

# ADVANCED CUSTOMIZATION GUIDE

September 2018, Release 1.1

This document contains information which is the property of Dialogue Technologies. All rights are reserved. Dialogue Technologies does not guarantee or in any way represent that the information is accurate and/or complete and assumes no responsibility for any errors, omissions or other issues with the information. The information in the document is provided AS IS. Dialogue Technologies does not assume any responsibility for any consequences of using the information contained herein.

Third Edition (September 2018)

This major revision obsoletes and replaces all previous versions. This edition applies to Release 2.0 of Ergo, and to all subsequent releases and modifications until otherwise indicated in new editions. Address any comments you may have on the document to:

Dialogue Technologies AB  
Ankdammsgatan 20  
S-171 43 Solna, Sweden, or  
[info@dialoguetech.com](mailto:info@dialoguetech.com)

When you send information to Dialogue Technologies, you grant Dialogue Technologies a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

@ Copyright Dialogue Technologies AB 2018. All rights reserved.

## CONTENTS

1. Overview of Ergo .....	9
1.1 The Query Interface .....	10
1.2 The Natural Language Engine .....	10
1.3 The Customization Tool .....	11
1.4 The need for Dialogue Technologies' Ergo .....	12
1.5 Database-centric and corpus-centric applications .....	13
1.5.1 Database-centric applications .....	13
1.5.2 Corpus-centric applications .....	13
1.5.3 Comparison between database-centric and corpus-centric applications .....	14
2. The Ergo application development cycle .....	16
2.1 Planning an Ergo project .....	16
2.2 Determining the target users .....	16
2.3 Gathering a representative corpus .....	17
2.4 Studying the RDBMS implementation .....	18
2.5 Identifying application areas .....	18
2.6 Studying the database design .....	19
2.7 Identifying application options .....	20
2.8 Selecting the application .....	20
2.9 Pre-customizing .....	20
2.10 Customizing .....	21
2.11 Testing and evaluation .....	21
2.12 Maintaining the application .....	22
2.12.1 User logs .....	22
3. Studying the database design .....	23
3.1 Identifying the application .....	23
3.2 Identifying relational databases .....	23
3.3 Examining tables and columns .....	24
3.3.1 Degree of normalization .....	24
3.3.2 Primary keys .....	26
3.3.3 Inclusion dependency .....	26
3.3.4 Data-value contents .....	27
3.3.5 Coded data-value cardinality .....	28
3.4 Understand conceptual content .....	28
3.5 Selecting the application .....	29
4. Pre-customization tasks .....	30
4.1 Pre-customization strategies .....	31
4.1.1 Compiling a list of queries .....	31
4.1.2 Designing a conceptual model .....	32
4.1.3 Filling out pre-customization forms .....	32
4.1.4 A combination of these strategies .....	33
5. Using the Customization Tool .....	34
5.1 Enter DB information .....	35
5.1.1 Enter table information .....	35
5.1.2 Joins .....	37
5.1.3 Inclusion dependencies .....	39
5.1.4 Excluding columns .....	39
5.2 Create entities .....	40
5.2.1 Creating a new entity .....	41

5.2.2	Naming an entity .....	42
5.2.3	Classifying an entity .....	43
5.2.4	Subclass and instance entities .....	43
5.2.5	Consists of .....	45
5.2.6	Units of measure .....	46
5.2.7	Creating terms.....	47
5.2.8	Specifying syntax .....	48
5.2.9	Specifying an SQL statement.....	49
5.2.10	Creating verbs.....	50
5.2.11	Subclasses.....	51
5.2.12	Creating adjectives.....	51
5.2.13	Intersect .....	52
5.2.14	Instances.....	53
5.3	The relation action.....	54
5.3.1	Defining relationships.....	55
5.3.2	Creating or changing a relationship.....	55
5.3.3	Removing a relationship.....	56
5.3.4	Correcting invalid relationships. ....	56
6.	Customization Phase 1 - Processing entities and defining relationships .....	58
6.1	Processing entities .....	58
6.2	Naming the entities.....	58
6.3	Classifying the entities.....	59
6.3.1	Classifying identifiers and names.....	62
6.3.2	Specifying terms for entities .....	63
6.3.3	Types of terms .....	64
6.3.4	Specifying syntax for entities.....	65
6.3.5	Specifying SQL for entities .....	66
6.4	Defining entity/entity relationships .....	66
6.5	Defining column/table relationships.....	66
6.6	Defining identify relationships.....	68
6.7	Defining name relationships .....	68
6.8	Defining possessive relationships .....	68
6.9	Defining other conceptual relationships.....	69
6.10	Defining table/table relationships.....	69
6.11	Defining other entity/entity relationships.....	69
6.12	Non-3NF considerations.....	69
6.13	Multi-column primary key considerations .....	70
7.	Customization Phase 2 - Creating new entities .....	71
7.1	Composite entities.....	71
7.2	Creating composite entities for identifiers.....	72
7.3	Creating composite entities for names .....	73
7.4	Creating composite entities for attributes .....	74
7.5	Creating composite entities representing embedded concepts .....	74
7.6	Creating composite report entities.....	74
7.7	Creating composite report entities with numerical components .....	75
7.8	Creating composite sorters .....	75
7.9	Creating dummy table entities .....	76
7.10	Creating verbs .....	78
7.10.1	Defining the verb complement.....	78
7.10.2	Specifying prepositional complements .....	79

7.10.3	Defining verb relationships .....	81
7.11	Creating passive verbs .....	82
8.	Customization Phase 3 - Extending the conceptual model.....	84
8.1	Creating subclass entities .....	84
8.2	Creating identifiers for subtypes .....	85
8.3	Creating instance entities .....	85
8.4	Writing DB images.....	86
8.5	Creating adjectives.....	87
8.5.1	Prepositional complements: .....	89
8.6	Handling adverbs .....	90
8.7	Creating special information entities.....	91
8.7.1	Entities referring to DATE data.....	92
8.7.2	Entities referring to numerical data.....	92
9.	Customization Phase 4 - Enhancing the model and refining the syntax .....	94
9.1	Asking test questions .....	94
9.2	General rules of thumb for asking questions .....	95
9.3	Pre-customizing for further work.....	97
9.3.1	Identifying and categorizing questions .....	97
9.3.2	Mini-modeling.....	98
9.3.3	Depth-first customizing.....	98
9.4	Model enhancement -- adding to the model .....	98
9.5	Adding entities.....	99
9.5.1	Adding new base entities .....	99
9.5.2	Adding dummy table entities .....	99
9.5.3	Adding composite entities .....	100
9.5.4	Adding subclass entities.....	100
9.5.5	Adding instance entities .....	100
9.5.6	Create special information entities .....	101
9.5.7	Adding verbs .....	101
9.5.8	Adding adjectives .....	101
9.5.9	Adding adverbial constructs .....	101
9.6	Adding new conceptual relations.....	102
9.7	Adding language terms .....	102
9.7.1	Adding new language (syntax) relations .....	102
9.8	Types of questions .....	102
9.9	Imperatives.....	102
9.9.1	Examples of imperatives using single objects .....	104
9.9.2	Examples of imperatives using compound objects.....	104
9.9.3	Examples of imperatives using compound objects and backward reference .....	105
9.9.4	Examples of imperatives with subordinate qualifying clauses .....	105
9.9.5	Examples of imperatives using prepositional phrases.....	106
9.10	Direct questions.....	106
9.10.1	Examples of questions using single subjects .....	106
9.10.2	Examples of questions using compound subjects.....	107
9.10.3	Examples of questions using single objects .....	107
9.10.4	Examples of questions using compound objects.....	107
9.10.5	Examples of questions using single predicates.....	108
9.10.6	Examples of questions using compound predicates without direct objects	108

9.10.7	Examples of questions using single subjects and compound predicates	108
9.10.8	Examples of questions using compound subjects and compound predicates.....	109
9.10.9	Examples of questions using compound predicates and compound objects	109
9.10.10	Examples of questions using predicates with prepositions .....	110
9.10.11	Examples of questions using strings of modifying clauses .....	110
9.10.12	Examples of questions using single adjectives.....	111
9.10.13	Examples of questions using compound adjectives with single subject	111
9.10.14	Examples of questions using compound adjectives with single objects	112
9.10.15	Examples of questions using compound adjectives with compound objects	112
9.10.16	Prefaced questions.....	112
9.11	Generalized query types .....	113
9.11.1	Examples of questions using intransitive verbs.....	113
9.11.2	Examples of questions using both transitive and intransitive verbs...	113
9.11.3	Examples of questions using anaphora .....	114
9.11.4	Examples of questions using date ranges.....	114
9.11.5	Examples of questions using numerical ranges .....	114
9.11.6	Examples of questions asking for sums .....	115
9.11.7	Examples of questions using adverbials .....	115
9.11.8	Examples of questions with indirect objects.....	115
9.11.9	Examples of with deictic time expressions .....	116
9.11.10	Examples of questions using negations .....	116
9.11.11	Examples of questions using passives .....	117
9.11.12	Examples of questions using temporals .....	117
9.11.13	Examples of questions using locatives .....	117
9.11.14	Examples of questions referring to previous questions .....	118
9.11.15	Examples of questions using elliptical phrases.....	118
9.12	Testing the model.....	119
9.13	Application development speed .....	121
10.	Final test and evaluation .....	123
10.1	Testing .....	123
10.1.1	Release to target users .....	123
10.2	Application evaluation .....	124
10.3	Customization performance implications.....	124
11.	CASE 1: Joins and inclusion dependencies.....	127
12.	CASE 2: To be added .....	131
13.	CASE 3: Entities with numerical data - unit of measure.....	132
14.	CASE 4: Ranges (date functions) .....	134
15.	CASE 5: Union of 2 identical non-3NF tables .....	135
16.	CASE 6: Conceptual entities and multiple-column concepts: .....	137
17.	CASE 7: Structured entities .....	140
18.	CASE 8: Entities with numeric data - measured by .....	142
19.	CASE 9: Non-3NF tables - tables processed as verbs .....	144
20.	CASE 10: Verbs with embedded prepositions .....	146
21.	CASE 11: Adverbials .....	147

22.	CASE 12. Compound verbs with single subjects .....	148
23.	CASE 13: Non-3NF databases - ambiguous data/multiple column primary key 150	
24.	CASE 14: Subtypes of entities with multi-column keys .....	152
25.	CASE 15: Entities with numerical data – counted-by relations .....	154
26.	CASE 16: Non-3NF database: migrated hierarchical tables .....	157

Appendices

# Preface

The objectives of this document are:

1. To document our knowledge of and experiences with Dialogue Technologies Ergo application customization to enable customers to have access to all the information they need to customize their application.
2. To provide a reference document to facilitate both customization and support efforts by demonstrating how to solve specific kinds of generic customization problems and by identifying relevant product limitations for minimizing customization time.



# PART 1. Advanced Customization Guide

## 1. OVERVIEW OF ERGO

Dialogue Technologies Ergo is a query product, which uses advanced grammar-based Natural Language Processing (NLP) technologies. It provides end-users and business professionals using their own natural language, such as English, with the ability to access information stored in relational database management systems (RDBMSs) or to control applications.

The components of Ergo are:

i) Query Interface (QI) for interactions between the users and the database:

- QI for voice users
- QI for text input

ii) Natural Language Engine (NLE) with an API for processing the queries

- Natural language analyzer
- Natural language generator

iii) Customization Tool (CT) for customizing applications:

- Conceptual modeling facility
- Model transfer facilities

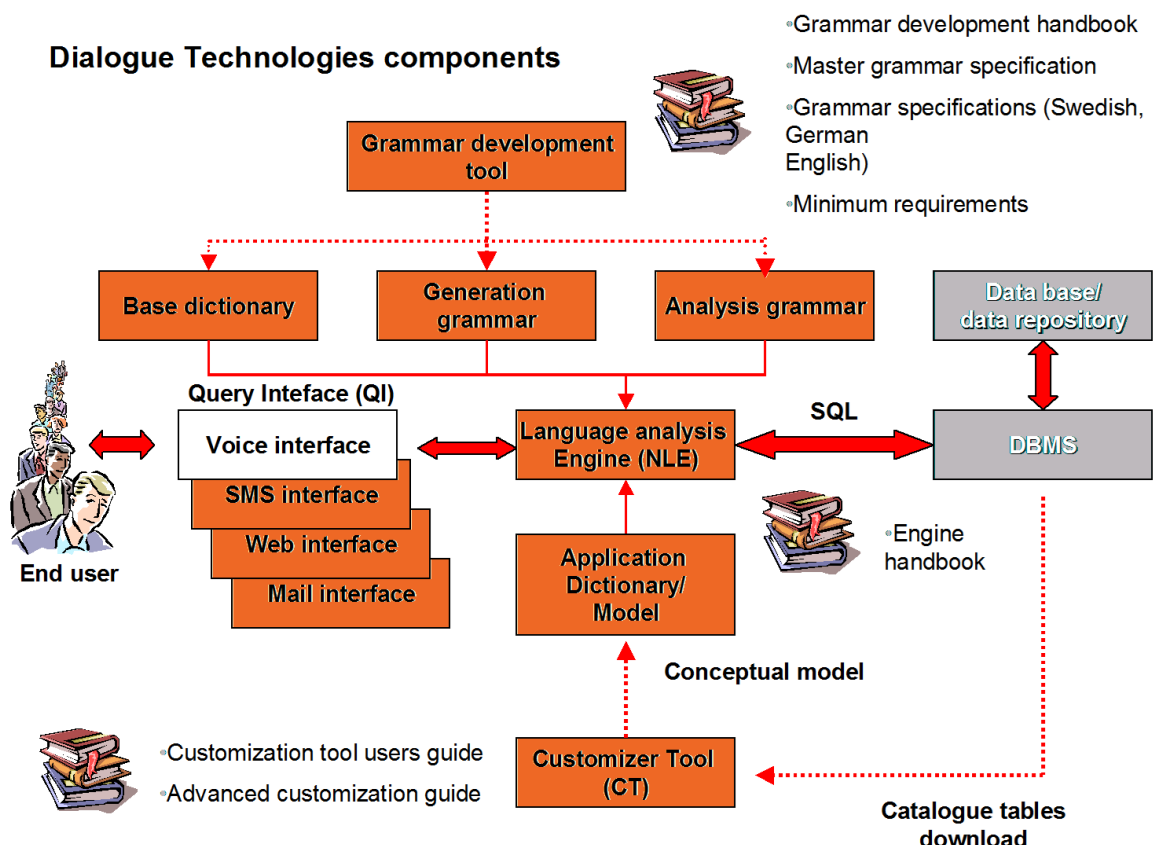


Figure 1 Components of an application

## 1.1 The Query Interface

The query interface handles communication between the user and the database. It can be text-oriented for applications with text input via e.g. the web, a special input window, SMS, e-mail, etc., or based on using a voice front-end for converting speech to text.

## 1.2 The Natural Language Engine

The Natural Language Engine (NLE) is the Ergo component that transforms queries into structured query language (SQL) statements (or extensive markup language, XML, etc.) that is sent to the database management system, which retrieves the answer. The input to this component is a natural language query. The output is the SQL command into which the input query has been translated as well as a set of paraphrases of the input query.

The NLE is designed with an Application Programming Interface (API) so it can be used as an embedded component in other products. In principle, the API can be used by any program. It is up to the calling program to handle user interactions, to use the generated SQL commands for retrieving data, and to answer users' questions based on the data and the information returned from the API.

When an end user submits a natural language query, it is parsed. Valid sentences are translated to an internal format using an analysis grammar for the language of the query. The vocabulary that is used for parsing consists of the application-independent words contained in the built-in dictionary and the application-specific words that have been added during the customization process. The query is translated back into natural language as a paraphrase to permit the user to confirm that the query has been understood correctly by the computer. In cases of ambiguity, alternative paraphrases are presented to the user for selection of the correct one. After confirmation, the translated query is sent to the database and the answer is displayed for the user. The query can be of the yes/no sort. For example, the user types in the following query:

*Which senior manager works at the head office?*

which results in the interpretation:

*Find senior managers that work at departments named head office.*

Ergo can handle natural language questions, which cannot be expressed in a single SQL query. The natural language query is translated into an answer set, which is beyond pure SQL. Besides SQL statements, the answer set contains information on the data representation (yes/no, report) derived from the input natural language question and, in cases when the data cannot be retrieved by a single SQL query. It also includes SQL statements for creating intermediate relations. Intermediate relations are temporary relations created as SQL tables containing data to be retrieved by a query. When the user selects an interpretation, the corresponding SQL is sent off to the Database Management System (DBMS).

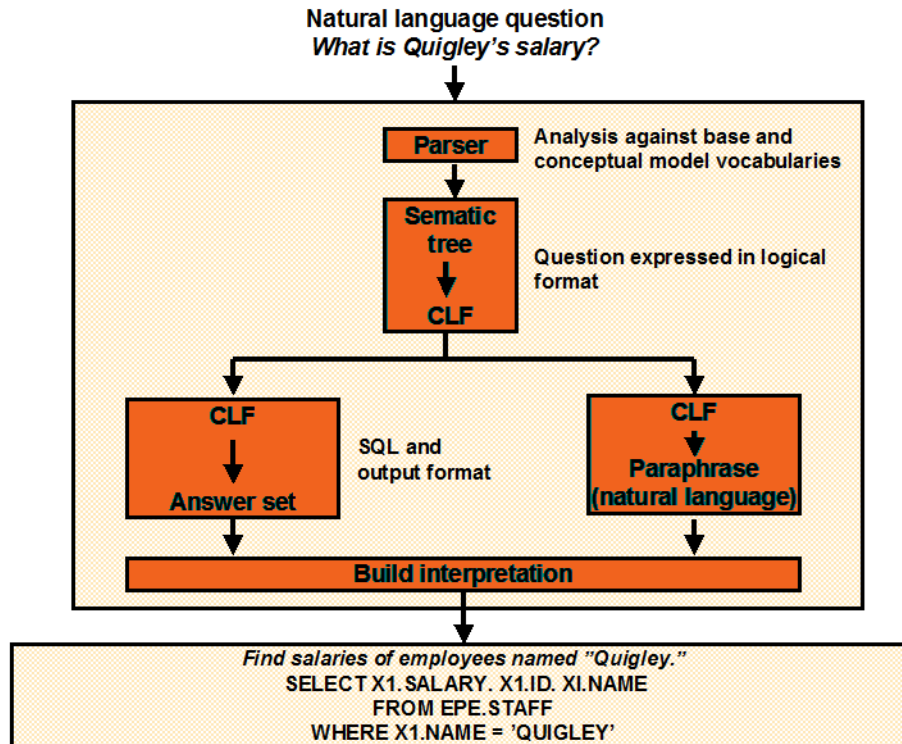


Figure 2 The Natural Language Engine

### 1.3 The Customization Tool

The Customization Tool (CT) is used to create a conceptual model by defining relations between concepts and the database tables and columns, specifying relevant natural language terms, and establishing relations between the words and the database elements. When selecting tables and columns for an application, and when processing the basic entities generated automatically by the CT, the customizer is prompted to enter information in a series of dialog boxes. The customizer selects the suggestions he wants or enters the information requested. Once this basic customization is complete, the customizer defines more complex relations in the conceptual model he or she builds.

An application in Ergo is a *conceptual model*. This conceptual model is a collection of prolog facts about the tables and columns in the database, including all information about corresponding entities, terms, and relations the NLE requires to process queries. When users ask queries, Ergo makes use of these facts to process the query and generate an answer. The task of providing Ergo with application-dependent information (i.e., an application conceptual model, which includes an application dictionary) is called customization. Applications are customized by a customizer (i.e., the one who customizes Ergo for a specific application).

A conceptual model in Ergo is used as a formal representation between the database and the language. It is connected to:

- A database through SQL-statements, and

- A language through natural language terms.

Using the Ergo CT the customizer creates a conceptual model. This model describes all the objects, which are of interest to the users. In other words, it is a model of the universe of discourse, which is selected portion of the real world or a postulated world dealt with in the application. The conceptual model is represented graphically to the customizer. The modeling technique used by Ergo is the binary ER (entity/relationship) model. This was chosen because the expressive power of the E/R approach and that it is simple and can easily address language concepts and the links between them.

The conceptual model is composed of concepts (entities) and relationships (links) between them. It is connected to the natural language terms from one side and to the database from the other side. These can either be nouns, verbs, and adjectives. Relationships are those that connect entities together which may denote possession, time, place, verb and prepositional relationships.



**Figure 3 Binary Entity Relationship Model**

In the customization process, words needed for a given application are specified in the model together with associated grammatical information. The relation of the words to each other and their relationship to information in the database is also defined. Many users can access the same customized application.

### 1.4 The need for Dialogue Technologies' Ergo

The need to use Ergo may vary. Ultimately there will be a host of applications and devices, which are controlled via voice or text commands. Initially two major application areas have been defined. These include:

- i. Command applications: These are applications where a device, application, or service is controlled by issuing commands in natural language. This way advanced operations can be performed without the user having to learn a set of complicated commands. A special case is mobile services. Voice control using natural language is one of the strongest user interfaces for mobile applications.
- ii. Customer support: Since Ergo supports asking questions in natural language the same Ergo application can service end-users both via a voice interface and a web-

based query interface. The result is significant savings in customer support costs and an increased level of service. These applications include support and help-desk applications as well as information applications.

There are certainly more areas where language-based user interfaces will become a reality over time.

The examples in the rest of this document generally reflect a data-mining application in which various decision makers in a company would like to access business-critical data from the company database. A customer support application can in this context be understood as entering frequently asked questions and manuals into an SQL database and allow end-users to query that database for information.

## 1.5 Database-centric and corpus-centric applications

There are two main types of Ergo applications, database-centric and corpus-centric. In this context a corpus is a set of type questions and answers that describe the domain, or universe of discourse (UOD), that the application should cover. A type question is a formulation of a question requesting a specific piece of information. E.g. *How long is the ship?* represents a question about the length of a particular ship. The same question can be formulated in a number of different ways (e.g. *What is the length of the ship?*, *How long is the vessel?*, etc.). Each of these requests the same information and each formulation can serve as representation for the type question.

### 1.5.1 Database-centric applications

Ergo can be used for building database-centric applications, taking its starting point in the *answers*, i.e. the existing databases. A typical example is the Yellow Pages. The Yellow Pages contain information about names, addresses, telephone numbers, etc. usually stored in a huge database. The purpose is to make this, and only this, information available to users.

In this case the knowledge domain of the application is completely defined. In other words, the answers that the application can provide are known *a priori*. As a consequence, all different questions that can be asked to retrieve these answers are limited in number, and can, in principle, be listed. Database-centric Ergo applications can thus reach a coverage of the user questions approaching 100%. The pre-study of a database-centric application is more focused on the design of the databases and how this design can be represented in the domain model, than on how questions can be formulated. This is not surprising since the database content restricts how questions can be posed.

### 1.5.2 Corpus-centric applications

In corpus-centric applications Ergo takes its' starting point in the users' *questions*, i.e. what the users actually want to know. The knowledge domain or UOD is more difficult to define in this type of applications since it is hard to anticipate *what* users would like to know and *how* they are going to ask about it. For corpus-driven applications pre-study mainly consists of:

- i) Pin-pointing the knowledge domain (*what*) and

- ii) Gathering as many authentic end-user formulations as possible about this domain (*how*).

It is very important that the collected data is as authentic, and as large as possible, since it is almost impossible to cognitively foresee user questions. This data makes up the corpus.

Based on the corpus a suitable database structure is defined, including defining tables and columns in the tables.

### 1.5.3 Comparison between database-centric and corpus-centric applications

The main difference between the two types of applications is in how the UOD is determined and how the database structure is defined.

In the database-centric approach the database, including all data records, is given from the beginning, something which largely also defines the corpus.

In the corpus-centric approach the corpus, and the database structure, is refined in a series development stages, each involving testing on a group of users. The database is populated with data during the development process. This is described in more detail in Appendix B.

	Database centric	Corpus centric
User question	Induced from database	Empirical pre-study and study of question logs
Model	Governed by database design	Governed by user questions
Database	Given	Governed by model
Knowledge domain	Identical with content of database	Combination of initial hypothesis, pre-study and analysis of user data.

**Table 1 Characteristics of the two types of applications**

A particular strength of Ergo is that both types of applications can be combined.

For example, two knowledge domains belonging to database-centric and corpus-centric applications, respectively, were combined into a single customer service application. The database-centric knowledge domain contained area codes for telephony (e.g. allowing questions like *Where does 0611 go?* and *What's the area code for Härnösand?*). The underlying database contained one table with three columns with:

- Area code (e.g. 0611),
- Area (e.g. Härnösand) and
- Country (e.g. Sweden).

The corpus-centric knowledge domain covered user questions about telecommunication products and services (e.g. allowing questions like *Can I order*

*ADSL 8.0 Mbit?* and *How often does the phone bill come?*). The structure of this database was determined by user queries.

In the combined application users could ask both about area codes and general questions about products and services, with the application querying two different databases.

## 2. THE ERGO APPLICATION DEVELOPMENT CYCLE

Development of an Ergo application usually entails a project involving participants from several parts of an organization, including target users, systems programmers, Database Administrators (DBAs), and Ergo administrators and customizers. It is useful for the project team to have an overview of the tasks required for developing an Ergo application from start to finish. This chapter discusses the things the project team, especially the *customizers* (the person building the application), must think about before starting work on an Ergo application.

### 2.1 Planning an Ergo project

In general terms, a project can be identified by these characteristics:

- A well-defined objective,
- A definite start date,
- A definite end date, and
- A well-defined end product.

For an Ergo application development project, the objective is to enable users to access relational database information by using their own natural language. Given a start date and an end date, the end product of an Ergo application development project is a fully customized application that target users can use to ask natural language questions. There are a set of standard tasks that an Ergo application development project team should consider as they set a project plan, especially the first time they use the product. These include:

1. Determining the target users
2. Gathering a representative corpus  
(in the case of corpus-centric application)
3. Studying the DBMS implementation  
(in the case of a database-centric application)
4. Identifying application areas
5. Studying/defining the database design
6. Identifying application options
7. Selecting the application
8. Pre-customizing - preparing for customization
9. Customizing
10. Testing and evaluation
11. Maintaining the application

### 2.2 Determining the target users

With knowledge of Ergo and the product, the project team should make certain that the customers' target users are well defined in terms of:

1. Their role
2. Their number
3. Their need to use Ergo.

Selecting the target users does not have to be done first, but it is normal after appreciating the advantages of Ergo to begin planning by examining who might use a



potential application. Later, after the database has been examined and the application selected, it may turn out that the original set of target users may have shifted because of the information contained in the application selected. This will vary from case to case, and it may also depend on the rigidity with which the set of target users are identified.

### **Role**

There may be different strategies and objectives if the target users are internal to the company or external, professional or private users, consumers, managers or non-managers, etc. This must also be coordinated with the number of target users involved in the project and with the size and seriousness of the application.

Where managers or organizational planners are end users, it is very likely that the information is vital for decision-making. If increased information access is only a general intention, applications may be more incrementally enhanced and on a more leisurely trial-and-error basis. Applications addressing large numbers of end-users require serious consideration.

The extent to which users are distributed over several locations should also be planned for. Because users will provide feedback to customizers, the utility of this in the context of organizational procedures must be considered.

The most demanding applications are those addressing consumers, citizens or other large groups of end-users, particularly for corpus-centric applications.

### **Number**

It is important to have an idea of the number of target users there will be. The number of users will affect:

- The vocabulary in the application
- The pre-customization strategy
- The post-project evaluation

If the number of target users is large, beyond some threshold, it may be expected that they will have less of an opportunity to directly affect the determination of the vocabulary in the application. In addition, it may be expected that there will be a broader degree of individual differences with respect to language use. This will impact the pre-customization strategy and the implementation of the application.

If the number of target users is large, it may make it more difficult to generate a complete *'list of desirable questions,'* which will necessitate an alternative strategy.

If the number of target users is large, it may be more difficult to get immediate feedback or quick answers to questionnaires that may be part of post-project evaluation.

## **2.3 Gathering a representative corpus**

In the case of a corpus-centric application the application domain, or UOD (Universe of Discourse), is not necessarily well known in advance. There are several ways to

try to determine the application domain, including:

1. Studying documentation describing the target UOD, e.g. manuals, various instruction books, etc.
2. Interviewing individuals with a knowledge about what the users might want to ask, e.g. customer service staff, people involved in analysis of consumer behavior, marketing staff, etc.
3. Focus group studies.
4. Logging and analyzing actual user queries.

Of these the last one is the most powerful, although use of 1-3 might be valuable. In either case the best method is to make an initial assessment of the corpus, deploying the system and then testing it, logging all questions, on a small group of users and the gradually increasing the corpus and UOD in successive stages. In each new stage the application is tested on a larger user group.

## 2.4 Studying the RDBMS implementation

For database-centric applications the project team may desire to focus on a particular part of their database, especially if they have more than one DBMS implementation, with one being more suitable for an Ergo application. They may also be in the process of migrating information, redesigning their databases, normalizing them, for example. Thus, the current status of their database design, together with their business objectives, will determine their needs and interests.

The application development goals will very likely vary depending on whether the project is an early test for getting familiar with the technology or on an acute need for an immediate production application.

Here security will also play a role in the selection of subsets of the database. The project team will very likely manage key confidential information very carefully.

*Note:* An Ergo application can access only one database subsystem at a time.

## 2.5 Identifying application areas

After studying the database implementation, the project team will select a set of subsets of the database for consideration for inclusion in the application development project. These are application areas.

For database-centric applications the application areas are often subsets of larger total data base structures. We are making a distinction between application areas and application options because experience with "*real life*" data bases indicate that strategic corporate information is usually organized into information subsystems - finance, personnel, marketing, R&D, inventory, purchasing, and so forth. Experience also shows that actual applications for most users will be subsets of such application areas, unless a senior manager would like an entire database or application area

customized. The ability to modularize applications efficiently would seem to render the customization of a very large database unnecessary because some kinds of questions linking particular sets of tables and columns would never be asked. For example, we would probably never ask:

*What was the bottom line for the green bolts in the administration dept?*

A question such as this would imply linking tables and columns from finance, personnel, and inventory application areas. Of course, linkages between these application areas are conceivable, but practice shows the usefulness of making a distinction between application areas and application options. For application areas we are assuming that:

1. The data base(s) selected will serve as application area superset(s) from which a set of alternative application area subset(s) can be identified
2. There can, but need not be, more than one application area selected
3. The selection of the application area depends solely on the design of the data base and user needs of the information system
4. The application area itself is a superset of a set possible application options related to that area.

The identification of application areas often depends on the storage structure of the relational data base management system and the particular way the local Database Administrator (DBA) has structured his information. The selection of these application areas will depend on:

- The application development goals
- The needs and interests of Ergo users the organizational security interests
- The design of the database.

## **2.6 Studying the database design**

Studying the database design is a major task the project team has to do. Knowledge of the database is also important for the customizer. When studying the database design, the project team must answer these questions:

1. Is the database truly relational?
2. How many tables and columns are there?
3. In what normal form is it?
4. Does every table have a primary key?
5. Is there a preponderance of numerical data?
6. Is there a lot of coded data?
7. What kind of data values are dates?

