to the non-key as shown in the figure below:





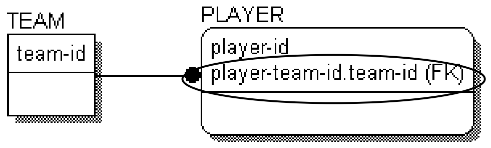
Since the migrated keys in a non-identifying relationship are not part of the primary key of the child, non-identifying relationships do not result in any identification dependency. In this case, PLAYER is considered an independent entity, just like TEAM.

However, the relationship can reflect existence dependency if the business rule for the relationship specifies that the foreign key cannot be NULL (missing). If the foreign key must exist, this implies that an instance in the child entity can only exist if an associated parent instance also exists.

Note: Identifying and non-identifying relationships are not a feature of the IE methodology. However, this information is included in your diagram in the form of a solid or dashed relationship line to ensure compatibility between IE and IDEF1X methods.

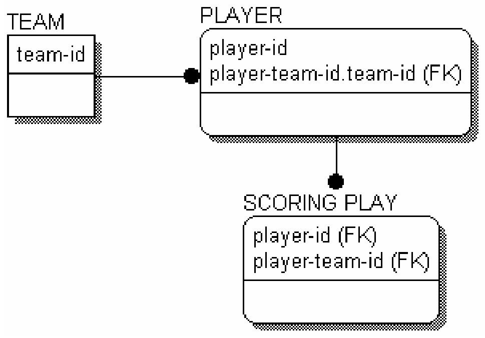
Rolenames

When foreign keys migrate from the parent entity in a relationship to the child entity, they are serving double-duty in the model in terms of stated business rules. To understand both roles, it is sometimes helpful to rename the migrated key to show the role it plays in the child entity. This name assigned to a foreign key attribute is called a rolename. In effect, a rolename declares a new attribute, whose name is intended to describe the business statement embodied by the relationship that contributes the foreign key.



The foreign key attribute of "player-team-id.team-id" in the PLAYER entity shows the syntax for defining and displaying a rolename. The first half (before the period) is the rolename. The second half is the original name of the foreign key, sometimes called the base name.

Once assigned to a foreign key, a rolename migrates across a relationship just like any other foreign key. For example, suppose that you extend the example to show which PLAYERs have scored in various games throughout the season. The "player-team-id" rolename migrates to the SCORING PLAY entity (along with any other primary key attributes in the parent entity), as shown in the figure below:



Note: A rolename is also used to model compatibility with legacy data models where the foreign key often had a different name from the primary key.

Chapter 5: Naming and Defining Entities and Attributes

This section contains the following topics:

Overview

It is extremely important in data modeling, and in systems development in general, to choose clear and well thought out names for objects. The result of your efforts will be a clear, concise, and unambiguous model of a business area.

Naming standards and conventions are identical for all types of logical models, including both the Entity Relationship diagrams (ERD) and Key-based (KB) diagrams.

Entity and Attribute Names

The most important rule to remember when naming entities is that entity names are always singular. This facilitates reading the model with declarative statements such as "A FLIGHT <transports> zero or more PASSENGERs" and "A PASSENGER <is transported by> one FLIGHT." When you name an entity, you are also naming each instance. For example, each instance of the PASSENGER entity is an individual passenger, not a set of passengers.

Attribute names are also singular. For example, "person-name," "employee-SSN," "employee-bonus-amount" are correctly named attributes. Naming attributes in the singular helps to avoid normalization errors, such as representing more than one fact with a single attribute. The attributes "employee-child-names" or "start-or-end-dates" are plural, and highlight errors in the attribute design.

A good rule to use when naming attributes is to use the entity name as a prefix. The rule here is:

- Prefix qualifies
- Suffix clarifies

Using this rule, you can easily validate the design and eliminate many common design problems. For example, in the CUSTOMER entity, you can name the attributes "customer-name," "customer-number," "customer-address," and so on. If you are tempted to name an attribute "customer-invoice-number," you use the rule to check that the suffix "invoice-number" tells you more about the prefix "customer." Since it does not, you must move the attribute to a more appropriate location, such as INVOICE.

You may sometimes find that it is difficult to give an entity or attribute a name without first giving it a definition. As a general principle, providing a good definition for an entity or attribute is as important as providing a good name. The ability to find meaningful names comes with experience and a fundamental understanding of what the model represents.

Since the data model is a description of a business, it is best to choose meaningful business names wherever that is possible. If there is no business name for an entity, you must give the entity a name that fits its purpose in the model.

Synonyms, Homonyms, and Aliases

Not everyone speaks the same language. Not everyone is always precise in the use of names. Since entities and attributes are identified by their names in a data model, you need to ensure that synonyms are resolved to ensure that they do not represent redundant data. Then you need to precisely define them so that each person who reads the model can understand which facts are captured in which entity.

It is also important to choose a name that clearly communicates a sense of what the entity or attribute represents. For example, you get a clear sense that there is some difference among things called PERSON, CUSTOMER, and EMPLOYEE. Although they can all represent an individual, they have distinct characteristics or qualities. However, it is the role of the business user to tell you whether or not PERSON and EMPLOYEE are two different things or just synonyms for the same thing.

Choose names carefully, and be wary of calling two different things by the same name. For example, if you are dealing with a business area which insists on calling its customers "consumers," do not force or insist on the customer name. You may have discovered an alias, another name for the same thing, or you may have a new "thing" that is distinct from, although similar to, another "thing." In this case, perhaps CONSUMER is a category of CUSTOMER that can participate in relationships that are not available for other categories of CUSTOMER.

You can enforce unique naming in the modeling environment. This way you can avoid the accidental use of homonyms (words that are written the same but have different meanings), ambiguous names, or duplication of entities or attributes in the model.

Entity Definitions

Defining the entities in your logical model is essential to the clarity of the model and is a good way to elaborate on the purpose of the entity and clarify which facts you want to include in the entity. Undefined entities or attributes can be misinterpreted in later modeling efforts, and possibly deleted or unified based on the misinterpretation.

Writing a good definition is more difficult than it may seem at first. Everyone knows what a CUSTOMER is, right? Just try writing a definition of a CUSTOMER that holds up to scrutiny. The best definitions are created using the points of view of many different business users and functional groups within the organization. Definitions that can pass the scrutiny of many, disparate users provide a number of benefits including:

- Clarity across the enterprise
- Consensus about a single fact having a single purpose
- Easier identification of categories of data

Most organizations and individuals develop their own conventions or standards for definitions. In practice you will find that long definitions tend to take on a structure that helps the reader to understand the "thing" that is being defined. Some of these definitions can go on for several pages (CUSTOMER, for example). As a starting point, you may want to adopt the following items as a basic standard for the structure of a definition, since IDEF1X and IE do not provide standards for definitions:

- Description
- Business example
- Comments

Descriptions

A description should be a clear and concise statement that tells whether an object is or is not the thing you are trying to define. Often such descriptions can be fairly short. Be careful, however, that the description is not too general or uses terms that are not defined. Here are a couple of examples, one of good quality and one that is questionable:

Example of good description:

A COMMODITY is something that has a value that can be determined in an exchange.

This is a good description since, after reading it, you know that something is a COMMODITY if someone is, or would be, willing to trade something for it. If someone is willing to give you three peanuts and a stick of gum for a marble, then you know that a marble is a COMMODITY.

Example of bad description:

A CUSTOMER is someone who buys something from our company.

This is not a good description since you can easily misunderstand the word "someone" if you know that the company also sells products to other businesses. Also, the business may want to track potential CUSTOMERs, not just those who have already bought something from the company. You can also define "something" more fully to describe whether the sale is of products, services, or some combination of the two.

Business Examples

It is a good idea to provide typical business examples of the thing being defined, since good examples can go a long way to help the reader understand a definition. Comments about peanuts,   marbles or something related to your business can help a reader to understand the concept of a COMMODITY. The definition states that a commodity has value. The example can help to show that value is not always measured in money.

Comments

You can also include general comments about who is responsible for the definition and who is the source, what state it is in, and when it was last changed as a part of the definition. For some entities, you may also need to explain how it and a related entity or entity name differ. For instance, a CUSTOMER might be distinguished from a PROSPECT.

Definition References and Circularity

An individual definition can look good, but when viewed together they can be circular. Without some care, this can happen with entity and attribute definitions.

Example:

- CUSTOMER: Someone who buys one or more of our PRODUCTs
- PRODUCT: Something we offer for sale to CUSTOMERs

It is important when you define entities and attributes in your data model that you avoid these circular references.

Business Glossary Construction

It is often convenient to make use of common business terms when defining an entity or attribute. For example, "A CURRENCY-SWAP is a complex agreement between two PARTYs where they agree to exchange **cash flows** in two different CURRENCYs over a period of time. Exchanges can be fixed over the **term** of the swap, or may **float**. Swaps are often used to **hedge** currency and interest **rate risks**."

In this example, defined terms within a definition are highlighted. Using a style like this makes it unnecessary to define terms each time they are used, since people can look them up whenever needed.

If it is convenient to use, for example, common business terms that are not the names of entities or attributes, it is a good idea to provide base definitions of these terms and refer to these definitions. A glossary of commonly used terms, separate from the model, can be used. Such common business terms are highlighted with bold-italics, as shown in the previous example.

It may seem that a strategy like this can at first lead to a lot of going back and forth among definitions. The alternative, however, is to completely define each term every time it is used. When these internal definitions appear in many places, they need to be maintained in many places, and the probability that a change will be applied to all of them at the same time is very small.

Developing a glossary of common business terms can serve several purposes. It can become the base for use in modeling definitions, and it can, all by itself, be of significant value to the business in helping people to communicate.

Attribute Definitions

As with entities, it is important to define all attributes clearly. The same rules apply. By comparing an attribute to a definition, you should be able to tell if it fits. However, you should be aware of incomplete definitions.

Example:

account-open-date

The date on which the ACCOUNT was opened. A further definition of what “opened” means is needed before the definition is clear and complete.

Attribute definitions generally should have the same basic structure as entity definitions, including a description, examples, and comments. The definitions should also contain, whenever possible, validation rules that specify which facts are accepted as valid values for that attribute.

Validation Rules

A *validation rule* identifies a set of values that an attribute is allowed to take; it constrains or restricts the domain of values that are acceptable. These values have meanings in both an abstract and a business sense. For example, “person-name,” if it is defined as the preferred form of address chosen by the PERSON, is constrained to the set of all character strings. You can define any validation rules or valid values for an attribute as a part of the attribute definition. You can assign these validation rules to an attribute using a domain. Supported domains include text, number, datetime, and blob.

Definitions of attributes, such as codes, identifiers, or amounts, often do not lend themselves to good business examples. So, including a description of the attribute's validation rules or valid values is usually a good idea. When defining a validation rule, it is good practice to go beyond listing the values that an attribute can take. Suppose you define the attribute “customer-status” as follows:

Customer-status: *A code that describes the relationship between the CUSTOMER and our business.* Valid values: *A, P, F, N.*

The validation rule specification is not too helpful since it does not define what the codes mean. You can better describe the validation rule using a table or list of values, such as the following table:

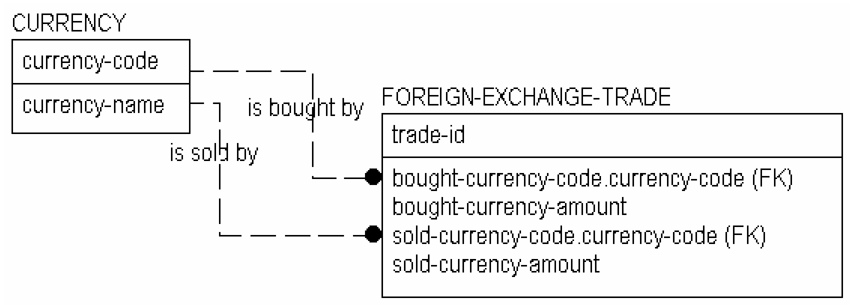
Valid Value	Meaning
A: Active	The CUSTOMER is currently involved in a purchasing relationship with our company.