In the next chapter we take up these questions in detail.

In the case of corpus-centric applications the project team must design the structure of the database, including defining tables and columns, based on the available corpus. The data is entered in the tables during the customization process. In this case it is important create a structure that can be expanded and refined as we get an

increasingly representative corpus.

## 2.7 Identifying application options

After identifying the database and studying its structure, it may be the case that a number of application options can be identified. This often depends on this size of the database. When there are a lot of tables, usually more than 15 or 20, it becomes apparent that subsets of the entire database can form applications of their own. User access to information is also important. In some organizations, users may not have access to an entire database. They may be restricted to accessing selected tables, columns, or views. In such cases, it is useful to identify all the application options and prioritize them both with respect to urgency for the organization and with respect to the speed the application can be developed.

## 2.8 Selecting the application

After the application options have been identified, the project team selects a set of tables and columns for customization. Criteria for selecting an application may include these:

- Priority of the application option
- Project duration
- Impact on organization.

Among the factors determining the project's duration are:

- Anticipated pre-customization time
- Anticipated customization time
- Projected time for introduction of the application into production.
- Projected period of end-user evaluation

Preparing for customization will take time. So will customizing the application and promoting for use by target users. And the users will use the product a given period before generating feedback to the customizer, perhaps for enhancing the application.

Finally, an application will have an impact on the organization.

## 2.9 Pre-customizing

After selecting an application, a clearly understood pre-customization strategy should be adopted. The customizer has to prepare for customization, especially if the application is large. There are at least four possible pre-customization strategies:

1. Compiling a query list
2. Drawing a conceptual model
3. Filling out customization forms
4. Some combination of these.

These approaches will be examined in a separate chapter.

## 2.10 Customizing

There are various phases and strategies of customization. There are four main, task-oriented phases of customization:

1. Phase 1: Processing entities and defining relationships
2. Phase 2: Creating new entities
3. Phase 3: Extending the conceptual model
4. Phase 4: Enhancing the model and refining the syntax.

Each of these phases will be examined in detail. When working developing an Ergo application, it is possible to conceive of two main customization strategies:

1. A Breadth-First Strategy
2. A Depth-First Strategy

**A Breadth-First Strategy:** A breadth-first strategy consists of concluding each customization step before proceeding with the next one. Very likely, a customizer's first customization effort will proceed in this fashion, especially if he customizes a database where answers to queries must be obtained after crossing many tables.

*Note:* We recommend starting your application with a breadth-first approach. This consists of the easiest tasks. Once completed, the work done on these tasks can be saved, and it will not have to be done again.

**A Depth-First Strategy:** A depth-first customization strategy consists of processing an individual cluster of entities as far as possible before moving on to the next cluster of entities. 'As far as possible' means that there is a limit to the 'depth' one can initially customize because the identification and definition of conceptual and language relations depends on the prior processing of other entities. Very likely, some experience will be required before a depth-first strategy can be successfully carried out.

*Note:* A depth-first strategy is useful when working on the model-enhancement phase of customization.

## 2.11 Testing and evaluation

Testing occurs in the process of application development as well as by end users after the application has been released to them. When testing and evaluating, we make sure that:

- Queries we expect to be processed are actually processed
- The SQL generated by the queries is correct
- Queries not processed are documented and sent to the customizer for implementation.

The application can be released incrementally to the target users, or the customizer may choose to achieve a given level of customization and language coverage before releasing it to users.

## 2.12 Maintaining the application

The number of questions you can ask in any natural language is infinite. This means that the development of an application will never be 'final'. We must accept what is practical and useful. It is also important that we recognize that applications can be enhanced incrementally. This makes proper maintenance of the application important.

Databases may also change. Usually, an application must be re-customized if the database normalization level has changed, or if its size has diminished. But if tables have been added to the database, or if some columns have been added to a table, the effort to update the application is easier, especially if the application is well maintained. To maintain an application:

Make sure you back up completed work you know is correct.

Give the latest correct model a recognizably different name from your working models to save time and confusion. Document queries that do not work, including any error messages they may generate. Document any solutions to difficult tasks you find. Do not let too much time pass between CT sessions, especially while working on an on-going project. Make plans for a back-up customizer and make sure pertinent information has been passed on to him/her.

Keeping these things in mind will contribute to the successful completion of a Ergo application development project.

### 2.12.1 User logs

All questions asked to the system are logged and can be analyzed with the analysis tool LogView. LogView, among other things, logs all questions that did not receive an answer and analyses and sorts them. These questions can then be used for detailed analysis and decision about how the application should be updated.

## 3. STUDYING THE DATABASE DESIGN

Studying the database design is the first major customization task. Understanding the entire database design is necessary before you can select a database, or a subset of one, for a Ergo Application. In addition, we must study the database structure in order to determine ahead of time which customization strategy to adopt. Readjusting the database tables through the creation of views or addition of columns may also be necessary. An optimum database for an Ergo application should

- Be a relational database.
- Be in third normal form (3NF) or use views to give the appearance of 3NF.
- Contain columns that can be used to identify and name or describe each record in a table.
- Contain a preponderance of CHAR data values.
- Avoid having null values in columns unless they make sense, i.e., unless they are semantically meaningful.
- Contain as little data as possible in coded form, such as GE for Germany.
- Express dates in DATE format.
- Express numbers in numerical formats.

The customizer must know the database well to use the CT correctly. The customizer has to process entities and define relations, often based on the structure of the information found in the database. If this is not done correctly, the users will not be able to ask all the questions they expect to be able to ask.

### 3.1 Identifying the application

Once the target users have been identified, the project team focuses on identifying and selecting an application. Usually this will be a subset of a database the users want information from. Identifying appropriate application subsets of the database and selecting one of them requires focusing on the following factors:

- Identifying a true relational database (if you have more than one DBMS, or non-relational implementations)
- Examining the tables and columns, identifying their:
  - degree of normalization
  - primary keys
  - inclusion dependencies
  - data-value content
  - coded data-value cardinality
- Understanding the conceptual content of the database for correct customization
- Selecting the subset of the database that most corresponds to the needs or expectations of the target users.

### 3.2 Identifying relational databases

The fact that information is stored in tables and columns and managed by a relational database management system does not mean that the database is a relational database. Be sure your database is, in fact, relational! For Ergo, the main criterion for

identifying a relational database is this:

*Each single data value in the database has one and only one semantic interpretation.*

A relational database contains records of information stored in tabular form. Each field of each record contains a data value or is blank. Ergo assumes that the data value or the blank contains only one meaning. If this is not the case, the database cannot be said to be relational.

For example, migrating from a hierarchical Data Base (DB) to tables and columns in a RDBMS, while storing the hierarchical sequential key, will lead to storing information in data fields that contain more than one meaning - here, a functional meaning and a meaning with respect to a place in the hierarchy. After some database migrations, the hierarchical sequential key is broken down into its components. Of course, Ergo can work with these databases; but some of the intended meaning may be lost or difficult to customize, especially if complicated referential integrity relations become built into the design the information is migrated to. In short, migrated information should be examined closely.

Those that are most friendly are those, that are in Third Normal Form (3NF). When a database is in 3NF each table represents a separate concept, and the columns are usually attributes of the concept. An application can be easily developed when the database is in 3NF, because all the concepts and their attributes are clear in principle.

Non-3NF databases are more conceptually complex, and hence more complex to customize. Often Non-3NF tables contain embedded concepts. Developing an application for a non-3NF database requires considerably more thought and planning to isolate the concepts and their relations and customize them correctly.

You may have a database structure with a mixed design. Some tables may be in third normal form, first normal form, or not even in first normal form. For the first application, the project team should focus on subsets of the database that are in third normal form where possible.

### **3.3 Examining tables and columns**

It is important to study the structure of your database. This includes examining the degree of normalization, primary/foreign key relations, inclusion dependencies, data-value content, and coded data-value cardinality.

#### **3.3.1 Degree of normalization**

Database structures may be in

- Third Normal Form - each field data value is conceptually dependent on the key, the whole key, and nothing but the key.
- Non-Third Normal Form - some fields are not conceptually dependent on the primary key.



**Third normal form:** As noted above, Ergo works best with databases that are in Third Normal Form (3NF). When a relational database is in 3NF, each table usually corresponds to an independent, substantive concept, independent in a sense that its existence does not depend on the existence of another concept.

When a database is in 3NF, customization is easier because the main concepts and concepts that are attributes are easily identifiable and are clear in principle. Customization involves the specification of main concepts, and their relationships to other main concepts, and the possessive relationships they have to concepts that are attributes. The linking of language terms to each of these concepts become easier since each single data value has one and only one semantic interpretation. The NLE has a precise mapping of terms to concepts.

For example, an EMPLOYEE table corresponding to the concept employee, does not depend on the existence of a PROJECT table, corresponding to a concept, PROJECT. Each of the columns are usually attributes of the concept, such that the entity represented by the concept can be said to possess the attributes. In other words each attribute must be a fact about the key, the whole key and nothing but the key. Having an EMPLOYEE table with columns like EMPNO, ENAME, AGE, WAGE, etc., we may say that a given employee number, name, age and wage are attributes of a given employee, or that employees have an employee number, name, age, and wage.

**Non-third normal form:** When a database moves away from 3NF, the column entities may not be attributes of the table entity. Customization requires considerably more thought and planning to isolate the main concepts and their identifying attributes. Non-3NF databases are conceptually more complex, and hence more challenging to customize because non-3NF tables contain embedded concepts (whose attributes are not dependent on the primary key of the table).

For example, if a non-3NF Employee table had a LOCATION and LOCATION ADDRESS column, the concept of location would be embedded in the employee concept, because location' has an attribute that is not an attribute of the concept represented by the table. For example, a 'location address' is not an attribute of the employee, it is an attribute of location. Embedded concepts have to be identified, before an application can be customized correctly.

To summarize, a table is said to be Non-3NF if:

- Some sets of columns represent embedded concepts where some of the columns are identifiers, and others are names or attributes dependent on the embedded identifier.
- Some columns may represent ambiguous meanings determined by a flag in another columns.

An entire table may actually be an extension of another table linked by a join-path. All columns in the extension table could be attributes of the other table. In this case, some tables may be processed as verbs or redundant entities can be deleted. There are techniques to remedy and create the appearance of a normalized database

through the use of views or creating new conceptual entities to achieve the same effect as views.

### 3.3.2 Primary keys

The primary-key structure of a database is important. If the DB is in 3NF, then customization is straightforward, noting only that *some multiple-column-key tables represent complex concepts rather than a single concept and a set of attributes*.

Primary keys of the tables are the columns whose data values identify instances of the concept represented by the table. Thus, EMPNO '101', identifying EMPNAME 'Smith', identifies the instance of an employee named Smith in an EMPLOYEE table.

Foreign key columns are columns containing data values that are found among the data values of a primary key in another table. They can be excluded from the customization since the same information does not need to be found in two places. A *join-path* is established between primary keys and foreign keys to access the information from both tables. Often entities corresponding to foreign key columns can be deleted.

An exception to this is when the foreign key column represents a concept that is different from the concept of the primary key. An example of this would be if we have a EMPNO column that is the primary key identifying employees in an EMPLOYEE table, and a 'manager' column that is a column somewhere else in the database containing data values that is a subset of the data values in the EMPNO column.

*Note:* If a foreign key column represents a different concept than that represented by the primary key it references, the entity it corresponds to should not be deleted.

### 3.3.3 Inclusion dependency

If a column is a subset of another column then there should be an *inclusion dependency* relation between them. For instance, if you have a MANAGER column and an EMPNO column, then you could define an inclusion dependency between the two and specify that the MANAGER column is a SUBSET of the EMPNO column.

*Note:* In Ergo, an inclusion dependency may be found in a single table. This is the case where one column of the table contains a subset of data values contained in the primary key column. "These columns usually represent different concepts than that represented by the primary key column.

Usually, an inclusion dependency relation is a relation between a super-type concept, e.g., employee, and a subtype concept, e.g., manager. Conceptually, inclusion dependencies enable attributes of a super-type concept to be inherited by a subtype concept. For example, if an employee has an address, then a manager will have one, too.

**Non-3NF databases:** If the DB is not in 3NF, then at least one table contains an embedded table requiring special processing, usually setting up a concept entity that would represent another table if the DB had been in 3NF. Also, if a non-3NF DB

contains multiple-column keys, special situations arise for customizing linguistic subtypes, i.e., subclass entities needed to enable users to ask specific types of questions. If a non-3NF DB is large, with many multiple-column keys, then customization can become complex and sometimes the whole entity can be processed as verbs. See Creating verbs on page 78.

**Multiple-column primary keys:** In Ergo entities that represent the primary key of a table are classified as an identifier, which uniquely identifies each row of the tables as well as the concept represented by the table. If the table has a primary key with more than one column or a multiple column primary key, none of the individual entities identify the table. We have to create a composite entity that consists of the primary columns and let this entity identify the concept corresponding to the table. In others words, we must create a composite entity consisting of primary key columns. This composite entity will be the identifier. We usually classify the component key columns (entities) according to the concepts they represent. Only the composite entity is classified as an identifier.

*Note:* If components of a multi-column primary key represent different concepts other than an identifier, they may be treated as such, but the entities corresponding to them may not be deleted.

See 6. Customization Phase 1 - Processing entities and defining relationships on page 58, for a discussion on composite entities.

### 3.3.4 Data-value contents

Data values must also be scrutinized. In a relational DB, each field in a table represents a single data value, and by implication, each data value has a single semantic interpretation.

In migrations from hierarchical DB's, sometimes the hierarchical sequential key is stored. This data value has both a functional interpretation and an interpretation based on its place in the hierarchy. Strictly speaking, such DB's are not relational. Ergo can handle tables with such columns but all the semantic information cannot be preserved without altering the DB, usually by breaking up hierarchical-sequential-key column into its component columns.

Another example of this occurs if a column contains data values with multiple interpretations by design. For example, a financial application may contain a column like SSNO\_TAX\_NBR, containing a set of numbers, which are either a Social Security Number for individuals or a Tax Numbers for firms. In such a design, there is usually a separate flagging column.

In fact, this kind of design is not strictly relational either, for we cannot determine the meaning of a data value in the SSNO\_TAX\_NBR column simply by looking at it. We need more information. Ergo can also handle such cases, but the customization process for preserving semantic integrity is more complex. Thus, first applications should not include a lot of such columns.

**DATE data values:** DATE data values should be in DB2 DATE format. This will enable questions like:

*How many jobs were processed between January 1st and January 15<sup>th</sup>?*

Date data values in other formats (INTEGER, CHAR) can be processed by Ergo, but at the moment language coverage is limited for these formats.

**Numerical data values:** If you want to ask questions about numbers, the data values the queries refer to must have numerical formats. Language coverage is limited for number stored as CHAR, for example.

Databases with a majority of columns representing numerical data values pose special language problems and require careful customization. SQL with column functions (MIN, MAX, SUM, AVG, COUNT) or SELECTS with arithmetic operations can be linked to entities. Linking language terms to these entities require special language clarity to avoid ambiguity and conceptual confusion. Often homonyms must be specified. For example, the concept, '*total resource*', may mean SELECT (col1 + col2 + ... + coln), and it may also mean SELECT SUM (col1) or even SELECT SUM (col1 + col2... + coln). Some applications may require the creation of many entities with arithmetic operations. We recommend that customizers avoid as their first application databases having a large number of numerical columns, unless they can give clear and precise meanings to the arithmetic operations they want to represent conceptually.

### 3.3.5 Coded data-value cardinality

In a relational DB, each data value has its own meaning. Sometimes these meanings are coded, for example, when M and F are codes for male and female in a SEX column. We can create instances of linguistic subtypes for each coded data value in a column. If there are many columns with coded data values, or if there is a column with many coded data values, then this will add to the complexity of the conceptual model. If there are many columns with coded data values, then we recommend that the number of data value codes not be significantly greater than 10. If there are only a few columns with coded data values, then we recommend that the number of codes not be greater than 20.

The natural language engine interprets the set of coded data values as disjoint, mutually exclusive instances.

## 3.4 Understand conceptual content

It is important for the Ergo user to understand exactly what information is wanted in response to their queries. For example, in the question:

*List the branch offices,*

a user might want the list of branch office names together with their office numbers, if the branches have one. On the other hand, the user may only want the names of the branch offices, identified by their geographic location, or both. The customization

process of getting the expected answer to this query is different.

The matter is complex in DB's with multiple-column primary keys, the component key columns are foreign keys of different tables. For example, a table with CUSTNO, PRODNO, and SHIP\_DATE columns, where CUSTNO, from a CUSTOMER table, and PRODNO, from a PRODUCT table, together constitute the primary key, it is important to understand what this table may mean conceptually. Here, we would expect it to mean that

*A product was shipped to a customer on a particular date.*

This is different from an ADDRESS column that may represent an attribute of an employee. Complex conceptual content requires special customization procedures.

### **3.5 Selecting the application**

Large DB's may contain many application areas. An application area may also contain several semantically meaningful subsets, all of which have specified limited access for information security reasons. All of these meaningful DB subsets correspond to application options.

For fast application development, the lower limits of information required for each possible application should be identified and adopted as the basis for the application. Later, if information requirements change or augment, tables and columns can be added to the application.

The final selection of the application should be made with well-defined project start dates and end dates. The first application should be manageable within a specific time frame.

## 4. PRE-CUSTOMIZATION TASKS

After identifying feasible subsets of the database, the project team will identify application options for implementation. Among the criteria for selecting an option are:

- The target users identified
- The application objectives
- The size of the databases
- The pre-customization strategy anticipated
- The estimated customization lead-time
- The total project time frame

**Target users:** An application is selected with a well-defined set of target users in mind.

In the process of customization, it is useful to involve end users in the implementation process. If the group is small, they may all possibly make contributions. If the group is large, a select group or representative can be designated to participate. End user participation is most useful in the pre-customization process and in the testing process. In the pre-customization process, end users can contribute ideas about syntactic and semantic coverage by listing possible queries, filling out pre-customization forms, or reviewing the pre-customization designs of the customizer.

In the testing process, the end users can assist the customizer in testing his application through the various phases of the customization process, providing incremental feedback as the customization process proceeds.

**Application objectives:** The business objectives of the application should be clear.

**Database size:** The first application should contain a total of no more than 100 columns. Usually first applications are pilot projects, leading to enhanced competence in customization skills.

**Pre-customization strategy:** When selecting an application, a clearly understood pre-customization strategy should be adopted. There are four basic approaches, all of which are discussed later in this chapter.

**Customization time:** For first applications, entities are usually processed at a rate of about 8-10 per hour through the first three customization phases. This includes time required to test the model during these phases. The model refinement process can take more time, which usually depends on the ability of the customizer.

**Time frame of the project:** The time frame of the project should be well defined. As soon as the starting date has been identified, the duration of the project should be established as definitely as possible. This will enable the expenditure of resources to be estimated with greater accuracy. Because unforeseen problems may arise that cause slips and delays, a plan for terminating the project should be in place, if these problems are deemed major, and if they indicate that excessive use of resources will result. Such problems should be anticipated as far as possible in advance, given a

full understanding of the state of the product.

## 4.1 Pre-customization strategies

There are at least four possible pre-customization strategies:

1. Compilation of a query list
2. Design of a conceptual model
3. Filling out customization forms
4. Some combination of these.

### 4.1.1 Compiling a list of queries

Compiling a list of queries is important for determining which tables and columns in an application area will be selected for customization at the test site. We will always need a query list to determine the data base scope of the application, regardless of the customization strategy selected. Compiling a query list for purely linguistic reasons can be useful if the application is small or if the number of target users is small. The query list can be compiled by the customizer, or he/she can solicit possible desirable queries from target users.

Note that there are two linguistic objectives to be achieved in the compilation of such a list:

1. Syntactic objectives
2. Semantic objectives.

Ideally, such a list of queries would be syntactically exhaustive in terms of the product's grammatical capabilities. But the customizer should be aware of this syntactic objective when he compiles his list of questions. Planning for syntactic coverage can be done by the customizer himself.

The customizer also has the semantic objective of including in the application dictionary all the words the target users wants to use. Solicit lists of queries from target users can be a good way of doing this.

Note, however, that as the size of the application increases, as the number of target users increases, or as the organization role of the target users varies, the usefulness of this approach to decide linguistic coverage declines, for these reasons:

- If the application is large, there is a greater probability that the query list provided by target users will be repetitive and not exhaustive, that is, they will not be able to think of all the questions they might really want to ask;
- If the number of target users is large, it may take a longer time to process and compile the lists of queries submitted, and not all the target users may submit questions, or all the questions, they may want to ask;
- If target users are managers, they may not have the time to devote their full attention to this task, thus making it perfunctory, and of little value to the customizer.

### 4.1.2 Designing a conceptual model

Designing a conceptual model can also be useful if the application is small or if the number of target users is small. The conceptual model can be designed by the customizer, or by a customization team.

Note that there are two linguistic objectives to be achieved in the design of a conceptual model:

1. Syntactic objectives
2. Semantic objectives.

Ideally, the entity specification of the conceptual model would be syntactically exhaustive in terms of the product's grammatical capabilities. But the customizer should be aware of this syntactic objective. The customizer also has the semantic objective of including in the application dictionary all the words the target users wants to use. The conceptual model should include synonyms for the terms that he associates with the entities in the model.

Note, however, that as the size of the application increases, the utility of designing a complete conceptual model declines, for these reasons:

- If the application is large, there is a threshold beyond which designing a complete conceptual model will take more time than is desirable;
- As the number of entities increases, there is a threshold beyond which the design of a complete conceptual model becomes impracticable;
- If the application is large, there is a greater probability that errors will emerge in the conceptual model design, and if there is a time limit involved, the customization process based on an inaccurate design, and this may necessitate modifications of the customization at some later point in time, thus delaying the project.

### 4.1.3 Filling out pre-customization forms

For large applications, or when the number of target users is large, or both, designing and filling out pre-customization forms would seem to be the best strategy. Such forms would include the following information:

- For tables, their primary keys and join-path links
- For tables, their entity names, terms, synonyms, and syntactic information
- For columns, their referential integrity information
- For columns, their entity names, terms, synonyms, and syntactic information
- Adjective entities to be added, with their names, terms, synonyms, and syntactic information
- Verb entities to be added, with their names, terms, synonyms, and syntactic information
- Sub-type entities to be added, with their names, terms, synonyms, syntactic information, and with their SQL image.

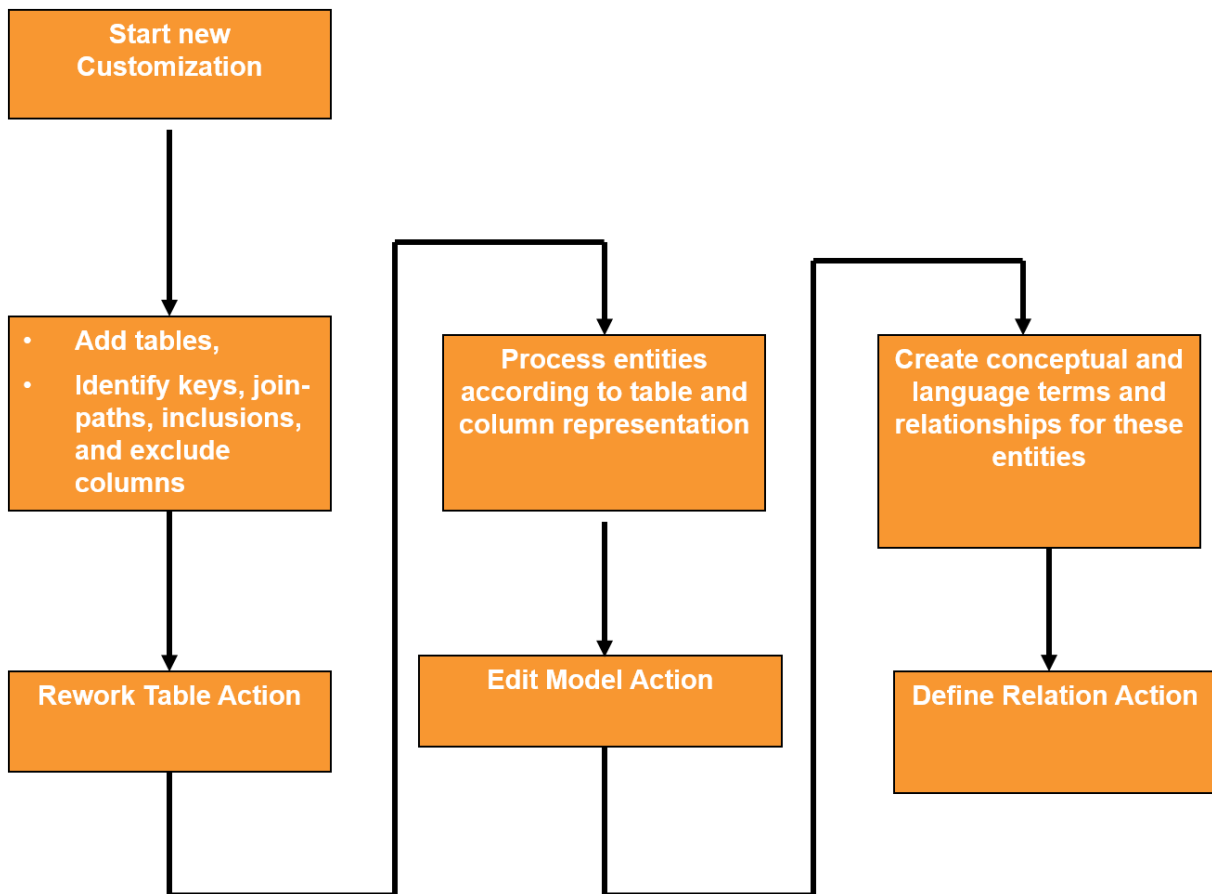


#### 4.1.4 A combination of these strategies

Very likely, the customizer will select some combination of these strategies, given the size of the customer application, the size of the target user group, and his time constraints. Generally speaking, the customizer will find these guidelines useful:

- A list of queries specifying syntactic and/or semantic coverage would be useful for guaranteeing a broader grammatical scope in the application. The customizer can do this himself.
- A rough conceptual model would be useful for identifying join-paths, referential integrity constraints, and required verbal relations.
- Pre-customization forms would be most useful in larger applications, especially if the number of entities exceed one hundred, or somewhere thereabouts.
- Participation of end users can provide important input and feedback, whatever pre-customization strategy is adopted.

## 5. USING THE CUSTOMIZATION TOOL



<p>In rework tables you specify what database structure the model has. You should specify the JOIN-PATHS and INCLUSION DEPENDENCIES that you have found in the pre-customization phase. You should specify the primary key and columns and exclude columns you do not need.</p> <p>You do all this using the <i>Mode</i> → <i>DB</i> and <i>Action</i> → <i>Add tables, Add columns</i>.</p>	<p>Customization links words that you are going to use in your queries to the database tables and column. You have to define these words as entities. This is done by assigning a term, classifying, defining the syntax and maybe adding SQL statements to the entity. Additional entities may need to be created if there is no data base representation for the entity</p> <p><i>Mode</i> → <i>Lang</i> and <i>Action</i> → <i>Add entity</i></p>	<p>As with any entity, additional entities must be related to other entities. Relationships are formed based on the entities classification. There are both Language and Conceptual relationships.</p> <p>You do this by dragging/dropping one entity on top of another in the graphical interface.</p>
--	--	---

Figure 4 Customization process

## 5.1 Enter DB information

In the CT you enter information about the database tables and columns – this becomes part of the domain model. The engine needs to know the name of the database to be used as well as information about tables and columns.

### 5.1.1 Enter table information

To enter table information press *Mode* → *DB* and *Action* → *Add tables*.

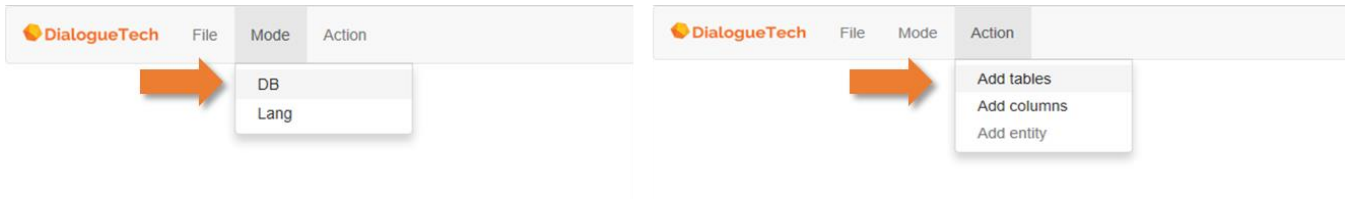


Figure 5 Opening of the "Add tables" window

When you press *Add tables* a screen appears that prompts you to enter the name of the database and tables.

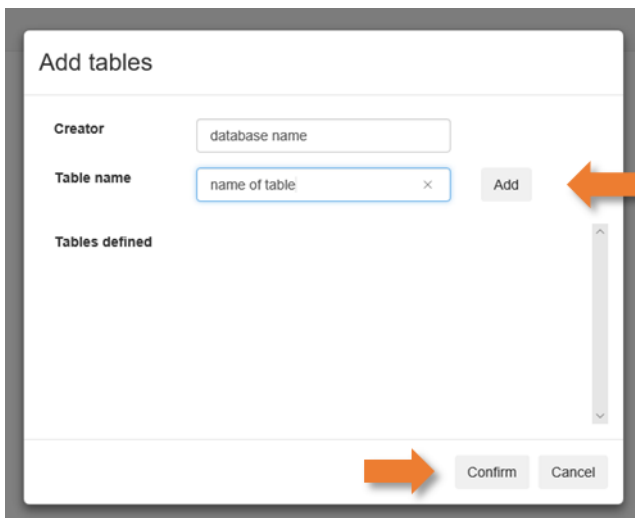


Figure 6 The "Add tables" window

Press *Add* and *Confirm* to complete the list of database tables

Once you have entered all tables you can add the columns of each table. To add the columns, you press *Action* → *Add columns*. Enter the name of each column for each table – press *Add* for each new column.