Valid Value	Meaning
P: Prospect	Someone with whom we are interested in cultivating a relationship, but with whom we have no current purchasing relationship.
F: Former	The CUSTOMER relationship has lapsed. In other words, there has been no sale in the past 24 months.
N: No business accepted	The company has decided that no business will be done with this CUSTOMER.

Rolenames

When a foreign key is contributed to a child entity through a relationship, you may need to write a new or enhanced definition for the foreign key attributes that explains their usage in the child entity and can assign a rolename to the definition. This is certainly the case when the same attribute is contributed to the same entity more than once. These duplicated attributes may appear to be identical, but because they serve two different purposes, they cannot have the same definition.

Consider the following example shown in the figure below. Here you see a FOREIGN-EXCHANGE-TRADE with two relationships to CURRENCY.



The key of CURRENCY is “currency-code,” (the identifier of a valid CURRENCY that you are interested in tracking). You can see from the relationships that one CURRENCY is “bought by,” and one is “sold by” a FOREIGN-EXCHANGE-TRADE.

You also see that the identifier of the CURRENCY (the “currency-code”) is used to identify each of the two CURRENCYs. The identifier of the one that is bought is called “bought-currency-code” and the identifier of the one that is sold is called “sold-currency-code.” These rolenames show that these attributes are not the same thing as “currency-code.”

It would be somewhat silly to trade a CURRENCY for the same CURRENCY at the same time and exchange rate. So for a given transaction (instance of FOREIGN-EXCHANGE-TRADE) “bought-currency-code” and “sold-currency-code” must be different. By giving different definitions to the two rolenames, you can capture the difference between the two currency codes.

Attribute/Rolename	**Attribute Definition**
currency-code	The unique identifier of a CURRENCY.
bought-currency-code	The identifier (“currency-code”) of the CURRENCY bought by (purchased by) the FOREIGN-EXCHANGE-TRADE.
sold-currency-code	The identifier (“currency-code”) of the CURRENCY sold by the FOREIGN-EXCHANGE-TRADE.

The definitions and validations of the bought and sold codes are based on “currency-code.” “Currency-code” is called a base attribute.

The IDEF1X standard dictates that if two attributes with the same name migrate from the same base attribute to an entity, then the attributes must be unified. The result of unification is a single attribute migrated through two relationships. Because of the IDEF1X standard, foreign key attributes are automatically unified as well. If you do not want to unify migrated attributes, you can rolename the attributes at the same time that you name the relationship, in the Relationship Editor.

Definitions and Business Rules

Business rules are an integral part of the data model. These rules take the form of relationships, rolenames, candidate keys, defaults, and other modeling structures, including generalization categories, referential integrity, and cardinality. Business rules are also captured in entity and attribute definitions and validation rules.

For example, a CURRENCY entity defined either as the set of all valid currencies recognized anywhere in the world, or could be defined as the subset of these which our company has decided to use in its day to day business operations. This is a subtle, but important distinction. In the latter case, there is a business rule, or policy statement, involved.

This rule manifests itself in the validation rules for "currency-code." It restricts the valid values for "currency-code" to those that are used by the business. Maintenance of the business rule becomes a task of maintaining the table of valid values for CURRENCY. To permit or prohibit trading of CURRENCYs, you simply create or delete instances in the table of valid values.

The attributes "bought-currency-code" and "sold-currency-code" are similarly restricted. Both are further restricted by a validation rule that says "bought-currency-code" and "sold-currency-code" cannot be equal. Therefore, each is dependent on the value of the other in its actual use. Validation rules can be addressed in the definitions of attributes, and can also be defined explicitly using validation rules, default values, and valid value lists.

Chapter 6: Model Relationships

This section contains the following topics:

Relationships

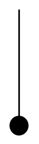
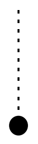
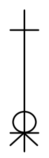
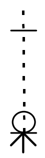
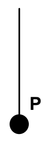
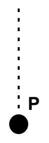
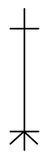
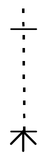
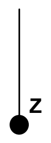
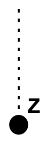
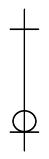
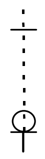
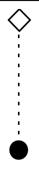
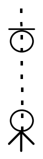
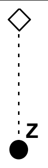
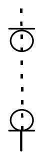
Relationships are a bit more complex than they might seem at first. They carry a lot of information. Some might say that they are the heart of the data model, since, to a great extent, they describe the rules of the business and the constraints on creating, modifying, and deleting instances. For example, you can use cardinality to define exactly how many instances are involved in both the child and parent entities in the relationship. You can further specify how you want to handle database actions such as INSERT, UPDATE, and DELETE using referential integrity rules.

Data modeling also supports highly complex relationship types that enable you to construct a logical model of your data that is understandable to both business and systems experts.

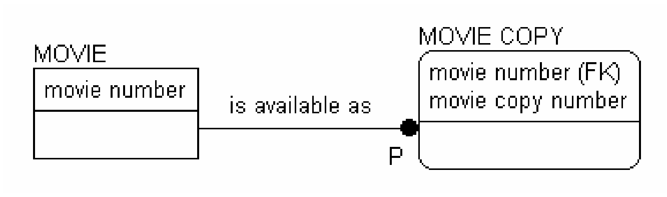
Relationship Cardinality

The idea of many in a one-to-many relationship does not mean that there has to be more than one instance of the child connected to a given parent. Instead the many in one-to-many really means that there are zero, one, or more instances of the child paired up to the parent.

Cardinality is the relational property that defines exactly how many instances appear in a child table for each corresponding instance in the parent table. IDEF1X and IE differ in the symbols that are used to specify cardinality. However, both methods provide symbols to denote one or more, zero or more, zero or one, or exactly N, as explained in the following table:

Cardinality Description	**IDEF1X Notation**		**IE Notation**	
	Identifying	**Non-identifying**	**Identifying**	**Non-identifying**
One to zero, one, or more				
One to one or more				
One to zero or one				
Zero or one to zero, one, or more (non-identifying only)				
Zero or one to zero or one (non-identifying only)				

Cardinality lets you specify additional business rules that apply to the relationship. In the following figure, the business has decided to identify each MOVIE COPY based on both the foreign key “movie-number” and a surrogate key “copy-number.” Also, each MOVIE is available as one or more MOVIE COPYs. The business has also stated that the relationship is identifying, that MOVIE COPY cannot exist unless there is a corresponding MOVIE.



The MOVIE-MOVIE COPY model also specifies the cardinality for the relationship. The relationship line shows that there will be exactly one MOVIE, and only one, participating in a relationship. This is because MOVIE is the parent in the relationship.

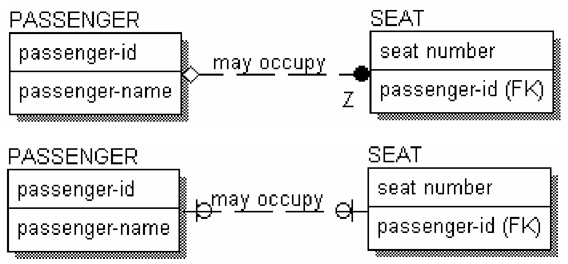
By making MOVIE-COPY the child in the relationship (shown with a dot in IDEF1X), the business defined a MOVIE-COPY as one of perhaps several rentable copies of a movie title. The business also determined that to be included in the database, a MOVIE must have at least one MOVIE-COPY. This makes the cardinality of the *is available as* relationship one-to-one or more. The *P* symbol next to the dot represents cardinality of one or more. As a result, you also know that a MOVIE with no copies is not a legitimate instance in this database.

In contrast, the business might want to know about all of the MOVIEs in the world, even those for which they have no copies. So their business rule is that for a MOVIE to exist (be recorded in their information system) there can be zero, one, or more copies. To record this business rule, the P is removed. When cardinality is not explicitly indicated in the diagram, cardinality is one-to-zero, one or more.

Cardinality in Non-Identifying Relationships

Non-identifying relationships contribute keys from a parent to a child entity. However, by definition, some (or all) of the keys do not become part of the key of the child. This means that the child will not be identification-dependent on the parent. Also, there can be situations where an entity at the *many* end of the relationship can exist without a parent, that is, it is not existence-dependent.

If the relationship is *mandatory* from the perspective of the child, then the child is existence-dependent on the parent. If it is *optional*, the child is neither existence nor identification-dependent with respect to that relationship (although it may be dependent in other relationships). To indicate the optional case, IDEF1X includes a diamond at the parent end of the relationship line and IE includes a circle.



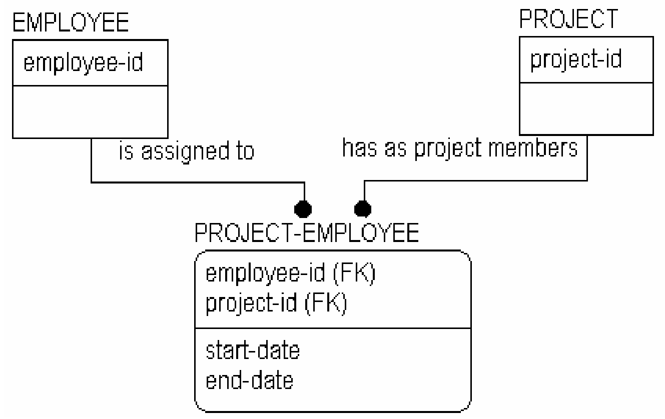
In this figure, the attribute "passenger-id" is a foreign key attribute of SEAT. Since the "passenger-id" does not identify the SEAT, it identifies the PASSENGER occupying the SEAT, the business has determined that the relationship is non-identifying. The business has also stated that the SEAT can exist without any PASSENGER, so the relationship is optional. When a relationship is optional, the diagram includes either a diamond in IDEF1X or a circle in IE notation. Otherwise, the cardinality graphics for non-identifying relationships are the same as those for identifying relationships.

The cardinality for the relationship, indicated with a *Z* in IDEF1X and a single line in IE, states that a PASSENGER <may occupy> zero or one of these SEATs on a flight. Each SEAT can be occupied, in which case the PASSENGER occupying the seat is identified by the "passenger-id," or it can be unoccupied, in which case the "passenger-id" attribute is empty (NULL).

Referential Integrity

Since a relational database relies on data values to implement relationships, the integrity of the data in the key fields is extremely important. If you change a value in a primary key column of a parent table, for example, you must account for this change in each child table where the column appears as a foreign key. The action that is applied to the foreign key value varies depending on the rules defined by the business.

For example, a business that manages multiple projects might track its employees and projects in a model similar to the one in the figure below. The business has determined already that the relationship between PROJECT and PROJECT-EMPLOYEE is identifying, so the primary key of PROJECT becomes a part of the primary key of PROJECT-EMPLOYEE.



In addition, the business decides that for each instance of PROJECT-EMPLOYEE there is exactly one instance of PROJECT. This means that PROJECT-EMPLOYEE is existence-dependent on PROJECT.

What would happen if you were to delete an instance of PROJECT? If the business decided that it did not want to track instances in PROJECT-EMPLOYEE if PROJECT is deleted, you would have to delete all instances of PROJECT-EMPLOYEE that inherited part of their key from the deleted PROJECT.

The rule that specifies the action taken when a parent key is deleted is called *referential integrity*. The referential integrity option chosen for this action in this relationship is Cascade. Each time an instance of PROJECT is deleted, this Delete cascades to the PROJECT-EMPLOYEE table and causes all related instances in PROJECT-EMPLOYEE to be deleted as well.

Available actions for referential integrity include the following:

Cascade

Each time an instance in the parent entity is deleted, each related instance in the child entity must also be deleted.

Restrict

Deletion of an instance in the parent entity is prohibited if there are one or more related instances in the child entity, or deletion of an instance in the child entity is prohibited if there is a related instance in the parent entity.

Set Null

Each time an instance in the parent entity is deleted, the foreign key attributes in each related instance in the child entity are set to NULL.

Set Default

Each time an instance in the parent entity is deleted, the foreign key attributes in each related instance in the child entity are set to the specified default value.

<None>

No referential integrity action is required. Not every action must have a referential integrity rule associated with it. For example, a business may decide that referential integrity is not required when deleting an instance in a child entity. This is a valid business rule in cases where the cardinality is zero, one to zero, or one or more, since instances in the child entity can exist even if there are no related instances in the parent entity.

Although referential integrity is not a formal part of the IDEF1X or IE languages, it does capture business rules that indicate how the completed database should work, so it is a critical part of data modeling. This provides a method for both capture and display of referential integrity rules.

Once referential integrity is defined, the facilitator or analyst should test the referential integrity rules defined by the business users by asking questions or working through different scenarios that show the results of the business decision. When the requirements are defined and fully understood, the facilitator or analyst can recommend specific referential integrity actions, for instance Restrict or Cascade.

Referential Integrity Options

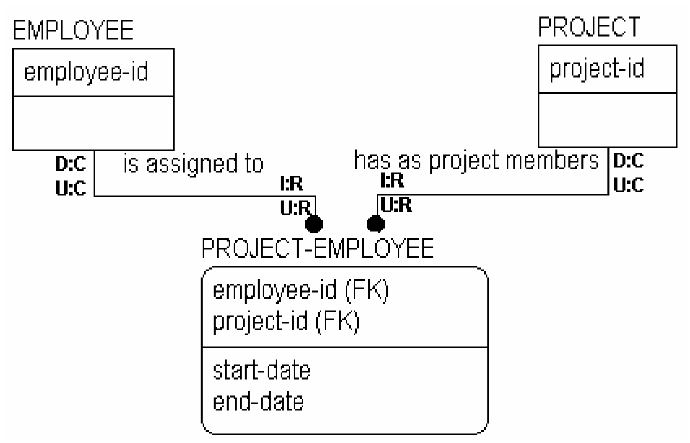
Referential integrity rules vary depending on:

- Whether or not the entity is a parent or child in the relationship
- The database action that is implemented

As a result, in each relationship there are six possible actions for which referential integrity can be defined:

- PARENT INSERT
- PARENT UPDATE
- PARENT DELETE
- CHILD INSERT
- CHILD UPDATE
- CHILD DELETE

The following figure shows referential integrity rules in the EMPLOYEE-PROJECT model:

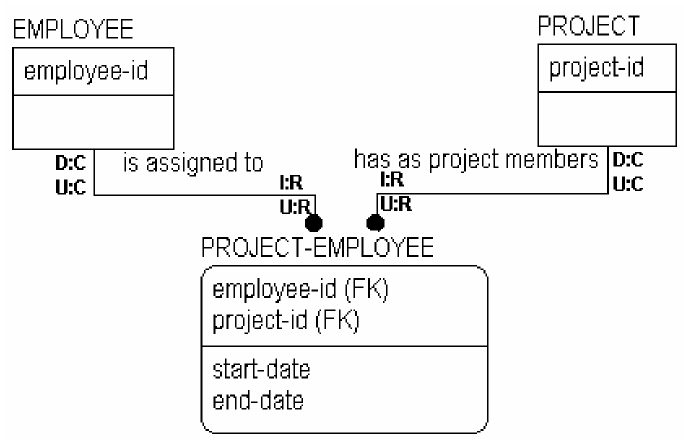


The referential integrity rules captured in the figure show the business decision to cascade all deletions in the PROJECT entity to the PROJECT-EMPLOYEE entity. This rule is called PARENT DELETE CASCADE, and is noted in the figure by the letters *D:C* placed at the parent end of the specified relationship. The first letter in the referential integrity symbol always refers to the database action: I(Insert), U(Update), or D(Delete). The second letter refers to the referential integrity option: C(Cascade), R(Restrict), SN(Set Null), and SD(Set Default).

In the figure, no referential integrity option was specified for PARENT INSERT, so referential integrity for insert (I:) is not displayed on the diagram.

RI, Cardinality, and Identifying Relationships

In the figure below, the relationship between PROJECT and PROJECT-EMPLOYEE is identifying. Therefore, the valid options for referential integrity for the parent entity in the relationship, PROJECT, include Cascade and Restrict:



Cascade indicates that all instances of PROJECT-EMPLOYEE that are affected by the deletion of an instance of PROJECT should also be deleted. Restrict indicates that a PROJECT cannot be deleted until all instances of PROJECT-EMPLOYEE that have inherited its key have been deleted. If there are any left, the Delete is restricted.

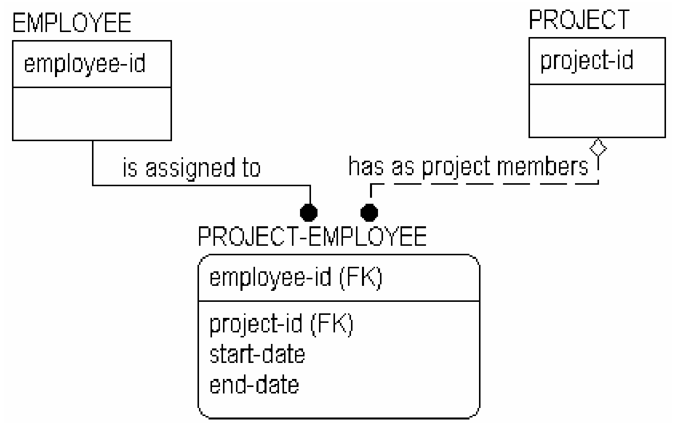
One reason to restrict the deletion might be that the business needs to know other facts about a PROJECT-EMPLOYEE such as the date started on the project. If you Cascade the Delete, you lose this supplementary information.

When you update an instance in the parent entity, the business has also determined that the updated information should cascade to the related instances in the child entity.

As you can see in the example, different rules apply when an instance is inserted, updated, or deleted in the child entity. When an instance is inserted, for example, the action is set to Restrict. This rule appears as *I:R* placed next to the child entity in the relationship. This means that an instance can be added to the child entity *only* if the referenced foreign key matches an existing instance in the parent entity. So, you can insert a new instance in PROJECT-EMPLOYEE only if the value in the key field matches a key value in the PROJECT entity.

RI, Cardinality, and Non-Identifying Relationships

If the business decides that PROJECT-EMPLOYEEs are not existence- or identification-dependent on PROJECT, you can change the relationship between PROJECT and PROJECT-EMPLOYEE to optional, non-identifying. In this type of relationship, the referential integrity options are very different:



Since a foreign key contributed across a non-identifying relationship is allowed to be NULL, one of the referential integrity options you can specify for PARENT DELETE is Set Null. Set Null indicates that if an instance of PROJECT is deleted, then any foreign key inherited from PROJECT in a related instance in PROJECT-EMPLOYEE should be set to NULL. The Delete does not cascade as in our previous example, and it is not prohibited (as in Restrict). The advantage of this approach is that you can preserve the information about the PROJECT-EMPLOYEE while effectively breaking the connection between the PROJECT-EMPLOYEE and PROJECT.

Use of Cascade or Set Null should reflect business decisions about maintaining the historical knowledge of relationships, represented by the foreign keys.

Additional Relationship Types

As you develop a logical model, you may find some parent/child relationships that do not fall into the standard, one-to-many relationships. These relationship exceptions include:

Many-to-many relationships

A relationship where one entity <owns> many instances of a second entity, and the second entity also <owns> many instances of the first entity. For example, an EMPLOYEE <has> one or more JOB TITLEs, and a JOB TITLE <is applied to> one or more EMPLOYEEs.

N-ary relationships

A simple one-to-many relationship between two entities is termed binary. When a one-to-many relationship exists between two or more parents and a single child entity, it is termed an *n-ary relationship*.

Recursive relationships

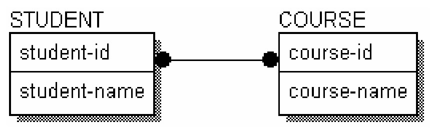
Entities that have a relationship to themselves take part in recursive relationships. For example, for the EMPLOYEE entity, you could include a relationship to show that one EMPLOYEE <manages> one or more EMPLOYEEs. This type of relationship is also used for bill-of-materials structures, to show relationships between parts.

Subtype relationships

Related entities are grouped together so that all common attributes appear in a single entity, but all attributes that are not in common appear in separate, related entities. For example, the EMPLOYEE entity could be subtyped into FULL-TIME and PART-TIME.

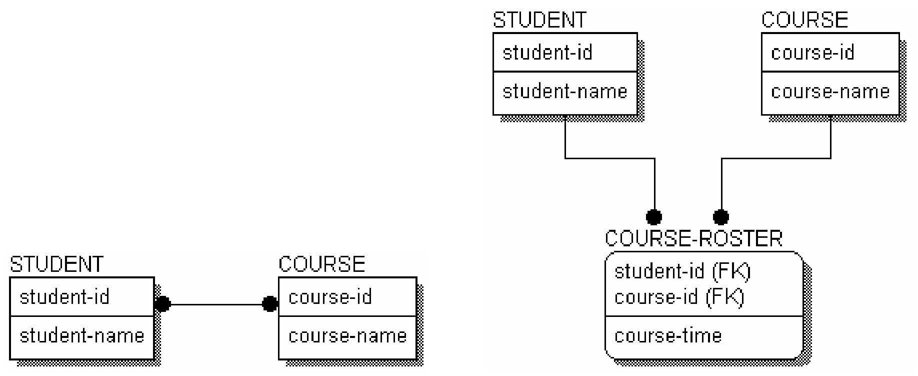
Many-to-Many Relationships

In key-based and fully-attributed models, relationships must relate zero or one instances in a parent entity to a specific set of instances in a child entity. As a result of this rule, many-to-many relationships that were discovered and documented in an ERD or earlier modeling phase must be broken down into a pair of one-to-many relationships.



This figure shows a many-to-many relationship between STUDENTs and COURSEs. If you did not eliminate the many-to-many relationship between COURSE and STUDENT, the key of COURSE would be included in the key of STUDENT, and the key of STUDENT would be included in the key of COURSE. Since COURSEs are identified by their own keys, and likewise for STUDENTs this, creates an endless loop.

You can eliminate a many-to-many relationship by creating an associative entity. In the following figure, the many-to-many relationship between STUDENT and COURSE is resolved by adding the COURSE-ROSTER entity.

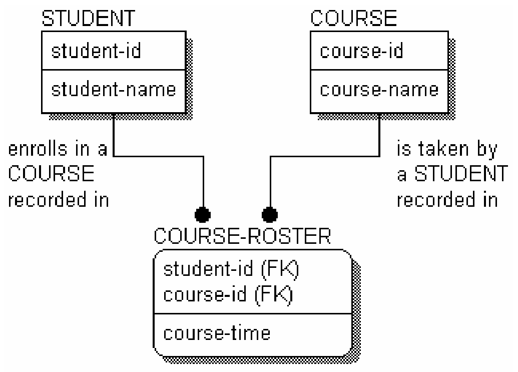


COURSE-ROSTER is an associative entity, which means it is used to define the association between two related entities.

Many-to-many relationships often hide meaning. In the diagram with a many-to-many relationship, you know that a STUDENT enrolls in many COURSEs, but no information is included to show how. When you resolve the many-to-many relationship, you see not only how the entities are related, but uncover additional information, such as the "course-time," which also describes facts about the relationship.

Once the many-to-many relationship is resolved, you are faced with the requirement to include relationship verb phrases that validate the structure. There are two ways to do this: construct new verb phrases or use the verb phrases as they existed for the many-to-many relationship. The most straightforward way is to continue to read the many-to-many relationship, through the associative entity. Therefore, you can read A STUDENT <enrolls in> many COURSEs and A COURSE <is taken by> many STUDENTs. Many modelers adopt this style for constructing and reading a model.