Add additional information for each column – define the type of data you have in each column by double-clicking on each column name, and select the data type in the Edit column screen:

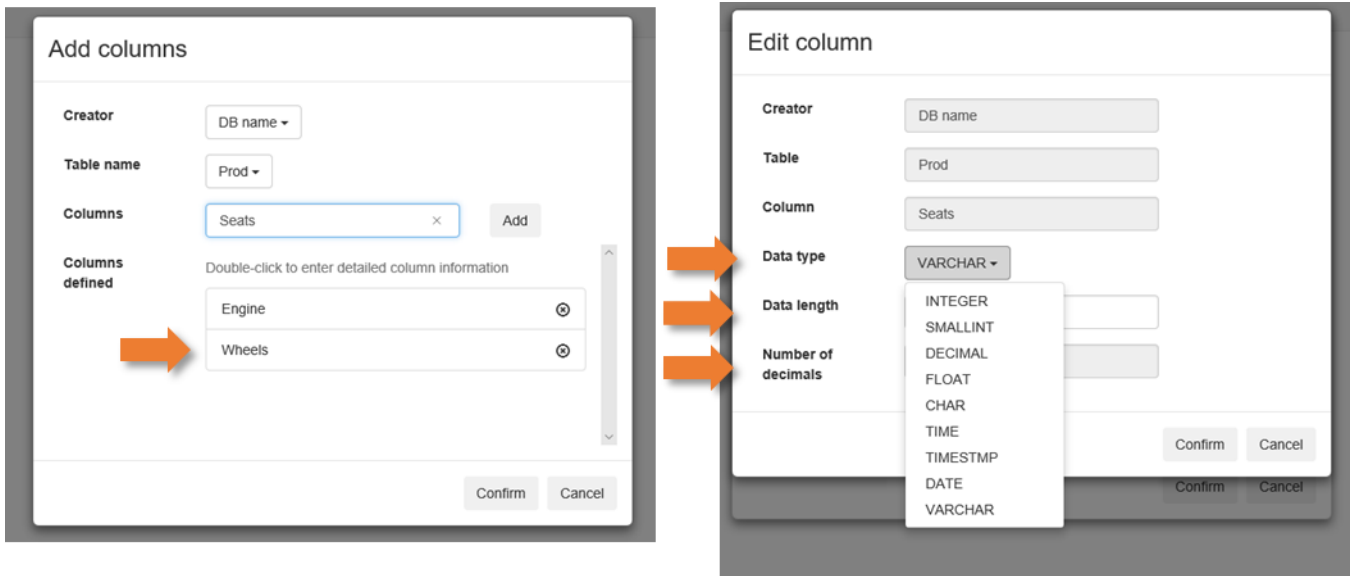


Figure 7 "Add columns"and "Edit columns" windows

The result from the example above look like:

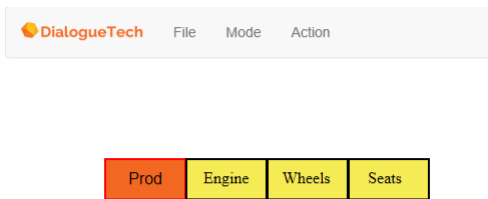


Figure 8 Graph DB mode

You can now create keys, join-paths, inclusion dependencies and exclude columns that you decided on in the previous phase. You should also exclude foreign keys and any other columns that you do not want to include in your model. Open the "Properties of columns" window by double-clicking on the column in the graph:

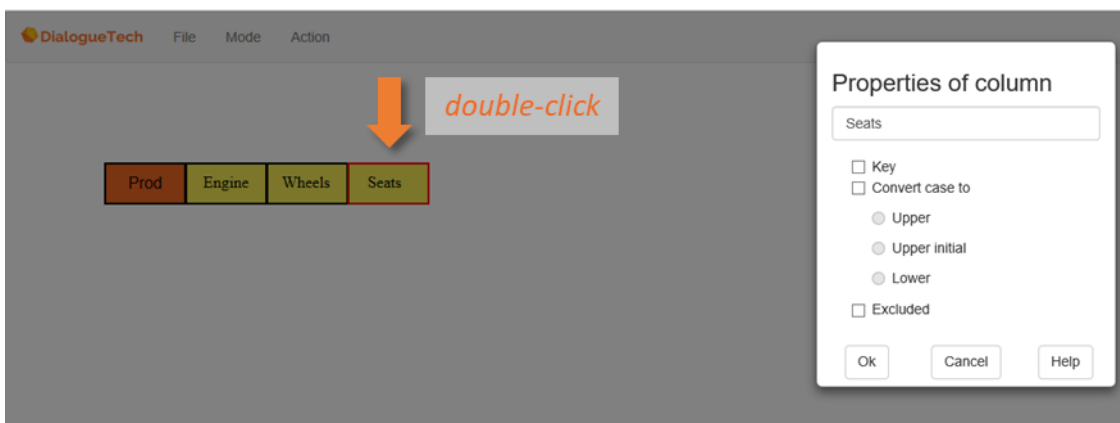


Figure 9 Defining columns and keys

### 5.1.2 Joins

A join-path is established between the primary key of one table and its foreign key in another table. This enables the Natural Language Engine to select information from different tables in a single query. More importantly, if a concept from one table contains an attribute associated with another table, a join-path enable a relationship to be established, thus linking entities across tables. For instance, if you have the EMPNO column in two tables, e.g., EMPLOYEE and DEPT tables, then you could connect the two tables with a join-path. If an employee has a department phone number, e.g., DEPT PHONE, then this relationship can be defined.

To define a join path, click on one of the tables and drag it on top of another table – this will open the window below:

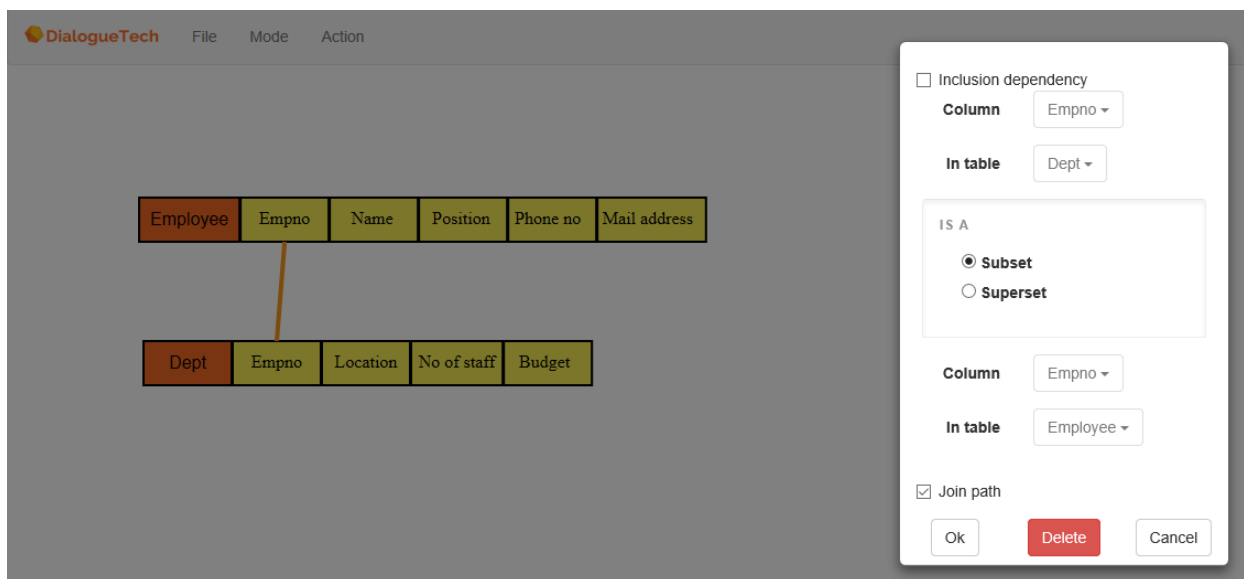


Figure 10 Defining joins and inclusions

Tables must only be joined with one join-path.

*Except for multi-column primary key whose components are multi-column foreign keys, only one single-column join-path should be made between the two tables.*

So, if these two tables, EMPLOYEE and DEPT also have a column called DEPTNO, or 'department number', you can choose which join-path to make: either the join-path between the EMPNO columns or between the DEPTNO columns, but NOT BOTH.

If you define more than one direct join-path between two tables, Ergo assumes at least one table has multiple-column primary keys. If there are more than one join-path links between a pair of tables Ergo interprets this as a multi-column join-path. For example if a table has a two-column key that can be found as a foreign key in another table with, say, a three-column primary key, you may join both columns to the corresponding columns in the second table.

Join-paths cannot be circular. If you have connected the employee table and the department table with a join-path and you have connected the department table with

the address table then the employee table and the address table automatically has a connection between them, and you should not make a join-path between them. What order to connect the tables is somewhat arbitrary, but connections should be made to the most important table. So, if you are dealing with an employee application then you could connect the employee table to all the other tables like spokes in a bicycle wheel.

Concerning designing what join-paths should be the best paths let us take this hypothetical example.

Table: Insurance Policy  
COL1: POLNO(Primary key)  
COL2: REGION\_CD  
COL3: DISTRICT\_CD  
COL4: BRANCH\_CD

Table: REGION  
COL1: REGION\_CD (Primary Key)

Table: DISTRICT  
COL1: REGION-CD(Primary Key)  
COL2: DISTRICT CD(primary Key)

Table: Branch Office:  
COL1: REGION\_CD (Primary key)  
COL2: DISTRICT\_CD(primary Key)  
COL3: BRANCH CD(primary Key)

Given these insurance tables, we could link Region, District and Branch Office to Insurance Policy individually. We can go from Insurance Policy to Branch Office to District to region etc. There are various possibilities to join these tables. However, what decides the best join-path? Usually, the kind of questions and the complexity of the SQL generated dictate the choice of a join-path. If the application mostly concerns insurance policies and agents in branch offices, we might link Insurance\_Policy via Branch Office. If the application concerns policies and regions, then we may link via regions. If all three are equally important, then we will link all these to Insurance\_Policy.

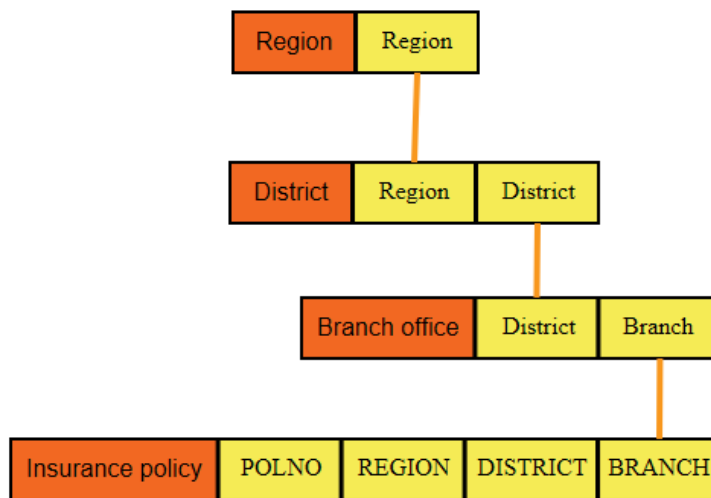


Figure 11 Example of join-path and inclusion dependencies

### 5.1.3 Inclusion dependencies

Primary-Key/Foreign-key relations should be scrutinized.

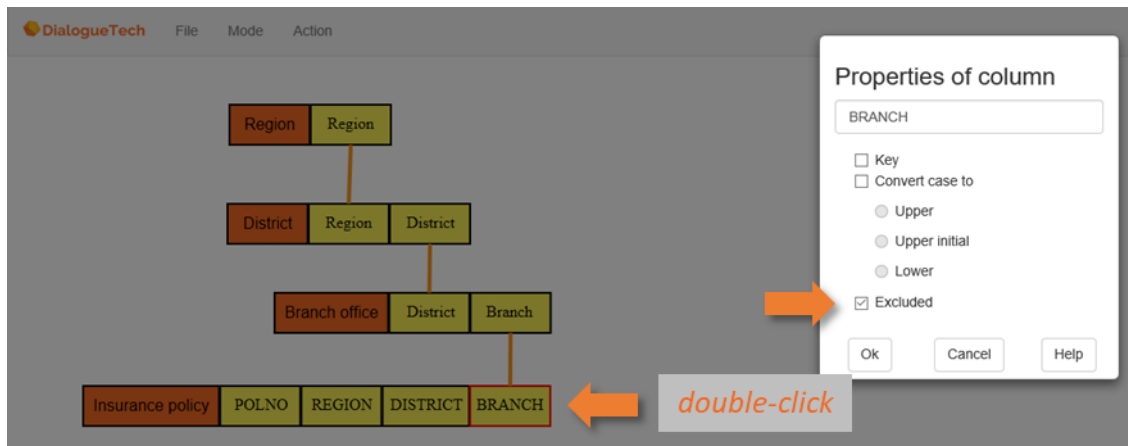
If the foreign-key column represents a different concept than the primary-key concept (e.g. Tab 1. employee/Tab 2. manager), then an Ergo inclusion dependency is usually specified, and the conceptual relationships of the super-type entity will be inherited by the subtype entity.

If a foreign key column represents the same concept as the primary key (e.g., Tab1\_dept/Tab1dept) then customization is straightforward, and usually a join-path is set up between the tables to permit the specification of relationships between entities representing concepts related to different tables.

### 5.1.4 Excluding columns

In table mode, those columns that are excluded, are those, which are not part of the application, which are foreign keys columns representing the SAME concept as the primary key. An example of this is an EMPLOYEE\_DEPARTMENT and DEPARTMENT\_NO representing the same concept so EMPLOYEE\_DEPARTMENT can be excluded. However the EMPLOYEE.EMPNO and DEPARTMENT.MANAGER cannot be excluded because it is a subtype (subclass) of employee representing a different concept.

To exclude a column, double-click on the column graph and select “Excluded” from the “Properties of column” window.



**Figure 12 Exclude column**

Constituents of a multi-column primary key should not be excluded.

All the work in the pre-customization phase and in rework tables should be checked before you go on. It is important that this phase is correct before you move on to edit the model. See Case 1. Joins and Inclusion Dependencies, Fine Tuning the Model.

## 5.2 Create entities

Use the Lang ( language) Mode to create and define entities in the conceptual model. This is what you must do when creating/defining a new entity:

1. Create an entity by giving it a name
2. Classify the entity
3. Give it a language term if you want to use it in a query
4. Define the morphology of the term if you defined one.

These are the tasks you can use when you edit the conceptual model.



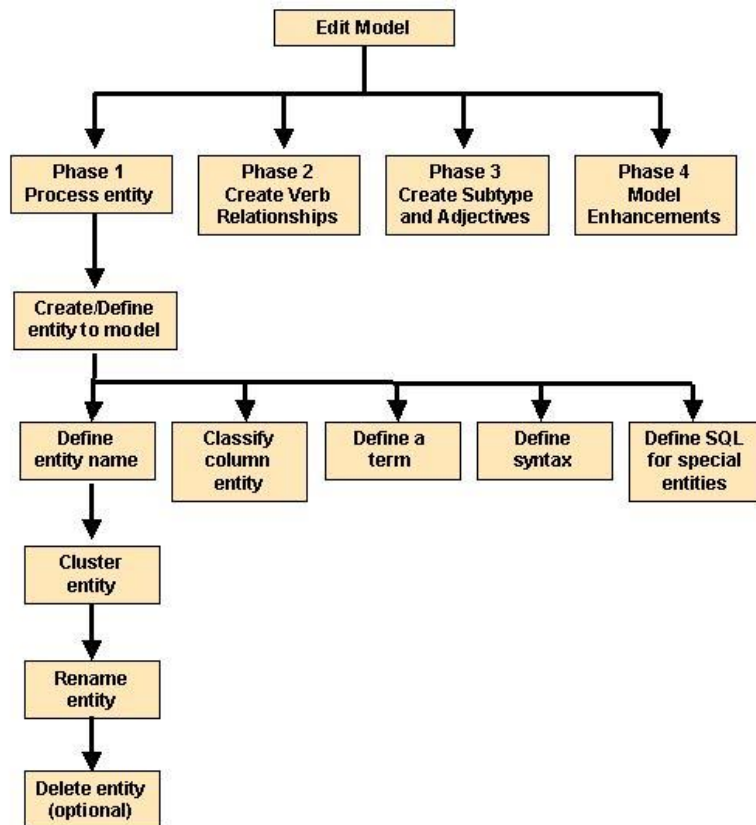


Figure 13 Process for entity creation

### 5.2.1 Creating a new entity

You create new entities when defining subclasses, instances, verbs and adjectives, to enhance the conceptual model. To create a new entity, select:

*Mode* → *Lang*

*Action* → *Add entity*

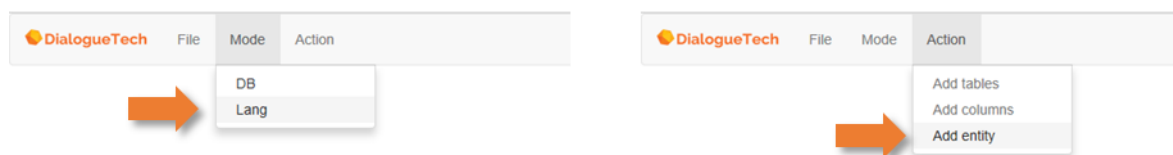
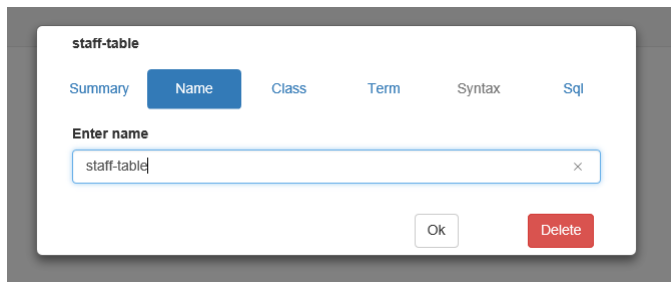


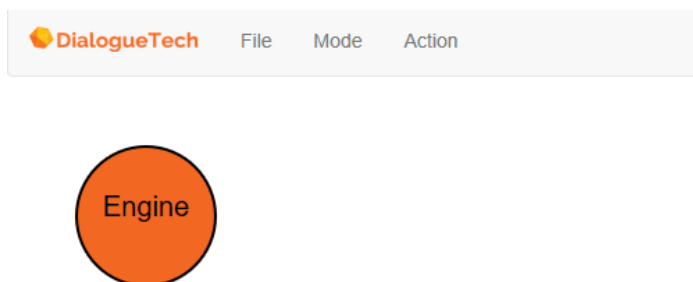
Figure 14 Opening of the "Add entity" window

Once you have opened the *Add entity* window you can enter Name of entity, classify it, add terms, syntax and SQL



**Figure 15 Add entity window**

Once an entity is defined and saved it will appear in the DB-mode graph. To edit the entity you double-click on the icon.



**Figure 16 Graph of an (undefined) entity**

### 5.2.2 Naming an entity

The entity window contains an empty entry field, in which you enter a name for the entity.

To change the name of an existing entity, select *Name*. The existing name appears in the entry field. Each entity name must be unique. It can contain up to 25 characters.

1. Type the entity name in the entry field.
2. Press **OK** to enter the input and remove the window or press **Class** to add classification

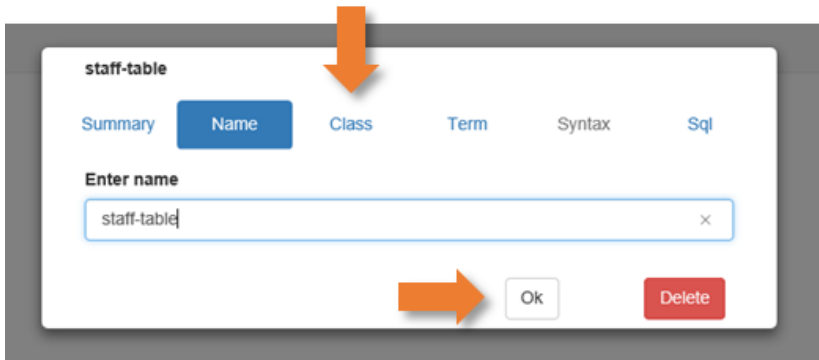


Figure 17 Specifying an entity name

### 5.2.3 Classifying an entity

Classify each entity in your conceptual model as a subclass or instance of at least one other class. The define classifications window shows existing classifications which are shown as a tree. The tree can be expanded and contracted.

### 5.2.4 Subclass and instance entities

1. Press the *Class* button to display the classification window:

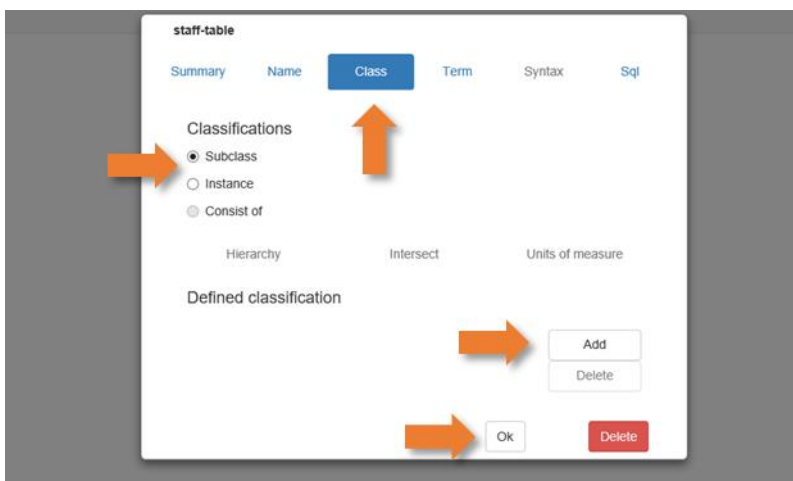
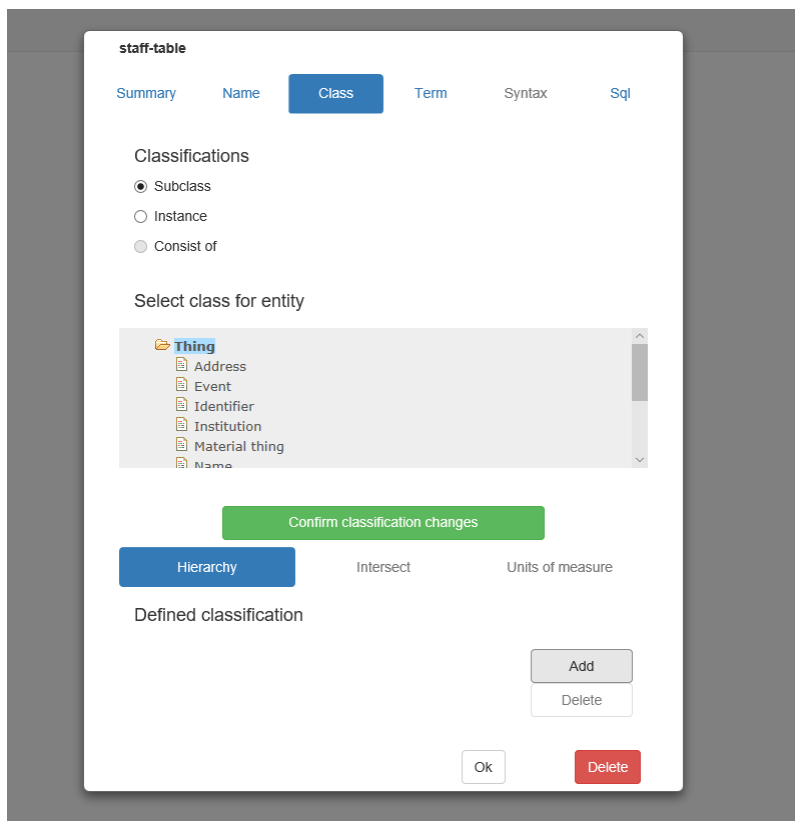




Figure 18 Entity classification window


2. Select *Subclass* or *Instance*.
3. Press *Add* → *Thing* to display the hierarchy of classes:



**Figure 19 Class hierarchy**

Some classes in the hierarchy already have subclasses. This is marked by a  symbol. To expand the hierarchy and view the subclasses, click on the  symbol.

4. Select the class under which you want to add your entity.
5. Press *Confirm classification changes* to add the entity.

If the class you select is collapsed, a  appears in front of it to show that you can now expand that class. If you click on it, you see how your entity has been added to the hierarchy:

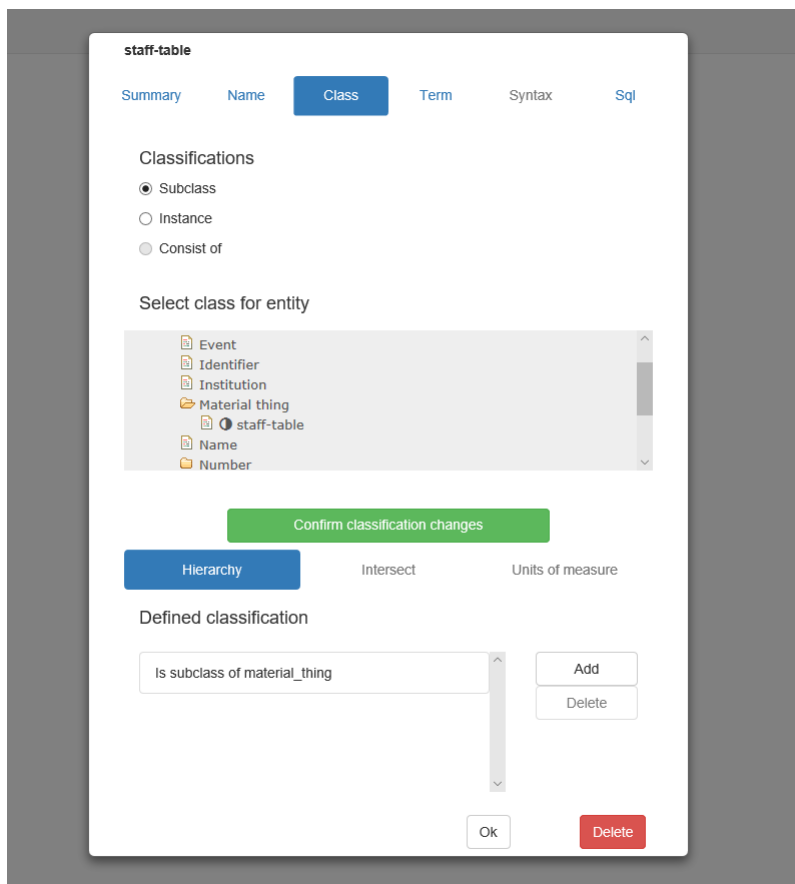


Figure 20 Adding a subclass to the hierarchy

### 5.2.5 Consists of

Use the *Consists of* selection if you have to define a structured entity that consists of a combination of two or more entities that have an SQL statement. As an illustration you may have an entity car and an entity fast (connected to the terms fast, quick, etc.) and define a structured entity fast car.

To classify a structured entity,

1. Press **Class**.
2. Add the entity as a subclass of one of the classes in the hierarchy.
3. Press **OK** in the Define classification window.
4. Select **Consists of** in the **Define Entity** window, and press **Add**.
5. Select the constituent entities in the same order as products users will use them in their queries when they enter data values of the corresponding entities. Avoid more than 3 or 4 constituents for each composite entities. Otherwise questions involving composite entities may cause memory problems. To remove a selected entity, click on it once in the list in the upper part of the window. The highlighting then disappears, and the entity is removed from the *Ordered selection* list.
6. Press **OK**.

### 5.2.6 Units of measure

When you classify an entity as a subclass of *quantified\_property* (or a subclass of a subclass of *quantified\_property*), you can specify its unit of measure. The customization tool contains units of measure for age, area, currency duration, length, volume, and weight. When you specify a unit of measure, Ergo users can use that unit of measure in their questions, for example:

*How many dollars does a chisel cost?*

#### **Selecting an existing unit of measure**

When you classify an entity as a subclass of *quantified\_property* (or a subclass of a subclass of *quantified\_property*), you can specify its unit of measure. The customization tool contains units of measure for age, area, currency duration, length, volume, and weight. When you specify a unit of measure, Ergo users can use that unit of measure in their questions, for example:

*How many dollars does a chisel cost?*

#### **Selecting an existing unit of measure**

To select an existing unit of measure:

1. Classify the entity under *quantified\_property* (for example, as a subclass of *price*).
2. Press *Confirm classification changes*.
3. Press *Unit of measure* in the *Class* window.
4. Select the unit of measure that you want from the list (for example, *dollar*).
5. Press *Confirm unit of measure changes*.

**Testing**

Summary   Name   **Class**   Term   Syntax   Sql

**Classifications**

Subclass  
 Instance  
 Consist of

**Select measurement for entity**

Units of measure   Scale factor

Cent  
**Dollar**  
Penny  
Pound

1.0

**Confirm unit of measure changes**

Hierarchy   Intersect   **Units of measure**

**Defined classification**

Add  
Delete

Ok   Delete

**Figure 21 Specifying units of measure**

You can also specify a scale factor. For example, if your database contains weights measured in pounds, but users instead want to talk about ounces in their queries, select *ounce* as the unit of measure and specify *16* as the scale factor because 16 ounces = 1 pound.

Note that the scale factor is used only to convert the unit of measure in a question so that it corresponds to the unit of measure used for data values in the column. The answer to a question is not converted.

### 5.2.7 Creating terms

Create one or more terms for each entity that users want to refer to. The terms you create can be either nouns, verbs, adjectives, or proper names. Each term that refers to the same entity must have the same category; if you give an entity a term that is a noun, the additional terms you create for that entity are automatically nouns. Do not confuse terms with entity names. The name is used to distinguish the entity icon in the diagram. Terms are used in questions. To create terms:

1. Press *Term* in the *Add entity* window.
2. Enter your term: for example, *employee*.
3. Select a category: *Noun*, *Verb*, *Adjective*, or *Proper name*.
4. Press *Add*.

If your term is a noun, verb or adjective, a window appears where you specify the grammar of the term.

Figure 22 Creating a term

### 5.2.8 Specifying syntax

In the customization tool, syntax refers to how terms are used in a question. You must specify syntax for each verb entities. Syntax is optional for noun and adjective entities.

#### Syntax for verbs, nouns and adjectives

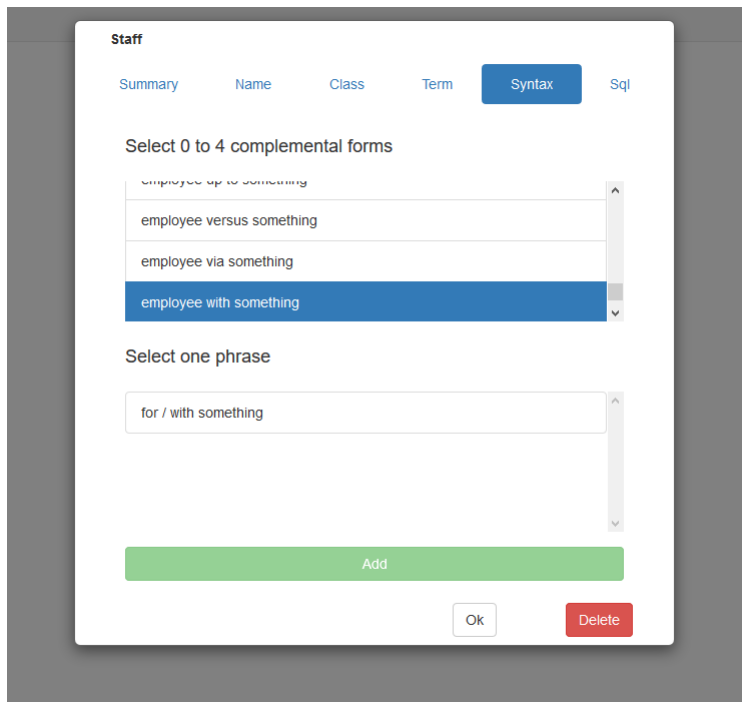
In the customization tool, syntax refers to how terms are used in a question. You must specify syntax for each verb entities. Syntax is optional for noun and adjective entities.

#### Syntax for verbs, nouns and adjectives

You can optionally specify syntax for adjectives and nouns.

1. Press *Syntax* in the *Add entity* window.  
The Preposition complement window appears.
2. Select the prepositions that you want to use with your entity. Verb syntax is discussed in detail in 7.10 Creating verbs.
3. Press *Add*





**Figure 23 Adding syntax**

4. If you select more than one preposition, the Syntax window appears with proposed phrases using your term.
5. Select the phrase that suits your term.
6. Press **OK**.

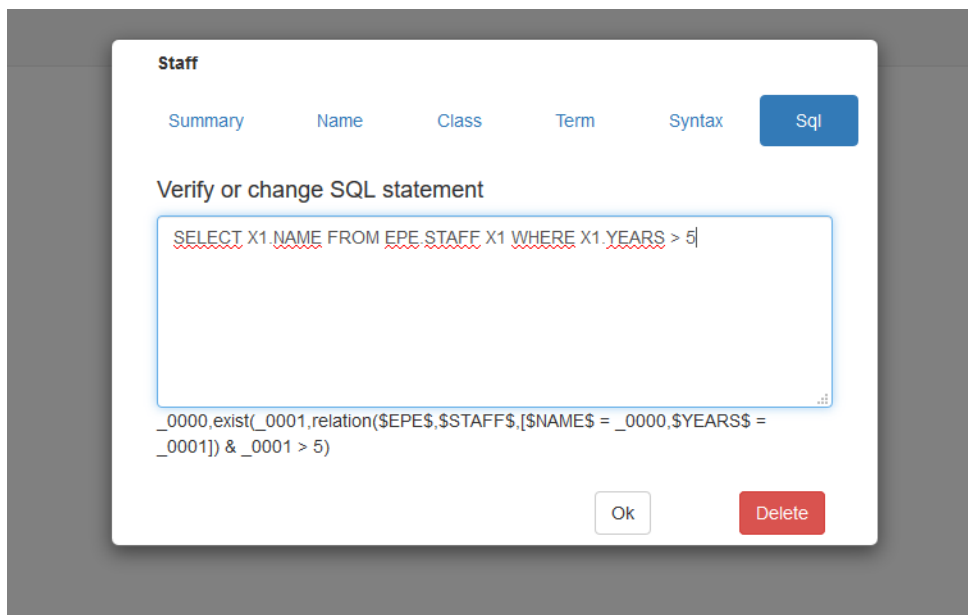
### 5.2.9 Specifying an SQL statement

We use SQL to define the SELECT statement for an instance, adjective, or noun entity that you create. If the column contains NULL values, you can change the SELECT statement so that the NULL values are disregarded in the answer to a query:

```
SELECT X1.COMM
FROM EPE.STAFF X1
WHERE X1.COMM IS NOT NULL
```

To specify an SQL SELECT statement:

1. Press **SQL** in the *Add entity* window.
2. Type the SELECT statement in the entry field, or copy text into the window from, for example, an application program.
3. Press **OK** to enter the input and remove the window.



**Figure 24 Specifying SQL**

Note that if you modify the SQL statement for a column entity go to table mode, then return to entity mode, the customization tool generates a new entity with the original SQL statement. The column entity with a modified SQL statement remains unaffected in the conceptual model.

See 8.3 Creating instance entities on SQL considerations for entities.

### 5.2.10 Creating verbs

Verbs are used to describe an action and link nouns together. In creating a verb, one should perform the following:

1. Select *Add entity* from the *Action* bar
2. Give the entity a name
3. Classify the verb as event
4. Define the term and choose the verb grammar
5. Choose *Verb Syntax*
  - Choose verb complements
  - Choose prepositional complements
  - Select the final syntax

To use the same verb with many different nouns, you can:

- Link the verb to a noun with many subclasses. (Because relationships are inherited, the verb can also be used by any noun subclasses of that noun.)
- Create two or more verb entities with the same terms. You must do this if:
  - The nouns are not all in the same table or tables that are joined together (directly or indirectly).
  - The verb has a different syntax or different meaning when used with different nouns. For example, if programmers *write* programs and writers *write* manuals about programs, define two verb entities with the term *write*.

### 5.2.11 Subclasses

Subclasses define data that are a subset of the data in the database.

- A subclass is one that is wholly contained within another class already defined.

Examples:

Clerk is defined as a subclass of staff, where job is clerk.

Tools are defined as a subclass of product, where the product group is tool.

- Subclasses use SQL to select a defined subset from the database.

#### Defining a subclass

1. Create and name a new entity. You could give the name an extension of \_subclass.

2. Classify the entity under the entity to which it belongs.

Examples:

Classify clerk\_subclass under staff\_table.

Classify tools\_subclass under product\_table.

3. In the *Terms* window, make the entity a noun and give it suitable terms and grammar.

4. Select any prepositional syntax required.

5. Give the entity an SQL SELECT statement that will retrieve the required data. The column selected is normally the identifier (primary key) of the concept.

Example: The SQL for clerk subclass is:

```
SELECT X1.ID FROM EPE.STAFF X1  
WHERE X1.JOB = 'CLERK'
```

If the column is in another table, you need not change the default SQL for the column entity. Appropriate SQL is derived from the inclusion dependency defined in *Table* mode.

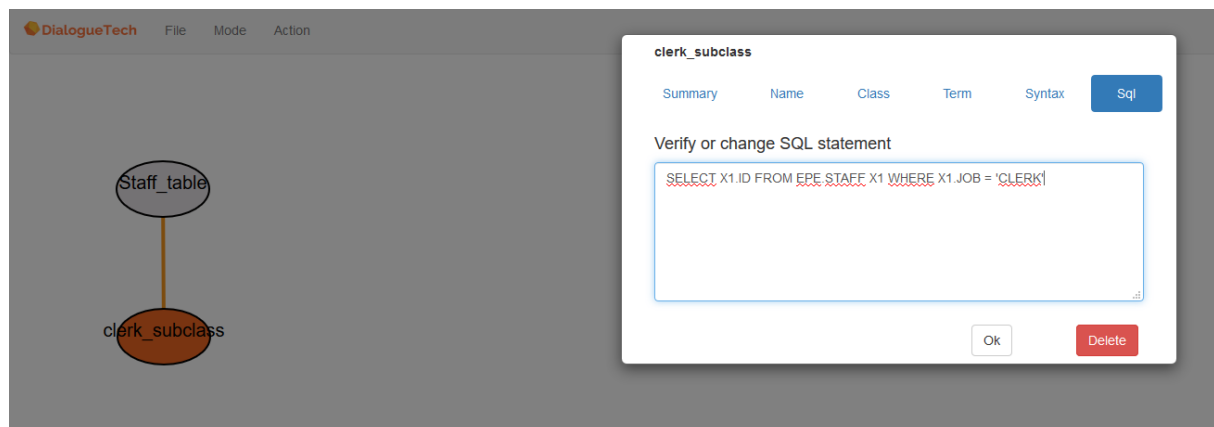


Figure 25 Subclass example

### 5.2.12 Creating adjectives

Adjectives in Ergo are used to select part of the data in a database. As with subclasses, this selection is done with an SQL statement.

#### Specific adjectives

1. Create and name an entity. No extension is needed.
2. Make the entity a subclass of the noun it qualifies.

3. In the *Terms* window, make the entity an adjective and choose suitable adjective terms and grammar.
4. Select any prepositional complements that may be required.
5. Enter the SQL that selects the required set of data.

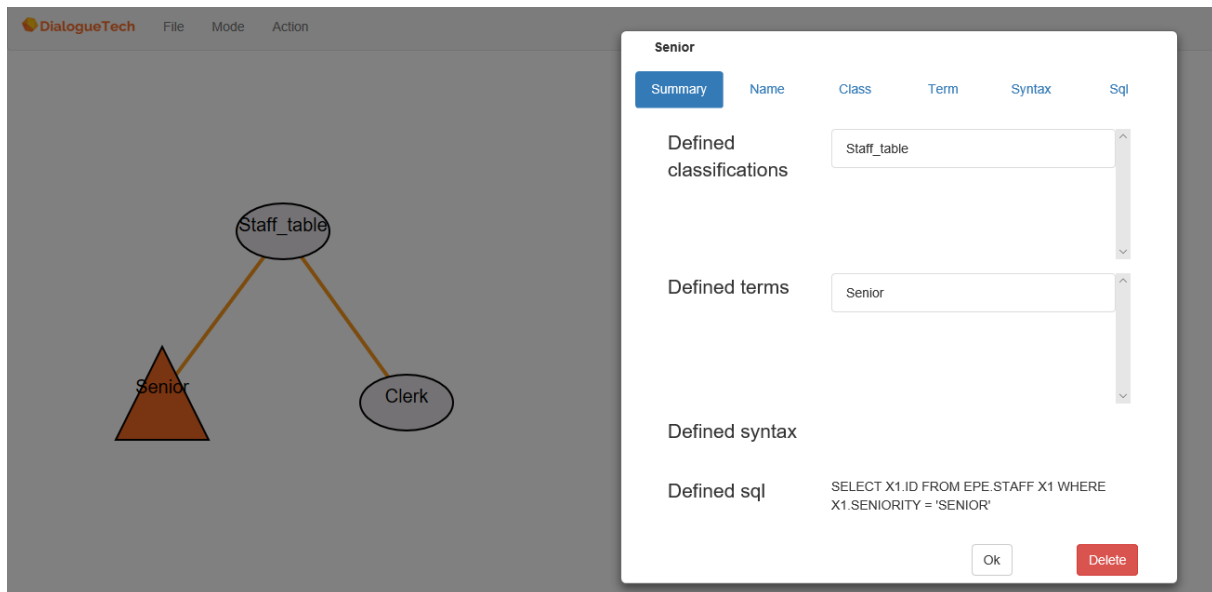


Figure 26 Adjective

### Subtype adjectives

If you create a compound term and define it as adjective + noun, Ergo creates a subtype adjective. Example:

If you define electrical product as adjective + noun, but without creating the adjective electrical, you can use the subtype adjective to ask questions such as *List products that are electrical*.

If you define the compound term *electrical product* as a single noun, you can only ask questions such as *List electrical products*.

#### For Each Subtype Adjective

1. Create a subclass entity
2. Classify the entity under the noun to be qualified
3. Give the entity a compound term (Adjective + noun)
4. Give the entity the SQL that selects the required subset.

### 5.2.13 Intersect

The classification window has an Intersect button that lets you select whether an entity intersects with other entities at the same classification level or is disjoint from them. Intersecting entities represent concepts that can be combined.

Example:

Senior is intersect with clerk\_subclass because a person can be both a clerk and senior.

Disjoint entities represent concepts that are mutually exclusive.

Example:

Clerk\_subclass is disjoint with manager\_subclass because a person cannot be both a manager and a clerk.

Correct selection of intersect/disjoint improves performance.

### Defining intersect entities

1. After classifying the entity, press *Apply*.
2. Press *Intersect*. A list of appropriate entities appears.
3. Select entities that are to be disjoint.
4. You only need to do this for one of the two entities involved.

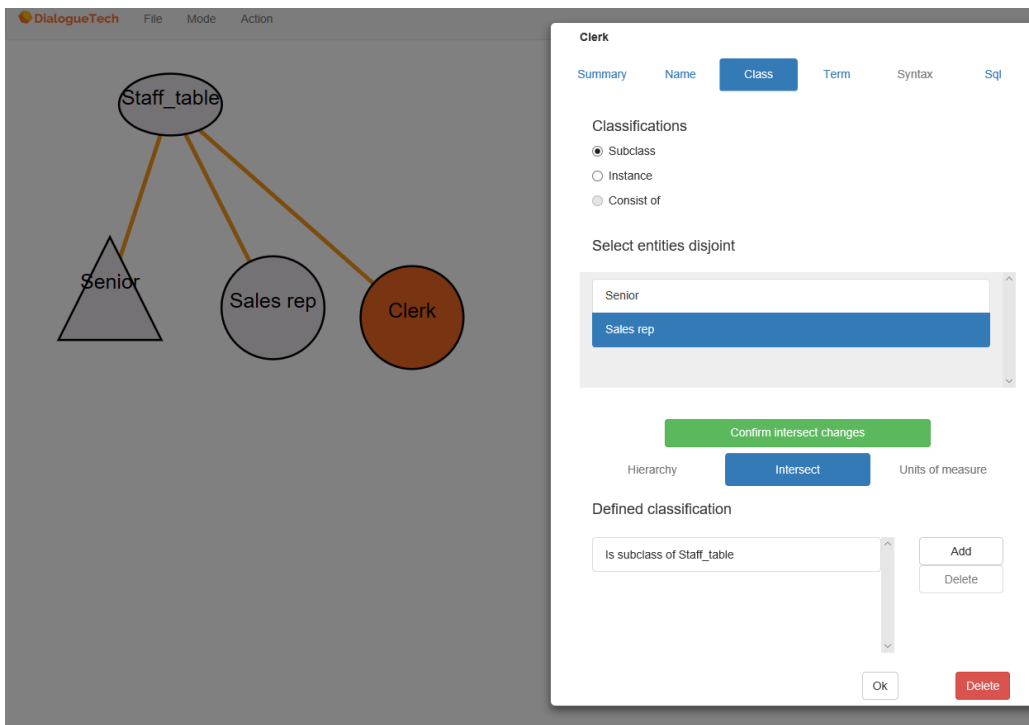


Figure 27 The intersect window

### 5.2.14 Instances

Use instances to identify a specific row in a table. You may want to do this if an item in the database has more than one term.

Examples:

Head Office is also known as HQ by the users.

A generator can also be called a dynamo.

The state of Texas is referred to as TX in the database.

You need to inflect the name of a database item.

Example:

You need to refer to cam and to cams, when CAM is the product name in the database.

Figure 15 Instances

**Lyckas inte förstå hur man gör detta på rätt sätt!! Se original-illustration**

### Creating instances

1. Create and name an entity. You could give the name an\_inst extension.
2. Make it an instance of the concept to which it belongs.

Example:

Make cam\_inst an instance of product\_table.

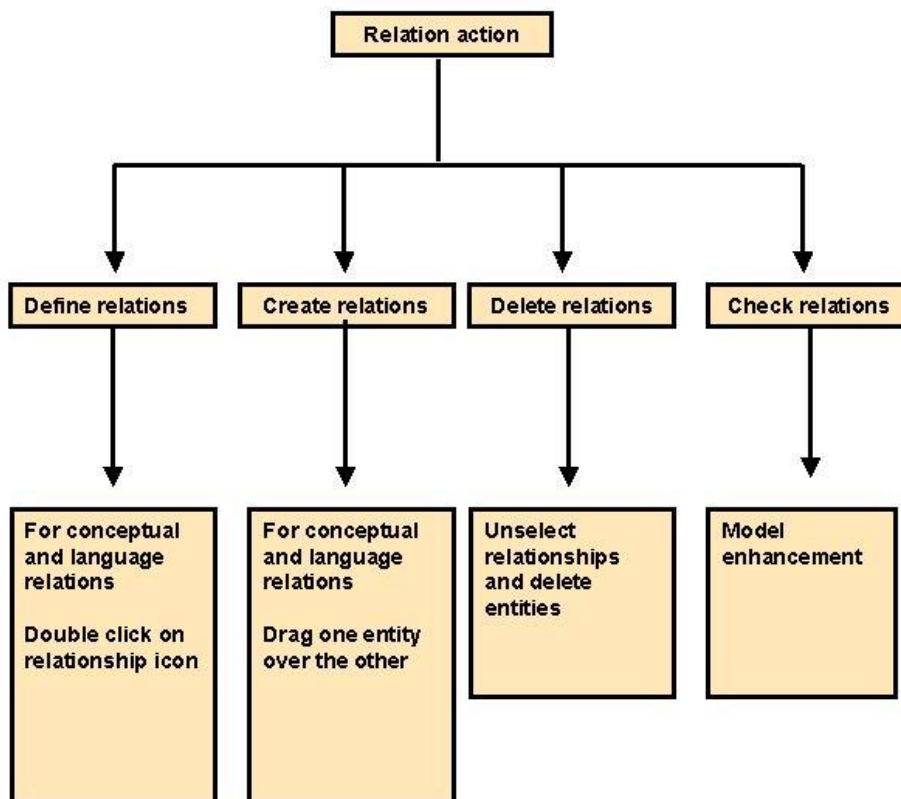
3. Select suitable terms and grammar.
4. Select prepositional complements if required.
5. Enter SQL to select the specific row, using the primary key.

Example:

To select cams, use  
SELECT X1.PRODNUM  
FROM EPE.PRODUCTS X1  
WHERE X1.PRODNUM = 150

### 5.3 The relation action

Use the Relation action to relate an additional entity you created with the Entity action. You can create language and conceptual relations. You also can delete relations. This is the way the *Relation* action works.



### 5.3.1 Defining relationships

A relationship is formed between two entities. Only one type of relationship can exist between two entities, however, an entity can have as many relationships as conceptually and grammatically needed. There are two types of relationships:

#### Conceptual relationships

1. Identifies

Used between an entity classified as an identifier and the entity it uniquely identifies.

Example:

*What uniquely identifies staff? employee\_id*

2. Names

Used between an entity classified as name and the entity it names. The name should be the column by which the row is generally described. Example:

*What is the name of Staff? employee\_name*

#### Language relationships

There are additional relationships that denote possession, place, time and use of prepositions.

1. Possessive relationships - denotes possession, for example, between the manager and employee entities you define:

*What does manager have? Employees*

2. Place relationships - denotes location, for example, between the manager and department entities you define:

*Where is the manager? Department*

3. Time relationships - denotes a duration of time, for example, between project and end date entities you define:

*Till when is the project? end date*

4. Prepositional relationships - denotes usage of a preposition between two entities (i.e. like a prepositional phrase). For example, between the verb entity works and the entity clerk, a preposition as is defined.

*Who works as a clerk*

Relationships are represented by lines in the entity diagram. A solid line is drawn between two entities if

- You define a relationship between the entities
- One entity is a subclass or instance of the other
- One entity consists of several entities, including the other.

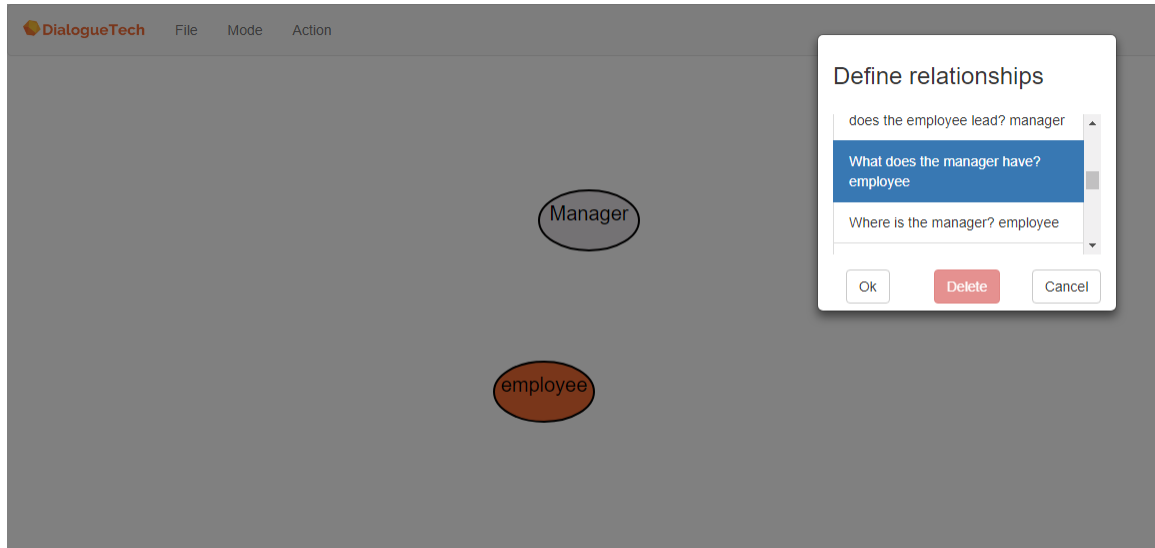
### 5.3.2 Creating or changing a relationship

To define a new relationship or change a relationship that you have already defined:

1. Specify that you want to define a relationship, in either of these ways:

- Hold down the right mouse button and drag one entity icon onto the other.
- If a line (dotted or solid) already exists between the two entities, double click on the line. To define a relationship between cluster icons, first expand the clusters, then double click on the relationship line.

The *Define relationships* window appears.



**Figure 28 Selecting a relationship**

2. The possible relationships are listed. Select the relationships that you want. The possible relationships depend upon the:
  - Class of each entity
  - Type of terms for each entity (noun, adjective, or verb)
  - Syntax of each entity.

See Chapter 6 on Defining Relationships for more details

3. Press OK. A solid line is drawn between the two entities in the diagram.

Relationships need not be chosen from the Select relationships window when you have defined a subclass, instance or structured entities using the “consists of” function. If no relationships appear in the select relationships window, then you may have forgotten to classify or give a term to the entity.

### 5.3.3 Removing a relationship

To remove a relationship, double click on the line representing the relationship and deselect the relationships that are no longer valid in the Define relationships window. Alternatively – press Delete and the relationship will be erased.

### 5.3.4 Correcting invalid relationships.

Relationships that you previously defined may become invalid if you change the:

- Class of either entity
- Type of terms (noun, adjective, or verb) for either entity
- Syntax of either entity.

This is how you check and correct an invalid relationship:



1. Double click on the relationship line to display the Define relationships window. A selected but invalid relationships appears within brackets:

[What is the name of department? location]

2. Click once on the invalid relationships to deselect it. The highlighting disappears.

3. Select any of the possible relationships that you want.

4. Press OK.

## 6. CUSTOMIZATION PHASE 1 - PROCESSING ENTITIES AND DEFINING RELATIONSHIPS

In following a breadth-first customization strategy, the first phase of the customization process consists of processing entities in the Define entity dialog box and then defining relationships between the entities. The CT automatically generates a data model of entities. The customizer creates a conceptual model that will contain necessary application information as the entities are processed and relationships between them defined. When this phase is completed, the information may be saved. Usually, this work will not have to be done again. If customization errors are made later, the customizer can always come back to this point, or any later phase that is done accurately and saved.

### 6.1 Processing entities

These are the main tasks for processing entities:

1. Naming the entities
2. Classifying the entities
3. Specifying terms for the entities
4. Specifying syntax for the terms linked to the entities
5. Specifying SQL for newly created entities

As these tasks are completed, the state of an entity changes. The aim of this process is to enter correct conceptual and language information so that users will be able to use the application containing this information to ask questions against a relational database.

### 6.2 Naming the entities

The names of the entities are used to help identify the entities, to distinguish them from other entities, and to be able to recognize what part of the database or what part of language they are associated with.

*Note:* The names themselves do not affect the kind of questions that you can ask with the query interface.

If there are two columns with the same name in different tables, then the Customization Tool adds a digit at the end of their names to distinguish the names from each other. Not only should names be distinguishable or identifiable within each table but across tables. In the creation of a composite entity, it is important to know where each entity comes from, and what table they belong to.

*Note:* Make sure that all names are recognizably different. This will make it easier to:

- Distinguish between each entity on the screen.
- Identify each entity during the creation of a composite entity, i.e., an entity consisting of a set of other entities.
- Know what part of the database or scope of language the entity is associated with.

Naming conventions can be useful to aid in the identification of these entities. This would help the customizer identify the entities and make the entities recognizable across each table.

It is useful if the entity name is similar to the primary term linked to the entity. The name could also have different extensions for different entities, for example: CENT for column entities, TENT for table entities, SENT for composite entity, VPENT for verbs in the passive voice, VAENT for verbs in the active voice and so forth. You can devise your own naming conventions accordingly.

### 6.3 Classifying the entities

When you classify an entity, you associate that entity with a concept found in the classification hierarchy. Since you will next link a language term to the entity, you must make sure that the classification you choose for an entity fits the language terms you want to use.

*Note:* You can also add your own classifications for selected classes not found in the hierarchy by creating an entity and classifying it where you expect it to belong. Such an entity requires a DB image, i.e., SQL must be specified for it. The main objective for creating such an entity would be to add conceptual clarity to the model.

The figure below shows the classification hierarchy at its highest level.

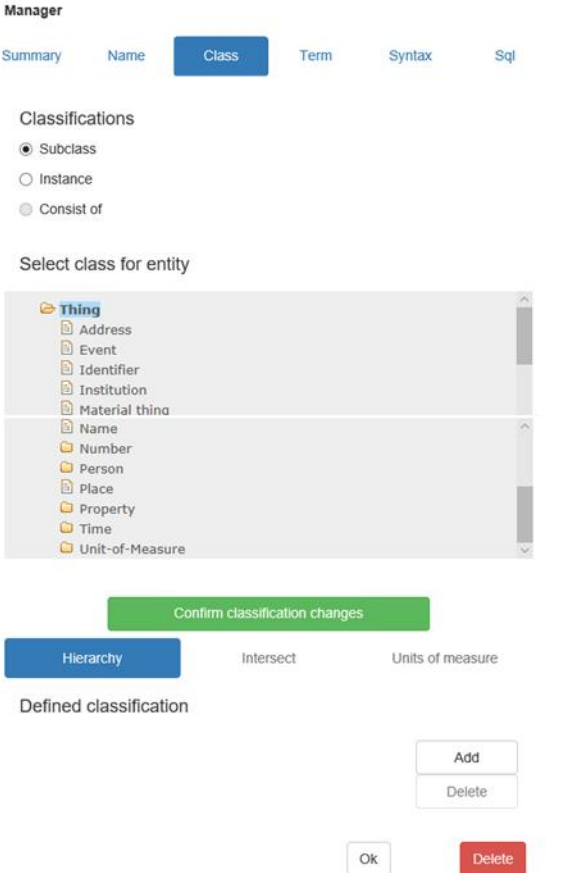


Figure 29 Classification hierarchy

Remember these things when you classify entities:

1. Classifications pass on syntactic properties to entities. A category in the classification hierarchy enables certain kinds of questions to be asked. Classifying an entity as a *place* will enable you to use 'Where' in your questions.
2. The classification of the entities will enable you to define different kinds of relationships between entities depending on how you have classified the entities. Classifications also effect relationship options. If you classify a '*department*' entity as a *place*, then you will get the option to select relationships like '*Where is the employee? department*' when you select relationships between the '*employee*' and the '*department*' entities. If you classify '*manager*' as a *person*, then you will get the option to select relationships like '*Who is the manager of the department? manager*' when you select relationships between the '*manager*' and '*department*' entities.
3. Do not classify entities where it should not be classified. For example, you should not classify *salary* as a *thing*, if you want to refer to salaries by asking '*how much...*' in a query. Classifying *salary* as *quantity* is correct because *salary* is usually used in a context, which denotes quantity, e.g. '*How much is Joe's salary?*'

Most of the classifications are obvious -- a *person* is a person, and a *date* is a date. But what if the entity does fit in any obvious classification? Then you can usually classify the entity as a material thing.

There are 37 classes already defined in the Customization Tool:

Thing	
	Address
	Event
	Identifier
	Institution
	Material_thing
	Name
	Number
	Quantity
	Person
	Place
	Property
	Quantified_property
	Age
	Area
	Depth
	Duration
	Height
	Length
	Price
	Size
	Value

		Volume
		Weight
		Width
	Time	
		Date
		Period
		Timestamp
	Unit_of_measure	
		Currency
		Unit_of_measure_age
		Unit_of_measure_area
		Unit_of_measure_duration
		Unit_of_measure_length
		Unit_of_measure_volume
		Unit_of_measure_weight

**Table 2 Define classes**

4. Entities can also have more than one classification. If for instance we have a 'university' entity, then that entity can be classified as both an *institution* and a *place*. This is not encouraged, however, since it will affect the performance of the natural language engine. There are some exceptions as follows:

*Example 1 - identifiers and names*

Some entities classified as names or identifiers can be classified as something else. Some entities can be classified as both identifier and name. For example, 'country' in a country table can be both an identifier and a name because there is only one and only one country that exists in this world having one and only one name. Note that it can also be classified as a place. Again this is dependent on the context on how it is used in a question.

*Example 2 - quantities and numbers*

Some entities can be classified both as a quantity or quantified property and number. Given an entity commission. If we were to ask

*List commissions > 500*

then commission would need to be classified as a number. If commission were to be used in this context

*List the top ten salesmen,*

then commissions would have to be classified as a quantity and a measured\_by\_relationship should be put in place between commission and perhaps the entity salesmen.

In general, multiple classifications should be avoided.

Performance of the Ergo Engine is affected by the number of relationships made possible by the classifications that have been specified. Multiple relationships based on multiple classifications generate multiple paraphrases. You could classify on the basis of the relationships you are sure you will need to ask the questions you want. Avoid multiple classifications that can generate redundant syntax. For example, if

'department' is classified as a place, it does not need to be classified as an institution, which gives you the same language-coverage possibilities.

### 6.3.1 Classifying identifiers and names

At this point keep in mind these important classes:

#### IDENTIFIER

This classification is used for entities that represent the primary-key data values of a table. Primary-key data values represent unique instantiations of a concept. For instance, the *'employee code'* column in an employee table will contain a set of data values representing unique instantiations of individual employees, where *'employee code'* is the primary key of the table.

**Subclass identifier note:** Identifiers that are subclasses of other identifiers can be classified as a straightforward identifier. Unless a term is required in the application for the identifier of a subclass, then it can be classified at the level of its super-class id. There are performance implications here because there is nothing to be gained in classifying it as a subclass of its super-class id.

**Non-3NF note:** If the table is not in third normal form, a column might be identifying another column in the same table. If this table were normalized, these columns would be in a separate table by themselves. In order to deal with this problem, you would usually create new concept entities, that would be identified by the entity this column. (See Chapter 7, Creating New Entities.)

**Multiple-column primary key note:** If the table represents a concept and has a multiple-column primary key, then you will create a composite identifier, which will be classified as an identifier. The components of the composite identifier do not have to be classified as identifiers. It is sufficient to classify them as things.

#### NAME

This classification is used to represent the name of instantiations of the concept represented by the table. For example, in an employee table the *'employee name'* column contains the name of instances of employees, since the table represents the concept, *'employee'*, and *'employee name'* is the name of instantiations of employees in the table.

**Subclass name note:** Names that are subclasses of other identifiers can be classified as a straightforward name. Unless a term is required in the application for the identifier of a subclass, then it can be classified at the level of its super-class name. There are performance implications here because there is nothing to be gained in classifying it as a subclass of its super-class name. This might occur if the DB is not in 3NF.