There is another style, which is equally correct, but a bit more cumbersome. The structure of the model is exactly the same, but the verb phrases are different, and the model is read in a slightly different way:

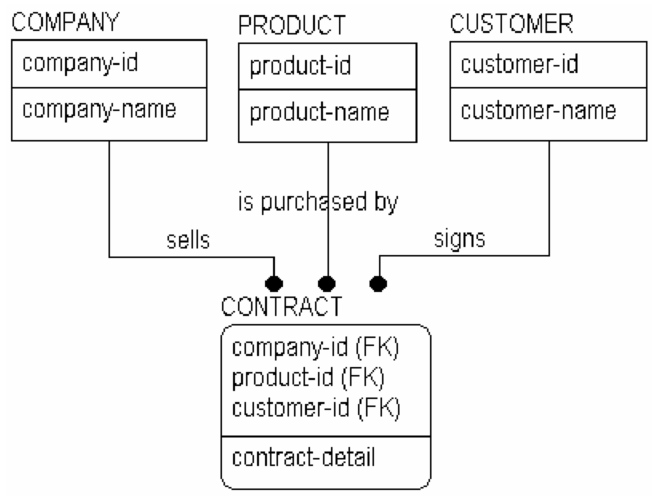


You would read: A STUDENT <enrolls in a COURSE recorded in> one or more COURSE-ROSTERs, and A COURSE <is taken by a STUDENT recorded in> one or more COURSE-ROSTERs.Although the verb phrases are now quite long, the reading follows the standard pattern; reading directly from the parent entity to the child.

Whichever style you choose, be consistent. Deciding how to record verb phrases for many-to-many relationships is not too difficult when the structures are fairly simple, as in these examples. However, this can become more difficult when the structures become more complex, such as when the entities on either side of the associative entities are themselves associative entities, which are there to represent other many-to-many relationships.

N-ary Relationships

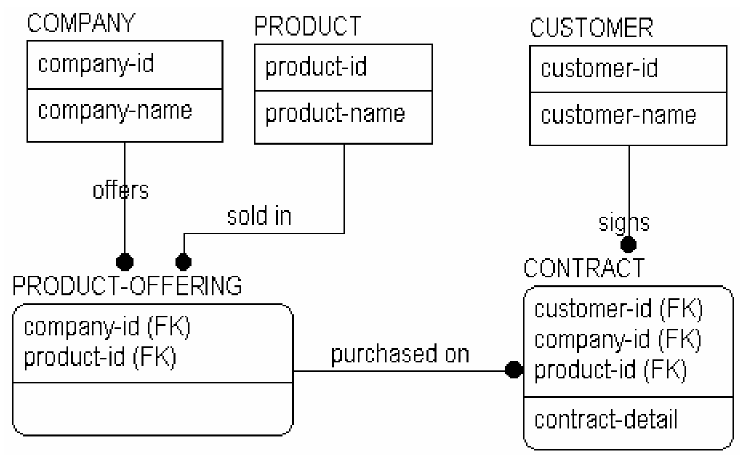
When a single parent-child relationship exists, the relationship is called binary. All of the previous examples of relationships to this point have been binary relationships. However, when creating a data model, it is not uncommon to come across n-ary relationships, the modeling name for relationships between two or more parent entities and a single child table. An example of an n-ary relationship is shown in the following figure:



Like many-to-many relationships, three-, four-, or n-ary relationships are valid constructs in entity relationship diagrams. Also like many-to-many relationships, n-ary relationships should be resolved in later models using a set of binary relationships to an associative entity.

If you consider the business rule stated in the figure, you can see that a CONTRACT represents a three-way relationship among COMPANY, PRODUCT, and CUSTOMER. The structure indicates that many COMPANYs sell many PRODUCTs to many CUSTOMERs. When you see a relationship like this, however, there are business questions that should be answered. For example, “Must a product be offered by a company before it can be sold?” “Can a customer establish a single contract including products from several different companies?” and, “Do you need to keep track of which customers 'belong to' which companies?” Depending on the answers, the structures may change.

For example, if a product must be offered by a company before it can be sold, then you would have to change the structure as follows:



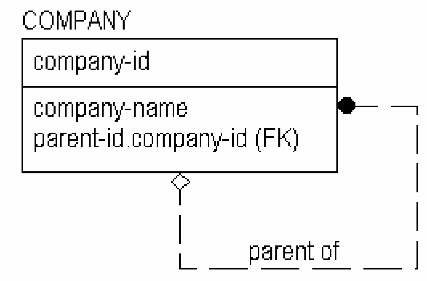
Since PRODUCTs must be offered by COMPANYs, you can create an associative entity to capture this relationship. As a result, the original three-way relationship to CONTRACT is replaced by two, two-way relationships.

By asking a variety of business questions, it is likely that you will find that most n-ary relationships can be broken down into a series of relationships to associative entities.

Recursive Relationships

An entity can participate in a recursive relationship (also called *fishhook*) where the same entity is both the parent and the child. This relationship is an important one when modeling data originally stored in legacy DBMSs such as IMS or IDMS that use recursive relationships to implement bill of materials structures.

For example, a COMPANY can be the parent of other COMPANYs. As with all non-identifying relationships, the key of the parent entity appears in the data area of the child entity. See the following figure:



The recursive relationship for COMPANY includes the diamond symbol to indicate that the foreign key can be NULL, such as when a COMPANY has no parent. Recursive relationships must be both optional (diamond) and non-identifying.

The "company-id" attribute is migrated through the recursive relationship, and appears in the example with the rolename "parent-id." There are two reasons for this. First, as a general design rule, an attribute cannot appear twice in the same entity under the same name. Thus, to complete a recursive relationship, you must provide a rolename for the migrated attribute.

Second, the attribute "company-id" in the key, which identifies each instance of COMPANY, is not the same thing as the "company-id" migrated through the relationship, which identifies the parent COMPANY. You cannot use the same definition for both attributes, so the migrated attribute must be rolenamed. An example of possible definitions follows:

company-id

The unique identifier of a COMPANY.

parent-id

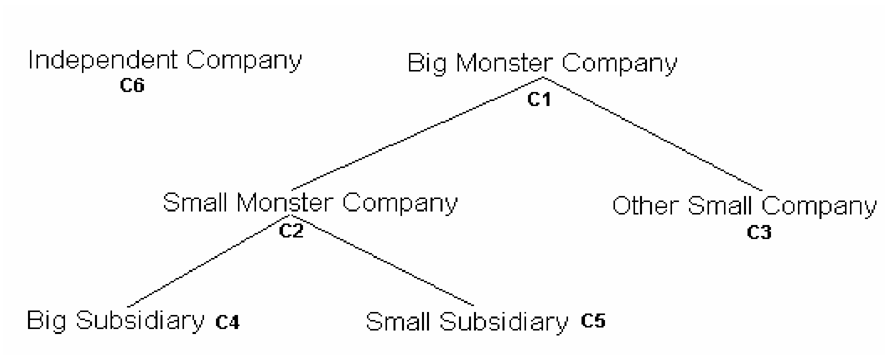
The "company-id" of the parent COMPANY. Not all COMPANYs have a parent COMPANY.

If you create a sample instance table, such as the one that follows, you can test the rules in the relationship to ensure that they are valid.

COMPANY

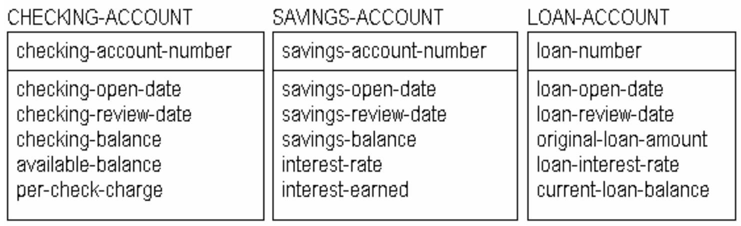
company-id	parent-id	company-name
C1	NULL	Big Monster Company
C2	C1	Smaller Monster Company
C3	C1	Other Smaller Company
C4	C2	Big Subsidiary
C5	C2	Small Subsidiary
C6	NULL	Independent Company

The sample instance table shows that Big Monster Company is the parent of Smaller Monster Company and Other Smaller Company. Smaller Monster Company, in turn, is the parent of Big Subsidiary and Small Subsidiary. Independent Company is not the parent of any other company and has no parent. Big Monster Company also has no parent. If you diagram this information hierarchically, you can validate the information in the table, as shown in the figure below:



Subtype Relationships

A subtype relationship, also referred to as a generalization category, generalization hierarchy, or inheritance hierarchy, is a way to group a set of entities that share common characteristics. For example, you might find during a modeling effort that several different types of ACCOUNTs exist in a bank such as checking, savings, and loan accounts, as shown in the figure below:

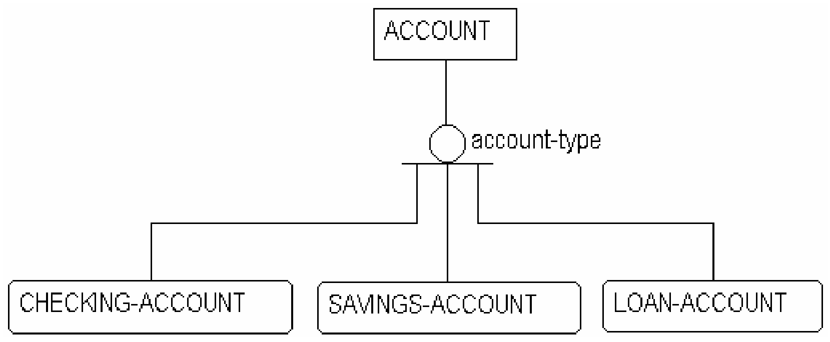


When you recognize similarities among the different independent entities, you may be able to collect attributes common to all three types of accounts into a hierarchical structure.

You can move these common attributes into a higher level entity called the *supertype entity* (or *generalization entity*). Those that are specific to the individual account types remain in the subtype entities. In this example, you can create a supertype entity called ACCOUNT to represent the information that is common across the three types of accounts. The supertype ACCOUNT includes a primary key of "account-number."

Three subtype entities, CHECKING-ACCOUNT, SAVINGS-ACCOUNT, and LOAN-ACCOUNT, are added as dependent entities that are related to ACCOUNT using a subtype relationship.

The result is a structure like the one shown in the figure below:



In this figure, an ACCOUNT is either a CHECKING-ACCOUNT, a SAVINGS-ACCOUNT, or a LOAN-ACCOUNT. Each subtype entity is an ACCOUNT and inherits the properties of ACCOUNT. The three different subtype entities of ACCOUNT are mutually exclusive.

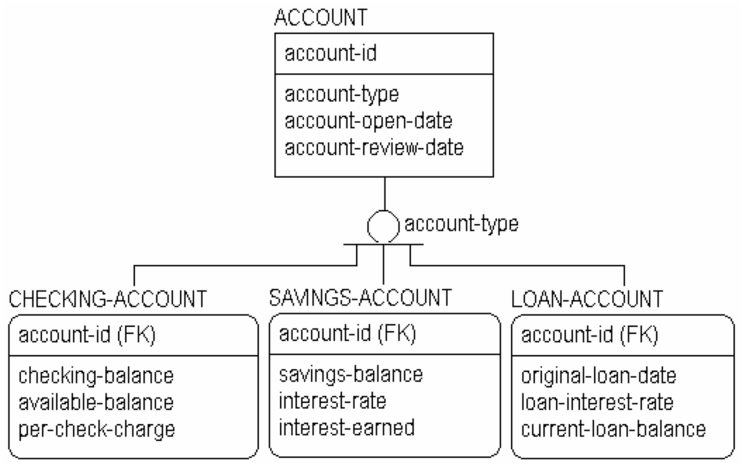
In order to distinguish one type of ACCOUNT from another, you can add the attribute "account-type" as the subtype discriminator. The subtype discriminator is an attribute of the category supertype (ACCOUNT) and its value will tell you which type of ACCOUNT it is.

Once you have established the subtype relationship, you can examine each attribute in the original model, in turn, to determine if it should remain in the subtype entities, or move to the supertype. For example, each subtype entity has an "open-date." If the definitions of these three kinds of "open-date" are the same, you can move them to the supertype, and drop them from the subtype entities.

You must analyze each attribute in turn to determine if it remains in the subtype entity or moves to the supertype entity. In those cases where a single attribute appears in only some of the subtype entities, you face a more difficult decision. You can either leave the attribute with the subtype entities or move the attribute up to the supertype. If this attribute appears in the supertype, the value of the attribute in the supertype will be NULL when the attribute is not included in the corresponding subtype entity.

Which alternative to choose depends on how many of the subtype entities share the common attribute. If most do, it is good practice, at higher level models, to move them up. If few subtype entities share the attribute, it is best to leave them where they are. In lower level models, depending on the purpose, it is often appropriate to leave the attributes in their subtype entity.

After analysis, the resulting model might appear as follows:

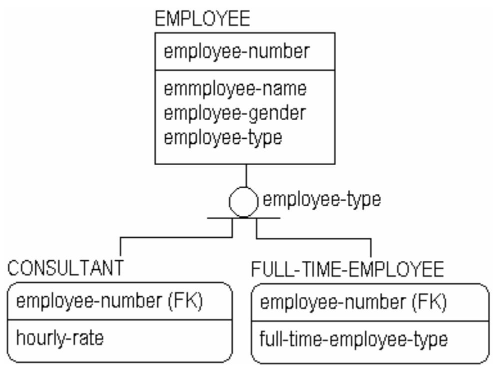


When developing a subtype relationship, you must also be aware of any specific business rules that you need to impose at the subtype level that are not pertinent to other subtypes of the supertype. For example, LOAN accounts are deleted after they reach a zero balance. You would not want to delete CHECKING and SAVINGS accounts under the same conditions.

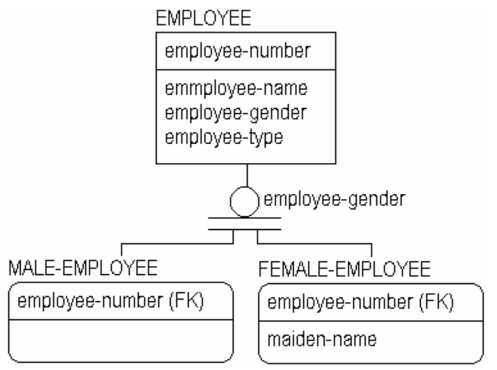
There can also be relationships that are meaningful to a single subtype and not to any other subtype in the hierarchy. For example, the LOAN entity needs to be examined, to ensure that any previous relationships to records of customer payments or assets are not lost because of a different organizational structure.

Complete Compared to Incomplete Subtype Structures

In IDEF1X, different symbols are used to specify whether or not the set of subtype entities in a subtype relationship is fully defined. An incomplete subtype indicates that the modeler feels there may be other subtype entities that have not yet been discovered. An incomplete subtype is indicated by a *single line* at the bottom of the subtype symbol, as shown in the figure below:



A complete subtype indicates that the modeler is certain that all possible subtype entities are included in the subtype structure. For example, a complete subtype could capture information specific to male and female employees, as shown in the figure below. A complete subtype is indicated by *two lines* at the bottom of the subtype symbol.



When you create a subtype relationship, it is a good rule to also create a validation rule for the discriminator. This helps to ensure that all subtypes have been discovered. For example, a validation rule for “account-type” might include: C=checking account, S=savings account, L=loans. If the business also has legacy data with account types of “O,” the validation rule uncovers the undocumented type and lets you decide if the “O” is a symptom of poor design in the legacy system or a real account type that you forgot.

Inclusive and Exclusive Relationships

Unlike IDEF1X, IE notation does not distinguish between complete and incomplete subtype relationships. Instead, IE notation documents whether the relationship is *exclusive* or *inclusive*. However, IDEF1X notation distinguishes between complete and incomplete; exclusive and inclusive.

In an exclusive subtype relationship, each instance in the supertype can relate to one and only one subtype. For example, you might model a business rule that says an employee can be either a full-time or part-time employee but not both. To create the model, you can include an EMPLOYEE supertype entity with FULL-TIME and PART-TIME subtype entities and a discriminator attribute called "employee-status." In addition, you can constrain the value of the discriminator to show that valid values for it include *F* to denote full-time and *P* to denote part-time.

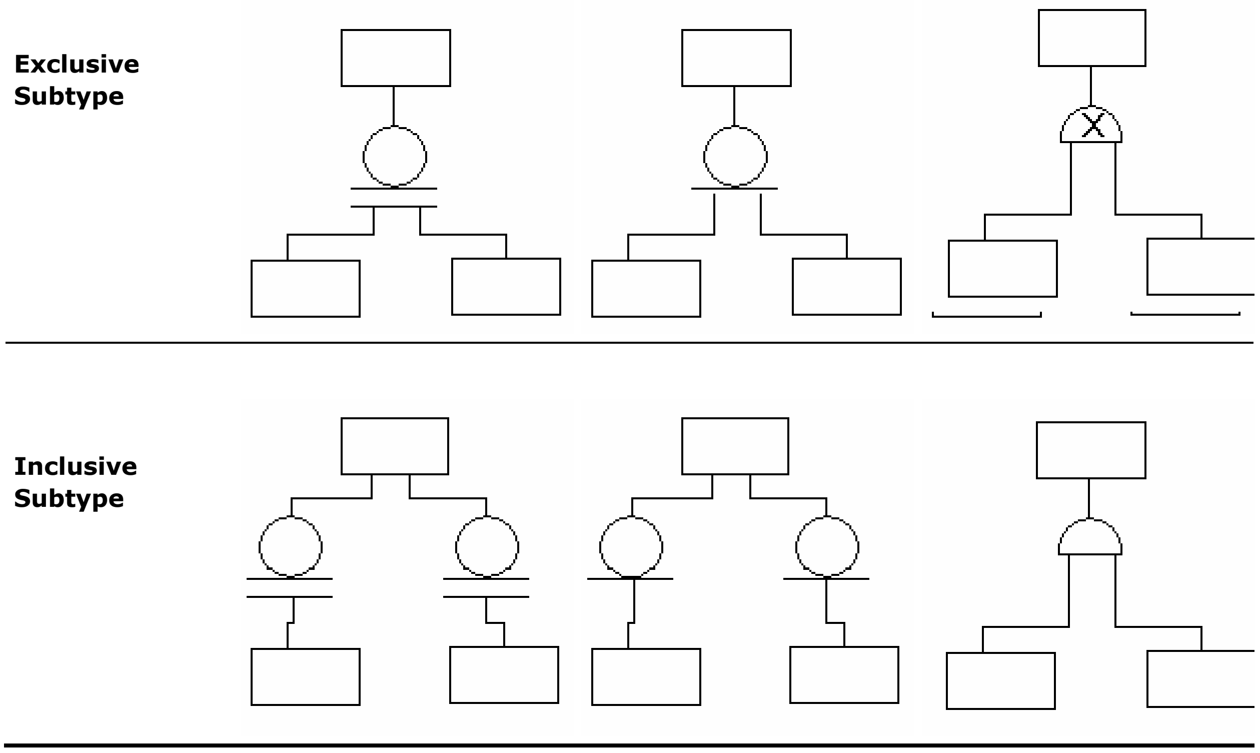
In an inclusive subtype relationship, each instance in the supertype can relate to one or more subtypes. In our example, the business rule might now state that an employee could be full-time, part-time, or both. In this example, you can constrain the value of the discriminator to show that valid values for it include *F* to denote full-time, *P* to denote part-time, and *B* to denote both full-time and part-time.

Note: In IDEF1X notation, you can represent inclusive subtypes by drawing a separate relationship between the supertype entity and each subtype entity.

IDEF1X and IE Subtype Notation

The following illustrates subtype notation in IDEF1X and IE:

IDEF1X Subtype Notation		IE Subtype Notation
Complete	**Incomplete**	



When to Create a Subtype Relationship

You should create a subtype relationship when:

- Entities share a common set of attributes. This was the case in our previous examples.
- Entities share a common set of relationships. This has not been explored but, referring back to the account structure, you can, as needed, collect any common relationships that the subtype entities had into a single relationship from the generic parent. For example, if each account type is related to many CUSTOMERs, you can include a single relationship at the ACCOUNT level, and eliminate the separate relationships from the individual subtype entities.
- Business model demands that the subtype entities should be exposed in a model (usually for communication or understanding purposes) even if the subtype entities have no attributes that are different, and even if they participate in no relationships distinct from other subtype entities. Remember that one of the major purposes of a model is to assist in communication of information structures, and if showing subtype entities assists communication, then show them.

Chapter 7: Normalization Problems and Solutions

This section contains the following topics:

Normalization

Normalization, in relational database design, is the process by which data in a relational construct is organized to minimize redundancy and non-relational constructs. Following the rules for normalization, you can control and eliminate data redundancy by removing all model structures that provide multiple ways to know the same fact.

The goal of normalization is to ensure that there is only one way to know a fact. A useful slogan summarizing this goal is:

ONE FACT IN ONE PLACE!

Overview of the Normal Forms

The following are the formal definitions for the most common normal forms.

Functional Dependence (FD)

Given an entity E, attribute B of E is functionally dependent on attribute A of E if and only if each value of A in E has associated with it precisely one value of B in E (at any one time). In other words, A uniquely determines B.

Full Functional Dependence

Given an entity E, an attribute B of E is fully functionally dependent on a set of attributes A of E if and only if B is functionally dependent on A and not functionally dependent on any proper subset of A.

First Normal Form (1NF)

An entity E is in 1NF if and only if all underlying values contain only atomic values. Any repeating groups (that might be found in legacy COBOL data structures, for example) must be eliminated.

Second normal Form (2NF)

An entity E is in 2NF if it is in 1NF and every non-key attribute is fully dependent on the primary key. In other words, there are no partial key dependencies-dependence is on the entire key K of E and not on a proper subset of K.

Third Normal Form (3NF)

An entity E is in 3NF if it is in 2NF and no non-key attribute of E is dependent on another non-key attribute. There are several equivalent ways to express 3NF. Another way is: An entity E is in 3NF if it is in 2NF and every non-key attribute is non-transitively dependent on the primary key. A final way is: An entity E is in 3NF if every attribute in E carries a fact about all of E (2NF) and only about E (as represented by the entity's entire key and only by that key). One way to remember how to implement 3NF is using the following quip: “Each attribute relies on the key, the whole key, and nothing but the key, so help me Codd!”

Beyond 3NF lie three more normal forms, Boyce-Codd, Fourth, and Fifth. In practice, third normal form is the standard. At the level of the physical database design, choices are usually made to *denormalize* a structure in favor of performance for a certain set of transactions. This may introduce redundancy in the structure, but it is often worth it.

Common Design Problems

Many common design problems are a result of violating one of the normal forms. Common problems include:

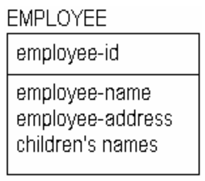
- Repeating data groups
- Multiple use of the same attribute
- Multiple occurrences of the same fact
- Conflicting facts
- Derived attributes
- Missing information

When you work on eliminating design problems, the use of sample instance data can be invaluable in discovering many normalization errors.

Repeating Data Groups

Repeating data groups can be defined as lists, repeating elements, or internal structures inside an attribute. This structure, although common in legacy data structures, violates first normal form and must be eliminated in an RDBMS model. An RDBMS cannot handle variable-length repeating fields because it offers no ability to subscript through arrays of this type. The entity below contains a repeating data group, “children's-names.” Repeating data groups violate first normal form, which basically states that an entity is in first normal form if each of its attributes has a single meaning and not more than one value for each instance.

Repeating data groups, as shown below, present problems when defining a database to contain the actual data. For example, after designing the EMPLOYEE entity, you are faced with the questions, “How many children's names do you need to record?” “How much space should you leave in each row in the database for the names?” and “What will you do if you have more names than remaining space?”