**Non-3NF note:** If the table is not in third normal form, in addition to redundant column information leading to subsets of data values and subclasses, a column might be naming another column in the same table. If this table is normalized, these columns would be in a separate table by themselves. In order to deal with this

problem, you would create new concept entities, which should be named by the entity corresponding to this column. (See Chapter 7, Creating New Entities.)

**Multiple-column primary key note:** If the table represents a concept and has a multiple-column name or description, then you will create a composite name, which will be classified as a name. The components of the composite name do not have to be classified as a name. It is sufficient to classify them as things.

### 6.3.2 Specifying terms for entities

You can have as many terms for an entity that you want and is practically useful. Memory is the main limitation. But it is unusual to have on the average more than 10 terms and synonyms linked to a given entity.

Here is a list of things to remember about terms:

1. The terms that you link to the entities are the words that the user will use to access information in the database. Because the entities are classified, they represent concepts. Except for verbs, all entities - either directly, or indirectly through an identifier - will have a DB image. It is the language-entity-database link that that enables your queries to be processed.
2. A subclass will inherit the default (primary term) of its super-class. Therefore, a subclass should not have any term from the super-class' term list in its term list. For example, if *'branch office employee'* is a subclass of *'employee'*, then *'employee'* and all its synonyms, e.g. *'staff'*, will be inherited by the subclass *'branch office employee'*.

An entity that is the subclass of another entity does not need nor should have as a synonym any of the terms linked to the super-class entity.

3. You can specify more than one term for an entity, and the user will be able to use all terms for an entity, but it is only the first one that the Query Interface will use to generate its paraphrase (interpretation) of the question. So you should make sure that the first term of all entities is different. For example, if you have two entities with the first term *'employee'*, then the query *'who are the employees'* will give you two identical paraphrases: *'list the employees'*. However if you make *'employee'* the second or third term of the entities and make the first term something else like *'headquarter employee'* and *'branch office employee'* then you will get the two paraphrases: *'list the headquarter employees'* and *'list the branch office employees.'* *That is, the most descriptive term should be the first term.* If a shorter form that has two different interpretations is used, then the shorter form should not be the first term for both entities.

The first term is the default term or primary term, which is used in paraphrases by the query interface. This term should be recognizably different from all other terms.

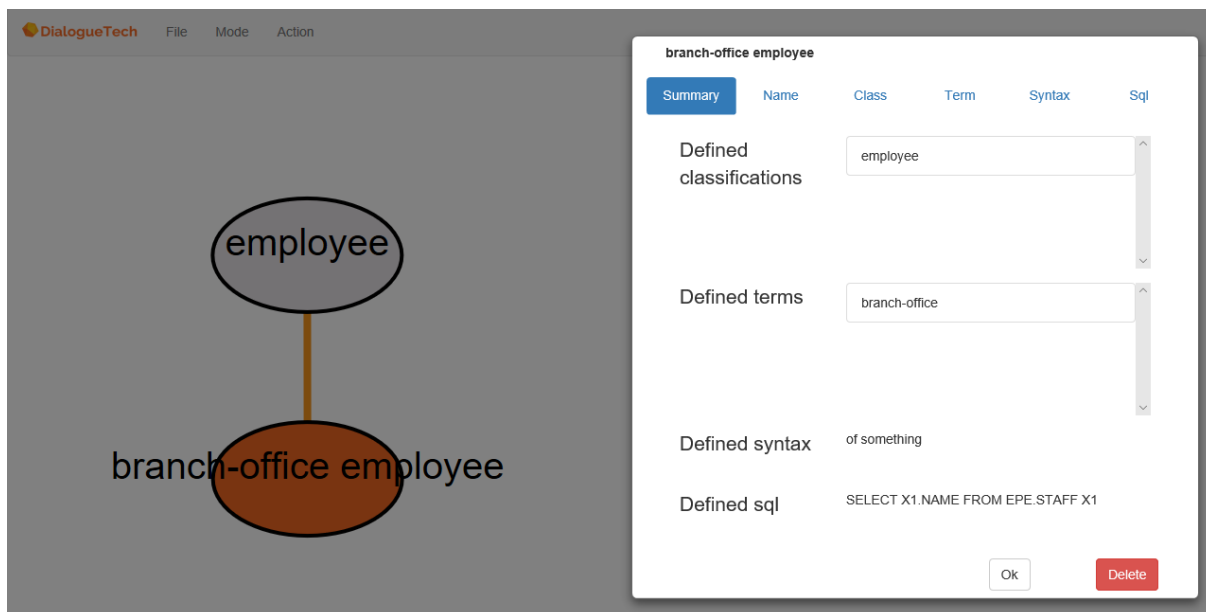


Figure 30 Example of subtypes and super-types sharing terms

### 6.3.3 Types of terms

The two most commonly used forms for terms are:

1. SINGLE NOUN. Use this form if you want treat the compound term as a single noun and you do not need to use the individual terms in any other combinations.

Example:

If the term is *'california location'*, you will always use the entire phrase together in each query you write: e.g. *'List the california locations'*.

2. ADJ + NOUN. This form is used if you also want to use the modifying adjective as a predicative adjective in your queries. This is also used to define term for an adjective entity.

Example:

We can say *'List the plants that are profitable'*, *'List the profitable plants'*. This adjective property is inheritive. So, for example, if we have a *'western plant'* subclass, then we can ask questions like: *'List the western plants that are profitable'*.

The following are other forms of compound terms that are rarely used:

1. NOUN + NOUN. Use this form if your compound term consists of two nouns, and you want to use the plural form of one or both of them either separately or in other combinations.

Example:

*'Woman doctor'* is a compound term whose components can both be inflected, e.g., women doctors.

2. ADJ + NAME. Use this forth if your compound term consists of an adjective and a proper name, and you want use one or both of them either separately or in combinations.



Example:

North America is a compound term whose parts can be used separately or in other combinations, such as South America or North Sea

3. NAME + NOUN. Use this form if your compound term consists of a proper name and a noun, and you want to use one or both of them either separately or in other combinations.

Example:

Texas customer is a compound term whose parts can be used separately or in other combinations such as Texas company or California customer.

4. NOUN + NAME. Use this form if the compound term consists of a proper name and a noun and you want to use one or both of them either separately or in combinations.

Example:

Accountant Smith is a compound term where parts can be used separately or in other combinations such as Accountant Jones or Captain Smith.

5. ADJ+ NOUN + NOUN. Use this form if the compound term consists of an adjective and two nouns, and you want to use at least one of the terms either separately or in combinations.

Example:

Local sports club is a compound term whose parts can be used separately or in other combinations, such as new tennis club or local tax authority.

*Note:* There is a way to define terms needing the form ADJ(ADV) + ADJ + NOUN. For instance, if you wanted to say both '*show senior clerical employees*' and '*show employees that are senior clerical*' you need to define '*senior clerical*' as an adjective. This is accomplished by writing '*senior clerical employee*' as the term and define it as ADJ + NOUN. Then change the forms of '*senior\_clerical*' in the *Adjective grammar* window. You change BIG '*senior clerical*' to '*senior clerical*'. You then can delete all other inflections of the adjectives, unless they are meaningful for the questions you want to ask.

#### 6.3.4 Specifying syntax for entities

Normally we save specifying syntax as our last task. This is because our classification specification will provide the initial necessary syntax. However, syntax must be specified for verbs.

*Note:* Remember to specify syntax for verbs. You must always specify to verb complement to be able to set up verb relationships to ask questions.

### 6.3.5 Specifying SQL for entities

For every new concept entity created must have a DB image. This is accomplished either by writing SQL directly to the entity representing the concept or by linking the new concept entity to an identifying entity that has an existing DB image.

## 6.4 Defining entity/entity relationships

You can define these kinds of entity/entity relationships:

1. Column/table entity relationships
2. Table/table entity relationships
3. Other Relationships
  - a. Column entity/column entity relationship
    - where the relationship is defined between entities linked to the same table
    - where the relationship is defined between entities linked to different tables
  - b. Newly created entities/any other entity

You can also define these entity/entity relationships based on conceptual criteria.

The conceptual relationships you can define include:

1. the identify relationship
2. the name relationship
3. the possess relationship
4. the locative (place) relationship
5. the temporal (time) relationship
6. the count\_by relationship
7. the measure\_by relationship.

There are also language relationships based on the parts of speech of the language terms linked to the entities. These include nouns, verbs, and adjectives.

## 6.5 Defining column/table relationships

When working with column/table relationships, you must take into account whether the table is in 3NF or not.

1. When a table is in 3NF, the column entities usually represent an attribute of the main concept represented by the table entity. Normally, all column entities will have relationships between them and their table entities defined (by activating the dashed lines between the column entities and the table entities). A possessive relationship is almost always defined or inherited. For example, if you defined a possessive relationship between the table entity, 'employee', and the column entity, 'salary', you will be able to ask questions like *'What are the salaries of the employees?'*

*Note:* Possess relationships also enable refer back pronouns, e.g., *'list the employees and their salaries'*.

Other kinds of relationships will depend on the classification of the entities, for example, as a place.

*Remember:* If the database is in 3NF, every column is usually an attribute or property

possessed by the concept represented by the table.

In order to define relationships between entities you need to click on the line between them. A list of possible relationships will show up on the screen, and you can select the ones that you feel are most appropriate. The list of relationships that show up will depend on how the entities were classified, so if a certain relationship that you want does not show up you might want to reclassify the entities. *The list of relationships also depends on whether terms have been defined for the entities, particularly for verb entities.*

These are some of the relationships that might show up:

*What does the Department have? Employee*

*What does the Employee have? Department*

These are "possess" relationships and will enable you ask questions like:

*What is the department of John?*

if John is an employee, and:

*Who are the employees of the marketing department?*

For locative expressions, you select:

*Where are the employee? Department*

This will enable you to get answers to questions like:

*From what department is John?*

and:

*Who is from the marketing department?*

The relationship:

*From where do the employee come? Department*

will let you answer questions like:

*From what department does John come?*

and:

*Who comes from the marketing department?*

*Note:* Some columns might be components of a composite entity possessed by the concept represented by the table. An example of this would be an address entity consisting of a street and a city column entity.

2. If the database is not in 3NF, then some column entities may not have a relationship to the table entity it is linked to. When this is the case, the customizer must understand how such entities are related to other entities.

In this case, some column entities may not be attributes of the table entity, and a relation between the column and the table entity may not have to be directly defined. The customizer should remember these things when working with non-3NF databases:

- Some sets of columns might represent embedded concepts where some of the columns are identifiers and others are names. See Case 6.
- Some columns may represent multiple (ambiguous) meanings determined by a flag in another column. See Case 13.
- An entire table may actually be an extension of another table linked by a join-path. All columns in the extension table could then be attributes of the other table. Usually in such cases, the primary key of one table is a foreign key of the primary key of another table. When this happens, table entities with the *'foreign key'* primary key can be deleted or processed as something other than nouns, e.g., processed as verbs. See Case 9.

## 6.6 Defining identify relationships

An identifier entity is classified as an IDENTIFIER, and you select the relationship *'What uniquely identifies < table entity > ? < column entity >'*

*Note:* All table entities representing main concepts have to have an identifier. Note also that not all tables represent main concepts.

The identifier has to be the primary key for the table in order to be the identifier when the database is in 3NF.

*Note:* If the table has a primary key with more than one column, none of the individual entities uniquely identifies the table. We have to create an entity that consists of all of the primary key columns, and let that entity be the identifier to the table, i.e. we must create a composite entity, classified as an identifier that consists of the primary key columns.

## 6.7 Defining name relationships

When a database is in third normal form, there is usually a column that *'names'* or describes instances of the concept represented by the table. The name of the instances of the concept represented by the table *'employee'* is the column *'employee name'*, or *'department name'* for the *'department'* table. These columns have to be classified as names, and you need to high-light the relationship *'What is the name of < table entity > ? < column entity > .'*

*Note:* As with the identifier, the name of the instances of the concept represented by the table can consist of more than one entity. For instance the *'city'*, *'state'* and *'country'* columns for a location table may together represent the location name. Here, we must create a composite entity, consisting of the describing/naming column. This entity is classified as a name.

## 6.8 Defining possessive relationships

If a column entity is an attribute of the concept represented by the table entity, a possessive relationship must be defined.

*Note:* A table entity representing a main concept can possess a column entity associated with another table. This can happen, for example, with non-3NF databases, or when the primary key of one table is the foreign key of another table. For queries to work, however, a join-path must link the two tables.

## 6.9 Defining other conceptual relationships

In addition to the identify, name, and possessive conceptual relationships, you can also define these:

1. the locative (place) relationship
2. the temporal (time) relationship
3. the count by relationship
4. the measure by relationship.

You must classify entities correctly to define these relationships. We examine these relationships more thoroughly in Chapters 8 and 9.

## 6.10 Defining table/table relationships

Just like the relationships between the columns and the tables, there may be relationships between the tables themselves. The same choices apply to these relationships as the column/table relationships.

*Note:* Table entities have no DB image, i.e. there is no SQL automatically specified for corresponding entities. The DB image is specified when an identify relationship has been defined. Of course, the customizer might later want to add SQL to table entities, but this rarely happens; and it would not happen if the table entity has an identifier specified.

## 6.11 Defining other entity/entity relationships

Other entity/entity relationships include:

- Column entity/column entity relationship where the relationship is defined between entities linked to the same table
- Column entity/column entity relationship where the relationship is defined between entities linked to different tables
- Newly created entities/any other entity

It may happen, especially if the database is not in 3NF, that relationships are required between entities linked to the same table or to different tables. In the latter case, be sure a correct join-path exists between the tables. And when you create new entities, be sure you clear about the relationships you want to define with them.

## 6.12 Non-3NF considerations

If your database is not in 3NF, then you should

1. Identify the main concepts and their identifiers.
2. Determine the attributes possessed by the main concepts, identify which entities in the model they correspond to, and define the relationship.

3. Determine what other relationships the main concepts have to other entities and define them.
4. Make sure join-paths are correctly set up to handle the relationships you want.

### **6.13 Multi-column primary key considerations**

If you have multi-column primary keys, you should

1. Identify multi-column join-paths to other tables.
2. Create a composite identifier and select the components in the order you want.
3. Link the identifier to the correct concept it identifies.

## 7. CUSTOMIZATION PHASE 2 - CREATING NEW ENTITIES

If you are following a breadth-first strategy, you have now processed all the entities of the initial data model and defined the main relationships you want. Your conceptual model is well on the way. You may now save the model and make a backup for future reference. The easiest work has been done. Now you will begin creating new entities. There are three main reasons for creating new entities:

1. You want to expand your language coverage.
2. You want to expand your access to the information in the database.
3. You want to add entities to optimize Ergo performance.

*Note:* We could add a fourth reason. If your database is not in 3NF, creating new entities also enable you to add conceptual clarity to your model. Maintaining conceptual clarity is equally important when customizing.

There are a variety of entities you can create, when adding entities to the model. These include:

1. Composite entities
2. Dummy table entities
3. Verb entities
4. Subclass entities
5. Instance entities
6. Adjective entities
7. Special information entities

In this chapter we will discuss the first three in this list. The others will be discussed in the next chapter.

### 7.1 Composite entities

Some concepts require links to several entities in the model. In the documentation entities corresponding to these concepts are called composite entities. A composite entity consists of more than one entity. The language term linked to a composite entity refers to information in more than one column in the database. You can have composite entities any class of concept, consistent with the information in the database. These include:

- Composite identifiers
- Composite names
- Composite attributes
- Composite embedded concepts
- Composite reports
- Composite sorters

A composite entity may consist of

- Entities that exist in the initial data model
- Entities that have been created.

For example, you may need an *address* composite entity, consisting of the column entities '*street*', '*city*', and '*state*'. The components of *address* are found in the initial data model

Or you may want an *income report* composite entity, consisting of the created entities, *'income'*, with a DB image including `SELECT (x1.salary + x1.commission)`, and *'income\_after\_tax'*, with a DB image including `SELECT ((x1.salary + x1.commission) – x1.tax)`, both of which give information about the gross and net income of salesmen. The components of *income report* must be created.

*Note:* A composite entity consisting of entities corresponding to columns or information from different tables requires that a join-path be specified, directly or indirectly linking the tables.

You create a composite entity by first classifying the entity and then choosing the *Consists of* option in the classification dialog box. The Customization Tool will then give you a list of all the entities in the conceptual model that are selectable as a composite entity components, so you can choose the entities that are appropriate. A composite entity is a concept whose entity components can only be selected from the *Consists of* window.

*Note:* You create composite entities during the classification phase of processing an entity. You can have up to 255 components in a composite entity.

## 7.2 Creating composite entities for identifiers

You need a composite identifier if

- the database contains a table having a multi-column primary key
- the conceptual model requires a subclass of an entity that has a composite identifier.

If several entities refer to the primary key columns of a table, you only need to

- create an entity
- classify it as an identifier
- select its component entities
- link it to the concept it identifies

to create and process a composite identifier. Generally, they do not need terms.

*Note:* If you create a subclass composite entity, the components of the subclass must be in the same order as they are in the super-class.

An identifier of a subclass whose super-class has a composite identifier must also be a composite identifier. Then the identifier of the subclass *must* consist of all the entities that make up the identifier of the super-class, except one. The last component of the identifier of the subclass - and it does not matter which of the components you select - must be a created entity with a DB image identifying the appropriate subset of the data values referred to by the subclass it identifies. This DB image will `SELECT` the data values referred to by the component entity and include a qualifying `WHERE` clause.

For example, if *'order'* has a composite identifier, *'order\_id'*, consisting of *'cust\_no'*



and *'order\_no'*, where *'order\_no'* might be non-unique because it is a date, a subclass, *'california\_order'*, would have a *'calif\_order\_id'* consisting of either *'calif\_cust\_no'* and *'order\_no'*, where *'calif\_cust\_no'* might have a DB image such as:

```
SELECT x1.cust_no
FROM tab x1
WHERE x1.ship_site='CA'
```

or it would have a *'calif\_order\_id'* consisting of either *'cust\_no'* and *'calif\_order\_no'* with a DB image like this:

```
SELECT x1.order_no
FROM tab x1
WHERE x1.ship_site='CA'
```

It does not matter which of these alternative ways of creating a composite identifier you choose. If there are  $n$  entities comprising a composite identifier of a super-class, there are at least  $n$  ways of creating a composite identifier of a subclass. You can select whichever one you want.

After defining the *identify* relationship you can ask:  
*list the California orders*

### 7.3 Creating composite entities for names

You need a composite name if

- the database contains a table having a more than one column naming or describing instantiations of the concept
- the conceptual model has an entity, whose name or description refers to more than one column of information in the database.

If several entities refer to the naming columns of a table, you only need to

- create an entity
- classify it as a name
- select its component entities, and
- link it to the concept it names

to create and process a composite name. Generally, the components will have terms, such as *'first name'*, *'middle initial'*, and *'last name'*, for example, where these terms refer to different columns in a table.

A name of a subclass whose super-class has a composite name will also be a composite name. But you do not have to create the composite name. You get it for free. As long as the subclass has a proper identifier, the NLE will be able to find the correct name as it parses the query and generates SQL.

## 7.4 Creating composite entities for attributes

Several component entities may together represent an attribute of a main concept. For example, an employee table may have the address columns STREET, CITY, and STATE. Entities representing these columns may together constitute an *'employee\_address'* attribute possessed by employees.

If you want to ask  
*list the employees and their addresses*

you would have to

- create a composite entity for *'employee\_address'*
- specify the term, *'address'*
- define a possess relationship between *'employee'* and *'employee\_address'*.

Similarly, employees might have a training history consisting of internal courses and external courses they have attended. You would create a composite entity, *'training history'* consisting of *'internal course'* and *'external course'*. Then you will get the information you expect when you ask  
*list smith's training history*

## 7.5 Creating composite entities representing embedded concepts

In principle, any entity can be combined with any other set of entities to form a new main concept. Non-3NF databases usually contain embedded concepts, and often these concepts have components. For example, a *'location'* concept embedded in a department table may consist of *'district'* and *'region'* entities. You might have an order-monitoring application where the concept, *'account'* consists of *'account\_no'*, *'subaccount\_no'*, and *'business\_unit'* entities corresponding to columns in an order table.

*Note:* All multi-column concepts should be identified in the pre-customization phase of the Ergo application development process. This will make it easier to customize when you reach the second phase of customization.

## 7.6 Creating composite report entities

You can create other composite entities if you want to create concepts consisting of several component entities, i.e. if you want a single term to refer to a group of entities, or if you wish to tailor a report consisting of several entities. Because you can select as the report entity's components entities corresponding to columns from different tables, report entities have an effect similar to creating a VIEW. The difference is that SQL syntax is more limited for CT-created DB images.

For example, employees might have incomes consisting of *'salary'* and *'commission'*. You very likely will want an *'income'* report. This could be a composite entity consisting of the *'salary'*, *'commission'* and also a new entity with (salary + commission) in the SELECT part of the DB image. Your report may or may not contain numerical components.

< The figure will be added in a later release >

*Figure 31 An example of a report composite entry*

The example of a report composite entity is shown above. It shows order report composed of customer, product id, product name, and sales quantity. It is used to show more information about the order entity than would be normally given by an identifier and a name.

A composite entity can be a means of generating a report that requests information from more than one column in an SQL statement. When creating reports be sure to keep make clear conceptual distinctions between them, if there is more than one of them.

## 7.7 Creating composite report entities with numerical components

Composite Entities can consist of created entities having arithmetic operations in the SELECT part of the DB image. These entities contain numerical information, subclasses, or instances. Subtypes and Instances are discussed in Chapter 8.

*Note:* Entities that contain SQL statements with Column Functions COUNT, SUM, AVG, MIN, and MAX cannot be a component of a composite entity for Ergo Release 1.

For example, if we create a Real\_Order\_Value concept, which consists of several entities, one of which contains an SQL expression to calculate the wholesale value  
SELECT (x1.quantity x x2.product price)  
FROM sales x1, products x2

and a second entity that contains the calculated retail value  
SELECT 1.10\*(x1.quantity x x2.product price)  
FROM sales x1, products x2

then the question '*What is the real order value*' should give us both the wholesale value and the retail price.

*Note:* You cannot create a composite entity out of a composite entity.

The components of composite entities must all have SQL. The entities in the model that contains SQL will be displayed in the *Consists of* window. Note that not all of these entities can be selected for the composite entity. All entities that are part of a composite entity must be associated with tables (as columns or subclass) that are linked by join-paths.

## 7.8 Creating composite sorters

If a report is needed consisting of the columns or entities that need to be ordered (order by colx, coly, colz) in a query, then we can take advantage of the composite entity approach and create a composite entity that is composed of these entity

columns in the manner that they will be ordered by. We can informally call such an entity a sorter. For example, if we create an entity, *'account\_order'*, composed of entities selected from the *Consists of window* in an order such as:

COL1, COL2, COL3, JOB, SYSTEM...

then create a possessive relationship between *account\_order* and another entity, e.g., *'client'*, consisting of information requiring the ordering of the entities in the sorter, we can now ask the question

*'List the clients by account order'*

and we will get the information ordered by the entities that *'account\_order'* consists of. With this solution an entity would have to be created for every kind of order-by question that the user would like to ask. If the user would normally order by the same permutation of entities, then this is a good solution. See Case 7.

## 7.9 Creating dummy table entities

Strictly speaking, any entity added to the conceptual model represents a new concept in one way or another. When we speak of a new dummy table entity here, we are referring to a single entity

- that either is directly linked to a DB image or is indirectly linked to a DB image by having an identifier with a DN image, and
- that has been added for the specific purposes of either enhancing language coverage or conceptual clarity, rather than for the specific purpose of accessing a subset of information in the database
- that would correspond to a table in the database if it were normalized.

Dummy table entities are often created if the database is not in third normal form. If you have a *'location'* table and you have an *'employee name'* column and a *'employee code'* column that identifies the employee name. If both of these columns are in same table and they are not identifying and naming a *'location'* itself, then the database is not in third normal form. The concept, *'employee'*, is embedded in the concept, *'location'*. This means that we have to create a dummy table entity called *'employee'*. The dummy *'employee'* entity will have the *'employee name'* entity as its name, and the *'employee code'* entity as its identifier. These dummy table entities often create the appearance that the table is in third normal form, since tables using these types of entities are normally non-3NF tables.

The name and the identifier of a dummy table entity (even though they correspond to real columns in the database) will usually not have relationships specified between them and the main table entity. Only the dummy table entity itself should have relationships to the table entity (if there should be any relationships at all). For example, if we have a non-3NF Agent table whose column entities contain embedded concepts referring to clients and agents, then we have to create a new concept called client, and establish relationships between the client id and client name to this new concept. We would not then define any relationships between these

entities and the 'agent' entity. Only the client entity will have a defined relationship such as a 'possess' or 'have' relationship if necessary.

*Note:* If the table were in third normal form, the dummy table entity would be a table by itself. Dummy table entities do not need a DB image, if there is an identifying column entity in the initial data model. If there is not, then an identifier must be created, or a DB image can be written directly to the entity.

### **Two kinds of dummy table entities**

There are basically two types of dummy table entities:

- Self-identifying dummy table entity - the identifier-of the concept is not reflected directly in the data model reflecting the structure of the database, i.e., the identifier is not physically a column in the table. In this case, you may write SQL directly to the concept.

< The figure will be added in a later release >

*Figure 32 A simple self-identifying concept*

- Non-Self identifying dummy table entity - the identifier exists as a physical column in the table, and it contains the SQL identifying the concept. An identifier may also be created with SQL written to the identifier, if the identifier did not exist as a physical data value in the table. However, this is not necessary. When this is done, it is usually to preserve clarity in the model.

*Note:* Refer to section on Creation of identifiers to subclass on corollary rules of thumb.

< The figure will be added in a later release >

*Figure 33 A non-self-identifying concept*

### **Dummy table entities and composite entities**

A dummy table entity may represent a new concept. As a new concept, it may be a composite entity; be identified or named, or both, by a composite entity; or lack an identifier or name, or both.

Both Numeric and Non-numerical data of data are supported for each category and a combination of several things are possible:

- Those that have an id (Single or Multiple column entities)
- Those that have an id, name (Single or Multiple column entities)
- Those that have only a name (Single or Multiple column entities)
- Those that have no id and no name (Single or multiple Column entities)

There are not clear rules for customizing, if the question is not resolved by the structure of the database or the language you want to use in your queries. There is also a trade-off between performance and conceptual clarity. You should create the

constructs that enable you to get your queries processed, and once processed, processed most efficiently.

*Note:* Dummy table entities (having no direct SQL) may have multiple-entity identifiers and/or names consisting of several components with semantically complex terms. All entities representing concepts need an identifier, or some SQL, to define the entity in order for it to be used in a query. Queries using terms linked to composite dummy table entities will be more efficient if identifying and naming columns are included among the list of components of the entity, rather than directly specified as identifier and name.

## 7.10 Creating verbs

A verb is a word or a group of words, which usually express action or a state of being. In Ergo, verbs concepts enable you to ask questions like

*'Where does John work' or 'What did the customers buy*

To create a verb, you must

- Create an entity
- Give it an entity name
- Classify it as EVENT
- Give it a term, selecting the *verb* option
- Select the *Syntax* option and specify it verb complement
- Continue with the *Syntax* option and specify it prepositional complement

*Note:* All verbs are classified as *Event*.

After creating the verb entity, the next step is to define its relationship to other entities in the model.

### 7.10.1 Defining the verb complement

The verb complement window allows you to select the proper syntax to be used corresponding to the verb. The verb complement is typically an object that follows the verb. Verbs that have direct objects are classified as transitive verbs, and those verbs that do not require a direct object are called intransitive verbs. For example:

#### **Intransitive with no object**

*Who assigns*

*Managers assign from departments.*

#### **Transitive with a direct object**

*Who assigns what*

*Managers assign projects*

#### **Transitive with indirect object**

*Who assigns what to whom*

*Managers assign projects to employees.*

## Transitive with 2 direct objects

*Who named whom whom/what what*

*Who named the modules MOD1?*

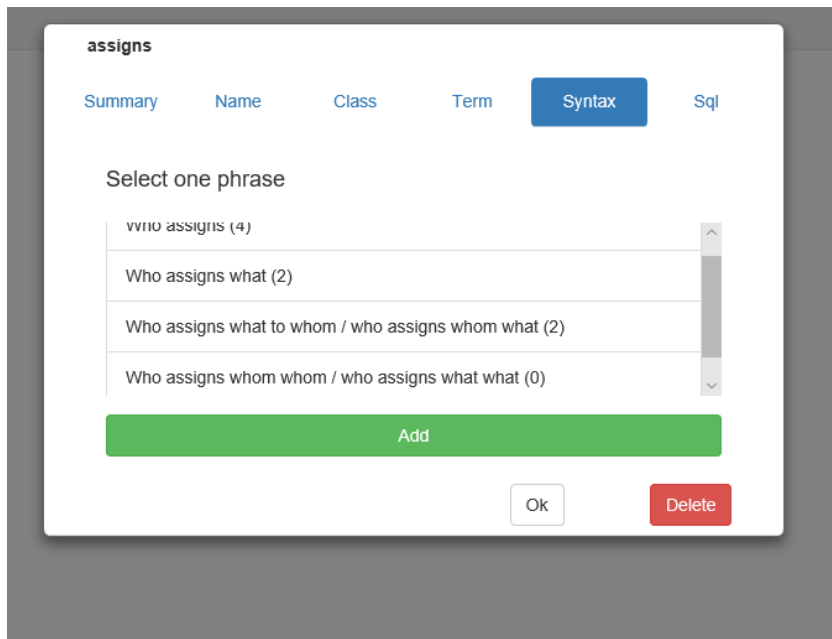


Figure 34 Subject predicate direct object indirect object

It is very important that the right verb syntax is selected, i.e. the right verb syntax terms of how you will be using the verb. Select the Intransitive form of the verb when the context of the sentence calls for that syntax. However, you are limiting the usage of the verb to that particular syntax ONLY.

*Note:* To use the verb 'assign' in all three of these grammatical contexts require that one single 'assign' verb be created.

- Selecting the third complement option will guarantee that the other two contexts will be covered provided that the first option 'Who assigns' does not require a prepositional complement. This is discussed in the next section.
- Selecting the first means you want to limit the use of the verb.