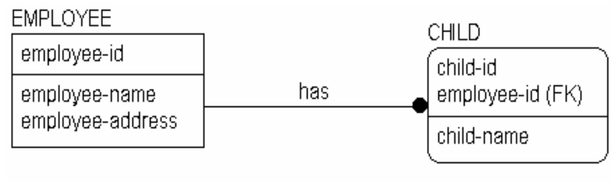


The following sample instance table might clarify the problem:

EMPLOYEE

emp-id	emp-name	emp-address	children's-names
E1	Tom	Berkeley	Jane
E2	Don	Berkeley	Tom, Dick, Donna
E3	Bob	Princeton	-
E4	John	New York	Lisa
E5	Carol	Berkeley	-

In order to fix the design, it is necessary to somehow remove the list of children's names from the EMPLOYEE entity. One way to do this is to add a CHILD table to contain the information about employee's children, as follows:



Once that is done, you can represent the names of the children as single entries in the CHILD table. In terms of the physical record structure for employee, this can resolve some of your questions about space allocation, and prevent wasting space in the record structure for employees who have no children or, conversely, deciding how much space to allocate for employees with families.

The following tables are the sample instance tables for the EMPLOYEE-CHILD model:

EMPLOYEE

emp-id	emp-name	emp-address
E1	Tom	Berkeley
E2	Don	Berkeley
E3	Bob	Princeton
E4	Carol	Berkeley

CHILD

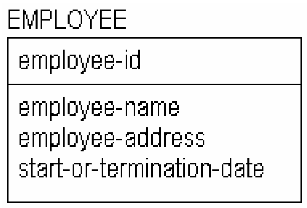
emp-id	child-id	child-name
E2	C1	Tom
E2	C2	Dick

E2	C3	Donna
E4	C1	Lisa

This change makes the first step toward a normalized model; conversion to first normal form. Both entities now contain only fixed-length fields, which are easy to understand and program.

Multiple Use of the Same Attribute

It is also a problem when a single attribute can represent one of two facts, and there is no way to understand which fact it represents. For example, the EMPLOYEE entity contains the attribute "start-or-termination-date" where you can record this information for an employee as follows:



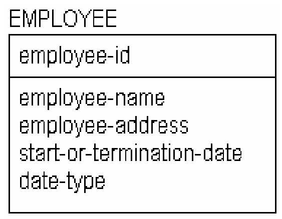
The following sample instance table shows start-or-termination date:

EMPLOYEE

emp-id	emp-name	emp-address	start-or-termination-date
E1	Tom	Berkeley	January 10, 2004
E2	Don	Berkeley	May 22, 2002
E3	Bob	Princeton	March 15, 2003
E4	John	New York	September 30, 2003
E5	Carol	Berkeley	April 22, 2000
E6	George	Pittsburgh	October 15, 2002

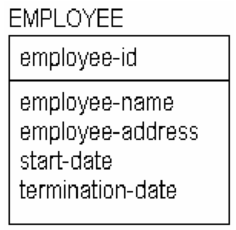
The problem in the current design is that there is no way to record both a start date, *the date that the EMPLOYEE started work*, and a termination date, *the date on which an EMPLOYEE left the company*, in situations where both dates are known. This is because a single attribute represents two different facts. This is also a common structure in legacy COBOL systems, but one that often resulted in maintenance nightmares and misinterpretation of information.

The solution is to allow separate attributes to carry separate facts. The following figure is an attempt to correct the problem. It is still not quite right. To know the start date for an employee, for example, you have to derive what kind of date it is from the "date-type" attribute. While this may be efficient in terms of physical database space conservation, it creates confusion with query logic.



In fact, this solution actually creates a different type of normalization error, since "date-type" does not depend on "employee-id" for its existence. This is also poor design since it solves a technical problem, but does not solve the underlying business problem-how to store two facts about an employee.

When you analyze the data, you can quickly determine that it is a better solution to let each attribute carry a separate fact, as in the following figure:



The following table is a sample instance table showing "start-date" and "termination-date":

EMPLOYEE

emp-id	**emp-name**	**emp-address**	**start-date**	**termination-date**
E1	Tom	Berkeley	January 10, 2004	-
E2	Don	Berkeley	May 22, 2002	-
E3	Bob	Princeton	March 15, 2003	-

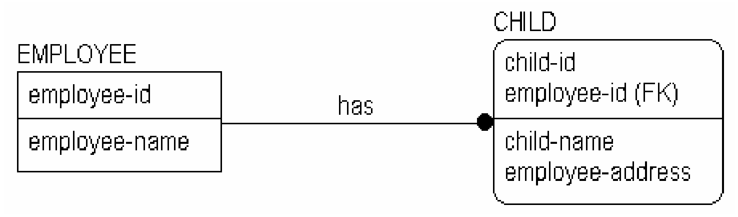
E4	John	New York	September 30, 2003	-
E5	Carol	Berkeley	April 22, 2000	-
E6	George	Pittsburgh	October 15, 2002	Nov 30, 2003

Each of the two previous situations contained a first normal form error. By changing the structures, an attribute now appears only once in the entity and carries only a single fact. If you make sure that all the entity and attribute names are singular and that no attribute can carry multiple facts, you have taken a large step toward assuring that a model is in first normal form.

Multiple Occurrences of the Same Fact

One of the goals of a relational database is to maximize data integrity. To do so, it is important to represent each fact in the database once and only once, otherwise errors can begin to enter into the data. The only exception to this rule (one fact in one place) is in the case of key attributes, which can appear multiple times in a database. The integrity of keys, however, is managed using referential integrity.

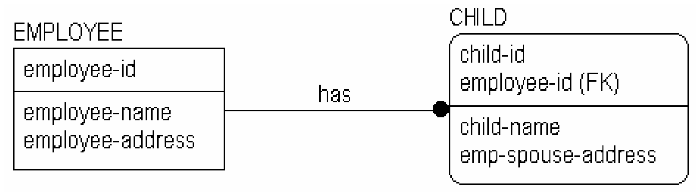
Multiple occurrences of the same fact often point to a flaw in the original database design. In the following figure, you can see that including "employee-address" in the CHILD entity has introduced an error in the database design. If an employee has multiple children, the address must be maintained separately for each child.



"employee-address" is information about the EMPLOYEE, not information about the CHILD. In fact, this model violates second normal form, which states that each fact must depend on the entire key of the entity in order to belong to the entity. The example above is not in second normal form because "employee-address" does not depend on the entire key of CHILD, only on the "employee-id" portion, creating a partial key dependency. If you place "employee-address" back with EMPLOYEE, you can ensure that the model is in at least second normal form.

Conflicting Facts

Conflicting facts can occur for a variety of reasons, including violation of first, second, or third normal forms. An example of conflicting facts occurring through a violation of second normal form is shown in the following figure:



The following two tables are sample instance tables showing "emp-spouse-address":

EMPLOYEE

emp-id	emp-name	emp-address
E1	Tom	Berkeley
E2	Don	Berkeley
E3	Bob	Princeton
E4	Carol	Berkeley

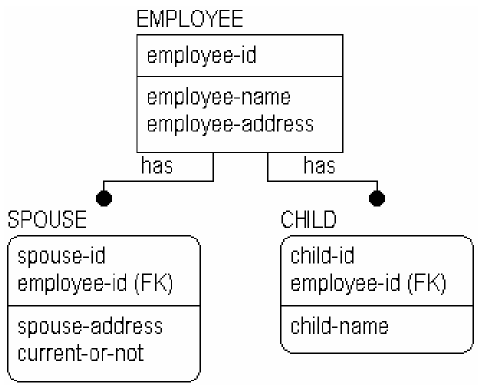
CHILD

emp-id	child-id	child-name	emp-spouse-address
E1	C1	Jane	Berkeley
E2	C1	Tom	Berkeley
E2	C2	Dick	Berkeley
E2	C3	Donna	Cleveland
E4	C1	Lisa	New York

The attribute named “emp-spouse-address” is included in CHILD, but this design is a second normal form error. The instance data highlights the error. As you can see, Don is the parent of Tom, Dick, and Donna but the instance data shows two different addresses recorded for Don's spouse. Perhaps Don has had two spouses (one in Berkeley, and one in Cleveland), or Donna has a different mother from Tom and Dick. Or perhaps Don has one spouse with addresses in both Berkeley and Cleveland. Which is the correct answer? There is no way to know from the model as it stands. Business users are the only source that can eliminate this type of semantic problem, so analysts need to ask the right questions about the business to uncover the correct design.

The problem in the example is that “emp-spouse-address”is a fact about the EMPLOYEE's SPOUSE, not about the CHILD. If you leave the structure the way it is now, then every time Don's spouse changes address (presumably along with Don), you will have to update that fact in multiple places; once in each CHILD instance where Don is the parent. If you have to update multiple places, you might miss some and get errors.

Once it is recognized that “emp-spouse-address” is a fact not about a child, but about a spouse, you can correct the problem. To capture this information, you can add a SPOUSE entity to the model, as shown in the following figure:



The following three tables are sample instance tables reflecting the SPOUSE Entity:

EMPLOYEE

emp-id	**emp-name**	**emp-address**
E1	Tom	Berkeley
E2	Don	Berkeley
E3	Bob	Princeton
E4	Carol	Berkeley

CHILD

emp-id	**child-id**	**child-name**
E1	C1	Jane
E2	C1	Tom
E2	C2	Dick
E2	C3	Donna
E4	C1	Lisa

SPOUSE

emp-id	**spouse-id**	**spouse-address**	**current-spouse**
E2	S1	Berkeley	Y
E2	S2	Cleveland	N
E3	S1	Princeton	Y
E4	S1	New York	Y
E5	S1	Berkeley	Y

In breaking out SPOUSE into a separate entity, you can see that the data for the address of Don's spouses is correct. Don has two spouses, one current and one former.

By making sure that every attribute in an entity carries a fact about *that* entity, you can generally be sure that a model is in at least second normal form. Further transforming a model into third normal form generally reduces the likelihood that the database will become corrupt; in other words, that it will contain conflicting information or that required information will be missing.

Derived Attributes

Another example of conflicting facts occurs when third normal form is violated. For example, if you included both a “birth-date” and an “age” attribute as non-key attributes in the CHILD entity, you violate third normal form. This is because “age” is *functionally dependent* on “birth-date.” By knowing “birth-date” and the date today, you can *derive* the “age” of the CHILD.

Derived attributes are those that may be computed from other attributes, such as totals, and therefore you do not need to directly store them. To be accurate, derived attributes need to be updated every time their derivation sources are updated. This creates a large overhead in an application that does batch loads or updates, for example, and puts the responsibility on application designers and coders to ensure that the updates to derived facts are performed.

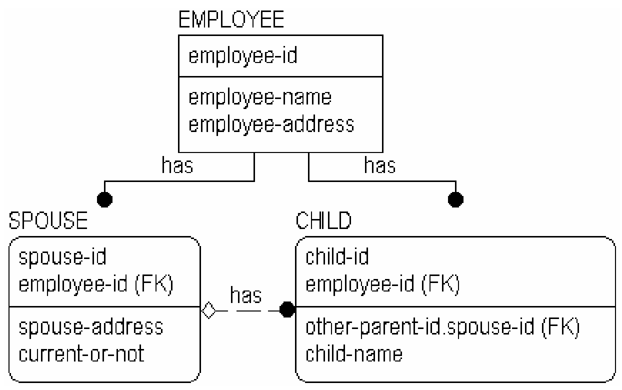
A goal of normalization is to ensure that there is only one way to know each fact recorded in the database. If you know the value of a derived attribute, and you know the algorithm by which it is derived and the values of the attributes used by the algorithm, then there are two ways to know the fact (look at the value of the derived attribute, or derive it by manual calculation). If you can get an answer two different ways, it is possible that the two answers will be different.

For example, you can choose to record both the "birth-date" and the "age"for CHILD. And suppose that the "age" attribute is only changed in the database during an end of month maintenance job. Then, when you ask the question, "How old is this CHILD?" you can directly access "age" and get an answer, or you can subtract "birth-date" from "today's-date." If you did the subtraction, you would *always* get the right answer. If "age" was not recently updated, it might give you the wrong answer, and there would *always* be the *potential for conflicting answers*.

There are situations, where it makes sense to record derived data in the model, particularly if the data is expensive to compute. It can also be very useful in discussing the model with those in the business. Although the theory of modeling says that you should never include derived data or do so only sparingly, break the rules when you must and at least record the fact that the attribute is derived and state the derivation algorithm.