### 7.10.2 Specifying prepositional complements

The next step is to select prepositions for the verb. If you feel that you need any special prepositions with the verb in order for the users to be able to use the verb right you can select those prepositions from a long list. For instance, if you have created the verb talk, and you wanted the user to ask questions like 'What did the customers talk about' then you need to specify that 'about' is a special preposition for the verb 'talk'. Verb should be binary (always joining two entities). You should have two different 'eat' verbs if you want to have both *Customers eat in restaurants.*  
*Employees eat food.*

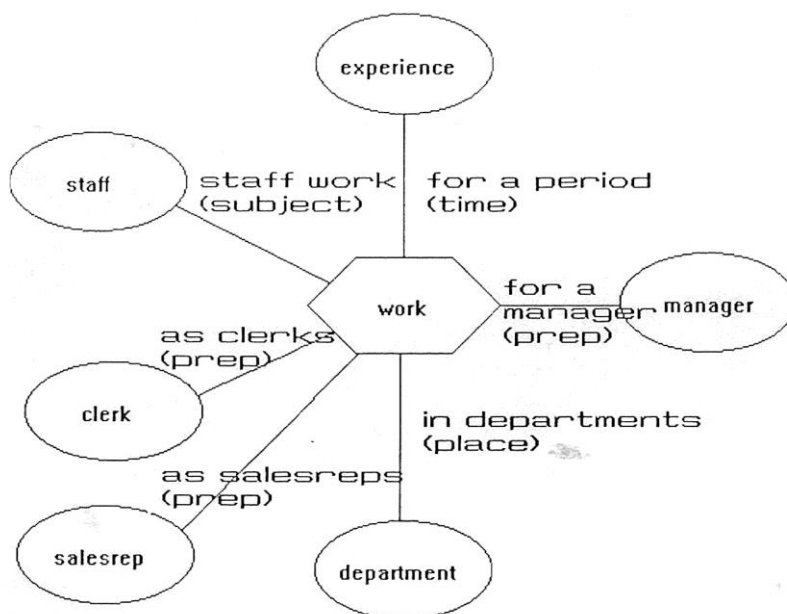
That is, make two verbs if the verbs are used in different grammatical contexts, as in this case an intransitive (without an object) and a transitive (with an object) context. Notice that the verb *EAT* is followed by the preposition *IN*. There are variations to how prepositions are defined with verbs.

In the previous example '*customers EAT IN restaurants*' *IN* is automatically defined for a place relationship. You do not need to define them at all since these prepositions come for free or automatically depending on the relationships of the entities. The next section on verb relationships talks more about this.

Prepositions can also be selected from the prepositional complement box. This results in the creation of prepositional relationships if one defines this verb with the preposition taken from the prepositional complement box, (VERB + PREP) then there is a lot more grammatical flexibility in this. One can ask questions such as:

*'What projects does John work with Smith on at LDG '*  
*'On which projects did John work?'*

Let s define the verb work as shown in the following diagram.



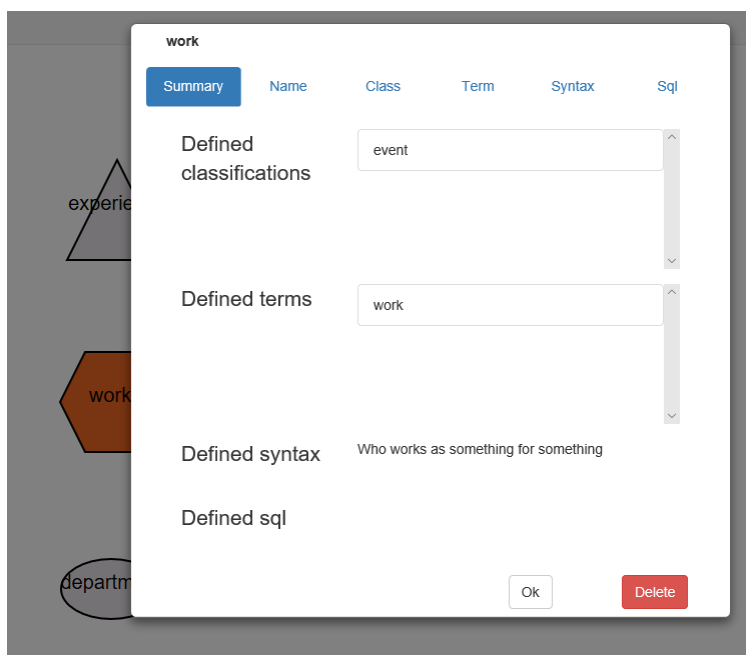
**Figure 35 Verb “work” with prepositional complements**

Here we can see from the diagram that three prepositional relationships have been defined:

- Staff work as clerks
- Staff works as sales-representatives
- Staff works for a manager

The verb syntax decides which relations will be available





**Figure 36 Defining syntax for a term**

Depending on the relationships of the entity, a preposition may be needed to be defined. Note that any relationships that include a preposition is inherited by a subclass only if the subclass has the same preposition defined. However since subtypes inherit the properties of their super-types, a verb should be created for the super-type, since this will be inherited by the subtypes.

The third way that prepositions can be defined is to define them within the term itself of the verb, in other words it accompanies the terms of the verb. For example we can define the verb 'work on' as on whole term for the verb as in 'John works on the project.' When we define the preposition that accompanies the term, then we are limited in using the verb in sentences where the preposition is always adjacent to the verb. For example, if we defined the verb 'work on' then we can only ask questions such as

'What does John work on?'  
 'Who works on project X?'

See Case 10

### 7.10.3 Defining verb relationships

After both verb and prepositional complements have been established one can define relationships between the verbs and their respective entities. These are either prepositional, locative, or time relationships.

#### Verb complement relationships:

*Who interviews?*  
*What is bought?*  
*To whom is someone assigned?*

#### Prepositional relationships:

*What does employee work as? Clerk*

**Locative relationship:**

*Where does employee work in? department*

**Time relationship:**

*Which employees worked yesterday? employees*

Depending on the relationships of each entity, prepositions can be generated automatically for use. In the above example, if we have a locative relationship that generated the preposition 'IN' automatically between work and department, if we need a preposition 'IN' between work and an entity classified as a material thing or noun, we need to define a preposition from the *Syntax box* because a noun or thing does not generate the preposition IN. For example: *Customer works in financial industry.*

Note that when defining verbs one should NOT define both a locative relation and a prepositional relationship. Use the prepositional relationship only when the preposition is not covered by one of the other relationships, and if only one or two relationships give the desired meaning.

For example: It is better to use locative or time relationships so that you can get 'At location', 'In location' automatically.

## 7.11 Creating passive verbs

A verb is in the active voice if the subject is the doer of the action and a verb is in the passive subject is the object of the action or it does not have a subject at all. For example:

*Who assigns projects?*

*Managers assign projects (assign in the active voice)*

*What were assigned by the managers?*

*The projects were assigned by the managers (assign in the passive voice)*

To customize a verb in the passive voice one should create a verb and give it a term in the active voice.

From the *Verb complement* box, choose the transitive form of

*'Who assigns what'.*

An object is needed to form an object relationship between the verb locate not between the projects and the verb assigned, choose the relationship

What is assigned? projects.?

From the *Relationships* windows select the relationship

*'Who assigns? manager'*

between manager and the verb assigns.  
A subject relationship.

Verbs in the passive voice can be easily recognized by a verb in the past participle preceded by the verb ' *TO BE* ' Example:

*is driven, was driven, were driven, has been driven*

See Cases 9, 10, and 11 for more information on Verbs.

## 8. CUSTOMIZATION PHASE 3 - EXTENDING THE CONCEPTUAL MODEL

Now you have completed implementing the main concepts in your model, including adding the main verbs you want to use to ask questions. Now you will extend your model by adding:

- Subclass entities
- Instance entities
- Adjective entities
- Special information entities

Adding these entities will enable you to expand your language coverage and ask questions more naturally.

### 8.1 Creating subclass entities

Subclasses will let you select certain subsets of data values or subsets of larger data sets on the basis of some property, quality or criterion, and it will let you do so by using language to refer to these subsets.

There are three kinds of subclasses:

1. A subclass may refer to a column of data values all of which are subsets of another column of data values. Often these are inclusion dependencies identified in table mode. They always represent a concept different from the concept represented by the super-type. The SQL for these subclasses are generated by the CT.
2. A subclass may refer to a subset of data values in a single column on the basis of some quality, property or criterion. The idea of creating a subclass is to select all those rows in the table that, represent this quality, property or criterion. In this case the idea of subclasses depends on the WHERE clause in a SQL statement. An example of a subclass is to create the entity '*administrative employee*' as a subclass of '*employee*'. The Natural Language Engine should in this case produce SQL that will look through the table and select all employees that work in administration. For example, it might check all employee numbers and departments selecting those where DEPT = '*ADMINISTRATION*'.
3. Subtypes also refer to subsets of larger data sets represented by a composite entity. When creating a subclass of a composite entity all of the components of the subclass may be the same as the structured entity except at least one. Note:
  - a. A subclass of a composite entity is also a composite entity.
  - b. At least one component of a subclass of a composite entity must be a subclass of a component of a composite entity. For example, if you have an composite entity '*address*' consisting of '*street*', '*city*' and '*state*', and you create another composite entity '*california address*' that is a subclass of '*address*', then '*california address*' will consist of '*street*', '*city*' and a third entity '*california*'. This entity is a subclass of '*state*', which is a component of '*address*'. In the '*california*' entity you would put the SQL which would select all addresses where STATE = '*CA*'.

*Note:* The order of the components of the subclass of a composite entity must be the same as the order of the components of the super-type. For instance, if *'address'* consists of *'street'*, *'city'* and *'state'* in that order, then *'california address'* must contain *'street'* and *'city'* in the same order. The entity *'california address'* should also contain an entity CA state as mentioned above.

Subtypes are usually classified as a subclass of the entity that they are subclasses of (selecting rows from). So, *'manager'* would be classified as a subclass of the employee entity. An exception to this rule might be identifiers that are subclasses of other identifiers. Unless a term is required in the application for the identifier of a subclass, subclass id can be classified at the same level as super-type\_id.

A subclass inherits the relationships defined for its class. If a product has a price, every subclass of product (such as tool) has a price. Any relationship that includes a preposition is inherited by a subclass (subclass) only if the subclass has the same preposition defined.

## 8.2 Creating identifiers for subtypes

A subclass may have the DB image directly linked to it by writing SQL when the entity is defined, or it may be linked to an identifier which has the DB image specified. Maintaining conceptual clarity is the main advantage for creating an identifier for a subclass.

So, if *'administrative employee'* is a subclass of an *'employee'* table entity and this table has an identifier, then the *'administrative employee'* subclass may also have an identifier. If, on the other hand, *'administrative employee'* is a subclass of the *'employee'* column (and this column does not have an identifier), then the *'administrative employee'* subclass will not need an identifier.

The identifier of a subclass does not have to have a term, and it should be classified as an identifier. It must also be connected with the subclass, and the identify relation must be selected.

*Note:* The SQL of a subclass may be written to the subclass itself or to the identify entity that identifies the subclass. Normally, we would write the SQL directly to the subclass. However, for the sake of clarity, it may be useful to make a distinction between a subclass and an entity that identifies a subclass. This is especially relevant when creating subclasses of structured entities that are also structured entities.

See Case 13, Ambiguous Data and Case 14, Subtypes of Entities with Multi-Column Keys

## 8.3 Creating instance entities

Instance entities are similar to subclass entities. They allow you to access specific subsets of information with specific language terms. But there are some special features of instance entities:

- Instance entities refer to data values in a single column of a single table in the database
- Instance entities are specially created to handle coded data values.
- Instance entities are created to give the coded data values meaning in natural language.

While an instance entity can be created from any combination of codes specified in the WHERE clause of the DB image, the default status of single-valued instance entities is disjoint, i.e., they do not intersect. For example, the subclasses, 'good employee' and 'senior employee' may intersect, but 'men' and 'women', instances of employees based on data values 'M' and 'F', do not.

*Note:* Instances may be created from data values that carry information as flags, such as 'M' and 'F' for 'male' and 'female', or actual codes, such as 'ADM' for 'administration'.

When specifying language terms for instance entities, you are allowed the same linguistic options as for subclasses.

## 8.4 Writing DB images

This section describes the form of SQL statements that may be used to specify the database representation of entities.

### General constraints

The syntax of SQL statements that can be used within Ergo entities are subject to these general restrictions:

- The SQL for an Ergo entity may only consist of a single SELECT statement. Consider creation of structured entities if more than one column is desired.
- The SELECT statement can return only a single column or a single expression.
- The names of tables or views must be specified with the name of the creator and with a correlation name.
- A column is specified by its correlation name and column name. Ergo uses this for SQL optimization.
- Host variables may not be used.
- GROUP BY, HAVING or ORDER BY clauses are not permitted.
- Concatenation and Sub-strings are not permitted.

An example of a valid Ergo entity SQL statement is:

```
SELECT X1.ID FROM EPE.STAFF X1 WHERE X1.JOB = 'CLERK'
```

*Note:* Refer to Customization Users Guide Handbook for a complete discussion on this subject.

### When writing SQL to an entity consider these points:

1. If you want to select rows from a table entity (the employee table), the SQL should look something like this:

```
SELECT x1.<table identifier>
FROM <creator user id>.<table name> x1
WHERE x1.department = 'ADMINISTRATION'
```

The < table identifier > is the column that identifies the table. You have to specify the database name for the table identifier in the SQL, and not the name or term that you have given it. If the table has more than one identifier, you should only specify one of the identifiers in the SELECT statement and not the composite entity or more than one of the identifiers. The < creator user id > . < table name > is the database name for the table. This is the same name that you saw for the table when you listed it in the ADD TABLES window in rework table mode. x1.department is the database name for a column in the < table name > table that tells us if an employee works in administration or not.

2. Often when databases are in non-3NF, it is desirable to create subclasses by specifying directly a subset of data values in a column. If you want to select rows from a column entity (for example the 'salaries' column), the SQL for 'managerial salaries' could look something like this:

```
SELECT x1.<column name> (= salary)
FROM <creator user id>.<table name> x1
WHERE x1.manager_flag = 'Y'
```

The <column name> is the database name for the column, in this case 'wage level', that you want to select rows from. The <creator user id> . <table name> is the database name for the table that the <column name> and 'wage level' is in. If 'manager\_flag' and <column name> (= salary) are in different tables then the SQL would look like this:

```
SELECT x1.<column name>
FROM <creator user id>.<table for column> x1,
     <creator user id>.<table for manager_flag> x2
WHERE x2.manager_flag = 'Y'
```

## 8.5 Creating adjectives

Adjectives are modifiers of nouns. In Ergo an adjective entity describes a quality or characteristic of another entity. For example, 'senior' in 'List the senior clerks', modifies the entity clerk. Adjectives work in principle like subclass entities. The difference is that the term defined to the adjective entity will always be used as a modifier to a noun entity. Thus an adjective entity specifies a subset of the data values represented by the concept it modifies on the basis of some quality, property or criterion. There are two types of adjective customization techniques:

1. The specific adjective: This adjective corresponds to a triangular entity in the model with a DB image, and whose term is specified to be an adjective. When a model reflects a 3NF DB, this type of adjective is usually used when the noun entity it modifies exists as a table or column in the database, or as a sub-class in the conceptual model. It is used only to qualify that particular entity. In this case, the

adjective is classified as a subclass of the noun it modifies. For Example:

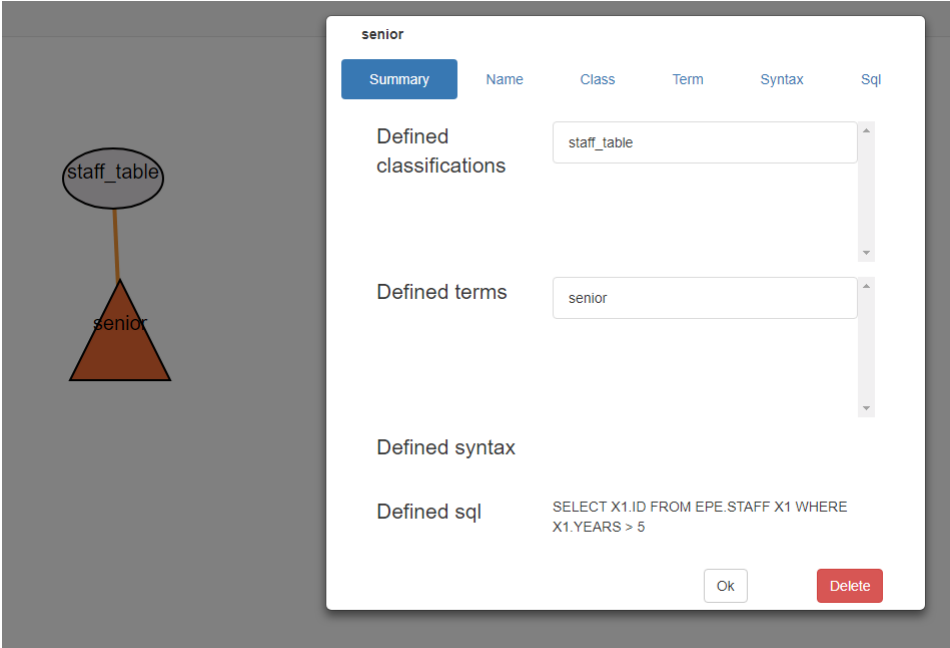


Figure 37 Defining an adjective entity

The *years* column in the STAFF table contains the number of years that an employee has been working in an organization. To allow questions like "Who are the senior employees", we create an entity and specify an adjective term, 'senior'. Just like a subclass, we specify a DB image for this entity to select rows that qualify the information in accordance to the SQL modification, for example, WHERE X1.YEARS > 5.

3. The implicit adjective: This type of adjective is specified when we process any entity and, in specifying its term, select it as the compound term option, ADJ + NOUN. For Example:



**European manager**

Summary   Name   Class   **Term**   Syntax   Sql

**Term**     

**Category**

- Noun
- Verb
- Adjective
- Proper name

**Category**

- Adjective + noun
- Noun + noun
- Adjective + name
- Name + noun
- Noun + name
- Adjective + noun + noun
- Single noun

Verify or change the following forms

**positive:**  

**comparative:**  

**superlative:**  

Verify or change the following forms

**one:**  

**two:**  

How do you refer to manager

**Figure 38 Defining an implicit adjective**

*Note:* All subtypes of a noun modified by an adjective entity inherit the modification.

### 8.5.1 Prepositional complements:

You can define prepositional compliments to accompany the adjective. If we have defined the adjective *experienced* and we wish to use it in the context '*experienced IN*', then, we choose a corresponding syntax of '*someone/ something is experienced IN*'. This enables queries such as '*List sales-representatives that are experienced in accounting.*'

Any other subclass of the concept that the adjective modifies inherits the properties of the adjective. For instance, if we had the adjectives '*profitable*' and '*western*' that modify the entity '*location*' then we can ask for the '*profitable western location*'. If '*california location*' is a subclass of location then we can ask '*list profitable, california locations*'.

An adjective is inherited by the super-type of the subclass entity that the adjective modifies and vice versa, the subclass inherits the adjective that modifies its super-type. This is on the condition that the entities INTERSECT each other or are not DISJOINT with each other.

The following diagram illustrates this:

< The figure will be added in a later release >

### *Figure 39 Inheritance*

Using the above example, we can ask the questions

*Who are the senior clerks*

*Who are the senior sales-representatives*

*Who are the senior employees*

If Senior modifies clerk and is defined as a subclass of clerk, we can still ask the same questions, because adjectives can be inherited also by the super-types.

*Who are the senior sales-representatives*

*Who are the senior employees*

*Who are the senior clerks*

IF we wanted to limit the adjective only to Clerk, then Senior should be defined as DISJOINT to sales-representatives in the INTERSECT definition.

## **8.6 Handling adverbs**

An adverb modifies a verb or other adverbs and adjectives. Example:

*She sang extremely well (modifies the adverb well)*

*He usually works hard (modifies the verb works)*

Adverbs can be distinguished in form, the most common of which is its ending in LY. However, note that there are adverbs that are not of this form such as *FAST* as in '*He ran FAST*', the adverb *WELL*, as in '*He played WELL*'.

In handling adverbs, we should first understand the context of how the adverb is used. It is very similar to the way an adjective is customized. We create an entity that imbeds the meaning of the adverb, and classify it as a subclass of the NOUN which it the doer of the action (verb). Like a subclass, and an adjective, the adverb concept, distinguishes a new "*concept*". For Example, '*Which Employees usually work hard*', is conceptually different from '*Which employees usually work slow*'. The difference in

customization between the adjective and adverb is that the adverb (term) is customized as part of the verb it modifies. In the conceptual model, the adverb is created between the verb and the noun that is the doer of the verb.

Key important steps to observe:

- "Adjectivize" the adverb and create a subclass concept embodying the adjective form.
- Customize the adverb as part of the verb
- Link the subclass and the verb (containing the adverb)
- Select appropriate syntax
- Consider other grammatical contexts

< The figure will be added in a later release >

*Figure 40 Illustration of the adverb work HARD*

See Case 11.

It is important to create a verb that contains the Adverb term itself. The verb is then customized as USUALLY WORK HARD. The concept (Entity) of the adverb need not have same term as the verb. Since subclasses HARD WORKER inherit the primary term of its super-type (*employees*). The subclass (adverb) contains the SQL to select the appropriate qualifying rows.

With this set up we can now ask the questions

*Who usually work hard?*

*Which sales representatives usually work hard?*

*Which manager usually works hard?*

*(assume salesrep and manager are a subclass of employee)*

< The figure will be added in a later release >

*Figure 41 Usually working hard adverb*

## **8.7 Creating special information entities**

When customizing, the customizer often focuses on specific information he wants from the database, usually qualified in some way. This information may be:

- An entire table
- An entire column
- Several columns from one table Columns from more than one table
- A subset of data values from a single column qualified by a range of those values
- A subset of data values from a single column qualified by a range of data values from another column

- A subset of data values from more than one column qualified by a range of those values
- A subset of data values from more than one column qualified by a range of data values from another column
- A column from one table qualified by a set of data values in another table
- A subset of data values from a single column qualified by a range of data values from a column in another table
- A subset of data values from more than one column qualified by a range of data values from more than one column in another table
- A subset of data values from a single column qualified by a range of data values from more than one column in another table
- A subset of data values from more than one column qualified by a range of data values from more than one column in another table

Entities can be created to handle all of these needs.

### 8.7.1 Entities referring to DATE data

Queries including *'last year'*, *'last month'*, etc.

### 8.7.2 Entities referring to numerical data

#### 1. Counted-by relations:

The counted by relations are used for columns that express an amount or a quantity. This relationship can be used if you have an entity *'inhabitants'* that list the number of people for each country and you want to ask *'how many inhabitants does Sweden have.'*

In order to make a counted-by relation you need to create a new entity. This entity can have the term *'population'* for example. The *'population'* entity has to be classified as a QUANTITY. A relation is then set up between the *'inhabitants'* column and the *'population'* column, and the relation *'How many INHABITANTS are there? POPULATION'* is selected.

See Case 15, Entities With Numeric Data - Counted By Relations.

#### 2. Measured-by relations:

The measured-by relation enables you to rank the instances of the entities. You will be able to ask questions like *'List the top ten sales reps.'*

If we assume that we have a *'sales'* concept which possesses a sales amount entity and a salesrep entity, then we can create the MEASURED-BY relation between the two entities. Sales Amount will be classified as a quantity. Thereby, the instances of the entity *'sales rep'* will be measured by the instances of the entity *'sales'*.

See Case 8.

#### 3. Unit-of-measure entities:

An entity classified as a quantified property has a corresponding unit of measure

associated with it. For example, if we classify an entity as '*value*' we can choose a unit of measure (UNIT OF MEASURE WINDOW) dollar corresponding to this. If we need to CREATE new units of measure that are, not available, we have to create new entities to embody this new type of unit of measure. We then classify these entities as a UNIT-OF-MEASURE.

Unit-of-measure entities are used to specify the unit that the instances of an entity should be measured in. If we wanted to measure sales in Pounds then we would have to create a unit-of-measure entity. The unit-of-measure entity would be classified under unit\_of\_measure value and given the term '*pound*'. Sales would then be classified as a subclass of value, and the UNIT OF MEASURE 'pound' would be specified for this classification.

See Case 3. Entities with Numerical Data - Unit of Measure.

## 9. CUSTOMIZATION PHASE 4 - ENHANCING THE MODEL AND REFINING THE SYNTAX

Now that all the basic entities and relations have been created, it is time to identify  
What kinds of questions can be asked  
What kinds of questions users still need to be able to ask  
What can be done to the model to meet the users' needs.

These tasks remain to be done:

- Asking test questions
- Pre-customizing for further work
- Adopting a customization strategy for model enhancement
- Enhancing the model
- Testing the model

### 9.1 Asking test questions

You should now test the basic model at this point. You should bring back the end-users to determine once again what kinds of queries still need to be supported. If special query areas or themes can be identified - e.g., sets of queries focused on payroll information, education information, or customer service information, to name a few - then this will be helpful, because the customizer can set up query logs, which will make it easier for the users to ask questions.

*Note:* The customizer must make clear to the users that they cannot ask questions any way they want. They can only ask questions in accordance with what the customizer has made possible while developing the conceptual model. The customizer must tell the users what kind of questions they can ask!

At this point, you should be able to ask these kinds of questions:

1. Queries asking for all instances of a concept  
*Example:* *list the employees*
2. Queries using the possessive relationship for all instances  
*Example:* *list the salaries of the employees*
3. Queries using the possessive relationship for a  
*Example:* *what is smith's salary*
4. Queries using a composite entity  
*Example:* *list the addresses of the employees*
5. Queries using a subclass entity  
*Example:* *list the price of the electrical products*
6. Queries using a verb  
*Example:* *which customers ordered electrical products*
7. Queries using an adjective  
*Example:* *what products are electrical*
8. Queries asking for a sum of instances of a concept  
*Example:* *how many employees are there*
9. Queries for a DATE interval  
*Example:* *which customers ordered electrical products in June*

10. Queries using the locative relationship for all instances of a concept

*Example: who works in new york*

11. Queries using the temporal relationship for all instances of a concept

*Example: how long has smith worked as a clerk*

12. Queries using backward possessive adjective reference

*Example: list the employees and their wage level*

13. Queries asking for a measure of a particular value

*Example: list the top 5 customers*

14. Yes/No queries for a single instance

*Example: Is smith a clerk*

If you cannot ask these types of queries and you expect to be able to, there is something in the model that should be corrected. If problems arise, you can use the CT Reporting Facility to produce a report of your model. You can study the information about the entities and relations to see if there is anything that should be changed. Very likely, you have done most things correctly up to this point.

## 9.2 General rules of thumb for asking questions

Here is a summary of general rules that should be followed when you ask questions:

1. You can use the SQL wildcards, % and \_: in queries just as you might if writing SQL. Thus, for example, you can ask,

*list employees with names like s%*

2. If a term or name contains an apostrophe ('), then the whole term or name must be placed between single quotation marks, e.g.:

*who works for 'o'brien'*

3. In order to create charts, at least one term in the query must refer to a column in the database that contains numerical data values.

4. If you want to ask a query, ordering the results, like

*list products by year by division by manager by orders*

you can create a composite entity consisting of the 'year', 'division' 'manager' and 'order' entities. If you give that composite entity the term 'product order' then you can ask:

*list the products by product order*

and have the results ordered.

5. Many queries you want to ask may presuppose something. For example,

*who wrote more prolog modules than lisp modules*

can presuppose that someone writes both prolog modules and lisp modules. Ergo

provides an answer based on this presupposition. If you want the case where someone writes at least 1 prolog module and no lisp modules (hence more prolog modules than lisp modules), you must re-phrase your question:

*who wrote at least 1 prolog module and no lisp modules*

However, the current version of Ergo will not give you all this information in the same report.

6. You should strive for logical clarity. If you ask,

*who works in new york and los angeles*

you probably mean

*who works in new york or los angeles*

assuming someone does not work in both places. Ergo is usually logically strict with these kinds of queries.

7. Modifiers and modifying clauses should be concatenated with the term modified. For example:

*of all new policies that cancelled before the year was out, how many that were effective in 1990 were cancelled for non-payment of premium.*

has a phrase, 'effective in 1990' split from the term it modifies, i.e., policies. Place modifying phrases next the term it modifies when it is correct grammatically. Here you can also re-phrase the question by putting 'how many' at the beginning and by changing the idiomatic expression 'year was out' to 'end of the year':

*how many new policies effective in 1990 that cancelled before the end of the year cancelled for non-payment of premium*

8. Avoid idiomatic expressions in your queries unless you explicitly customize them.
9. Arithmetic calculations can be made directly in the query. You can as

*list number of managers / number of employees*

to get the ratio of managers to employees.

10. You cannot embed a comparative expression in one element of a set of coordinated noun phrases. For example, you cannot ask

*How many programmers either programmed more modules than smith or programmed in prolog*

Here, 'programmed more modules than smith' is a comparative phrase that is



embedded in *'either ... or'* construction. You must ask two questions here.

11. You can refer to data values in a column by specifying the data value before the term of the concept representing the column. For instance, if *'product'* is the term for the *'product'* column, and *'motor'* is a data value in the *'product'* column then we can ask a question like: *'List motor products'*.
12. If you want to refer to data values that contain words that are in the base dictionary, then the data value has to be in quotes in the queries. For instance *'New York'* has to be in quotes since *'new'* is in the base dictionary.
13. In order to process YES/NO-type questions involving subtypes the verbs of the questions have to be directly linked to the subtypes, i.e. the subtype will not inherit the verb relationship from the super-type for YES/NO-questions.
14. It is not possible to use backward reference in a query that starts with *'Who are...'*.

### 9.3 Pre-customizing for further work

After identifying what has been done, you have to determine what must be done. You have to do some pre-customization work. The tasks include:

1. Identifying and categorizing questions you want to make work
2. Identifying what you have to do to the model to enhance your language coverage
3. Choosing a customization strategy.

#### 9.3.1 Identifying and categorizing questions

You now know that a lot of questions work, but there are a lot more that users want to ask. You have to identify these. In addition, you should categorize the questions according to the themes or subject matters. There are two reasons for this:

1. You can get the application out to your users faster if you create usable query logs as fast as possible, while you customize incrementally.
2. You can design mini-conceptual models for these query categories, enabling us to identify:
  - the entities you have to work with
  - the entities you have to create
  - the relationships you have to define
  - the language terms you need to specify.

The end-users can be brought in again to guide this process.

*Note:* The syntactic (grammatical) structure of each query should be studied carefully. Some questions may have to be changed before you can decide how to enhance the model to accommodate them. You should strive for generic solutions wherever possible.

### 9.3.2 Mini-modeling

While designing a complete conceptual model for the entire database in advance of customization may be ineffective, especially for large databases, modeling a cluster of entities for the purpose of enhancing language coverage is usually useful at this point. As soon as you have your queries categorized according to theme or subject matter, you can take a category and:

- identify the entities the queries are related to
- identify what entities you have to create
- identify what conceptual and language relations you have to define
- determine what additional language terms you have to add to meet the users' needs.

Mini-modeling clusters of entities makes model enhancement much easier. The part of the model you are working with is limited, making it easier to comprehend conceptually what has to be done. Your customization goals at this phase of the process becomes clearer, especially when needed queries have been identified. And any potential errors made can be corrected quickly.

### 9.3.3 Depth-first customizing

So far you have followed a breadth-first customization strategy. Very likely, you have done most things correctly. You have also probably saved your backed up your file, so you will not have to do all this work again.