Missing Information

Missing information in a model can sometimes result from efforts to normalize the data. In the example, adding the SPOUSE entity to the EMPLOYEE-CHILD model improves the design, but destroys the implicit relationship between the CHILD entity and the SPOUSE address. It is possible that the reason that "emp-spouse-address" was stored in the CHILD entity in the first place was to represent the address of the other parent of the child (which was assumed to be the spouse). If you need to know the other parent of each of the children, then you must add this information to the CHILD entity.



The following three tables are sample instance tables for EMPLOYEE, CHILD, and SPOUSE:

EMPLOYEE

emp-id	**emp-name**	**emp-address**
E1	Tom	Berkeley
E2	Don	Berkeley
E3	Bob	Princeton
E4	Carol	Berkeley

CHILD

emp-id	**child-id**	**child-name**	**other-parent-id**
E1	C1	Jane	-
E2	C1	Tom	S1
E2	C2	Dick	S1
E2	C3	Donna	S2
E4	C1	Lisa	S1

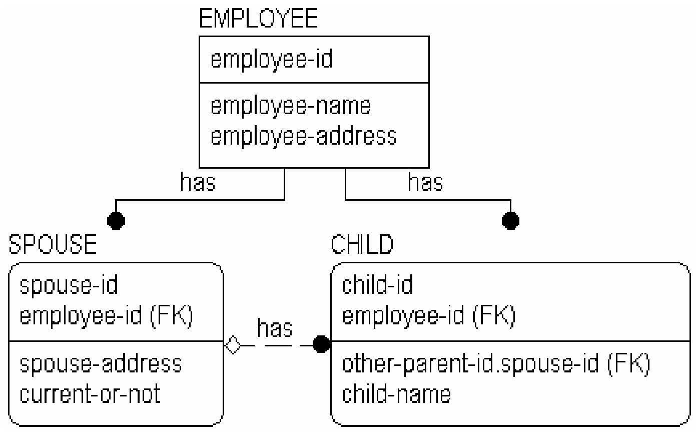
SPOUSE

emp-id	**spouse-id**	**spouse-address**	**current-or-not**
E2	S1	Berkeley	Y
E2	S2	Cleveland	N
E3	S1	Princeton	Y
E4	S1	New York	Y
E5	S1	Berkeley	Y

However, the normalization of this model is not complete. In order to complete it, you must ensure that you can represent all possible relationships between employees and children, including those where both parents are employees.

Unification

In the following example, the “employee-id” attribute migrates to the CHILD entity through two relationships: one with EMPLOYEE and the other with SPOUSE. You might expect that the foreign key attribute would appear twice in the CHILD entity as a result. Since the attribute “employee-id” was already present in the key area of CHILD, it is not repeated in the entity even though it is part of the key of SPOUSE.



This combining of two identical foreign key attributes migrated from the same base attribute through two or more relationships is called *unification*. In the example, “employee-id”was part of the primary key of CHILD (contributed by the “has” relationship from EMPLOYEE) and was also a non-key attribute of CHILD (contributed by the “has” relationship from SPOUSE). Since both foreign key attributes are the identifiers of the same EMPLOYEE, it is better that the attribute appears only once. Unification is implemented automatically when this situation occurs.

The rules used to implement unification include:

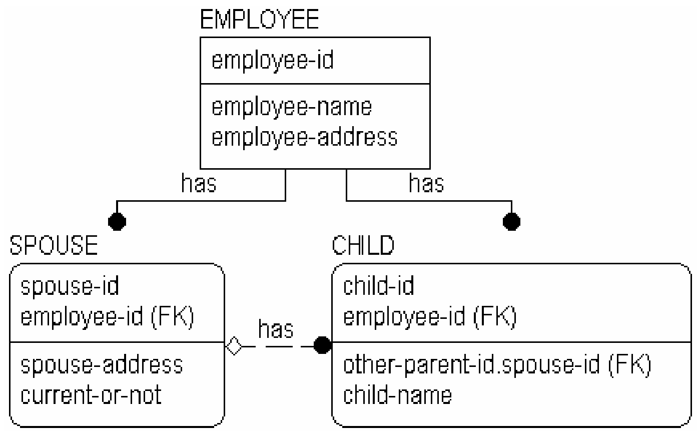
- If the same foreign key is contributed to an entity more than once, without the assignment of rolenames, then all occurrences unify.
- If the occurrences of the foreign key are given different rolenames, then unification does not occur.
- If different foreign keys are assigned the same rolename, and these foreign keys are rolenamed back to the same base attribute, then unification occurs. If they are not rolenamed back to the same base attribute, there is an error in the diagram.
- If any of the foreign keys that unify are part of the primary key of the entity, then the unified attribute remains as part of the primary key.
- If none of the foreign keys that unify are part of the primary key, then the unified attribute is not part of the primary key.

Accordingly, you can override the unification of foreign keys, when necessary, by assigning rolenames. If you want the same foreign key to appear two or more times in a child entity, you can add a rolename to each foreign key attribute.

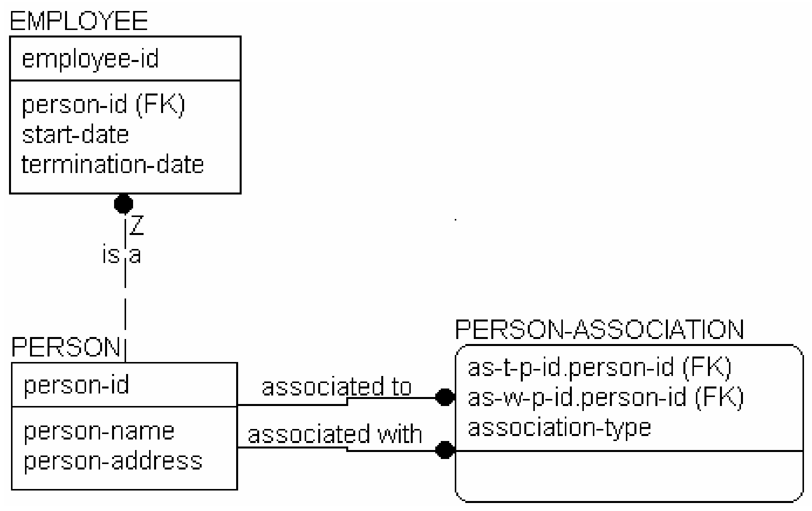
How Much Normalization Is Enough

From a formal normalization perspective (what an algorithm would find solely from the shape of the model, without understanding the meanings of the entities and attributes) there is nothing wrong with the EMPLOYEE-CHILD-SPOUSE model. However, just because it is normalized does not mean that the model is complete or correct. It still may not be able to store all of the information that is needed or it may store the information inefficiently. With experience, you can learn to detect and remove additional design flaws even after the pure normalization is finished.

Using the following EMPLOYEE-CHILD-SPOUSE model example, you see that there is no way of recording a CHILD whose parents are both EMPLOYEEs. Therefore, you can make additional changes to try to accommodate this type of data.



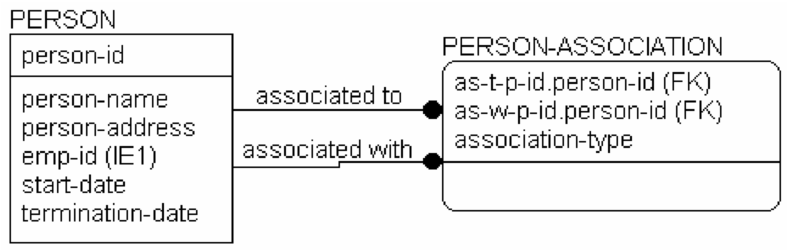
If you noticed that EMPLOYEE, SPOUSE, and CHILD all represent instances of people, you may want to try to combine the information into a single table that represents facts about people and one that represents facts about relationships. To fix the model, you can eliminate CHILD and SPOUSE, replacing them with PERSON and PERSON-ASSOCIATION. This lets you record parentage and marriage through the relationships between two PERSONs captured in the PERSON-ASSOCIATION entity.



In this structure, you can finally record any number of relationships between two PERSONs, as well as a number of relationships you could not previously record in the first model, such as adoption. The new structure automatically covers it. To represent adoption you can add a new value to the "person-association-type" validation rule to represent adopted parentage. You can also add legal guardian, significant other, or other relationships between two PERSONs later, if needed.

EMPLOYEE remains an independent entity, since the business chooses to identify EMPLOYEEs differently from PERSONs. However, EMPLOYEE inherits the properties of PERSON by virtue of the *is a* relationship back to PERSON. Notice the *Z* on that relationship and the absence of a diamond. This is a one-to-zero or one relationship that can sometimes be used in place of a subtype when the subtype entities require different keys. In this example, a PERSON either *is an* EMPLOYEE or *is not an* EMPLOYEE.

If you wanted to use the same key for both PERSON and EMPLOYEE, you can encase the EMPLOYEE entity into PERSON and allowed its attributes to be NULL whenever the PERSON is not an EMPLOYEE. You still can specify that the business wanted to look up employees by a separate identifier, but the business statements would be a bit different. This structure is shown in the following figure:



This means that a model may normalize, but still may not be a correct representation of the business. Formal normalization is important. Verifying that the model means something, perhaps with sets of sample instance tables as done here, is no less important.

Support for Normalization

Support for normalization of data models is supported, but does not currently contain a full normalization algorithm. If you have not used a real time modeling tool before, you will find the standard modeling features quite helpful. They will prevent you from making many normalization errors.

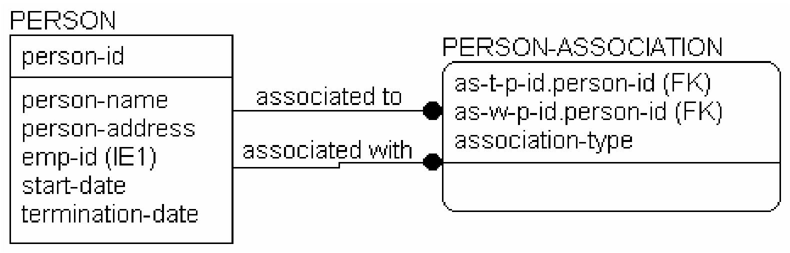
First Normal Form Support

In a model, each entity or attribute is identified by its name. Any name for an object is accepted, with the following exceptions:

- A second use of an entity name (depending on your preference for unique names) is flagged.
- A second use of an attribute name is flagged, unless that name is a rolename. When rolenames are assigned, the same name for an attribute may be used in different entities.
- You cannot bring a foreign key into an entity more than once without unifying the like columns.

By preventing multiple uses of the same name, you are prompted to put each fact in exactly one place. However, there may still be second normal form errors if you place an attribute incorrectly, but no algorithm would find that without more information than is present in a model.

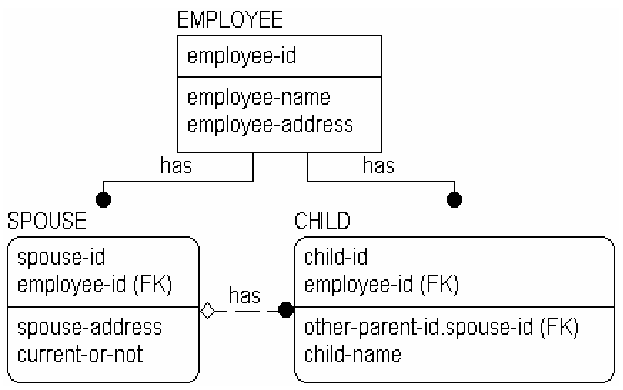
In a data model, CA ERwin DM cannot know that a name you assign to an attribute can represent a list of things. In the following example, CA ERwin DM accepts “children's-names” as an attribute name. So CA ERwin DM does not directly guarantee that every model is in first normal form.



However, the DBMS schema function does not support a data type of *list*. Since the schema is a representation of the database in a physical relational system, first normal form errors are also prevented at this level.

Second and Third Normal Form Support

CA ERwin DM does not currently manage functional dependencies, but it can help to prevent second and third normal form errors. For example, if you reconstruct the examples below, you will find that once “spouse-address” is defined as an attribute of SPOUSE, you cannot also define it as an attribute of CHILD. (Again, depending on your preference for unique names.)



By preventing the multiple occurrence of foreign keys without rolenames, you are reminded to think about what the structure represents. If the same foreign key occurs twice in the same entity, there is a business question to ask: Are we recording the keys of two separate instances, or do both of the keys represent the same instance?

When the foreign keys represent different instances, separate rolenames are needed. If the two foreign keys represent the same instance, then it is very likely that there is a normalization error somewhere. A foreign key appearing twice in an entity without a rolename means that there is a redundant relationship structure in the model. When two foreign keys are assigned the same rolename, unification occurs.

Chapter 8: Physical Models

This section contains the following topics:

Objective

The objective of a physical model is to provide a database administrator with sufficient information to create an efficient physical database. The physical model also provides a context for the definition and recording (in the data dictionary) of the data elements that form the database, and assists the application team in choosing a physical structure for the programs that will access the data. To ensure that all information system needs are met, physical models are often developed jointly by a team representing the data administration, database administration, and application development areas.

When it is appropriate for the development effort, the model can also provide the basis for comparing the physical database design against the original business information requirements to:

- Demonstrate that the physical database design adequately supports those requirements.
- Document physical design choices and their implications, such as what is satisfied, and what is not.
- Identify database extensibility capabilities and constraints.

Support for the Roles of the Physical Model

Support is provided for both roles of a physical model:

- Generating the physical database
- Documenting physical design against the business requirements

For example, in a logical/physical model, you can create a physical model from an ERD, key-based, or fully attributed model simply by changing the view of the model from Logical Model to Physical Model. Each option in the logical model has a corresponding option in the physical model. Therefore, each entity becomes a relational table, attributes become columns, and keys become indices.

Once the physical model is created, you can generate all model objects in the correct syntax for the selected target server directly to the catalog of the target server, or indirectly as a schema DDL script file.

Summary of Logical and Physical Model Components

The following table summarizes the relationship between objects in a logical and a physical model:

Logical Model	**Physical Model**
Entity	Table
Dependent entity	Foreign Key is part of the child table's Primary Key
Independent entity	Parent table or, if it is a child table, Foreign Key is NOT part of the child table's Primary Key
Attribute	Column
Logical datatype (text, number, datetime, blob)	Physical datatype (valid example varies depending on the target server selected)
Domain (logical)	Domain (physical)
Primary key	Primary key, Primary Key Index
Foreign key	Foreign key, Foreign Key Index
Alternate key (AK)	Alternate Key Index-a unique, non-primary index

Logical Model	**Physical Model**
Inversion entry (IE)	Inversion entry Index-a non-unique index created to search table information by a non-unique value, such as customer last name.
Key group	Index
Business rule	Trigger or stored procedure
Validation rule	Constraint
Relationship	Relationship implemented using Foreign Keys
Identifying relationship	Foreign Key is part of the child table's Primary Key (above the line)
Non-identifying relationship	Foreign Key is NOT part of the child table's Primary Key (below the line)
Subtype relationship	Denormalized tables
Many-to-many relationship	Associative table
Referential Integrity relationship (Cascade, Restrict, Set Null, Set Default)	INSERT, UPDATE, and DELETE Triggers
Cardinality relationship	INSERT, UPDATE, and DELETE Triggers
N/A	View or view relationship
N/A	Prescript or postscript

Referential integrity is a part of the logical model, since the decision about how to maintain a relationship is a business decision. Referential integrity is also a physical model component, since triggers or declarative statements appear in the schema. Referential integrity is supported as a part of both the logical and physical models.

Denormalization

You can also *denormalize* the structure of the logical model, or allow data redundancy in a table to improve query performance so that you can build a related physical model that is designed effectively for the target RDBMS. Features supporting denormalization include:

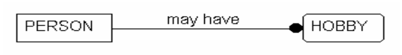
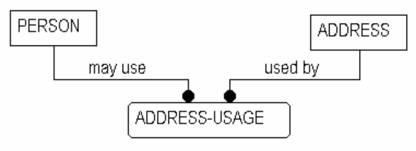
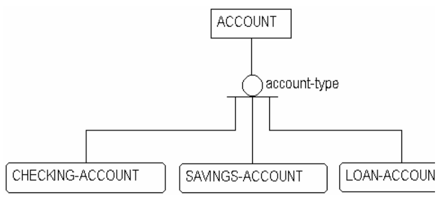
- *Logical only* properties for entities, attributes, key groups, and domains. You can mark any item in the logical model logical only so that it appears in the logical model, but does not appear in the physical model. For example, you can use the logical only settings to denormalize subtype relationships or support partial key migration in the physical model.
- *Physical only* properties for tables, columns, indexes, and domains. You can mark any item in the physical model physical only so that it appears in the physical model only. This setting also supports denormalization of the physical model since it enables the modeler to include tables, columns, and indexes in the physical model that directly support physical implementation requirements.
- Resolution of many-to-many relationships in a physical model. Support for resolving many-to-many relationships is provided in both the logical and physical models. If you resolve the many-to-many relationship in the logical model, the associative entity is created and lets you add additional attributes. If you choose to keep the many-to-many relationship in the logical model, you can still resolve the relationship in the physical model. The link is maintained between the original logical design and the new physical design, so the origin of the associative table is documented in the model.

Appendix A: Dependent Entity Types

This section contains the following topics:

Classification of Dependent Entities

The following table lists the types of dependent entities that may appear in an IDEF1X diagram:

Dependent Entity Type	Description	Example
Characteristic	A characteristic entity represents a group of attributes that occur multiple times for an entity, and is not directly identified by any other entity. In the example, HOBBY is a characteristic of PERSON.	
Associative or Designative	Associative and designative entities record multiple relationships between two or more entities. If the entity carries only the relationship information, it is termed a designative entity. If it also carries attributes that further describe the relationship, it is called an associative entity. In the example, ADDRESS-USAGE is an associative or designative entity.	
Subtype	Subtype entities are the dependent entities in a subtype relationship. In the example, CHECKING-ACCOUNT, SAVINGS-ACCOUNT, and LOAN-ACCOUNT are subtype entities.	 ACCOUNT account-type CHECKING-ACCOUNT SAVINGS-ACCOUNT LOAN-ACCOUN

Glossary

alternate key

An attribute or attributes that uniquely identify an instance of an entity. If more than one attribute or group of attributes uniquely identify an instance of an entity, the alternate keys are those attributes or groups of attributes not selected as the primary key. A unique index for each alternate key is generated.

attribute

Represents a type of characteristic or property associated with a set of real or abstract things (people, places, events, and so on).

basename

The original name of a rolenamed foreign key.

binary relationship

A relationship where exactly one instance of the parent is related to zero, one, or more instances of a child. In IDEF1X, identifying, non-identifying, and subtype relationships are all binary relationships.

BLOB

A dbspace that is reserved for storage of the byte and text data that makes up binary large objects, or BLOBs, stored in table columns. The BLOB dbspace can hold images, audio, video, long text blocks, or any digitized information.

cardinality

The ratio of instances of a parent to instances of a child. In IDEF1X, the cardinality of binary relationships is 1:n, where n can be one of the following:

- Zero, one, or more (signified by a blank space)
- One or more (signified by the letter P)
- Zero or one (signified by the letter Z)
- Exactly n (where n is some number)

complete subtype cluster

If the subtype cluster includes all of the possible subtypes (every instance of the generic parent is associated with one subtype), then the subtype cluster is complete. For example, every ACCOUNT is either a checking, savings, or loan account and therefore the subtype cluster of CHECKING-ACCOUNT, SAVINGS-ACCOUNT, or LOAN-ACCOUNT is a complete subtype cluster.

dependent entity

An entity whose instances cannot be uniquely identified without determining its relationship to another entity or entities.

denormalization

To allow data redundancy in a table to improve query performance.

discriminator

The value of an attribute in an instance of the generic parent determines to which of the possible subtypes that instance belongs. This attribute is known as the discriminator. For example, the value in the attribute "account-type" in an instance of ACCOUNT determines to which particular subtype (CHECKING-ACCOUNT, SAVINGS-ACCOUNT, or LOAN-ACCOUNT) that instance belongs.

domain

A group of predefined logical and physical property characteristics that can be saved, selected, and then attached to attributes and columns.

entity

An entity represents a set of real or abstract things (people, places, events, and so on) that have common attributes or characteristics. Entities can be either independent or dependent.

foreign key

An attribute that has migrated through a relationship from a parent entity to a child entity. A foreign key represents a secondary reference to a single set of values; the primary reference is the owned attribute.

identifying relationship

A relationship where an instance of the child entity is identified through its association with a parent entity. The primary key attributes of the parent entity become primary key attributes of the child.

incomplete subtype cluster

If the subtype cluster does not include all of the possible subtypes (every instance of the generic parent is not associated with one subtype), then the subtype cluster is incomplete. For example, if some employees are commissioned, a subtype cluster of SALARIED-EMPLOYEE and PART-TIME EMPLOYEE is incomplete.

independent entity

An entity whose instances can be uniquely identified without determining its relationship to another entity.

inversion entry

An attribute or attributes that do not uniquely identify an instance of an entity, but are often used to access instances of entities. A non-unique index for each inversion entry is generated.

logical model

The data modeling level where you create a conceptual model that contains objects such as entities, attributes, and key groups.

logical/physical model

A model type created where the logical and physical models are automatically linked.

non-key attribute

Any attribute that is not part of the entity’s primary key. Non-key attributes can be part of an inversion entry or alternate key, and can also be foreign keys.

non-identifying relationship

A relationship where an instance of the child entity is not identified through its association with a parent entity. The primary key attributes of the parent entity become non-key attributes of the child.

non-specific relationship

Both parent-child connection and subtype relationships are considered specific relationships since they define precisely how instances of one entity relate to instances of another. However, in the initial development of a model, it is often helpful to identify non-specific relationships between two entities. A non-specific relationship, also referred to as a many-to-many relationship, is an association between two entities where each instance of the first entity is associated with zero, one, or many instances of the second entity and each instance of the second entity is associated with zero, one, or many instances of the first entity.

normalization

The process by which data in a relational construct is organized to minimize redundancy and non-relational constructs.

physical model

The data modeling level where you add database and database management system (DBMS) specific modeling information such as tables, columns, and datatypes.

primary key

An attribute or attributes that uniquely identify an instance of an entity. If more than one attribute or group of attributes can uniquely identify each instance, the primary key is chosen from this list of candidates based on its perceived value to the business as an identifier. Ideally, primary keys should not change over time and should be as small as possible. A unique index for each primary key is generated.

referential integrity

The assertion that the foreign key values in an instance of a child entity have corresponding values in a parent entity.

rolename

A new name for a foreign key. A rolename is used to indicate that the set of values of the foreign key is a subset of the set of values of the attribute in the parent, and performs a specific function (or role) in the entity.

schema

The structure of a database. Usually refers to the DDL (data definition language) script file. DDL consists of CREATE TABLE, CREATE INDEX, and other statements.

specific relationship

A specific relationship is an association between entities where each instance of the parent entity is associated with zero, one, or many instances of the child entity, and each instance of the child entity is associated with zero or one instance of the parent entity.

subtype entity

There are often entities which are specific types of other entities. For example, a SALARIED EMPLOYEE is a specific type of EMPLOYEE. Subtype entities are useful for storing information that only applies to a specific subtype. They are also useful for expressing relationships that are only valid for that specific subtype, such as the fact that a SALARIED EMPLOYEE qualifies for a certain BENEFIT, while a PART-TIME-EMPLOYEE does not. In IDEF1X, subtypes within a subtype cluster are mutually exclusive.

subtype relationship

A subtype relationship (also known as a categorization relationship) is a relationship between a subtype entity and its generic parent. A subtype relationship always relates one instance of a generic parent with zero or one instance of the subtype.

Index