Now what we want to do is identify a cluster of entities and customize them as thoroughly as possible to provide maximum language coverage, based on our understanding of our end-users' needs and our understanding of the limitations of Ergo. It is very efficient at this point to start using a depth-first customization strategy. The advantages of this include:

- Enhancements to the model and language coverage will have a clear focus.
- Usable query logs can be created faster.
- Incremental results known to be correct can be saved and returned to should errors be made later.

## 9.4 Model enhancement -- adding to the model

The goal of this phase of the customization process is to expand language coverage. Expansion of language coverage can either be achieved by creating new entities and relationships or by refining existing entities and relationships. In model enhancement, the goal is to expand language coverage by adding new entities or relationships.

The model enhancement tasks include:

1. Adding entities
2. Adding new conceptual relations
3. Adding language terms
4. Adding new language (syntax) relations

## 9.5 Adding entities

Very likely you will want to add entities to your model so you can link language terms to tables, columns, or data values in your database. You may need to add these kinds of entities to your model:

- New base entities
- Dummy table entities
- Composite entities
- Subclass entities
- Instance entities
- Special information entities
- Verb entities
- Adjective entities
- Adverbial constructs

What you need to add depends on:

- What kind of questions you want to ask
- The structure of your database
- What you already have in your model.
- 

### 9.5.1 Adding new base entities

Adding a new base entity to the model suggests you have a concept that refers to a set of data values in a column. You may want all the data values, or only a subset of them. If you ask a question using language terms associated with the new concept, you expect this information returned.

*Remember:* all new base entities added to the model require a DB image. You can write the DM image directly by selecting the SQL option in the *Define* entity dialog box, or you can create a separate identifier with the DB image and link it to the new entity.

The DB image of any entity can only refer to a single column in the database. If your concept requires limiting the data values in the column on the basis of criteria elsewhere in the database, these are specified in the WHERE clause of the DB image. If the limiting criteria depend on information in another table, you should specify the appropriate table joins in the DB image.

The DB image may also be an operation on data values from more than one column that would appear in a report as a single column. For example, *'income'* may consist of *'salary + commission'*. As long as the result would appear as a single column in a report using SQL, you can specify such operation in the DB image. The major exception to this is character SUBSTR concatenations.

### 9.5.2 Adding dummy table entities

If your database is not in 3NF, then your database may contain an identifier or name of a concept that itself is not explicitly found in the database. If this is the case, then

very likely you will want to add and that would correspond to a table if the database were normalized. However, note these subtleties:

- If there is an identifier without a name, for example, a PURCHASE-CODE, identifying a purchasing method, then the values can be treated as coded data values, and instance entities may be created for them.
- If there is only a name, for example, CONTACT, embedded in a table, then the identifying DB image would contain a SELECT DISTINCT for the concept represented by the new entity. A problem may arise if more than 1 instance have the same name.
- If the table contains both an identifying column, e.g., CITY\_CD, and a name column, e.g., CITY, then the dummy table entity would not have a DB image itself, but would get one after it is linked to the identifying entity. In addition, in this case, it would be this dummy table entity that enters into relations with other entities.

### 9.5.3 Adding composite entities

You can add composite entities for

- concepts related to information in more than one column, e.g., address,
- concepts related to subsets of information in more than one column, e.g., chicago address,
- information you need from more than one column that would entail asking questions difficult to ask in natural language, e.g., a monthly report consisting of several columns, and
- structured reports, using composite sorters in the query.

Sometimes there are simple concepts associated with composite entities, as for addresses. Other concepts for complex information may be clear, as for monthly reports. But sometimes you have to be inventive to access exactly the information you want.

### 9.5.4 Adding subclass entities

If a subset of a set of data values has a particular meaning to users who ask questions, you can create subclass entities and add new language terms to the application dictionary. Remember that if you create a subclass of a composite entity, the order of the data values referred to by the subclass must be the same as those referred to by the super-class.

### 9.5.5 Adding instance entities

You can create instance entities to give specific language meaning to coded data values. The codes often have well-known meanings for users, who would like to use

terminology rather than codes in their queries. It may also happen that users remember the terminology but not the codes, especially in large DBs. Thus, you can expand language coverage by adding instances.

*Remember:* There is no functional difference between instance and subtype entities, since their DB images are written in accordance with the same principles. The engine processes them both exactly the same way. However, since instance entities are designed to handle coded data values, they are treated as disjoint by the engine. You can also select instances as a component of a composite entity.

Subtype entities are not treated as disjoint because data values associated with their DB images can intersect.

### 9.5.6 Create special information entities

You can create special information entities for

- Units of measure
- Arithmetic calculation
- Date expressions, like 'last month'

Often you want the database information in a special form. This often involves putting a special SQL in the information entity. For instance, you might need to add a '*base salary*' column to a 'sales commission' column and call that '*salary*'. To do this you would have to create an information entity called '*salary*' where you would write the following SQL:

```
SELECT X1.BASE SALARY + X1.COMMISSION...
```

Information entities are especially useful for arithmetic data.

### 9.5.7 Adding verbs

Only a few basic verbs like '*to be*' or '*to have*' are contained in the base dictionary of Ergo. All other verbs have to be customized.

### 9.5.8 Adding adjectives

Adjectives can also greatly enhance the model by defining language terms for subset of rows in a column or set of columns. Adjectives are usually used in conjunction with subtypes, and adjectives can either be defined within the subtype (in the terms), or separately as an adjective entity.

### 9.5.9 Adding adverbial constructs

Ergo does not directly support the use of adverbs, but there are ways, in which users can use adverbial constructs in queries. As adverbs are modifiers they also denote a subset of rows in a column or set of columns. They can therefore be customized similarly to adjectives by using subtypes.

## 9.6 Adding new conceptual relations

Some entities may need more than one classification. Most entities however should only have one classification; so it might be necessary to change the classification so that all the appropriate relationships can be defined.

## 9.7 Adding language terms

If there is no entity representing the concept, create a new entity with the appropriate term. If there is an entity representing the concept, then add a synonym language term to the entity. Note that if there are more than one entity with the same term, and these entities have similar language relationships, multiple paraphrases will be generated. Terms should therefore be as distinct as possible to avoid ambiguity.

### 9.7.1 Adding new language (syntax) relations

Language relations need to be specified for each type of query that you want to ask. If existing relations do not cover the query, new relations need to be specified.

As you work with the model, you can refine the syntax by:

- Adding new conceptual relationships to existing entities.
- Defining new language relationships between existing entities.
- Adding new prepositions.

## 9.8 Types of questions

In this section we take a look at some of the kinds of questions that can be asked, together with customization notes and comments concerning limitations. This list is not exhaustive. The list could have been presented in a different way. However, considering that language is infinite in scope, our intention is to provide a sense of the kinds of questions you can ask with Ergo, and what you have to think about while customizing. The questions below are divided into three categories:

1. Imperatives  
These questions start with an imperative verb (*List, Show, Display*). The subject of these questions is the assumed 'you'.
2. Direct Questions  
These questions contain a subject and a predicate and can either be WH-questions (*When, Where, What, Who*) or Yes/No-questions (*Are there...*).
3. Generalized Query Types  
These questions are categorized according to different linguistic categories.

Here is a summary of general rules that should be followed when you ask questions:

## 9.9 Imperatives

An imperative is a directive to do something. It has an omitted, but implied, subject, you, and a verb phrase. The verb phrase always consists of verb + direct object. The verb has its imperative form.

## Imperatives

*Note:* In Ergo, the main verbs that can be used in imperative queries are:

list	print	type	find
give me	give	show me	show
tell me	display	fetch	get
locate	identify	select	name
search for	report	extract	

The verb phrase may contain a complement. And the direct objects may reflect any noun related to any entity in the conceptual model, provided the entity has a DB image or consists of components having a DB image. The objects may also be modified by adjectives or complementary phrases. In Ergo the direct objects can be represented by entire phrases. Thus, in the question,

*list the California customers*

Ergo interprets the phrase, 'California customers', objectively, i.e., as an object, if it represents a complete term linked to a single entity.

### **How the NLE works - Things to remember**

1. The NLE breaks a query down into its components.
2. Any word or phrase specified as a term by the customizer is interpreted as being a component of a query. If a term or name contains more than one part, all the parts together make up the component.
3. Queries should be as straightforward and direct as possible. Queries should not include obvious ambiguities unnecessarily, since processing uses resources.
4. A query should not contain an evaluation unless it has been specifically customized. Ergo will not be able to figure out who does a good job, unless it is told.
5. The classification of the entity associated with a term determines the term's part of speech: entities linked to verbs are classed as events; nouns anywhere else; and adjectives as a subclass of the nouns they modify or as properties.
6. The part of speech of a term in the application dictionary determines how it can be used in queries.
7. Queries are turned into a logical form. So you should remember about this when you ask your queries. The more logically clear queries are, the easier it is for the NLE to process them.
8. The information in a query can be ordered by one or more concepts. The more concepts that a query is ordered by, the slower the processing of the query will be. The NLE will also not create any paraphrase for queries that are ordered by many concepts. This does not mean that you cannot order by several

columns. If you want to order by several columns, you can use a composite sorter, if the order is fixed.

9. To process YES/NO-type questions involving subtypes, the NLE requires that verbs in the questions be directly linked to the subtypes, i.e., the subtype will not inherit the verb relationship from the super-type for YES/NO-questions.
10. It is not possible to use backward reference in a *Wh*-query.
11. Conjunctions are handled in two ways by the NLE. The NLE either breaks the query up into two queries, i.e. the query *'List managers and department'* might become the two queries *'List managers'* and *'List departments'*, or the NLE thinks of the query as one question. The way the NLE handles the query will depend on how the questions is formulated and there is almost always a way to reformulate the query to get the NLE to handle the query in the desired way. For instance, if you ask: *'List valuable and electrical products'* the only interpretation that the NLE will generate is *'List valuable products'* and *'List electrical products'*. However, by appending the phrase *'by price'* at the end of the query, the NLE will generate the paraphrase *'List products that are valuable and expensive. Sort the information by price'*. Similarly *'List products that are valuable and electrical'* will generate a broken up interpretation since it contains a conjunction but no order-by. However, if you ask *'List valuable products that are electrical'* you will get the paraphrase *'List products that are electrical and that are valuable'*. This is because there is no conjunction in the query.

### 9.9.1 Examples of imperatives using single objects

Here the verb has only one object.

#### Examples:

*list employees*

*show the weekly sales information*

*draw a histogram of the sales-representatives' income*

**Customization notes:** The database-related terms used in the queries have to be specified in the application dictionary. In the second example, 'weekly sales information' could be a structured entity consisting of several components. These components must have DB-images. In the third, Ergo will display a graph. Here the direct object of the imperative, namely, *'histogram'*, is found in the base dictionary.

**Limitations:** The DB-image of a component of a structured entity like *'weekly sales information'* cannot contain entities with column functions.

### 9.9.2 Examples of imperatives using compound objects

Here the verb has more than one object.

#### Examples:



*list the orders and products*  
*show salaries + commissions*  
*display 1Q sales and 1Q profits*

**Customization notes:** In the first example, if 'customers' and 'account numbers' refer to information in different tables, a join-path must exist between the two tables. In the second example, each salary and commission together with the sum of the two, will be displayed for each row in the table. The third example generates a two-part query, one report for 1Q sales and another for 1Q profits.

**Limitations:** Without further qualification, the use of the conjunction 'and' generate two-part paraphrases and reports.

### 9.9.3 Examples of imperatives using compound objects and backward reference

Here the possessive adjective refers back to the object of the verb.

#### Examples

*List managers and their staff*  
*Show the divisions, products, and their orders by division*  
*Display the employees and their total wage breakdown*

**Customization notes:** A simple coordinated noun phrase like 'managers and their staff' will generate two-part queries. In the second example, the addition of the phrase, 'by division', will lead to a single-part paraphrase with the report sorted by the division column of the table. In the third example, 'total wage breakdown' can be a structured entity consisting of, for example, 'salary' and 'commission'. The use of anaphora implies that correct possessive relationships have been defined between all the entities referred to in the query.

**Limitations:** Imperatives with noun phrases having more than two components, as in the second example above, often cannot be paraphrased. However, you can still process the query and generate the correct SQL.

### 9.9.4 Examples of imperatives with subordinate qualifying clauses

Here the objects are modified by adjectival clauses.

#### Examples

*List employees who work on project 1401 and code in fortran*  
*Display branches and managers working in the new york region by branch*  
*List programmers who code in assembler and pascal*

**Customization notes:** At the moment, the NLE treats coordination of compound subjects and objects differently. Thus, in the second example, appending 'by branch' to the query will generate a single paraphrase and group the report by branch. If you

leave out *'by branch'*, you will get a 2-part paraphrase. (This will very likely change in a future version of the product.) However, in the third example, you will get a 1-part paraphrase because the coordinated nouns, *'assembler'* and *'pascal'* constitute a (compound) prepositional object, which the NLE can process.

**Limitations:** Compound subjects with compound predicates, an extension of the first example, do not always work. Thus, the NLE may fail if you ask, *'list the managers and employees who work on project 1401 and code in fortran'*.

### 9.9.5 Examples of imperatives using prepositional phrases

The prepositional phrases may refer to time or place.

#### Examples

*List the planes from France*

*List the employees at the Chicago office and their salaries*

*Show the salaries of the employees in June in a pie chart*

**Customization notes:** The prepositional relationships must be specified through classification or defined using the Syntax option in the Define entity dialog box. In the third example, *'head office'* happened to be a data value. If you want to use *'Chicago office'*, where *'Chicago'* is the data value as in the second example, a subtype entity for *'chicago office'* has to be created. However, we could say, *'list the employees in chicago and their salaries'* because *'in'*, which syntactically agree with *'chicago'* in this example, is passed on by classifying department as a place.

**Limitations:** For charging limitations, see Case Study 2 below.

### 9.10 Direct questions

Direct questions, also called Wh-questions, use the question words, who, what, when, where, and how. How is counted among the Wh-question words. Direct questions require a verb. The verb phrase may be active or passive, transitive or intransitive.

#### 9.10.1 Examples of questions using single subjects

In active-voice queries, the subject does the action. In passive queries, the subject is acted upon.

#### Examples

*What salesmen sold more than 10 policies in June*

*How many product were ordered in June*

*Are there any managers that earn more than 50000*

**Customization notes:** In these examples, the correct verb complement must be set up. In the third example, you can follow up with a refer-back question, *'who are they'*, and get the right answer.

**Limitations:** In direct questions, you cannot append *'sorted by ...'* or *'by ...'* at the end of the query.

### 9.10.2 Examples of questions using compound subjects

Here there is more than one subject.

#### Examples:

*Which managers and salesmen made more than 100000 dollars last year*  
*What customer and branch office placed sales order 122343*  
*Did any managers or staff receive the merchandise*

**Customization notes:** In the first example, using *'last year'* assumes that the date column is a DB2 DATE data value. Because *'last'* is not in the base vocabulary, *'last year'* must be customized as a subtype entity of the date column.

**Limitations:** Compound subjects will always generate paraphrases with more than one part.

### 9.10.3 Examples of questions using single objects

#### Examples

*How many programmers wrote assembler modules*  
*Who is Smith's manager*  
*Did any customers order motor products*

**Customization notes:** The second example assumes that a possessive relationship has been defined with *'manager'*. In the third example, *'motor'* is a data value, referring to a product description classified as a name.

**Limitations:** In the first example, the NLE will count the number of appropriate rows in a table. But if the value refers to a subset of quantities, the NLE will not currently process the query. In the third example, you cannot ask, *'how many customers ordered motors'*, unless *'motor'* corresponds to a subtype entity. But if *'motor'* is a data value in the PRODUCT column, you can ask the as shown here. If products possess a location, you can ask, *'show the Chicago products'*.

### 9.10.4 Examples of questions using compound objects

Here there is more than one object.

#### Examples

*Who programmed assembler modules and prolog modules*  
*Did any customers from 'New York' buy hammers and saws*  
*Who wrote 'The Idiot' and 'Crime and Punishment'*

**Customization notes:** In the third example, The Idiot and Crime and Punishment

were placed between single quotes because *'the'* and *'and'*, which are part of the book titles, are contained in the base dictionary. This is also the case for the second example, because *'new'* can be found in the base dictionary. Otherwise, the NLE can usually recognize direct objects of verbs, where the data values contain more than one word or have different character lengths, as long the correct language relationships have been defined.

**Limitations:** As noted, in queries, when using terms corresponding to data values that have components found in the base dictionary, the terms should be placed between single quotes.

### 9.10.5 Examples of questions using single predicates

#### Examples

*Are there any clients in Denver*  
*Where did flight 201 arrive from*  
*Which calls were replied to by department managers*

**Customization notes:** The second example has a *locative of source* relationship. This is selected from the *Define relationship* dialog box: *From where do flights arrive? incoming airport*, for example. Here, *'incoming airport'* must be classified as a place. In the third example, *'last month'* must be created with the correct DB-image. Note the passive form of this query.

**Limitations:** In the third example, the verb, *'reply to'* has to be created because the NLE cannot handle back-to-back prepositions.

### 9.10.6 Examples of questions using compound predicates without direct objects

Here the query contains more than one verb, at least one with not object.

#### Examples

*Who studied in California and works in New York*  
*Does anyone work in the head office and earn less than 30000*  
*Are there any flights that start in Los Angeles and fly via Amsterdam*

**Customization notes:** Both verbs, of course, must be linked to the subjects.

**Limitations:** The syntax for queries with *'is there...'* or *'does anyone...'* will not always be inherited by subtypes, as in the second and third examples. So if you want to ask, *'is there any manager who works in the head office and makes less than 30000'*, then the verbs must be linked to the subtype. However, this will generate multiple paraphrases for the single predicate queries.

### 9.10.7 Examples of questions using single subjects and compound predicates

Here the subject may do perform more than one action.

## Examples

*Which customers bought life insurance and live in 'new york'*  
*Does any manager earn more than 100000 and like to play golf*  
*How many salespersons sold motors and report to manager Smith*

**Customization notes:** These will all work as long as the correct language relationships have been defined. In the second example, you could customize the verb, 'to play', and then add a synonym, 'like to play', and conjugate 'like'.

**Limitations:** In the second example, it is assumed that 'salary' is classified as a number, quantity, or value. In addition, both verbs must be linked to 'manager'. The third example requires that 'motor' be a subtype (for example, of 'product'), otherwise the query must read, 'how many salespersons sold motor products and report to manager smith'. Many compounds generate 2-part paraphrases. For example, this will happen if you ask, 'how many salespersons have 6 years' experience and report to manager smith'.

### 9.10.8 Examples of questions using compound subjects and compound predicates

Here more than one subject may perform more than one action.

## Examples

*Did any customers and sales-representatives study statistics and live in New York*  
*Which managers and employees work in chicago and earned a commission*  
*Are there any staff and managers that work in finance and know SQL*

**Customization notes:** Both subjects must be linked to both the predicated.

**Limitations:** Currently you cannot use a comparative with compound predicate. Thus, for example, 'which managers and employees work in Chicago and earned more than 45000' will not work.

### 9.10.9 Examples of questions using compound predicates and compound objects

Here more than one object is acted upon in more than one way.

## Examples

*Did any customers buy and use hammers and saws*  
*Which customers watch and play tennis and golf*  
*Are there any programmers that code and test Pascal modules and PL/S module*

**Customization notes:** Here, of course, the direct object form of the verb complement, for example, who buys what, must be selected for each verb. The

assumption here is that there are two different verbs. But two verb synonyms can also be used.

**Limitations:** NLE performance decreases as the number of components of the coordinated verb and noun phrases increases.

#### 9.10.10 Examples of questions using predicates with prepositions

Typically prepositions involve place or time.

##### Examples

*Which communications were replied to by smith*

*Which accounts is Smith responsible for*

*Are there any products that were bought by preferred customers*

**Customization notes:** The first example requires that a verb, 'reply to', be specified because the NLE cannot process back-to-back prepositions. In the second example, you can select 'responsible for something' from the *Verb syntax* box. The third is example is a simple passive defined when the verb relationships are set up.

**Limitations:** Coordinated verb phrases in passive form will generate two-part paraphrases. Thus, for example, 'what salaries or commissions were earned by managers or employees' will generate a two-part paraphrase, one asking for salary information, the other for commission information.

#### 9.10.11 Examples of questions using strings of modifying clauses

Here subjects or objects are modified by modifying clauses.

##### Examples

*Did anyone process the orders from the agents concerning the households residing in the western region*

*What motors made in California plants were bought by new york customers*

*What manager confirmed the sale of property to clients during April*

**Customization notes:** If you know your users want to ask queries like these, it is useful to set up a mini-model of the entities involved to make sure that all the necessary relationships are identified and properly customized. In the first example, very likely you would select the *Syntax* option from the Define entity dialog box and chose 'agents concerning something' from the prepositional complement list. In the second example, note that 'new york' is not in single quotes. This is because 'new york' is part of a subtype term rather than a term referring to a data value in a table. The third example only looks tricky. Assume that 'manager confirm' and 'sale of property is confirmed'. Select 'who confirms what to what'. Clearly, 'what is something confirmed to' is 'client'. Finally, from the *Syntax* box, select the prepositional complement, 'clients during something.' The query will now work.

**Limitations:** You can only define four prepositions for each noun, in two sets of two

each. Thus, in the first example, agents can be linked to 'orders' in two ways and 'households' in two ways, e.g., orders from/by agents concerning/from households. If you need more prepositions, you have to create additional entities and re-define the appropriate relationships.

### 9.10.12 Examples of questions using single adjectives

#### Examples

*Are there any overdue accounts*  
*When does the most expensive flight arrive*  
*How many managers and programmers are senior*

**Customization notes:** Adjectives are inherited by subtypes. This you can ask, 'are there any overdue tax accounts', without linking the adjective to 'tax account', assumed here to be a subtype of account. Each adjective needs the correct SQL, usually selecting the identifying column of the entity it modified. Thus, the SELECT part of 'senior' in the third example, would be ID, for example, and the WHERE part would be WHERE YEARS > 5'. Note that adjectives can be used as predicate adjectives, as in the third query.

**Limitations:** The NLE does not yet support the comparative and superlative forms of the adjective. Thus, in the second example, the answer would be the same for 'expensive' or 'more expensive', unless a special entity was created for 'most expensive'. Coordinated subjects, as in the third example, will lead to two-part paraphrases.

### 9.10.13 Examples of questions using compound adjectives with single subject

Here more than one subject or object may be modified by more than one adjective.

#### Examples

*Which products are valuable and electrical list the new and profitable divisions*  
*Are there any loans that are large and due yesterday*

**Customization notes:** In the third example, 'yesterday' is found in the base dictionary. To make this query work, you have to link 'due' to a date column in DB2 DATE data value format. Here, 'due', as a participle, works like an adjective. For phrases like 'last week' or 'last year', you must create entities and define the correct relationships.

**Limitations:** Coordinated adjectives lead to two-part paraphrases. Thus, 'how many products are valuable or mechanical' will first ask for the number of electrical products, then for the number of mechanical products. You could ask, using the imperative form, 'list products that are valuable and electrical by price' and get a one-part paraphrase. Note that if you ask, 'list products that are valuable and electrical by product', you are asking for a sort on the same columns the report would be sorted

by from the beginning, and the NLE does not handle this. You could get a single-part paraphrase, however, with *'which valuable products are electrical'*.

#### 9.10.14 Examples of questions using compound adjectives with single objects

Here a single object can be modified in more than one way.

##### Examples

*Who bought saws that are long and not expensive*

*Which policy holders own new and expensive cars*

*Are there any plants that made profitable products that are not expensive*

**Customization notes:** Observe in the first example that customizing the negation of an adjective is usually not necessary. However, this example will generate a two-part paraphrase. To avoid this, you can ask, *'who bought long saws that are not expensive'*.

**Limitations:** Complex syntax like the third example may not generate a paraphrase. But the query will be processed with the correct SQL.

#### 9.10.15 Examples of questions using compound adjectives with compound objects

Here more than one object is modified in more than one way.

##### Examples

*Who bought saws and screwdrivers that are short and not expensive*

*Which salesmen sold profitable and small radios and TVs*

*Did any customers order red cars and trailers that are inexpensive*

**Customization notes:** If only saws and screwdrivers are short and not expensive, performance will be enhanced you create subtypes for products and link the adjectives to them.

**Limitations:** Multiple compounds like these examples often will not generate paraphrases. However, the query will be process with the correct SQL.

#### 9.10.16 Prefaced questions

##### Examples

*I want to know who bought saws and screwdrivers*

*I would like to see the products and their prices in a bar chart*

*I would like to know which customer placed the largest order last year*

**Customization notes:**



## Limitations:

Phrases you can use to preface a query:

*I want to know*

*I would like to know*

*I wish to know*

*I want to find out*

*I would like to find out*

*I wish to find out*

*I want to see*

*I would like to see*

*I wish to see*

## 9.11 Generalized query types

Examples of questions using transitive verbs

### Examples

*Which customers bought motors and serviced generators*

*Who completed modules yesterday*

*Which sales-representatives made their quotas last year*

**Customization notes:** These examples all require that correct the verb complement has been specified.

**Limitations:** As the first question indicates, you can have more than one transitive verb in a query. However, when you have more than one verb coordinated, you cannot embed a comparative phrase in either of the elements. For example you cannot ask, '*who bought motors and serviced more than 3 generators*'.

### 9.11.1 Examples of questions using intransitive verbs

Intransitive verbs have no object complements.

### Examples

*which customers live in new york*

*which students live in sweden and arrived yesterday*

*which prices rose or fell*

**Customization notes:** Many verbs can be used transitively or intransitively. If you think you may need to use an intransitive verb transitively, you should select one of the transitive verb complements when you process the verb.

**Limitations:** As the first question indicates, you can have more than one intransitive verb in a query. However, when you have more than one verb coordinated, you cannot embed a comparative phrase in either of the elements. For example you cannot ask, '*which students live in sweden and arrived later than smith*'.

### 9.11.2 Examples of questions using both transitive and intransitive verbs

You can use verbs with or without complements in the same query.

### Examples

*which customers live in new york and play golf*  
*which students live in sweden and study computer science*  
*which prices rose last year and effected the price index*

**Customization notes:** You could also use the same verb transitively or intransitively. For example, you could ask, '*which customers play in new york and play golf*', as long as a transitive verb complement is selected and the relationships defined.

**Limitations:** When you have more than one verb coordinated, you cannot embed a comparative phrase in either of the elements. For example you cannot ask, '*which students live in sweden and studied more subjects than smith*'.

### 9.11.3 Examples of questions using anaphora

#### Examples

*List the customers and their addresses by customer*  
*List the products and their prices by product*  
*List the managers and their yearly salaries by department*

**Customization notes:** These examples all require that correct possessive relationships be defined.

**Limitations:** You cannot use backward reference in the same sentence with '*who are..*', such as '*who are the employees and their staff*'. Use the imperative form for anaphora. Without the '*by ...*' tag, you'll still get 2-part paraphrases. Paraphrases may not be generated for queries like the third example, because '*department*' is not part of the main information set asked for. However, the SQL will be correct, as well as the information being asked for.

### 9.11.4 Examples of questions using date ranges

#### Examples

*Which products were sold between june 1st and june 15<sup>th</sup>*  
*What was the profit for the motor division between 1/1 and 6/1*  
*What tax accounts become overdue between 7/92 and 9/92*

**Customization notes:** These examples all require the correct possess and time relationships to be defined. If '*tax accounts*' is a subtype of '*accounts*', then '*tax accounts*' will inherit the relationships that '*accounts*' has. See Case 12 below.

**Limitations:** The date columns in the database have been declared as e.g. DB2 DATES.

### 9.11.5 Examples of questions using numerical ranges

#### Examples

*Who ordered between 10 and 15 disk drives*  
*What programmers earn between 50000 and 55000 dollars*  
*What managers have between 5 and 25 employees*

**Customization notes:** We assume that *employees* is an entity in the data model. A possess relationship has to be set up between the entity *'manager'* and the entity *'employee'*. *'Dollars'* is a predefined unit of measure that should be selected for the entity *'salary'*. A relationship should then be defined between *'earn'* and *'salary.'* *'Disk drives'* is a subtype of *'orders'*. A subject has to exist for the verb *'order'*.

**Limitations:** The entities referred to must be measurable and have a measured by relationship defined. See Case 13 below.

### 9.11.6 Examples of questions asking for sums

#### Examples

*How many people live in the USA*  
*How many customers bought motors in October*  
*What is the sum of the commissions for each branch office*

**Customization notes:** The proper place and possess relationships have to be set up.

**Limitations:** You cannot sum the data values of a subset of a column, i.e. you cannot sum the data values of a subtype. See Case 2 below. Note also that summing of rows in a database referring to subtype entities can take more system resources.

### 9.11.7 Examples of questions using adverbials

#### Examples

*Which employees usually perform well*  
*Who never orders motors*  
*Which sales reps always make their quotas*

**Customization notes:** The NLE can handle the negative adverbial of manner as in the second example. Otherwise, processing adverbials require a work-around. See Case 9 below.

**Limitations:** You cannot *'split the infinitive'* with an adverb without customizing around it. Thus, you cannot ask, *'who likes to often play golf'*. Also, you cannot interpose an adverbial between the auxiliary part of a verb and the main verb, as in *'which product has never been ordered'*.

### 9.11.8 Examples of questions with indirect objects

#### Examples

*What products do the sales-representatives in chicago sell to customers*  
*Which managers gave awards to their employees*  
*What plants made motors for customers that live in New York*

**Customization notes:** In order to use indirect objects, the verb has to be defined in the *Syntax* window as being able to use indirect objects. For the last example, two verbs have to be set up: *'make'* and *'live'*. The relationship *Customers Live* has to be set up, and the verb *'live'* does not have to be defined as being able to use an object or an indirect object. The verb *'give'* is not in the base dictionary and has to be created. In the first example, for *'sales-representatives in chicago'*, a possessive relation must exist between *'salesrep'* and *'location'*.

**Limitations:** You cannot append *'by ...'* to sort the report by a particular column when using indirect objects with predicates. Thus, you cannot ask, *'list products sold to preferred customers.'*

### 9.11.9 Examples of with deictic time expressions

#### Examples

*Who completed modules yesterday*  
*Who completed modules in June*  
*What was the profit last year*

**Customization notes:** *'Yesterday'* is in the base dictionary and therefore does not have to be customized. *'Last year'*, however, does have to be created by using the CURRENT-DATE function. See Case 12 below.

**Limitations:** This, that, these, and those are not parsed by the NLE. If required, special workarounds must be customized. The date columns in the database have been declared as e.g. DB2 DATES.

### 9.11.10 Examples of questions using negations

#### Examples

*Find sales-representatives who are not from the Atlanta region*  
*What plants are not profitable*  
*What employee who is not a manager earned the most*

**Customization notes:** Normally, negation requires no special customization. However, when using workarounds, as required by the third example, an effort must be made to make sure the SQL is correct. In this case, the SQL for *'most\_earning\_employee'* must include sub-select both for the case the employee is a manager and the case when he is not a manager.

**Limitations:** Two negations in the same query will not work. Thus, you cannot ask, *'list sales-representatives who are not managers and who do not work in the Atlanta region'*.

### 9.11.11 Examples of questions using passives

#### Examples

*What modules were assigned by the managers*

*List the salaries earned by the managers and staff at the head office'*

*What products were sold in the New York area*

**Customization notes:** If 'New York area' is not a sub-type, it must have single quotes around 'new york' because 'new' is found in the base dictionary. The entities 'manager', 'staff' and 'products' all have to be related to the verbs as objects. No subjects should be defined for these verbs. If the verbs are to be used in the active voice as well, then to different verbs should be defined: an active verb with a subject and a passive verb without a subject.

**Limitations:** The use of conjunctions as in the second query complicates the processing in the NLE. These types of queries will therefore take longer time to process. There are also two possible interpretations to a query that uses a conjunction:

- 1) One query:  
*'List salaries earned by both managers and staff' or*
- 2) Two queries:  
*'List salaries earned by managers' and 'List staff'.*

The NLE will sometimes choose one of these interpretations and continue processing using just that interpretation. If the NLE has disregarded one of the interpretations there are often a way to rephrase the question to that you get the other interpretation.

### 9.11.12 Examples of questions using temporals

#### Examples

*Who completed a module on or before July 17*

*What accounts became overdue after March 2*

*Which employees took vacation in January*

**Customization notes:** A temporal relationship has to be defined between the date entities and the verbs. See Case 12 below.

**Limitations:** The date columns in the database have been declared as e.g. DB2 DATES.

### 9.11.13 Examples of questions using locatives

#### Examples

*Which employees work at the Dialogue Technologies?*

*How many orphan accounts are there in the western region  
Which flights arrive via 'New York'*

**Customization notes:** The appropriate locative relationships have to be set up. If 'Orphan accounts' is a subtype of 'accounts' then 'orphan accounts' will inherit the relationships of the 'accounts' entity. So, if 'accounts' has a locative relationship with 'western region' then no such relationship need to be defined for 'Orphan accounts'. 'Western region' could be a subtype of 'location' where the states belonging to the western region are selected in the WHERE-clause of the 'western region' entity.

**Limitations:** You cannot use a sequence of locatives in your query, especially queries referring to data values having more than one lexical component. Thus, you cannot ask, *'which customers bought motors from sales-representatives in chicago, los angeles, and 'new york'.*

#### 9.11.14 Examples of questions referring to previous questions

##### Examples

*How many are there  
Where do they work and how much do they earn  
Are they expensive*

**Customization notes:** Simple refer back questions will work as long as the appropriate relationships have been specified. In addition, you have to set the 'Refer to questions' option on the Action bar of the Query Interface. You can refer back to up to five questions.

**Limitations:** You can only refer back to a previous question once without having to ask another question.

#### 9.11.15 Examples of questions using elliptical phrases

##### Examples

*customers  
products with price  
which manager earns the most*

**Customization notes:** The first two examples are short for 'list customers' and 'list products with price'. The NLE cannot parse the third example without a customized work-around. You can create an entity like 'max earning manager with the correct SQL, including a sub-select on the salary column, and link it to a verb, 'earn the most'.

**Limitations:** You cannot use coordinated noun phrases in these kinds of queries. Thus, you cannot ask, *'products and price'*. In the second example, *'with price'* works because products have prices, and the possessive relationship has been defined.

## 9.12 Testing the model

- Test your model incrementally as you develop it.
- Start by asking questions that use only nouns.
- Ask questions that include verbs.
- Ask questions that include adjectives, subclasses, and instances.
- Ask questions that test units of measure, prepositions, and other special relationships.

**What to look for:** For each question that you test, you should check that:

- The question is interpreted.
- At least one interpretation matches the sense of the question.
- The SQL generated matches the interpretation.
- The SQL is correctly executed.

**Possible problems:** Some problems you may encounter while developing your model are:

- No interpretations of your question.
- Words highlighted in your question.
- No interpretation matches your question.
- You get a lot of interpretations.
- The SQL does not match your question.
- The SQL is correct but the answer is not.
- You get an unexpected null value.
- A question that needs a temporary table is not executed.

**No interpretations:** If your question is not interpreted:

- Follow any guidance in the error messages.
- Check highlighted words.
- Try asking the question another way.
- Check that you have defined the terms and relationships used in the question.
- Break the question into simpler parts to identify the problem.
- Rebuild the question by degrees.

**Highlighted words:** If some words in your question are highlighted:

- The highlighting means that Ergo did not recognize the word.
- If the word is a data value, the highlighting does not indicate an error.
- If the word highlighted is a term used in the question:
  - The term has not been defined in the model, or
  - The term has been spelled incorrectly.

**Wrong interpretations:** If none of the interpretations matches the question you asked, check:

- the terms
  - the classification, or
  - the relationships,
- for all the entities used in the question.

**Many interpretations:** If your question produces many interpretations:

- There may be unnecessary relationships between entities.
- One of the entities may be classified incorrectly.
- The same term may be used by more than one entity. This is permitted and may not necessarily be an error.

**Wrong SQL:** If the SQL statement does not match the question you asked, check:

- the terms
- the classification, and
- the relationships

for all the entities used in the question.

**Wrong answers:** If the SQL appears to be correct but the answer is wrong, check that data values in the SQL exactly match those in the database.

**Null values:** If your question returns an unexpected null value:

- The SQL is executing an expression that includes a null value.
- You can avoid this by not allowing null values in your database.
- If this is not possible, add a clause WHERE ... IS NOT NULL to the entity SQL.

**Temporary tables:** If questions requiring temporary tables are not executed:

- These are generally questions referring to a range (from A to B) or a ranking (the three biggest..).
- Either you lack authority to create tables, or you do not have sufficient space for them.
- Contact your Ergo administrator to correct the problem.

### Common mistakes

- Table mode.  
Check that you have:
  1. Defined keys and other column information.
  2. Defined inclusion dependencies for subsets with a special meaning.
  3. Added join-paths between columns used to join tables.

### Common mistakes

- Names and identifiers.
  - The identifier should normally be associated with a key column.
  - The only classification should be identifier.
  - The only relationship should be the '*what uniquely identifies... relationship.*'
  - Do not give an identifier any term except one that expresses identity.
  - Treat names similarly, except for the association with a key column.

### Common mistakes

- Classifying subclasses.  
First classify the primary concept appropriately.  
Then classify the subclass under its super-class.  
Be sure to do this for subsets and supersets defined in *Table* mode.

Other mistakes:



- Identifying subtypes
- Classifying quantity vs. quantified property vs. number
- Column to exclude
- No referential Integrity
- Customizing Verbs in Passive Voice
- Date Functions
- Multi-Column Concepts - NON-3NF TABLES
- Understanding Unit of Measure

### Error logs

- All failed questions are kept in an error log.
- Consult your Ergo administrator to find out what the log-file is called
- This log can give useful information about how your application is working.

**Promoting a model:** When you consider that your model is ready for use, contact your Ergo administrator, who can then make it available to users on the host.

## 9.13 Application development speed

1. Make sure each entity has easily distinguishable names.
2. Make sure each entity has easily distinguishable primary terms.
3. Avoid unnecessary tasks:
  - a. Multiple classifications
  - b. Redundant terms and syntax
  - c. Incorrect SQL
  - d. Redundant relationships
  - e. Unnecessary subtypes or other entities

The following is a summary of Rules of Thumb for Customization

### General rules of thumb for customization

1. Entity names may have no more than 25 characters.
2. Entities that queries are linked to should be properly classified.
3. Language terms may have no more than 80 characters.
4. Language terms as used in queries must have the correct part of speech specified.
5. All language terms in the queries must be customized (as part of the application dictionary) or be found in the base dictionary.
6. Correct language (grammar) relations must be specified.
7. For verbs, the correct verb complement must be specified.
8. Prepositions not associated with entity classification must be specified by selecting prepositional object phrases from the *Syntax* option in the *Define* entity dialog box. For instance, if one entity is classified as a place, the *'in'* preposition can be used in queries if the entities in the query have a locative (place) relationship specified between them. If the *'in'* preposition is explicitly specified with the *Syntax* option, then multiple paraphrases can be generated for queries with these entities. To avoid redundant paraphrases, select *'in'* from the *Syntax* option when the object of the preposition is not classified as a place, for example.

9. All concepts used in queries must have a DB-image or be identified by an entity that has a DB-image.
10. Components of a structured entity should have a DB-image or be identified by an entity that has a DB-image.
11. Components of a structured entity may not contain entities with column functions (SUM, MIN, etc.) in the SELECT part of the DB-image.
12. If a structured entity consists of components intended to access information from different tables, the tables must be linked with a join-path, even if they are already linked with an inclusion dependency.
13. If an entity is measurable, only one '*measured-by*' relationship may be defined for it.
14. In order to create charts, at least one term in the query must refer to a column in the database that contains numerical data values. The corresponding entity must be classified as a number, quantity, or value.
15. The information in a query can be ordered by an entity if there is a possess relationship between that entity and an entity in the query. The order-by-entity can refer to a single column or be a composite entity that refers to several columns. If the order-by-entity is a composite entity, then the information in the query can be ordered by several columns.

## 10. FINAL TEST AND EVALUATION

The model should be tested as the work progresses. Normally, changes to the model will be incremental so that changes made will not affect work that has been done previously. It can therefore be sufficient to test only those areas of the models that have been added. At this stage, however, all the questions will have to be tested and the reports that are generated should be evaluated.

It is also useful to evaluate the training at this point, and to give feedback to the trainer. Also, the authors would gladly receive any comments that you might have about this guide or about the product.

### 10.1 Testing

Before the application is made available to target users, it would be prudent to test it with a set of queries. This can be done incrementally. There are three useful stages in the customization process for testing:

1. As soon as language terms and synonyms have been added to the basic data model generated by the *Edit Schema* display, users may pose queries such as:  
*list the agents*  
*list the agents and the general offices is smith an agent*

These questions assume the tables in the *Rework Tables* display have been correctly processed.

2. As soon as verb entities have been added to the conceptual model, users may pose queries like:  
*who works in new york*  
*does smith work in new york*  
*does smith work in new york or los angeles*
3. As soon as subtype and adjective entities have been added to the conceptual model, users may pose questions like:  
*who works in the new york office*  
*how many agents work in the new york office*  
*who is the best agent in the new york office*

#### 10.1.1 Release to target users

While some of the end users have participated in the pre-customization process, other may not have. In addition, release to target users can be incremental or at a point where the customizer decides his customization is satisfactory for most uses. Very likely, release will occur on a somewhat incremental basis, the exact degree depending on the project, number of target users, and so forth.

Nonetheless, these things must be kept in mind, as the application is released to the target users:

- The project plan should stipulate whether the end users should immediately be informed of the scope of the application dictionary. The only case they would not

be informed is when the project specifically aspires to test the performance of the meta-knowledge function in the product.

- The target users should be encouraged send feedback to the customizer when they discover that words they use are not in the application dictionary.
- The target users should be encouraged to examine the validity of the answers they get to their queries, to make sure that the customization is correct, or identify potential bugs in the NLE.
- The target users should be encouraged to make estimates and note their reactions to performance times, so that their comments can be included in the project evaluation.

## 10.2 Application evaluation

After customization has achieved a satisfactory level and after the period of time for use by the target users has ended, the project should be evaluated. Evaluation is a well-defined process. Generally, we would like to know:

1. for the customization process:
  - How many entities did the conceptual model contain?
  - How many entities were verbs?
  - How many entities were subtypes?
  - How long did the customization process take per customization phase?
  - How long did the customization process take per entity?
  - How many requests for word additions occurred?
  - Were there any errors in writing the SQL image for subtype entities?
  - Are there any non-specified functions desirable?

Many other questions can be added to this list.

2. for the target users:
  - How many queries were asked?
  - How many queries were successfully processed and answered?
  - How-many requests for synonyms were made?
  - What syntactical characteristics did the most successful queries have?
  - What syntactic structures did the failed queries have?
  - Were the response times satisfactory?
  - Were the answers to the queries the expected answers?

This evaluation process will assist in developing future applications.

## 10.3 Customization performance implications

The linguistic performance of Ergo is dependent on the customization of the database. A conceptual model is created to link language terms to the tables in the database.

Concerning performance as customizers, one has to look closely at the exact number of expected paraphrases, eliminate unnecessary language and conceptual relationships, eliminate the unnecessary classifications, and synonyms. Obviously

getting rid of redundancy in relationships or classifications, takes away additional semantic trees and or CLF's since each interpretation derives one or more semantic tree, which is traversed by the engine. Each semantic tree may have, depending on model facts, one or more equivalent CLF's which then get translated to equivalent SQL statements. Classification of a term such as the verb *work* provides its semantic meaning. As the generation grammar converts the CLF's closer to the Paraphrase or DBLF form, we notice that lesser conceptual objects are maintained and the query now looks more like its English form.

There are times when it may be better at to create two verbs (*work*) let us say if there are 10 dependent subjects that share the same verb syntax. Therefore, for performance reasons, it may be better to have 5 of those subjects linked to one verb and the other 5 linked to a newly created verb. The same performance aspects are applicable in the creation of structured entities. The more entities a structured entity is composed of affect engine performance. If a sentence's syntax becomes too complex, this may result in either lexical or structural ambiguity, or both. Therefore, paraphrasing is essential to disambiguate sentence structures that are rendered ambiguous, such as these queries:

*Who works for Smith at LDG?*  
*Can flying planes be dangerous?*

Queries where a data value refers to any number of parts of a structured entity will have a process time proportional to the number of parts in the structured entity in question.

Ergo requires on an average, a couple of seconds to interpret the question. This time is dependent on the number of words in the vocabulary, and the length and complexity of the question.

# PART 2. Case Studies

## 11. CASE 1: JOINS AND INCLUSION DEPENDENCIES

Joins and inclusion dependencies are important in establishing inter- and intra-table relationships. In Ergo terms this means that entities representing concepts of their respective table can establish relationships or can be linked to other entities representing concepts of other tables.

The following are examples of how we use joins and inclusion dependencies between tables.

### A. Intra-table inclusion dependency:

Inclusion dependencies occur when a column is a subset of another column in the same table. This can happen between two columns within the same table.

We can say that the '*superior department*' column is an inclusion dependency of the '*department*' column because '*superior department*' refers to a subset of the data values present in the '*department*' column. The effect of this in entity mode is that '*superior department*' entity has to be classified as a subtype of the main concept SDEP (department\_table). It is this subtype relationship and not the inclusion dependency that enables 'superior department' to inherit the relationships of SDEP (department\_table). The inclusion dependency is needed however to optimize the SQL generated. Now we can ask the following question:

*Who is the manager of the superior department?*

### B. Inclusions between tables

Inclusion dependencies are more obvious and clearer when it exists between tables. It is used to represent columns of data values which are subsets of another column of data values. In the perspective of Ergo this means that these two columns represent different concepts.

*Figure 42 Intra-table inclusion dependency*

**Inclusions between tables:** An inclusion dependency is defined between manager and employee\_id, and a subtype definition between manager and employee entity. The subtype is responsible for generating this SQL statement that shows a join between manager and employee\_id, and not the inclusion.

```
SELECT DISTINCT X1.SALARY, X1.ID, NAME
FROM EPE.STAFF X1, EPE.ORG X2
WHERE X2. MANAGER = X1.ID
```

Therefore, in this example, manager will inherit all attributes of the department table. So if the employee (EMP table) has commission and salary, then manager will inherit or possess automatically commission and salary, and thus will enable a question such as

*What are the manager's salaries?*

*What are the manager's commissions?*

Columns linked by inclusion dependencies are normally defined as subtypes. It is important to emphasize that it is the subtype relationship (ENTITY MODE) that enables subtype entities to inherit the properties of their super-types. The inclusion dependency itself, is not required, but the SQL code will be optimized if it is included.

Therefore as long as an entity is a subtype of another entity, it can inherit the relationships of that entity, no matter what joins or inclusion dependencies have been defined.

### C. Single-column joins

Join-paths establish a connection between columns of different tables with the same concept. This enables the creation of OTHER relationships between ANY entities in the tables that are joined. In other words you can address individual entities from both tables in a single query. For example you can ask a query

*'List location of departments for staff.'*

**A join-path generates in the WHERE clause of an SQL statement a "Join predicate" i.e. where Col X = Col Y**

*Figure 43 Single-column joins*

If join-paths are defined, the tables will be joined in the WHERE-clause on the columns that are in the join-paths.

For the question:

*List location of department for staff*

If there is a join-path between STAFF.DEPT and ORG.DEPTNUM then the WHERE-clause will contain:

WHERE X2.DEPT= X1.DEPTNUM

If there is a join-path between STAFFAD and ORG\_MANAGER then the WHERE-clause will be:

WHERE X2.ID =X1.MANAGER

If both joins are defined, then the WHERE-clause will contain both:

WHERE X2.DEPT = X1.DEPTNUM AND X2.ID =X1.MANAGER

### D. Multiple-column primary key joins

Join-paths are necessary for creating a structured entity or when we wish to join a set of matching multiple column primary keys in one table to another set of multiple primary keys in another table.

If we define more than one direct join-path between two tables, Ergo assumes that at least one of the tables has a multi-column primary key. For example if a table has a



two-column key that can be found as a foreign key in another table with a three column primary key, you may join both columns to the corresponding columns of the second table.

A multiple-column primary key that has more than one join-path will generate several JOIN PREDICATES in the WHERE CLAUSE of an SQL statement Where COL W = COL X and COL Y = COL X

< The figure will be added in a later release >

Figure 44 Multiple-column primary key join

**E. Combined inclusion dependency and joins**

It is important to understand that what governs the need to establish both a join-path and an inclusion dependency on 2 tables depend very much on the context of the application. There are instances which show that tables may have an inclusion dependency and a join on the same column of two tables or may have an inclusion dependency and a join but on different columns of two tables.

In one example, a join is needed in order to enable us to create a structured entity. We join the tables using a pair of columns. However, if these pair of columns denote different concepts, an inclusion dependency is necessary. In this example, a join is needed for the structured entity, 'order info', but because the columns *prodnum* (in PRODUCTS) identified a DIFFERENT concept ('product') than *prodno* (in SALES), which is ('an ordered product') we need both a join-path and an inclusion dependency. The inclusion here is used in order to optimize the SQL and will not by itself enable us to create the structured entity.

< The figure will be added in a later release >

Figure 45 Combined inclusion dependency and joins

**F. Referential integrities, primary and foreign keys.**

The CT will automatically pick up information about keys and referential integrities from the system tables in e.g. SQL/DS and DB2. When the information from the database is fetched through the CT Table Download utility CTTABS, the two files SYSTAB.DAT and SYSCOL.DAT are created. The new thing is a third file - SYSREFDAT - which contains all information about the existing referential integrities in the database. The format of this file is:

CREATOR_1	TABLE_1	COLUMN_1	CREATOR_2	TABLE-2	COLUMN_2

Columns 1-3 of this file refer to those columns in the database, which are foreign keys. Columns 4-6 refer to primary key columns. Thus one row of this file represents a referential integrity or, in case of a multiple column key, it represents part of a referential integrity.

The CT will draw inclusion dependencies between all pairs of COLUMN\_1 and COLUMN\_2. The columns in COLUMN\_1 will be indicated as excluded and columns in COLUMN\_2 will be marked as primary keys.

In case several tables are linked in a chain, it is possible that one column will be both excluded and primary key at the same time. In the case where a multiple column key is part of a referential integrity, the CT will draw one inclusion dependency between every pair of foreign/primary key columns, i.e. if the key is built up by three columns, three dependencies will be drawn.

### Restrictions

When a column is not part of a referential integrity in the database, the CT will not show the column as a primary key even if the user has defined it as a primary key in the database.

See Rework Tables Section on page **Fel! Bokmärket är inte definierat.**, for discussions on joins.

## 12. CASE 2: TO BE ADDED

## 13. CASE 3: ENTITIES WITH NUMERICAL DATA - UNIT OF MEASURE

### A. Unit of measure

When you classify an entity as a subclass of `quantified_property`, you can specify its unit of measure. The Customization Tool contains units of measure for age, area, currency, duration, length, volume, and weight. When you specify a unit of measure, Ergo users can use the unit of measure in their questions, for example: You can:

- Add units to an existing quantified property.
- Create new quantified properties and units.

#### Defining Units of Measure

You can define units of measure for numerical entities. Do this by classifying the entity under a suitable heading of `quantified_property`.

- Select the classification and press *Apply*. Then press *Unit of measure*.
- Select the unit required and any scale factor.

Example:

If you want to use ounces in questions when the database values are in pounds, set the scale factor to 16.

#### Defining new units of measure

You can create your own units if the existing ones are not applicable. For example, to define nautical mile as a unit of distance:

- Create an entity for nautical mile.
- Classify it as an instance of unit of measure length.
- Give the entity the term nautical mile.

The new unit will be available for any entity classified under length.

#### Creating a new quantified property

To create the quantified property temperature, with unit of measure Kelvin:

- Create an entity for temperature. Make it a subclass of `quantified_property`.
- Create an entity for unit of measure temp. Make it a subclass of `unit of measure`.
- Between temperature and unit of measure temp, create the relationship:

*What is the unit type for temperature? Unit of Measure temp*

#### Creating a new unit of measure

Create an entity for the unit you want to use. Make it an instance of `unit_of_measure_temp`, and give it the term you want (Kelvin). You can now use Kelvin as the unit for any entity classified under temperature.

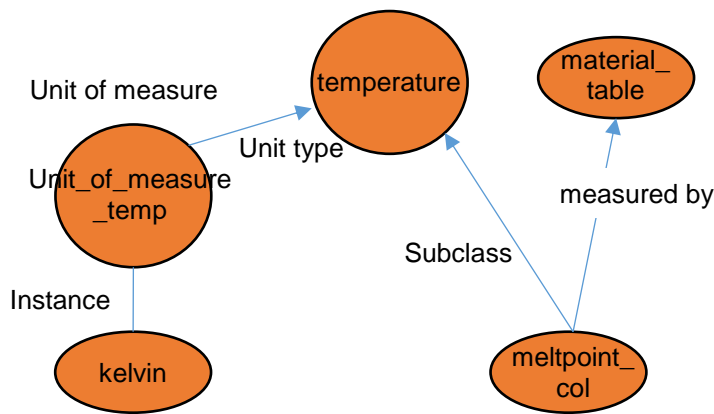


Figure 46 Units of measure

## B. Applying units of measure to numeric ranges

The following are some examples on how to use units of measure. It is important to note that data should be defined as numeric (decimal or integer) and not character to be able to perform this function.

*Which clients borrowed between 1000 and 100000 dollars?*

We can define dollars as a unit of measure for the 'borrowed amount entity'. We classify borrowed amount as a 'value' under quantified property and then select the unit of measure 'currency' instance of 'dollar' for this. Dollar is already predefined.

*Which employees have between 5 and 10 years experience*

We can define 'years' as a unit of measure for the experience entity. We can classify years as a duration under quantified property and then select the unit of measure duration instance of 'years'. Years have already been predefined.

*Which customers ordered between 15 and 100 items?*

This is a different case, since we need to create a new quantified property and a new unit of measure. Using the previous steps as guidelines, we first create a quantified property 'order' and classify it as a quantified property. Create a unit of\_measure\_order and make it a subclass of unit of measure Create a relationship between order and Unit\_of\_measure\_order:

*What is the unit type for orders? Unit of measure orders*

Create a new entity (item) and make it an instance of Unit\_of\_measure order. We can now classify 'ordered quantity' as orders and then select the unit of measure instance of 'items' as appropriate.

## 14. CASE 4: RANGES (DATE FUNCTIONS)

Entities that express date or time are classified as a subclass of DATE or TIME. This enables Ergo to perform date and time arithmetic to express a duration. A duration is a number that represents an interval of time which can either be a date duration or a time duration. Ergo automatically generates SQL to perform date and time arithmetic to express duration. Examples of SQL statements that may express the following contexts in terms of date and time durations are as follows:

*Who completed modules yesterday?*

SQL:

```
SELECT X1 module_name from table.module x1
WHERE DATE(ADATE)= DATE(CURRENT DATE -1)
```

An entity ADATE exists and is classified as date

*Who will complete modules tomorrow?*

SQL:

```
SELECT X1 module_name from table.module x1
WHERE DATE(ADATE)= DATE(CURRENT DATE + 1)
```

*What policies four (4) years old?*

SQL:

```
SELECT X1 policy name from policy
WHERE years(currentdate - issueyr ) <4
```

An entity 'issueyr' exists and is classified as date. Ergo allows you to define SQL to satisfy date durations to create new entity types.

*Who are the experienced agents?*

SQL:

```
SELECT X1 aname from table.agent x1
WHERE year(current date) - year(x1.hiredate) > 6
```

An entity 'experience' contains this SQL. Experience is defined as agents who have been in the company for 6 years or more. Experience is classified as duration.

*Who completed modules in June?*

SQL:

```
SELECT X1 module_name from table.module x1
WHERE date (policy) >= '06-01-98' and date(issueyr) <= '06-08-98'
```

We classify the 'june' entity as a duration and set up the SQL in it.

## 15. CASE 5: UNION OF 2 IDENTICAL NON-3NF TABLES

Some tables may contain historical information that acts as an archive of a current working table. This means that these tables are structurally similar, but their contents have different meanings. A customer CURRENT\_ORDER table versus a customer OLD\_ORDER table may illustrate this point.

CURRENT_ORDER			OLD_ORDER		
CUSNO]	MACHINE]	ORDERNO	CUSNO ]	MACHINE]	ORDERNO
123	3090	789	123	3090	233
234	AS400	434	455	4331	657
456	9370	233	444	4241	324

**Table 3 Non-identical tables**

It is possible that there exists a subset of employees in either table that belong to the other table. We are assuming here too that order numbers may be repeating after a given limit.

The main objective here would be to address a query, which asks information about all orders made by all customers. This would need the information from both tables. An intuitive approach to this would be to address both tables as a single concept in Ergo. It may have been conceived that the use of a VIEW to get the UNION of both tables would have been an ideal solution in order to create a single physical table that corresponds to a single language concept of *"all customer orders"*. Other means have to be investigated.

Note that an attempt to join the two tables on the primary key CUST\_NO cannot merge the information on the two tables because we are not only interested in finding the intersecting information (customers with both old and new orders), but the entire set of customers.

### Customization steps

We will try to bridge the information on both the CURRENT\_ORDERS table and the OLD\_ORDERS table by creating a new table with a single column that contains the information of all customers that belong to either table.

```
CREATE TABLE ALL_CUS_ORDERS  
(SUP CUSTNO CHAR(6))
```

Populate the table with customer numbers from both tables in one of several ways i.e. UPDATE, INSERT statements.

*Note:* Each application that updates old and current orders should update the new ALL\_CUS\_ORDER table

In TABLE Mode, join-paths must be established between the tables OLD\_ORDERS and CURRENT\_ORDERS and ALL\_CUS\_ORDERS. Join the primary key of CUSTNO of each table to the single column SUP\_CUSTNO.

In Entity Mode, we now have a new concept called All\_Customer\_Order corresponding to the table ALL\_CUS\_ORDER.

The single column entity called SUP\_CUSTNO will be customized as an identifier.

CREATE an entity called DISTINCT\_CUSTNO and make it a subtype of the entity SUP\_CUSTNO. This entity will be classified as an identifier The SQL image for this entity is as follows:

```
SELECT DISTINCT SUP_CUSTNO  
FROM ALL_CUSTORDERS
```

Make the identifier CUR\_CUST\_NO of the entity CURRENT\_ORDERS a subtype of entity DISTINCT\_CUSTNO.

Make the identifier OLD\_CUS\_NO of the entity OLD\_ORDERS a subtype of the entity DISTINCT\_CUSTNO.

Now that we have these in place, if we ask a question

*List all customers with 3090 orders.*

the NLE will look for the entity with the term 'customers' which is found in the entity SUP\_CUST\_ORDERS which is defined in the entity SUP\_CUSTNO.



## 16. CASE 6: CONCEPTUAL ENTITIES AND MULTIPLE-COLUMN CONCEPTS:

### Example 1: Contract application

Tables may contain imbedded concepts, which make them NON-3NF. These imbedded concepts are attributes of a table, foreign keys or primary keys that do not represent the main concept of a table. The following situation illustrates this point Hours Worked Table:

ID	CON_ID	CON_Name	CONTRACT VALUE	HRSWORK
123	10	SitePrep	20000	155
345	20	Concreting	15000	168
456	30	Bushwork	25000	167
557	10	SitePrep	20989	200

**Table 4 Non-3NF table**

The Primary Key of this table is ID. As the columns\_Con name and Con\_val are functionally dependent on CON\_ID, this table is in 2nd Normal form (NON-3NF)

Assume we have an employee table to be joined to the *Hours\_Worked* table and the *employee* table has an Employee name and ID columns. We can join the hours worked table to the *employee* table (Employee\_id) and exclude ID (*Table mode*).

### Solution 1

- Create a conceptual entity CONTRACT
- Classify CON\_ID as an identifier and make it identify the contract entity
- Classify CON\_NAME as a name and make it name the CONTRACT entity
- Establish the appropriate relationships between CONTRACT\_VALUE and CON\_VAL and the Contract Entity. For instance we can establish a measured\_by relationship or a possess relationship between the two entities.
- Create a new verb entity Work.
- We link '*Employee*' (table entity) and '*Number of hours worked*' to the '*worked*' choosing the syntax '*who work for something*' and '*who works for number of hours*'
- Link CONTRACT entity to verb Work with the proper syntax, for example the syntax '*who works on something*' to denote employee works on contracts.

### Solution 2

The use of views to create the appearance of a normalized database can also be used.

This is done through the following view:

```
CREATE VIEW V1 (CID, CNAME, CVAL)
AS SELCT DISTINCT CONID, CONANIE,CONVAL
FROM HOURSWORKED
```

We can now ask the questions like:

*Who worked on contract 123?*

*List the total number of hours worked by smith.*

*Who has the highest number of contracts worked on?*

*List contracts by hours worked.*

## Example 2. Mailing application

This is an exercise to demonstrate how to create new concepts from existing complex concepts such as structured entities:

Table: LOCATION

Col1: Location\_Id

Col2: City\_ID

Col3: City\_Name

Col4: State\_ID

Col5: State\_Name

Col6: Country\_ID

Col7: Country\_Name

- Create Conceptual Entity with ID and Name: CITY  
Link CITY\_ID to CITY with the relationship  
*What uniquely identifies City? CITY\_ID*  
  
Link CITY\_NAME to CITY with the relationship  
*What is the name of CITY? CITY\_NAME*
- Create Conceptual Entity with ID and Name: STATE  
  
Link STATE\_ID to STATE with the relationship  
*What uniquely identifies STATE? STATE-ID*  
  
Link STATE\_NAME to STATE with the relationship  
*What is the name of STATE? STATE\_NAME*
- Create Conceptual Entity with ID and Name: COUNTRY  
  
Link COUNTRY\_ID to COUNTRY with the relationship:  
*What uniquely identifies Country? Country\_ID*  
  
Link Country\_name to Country with the relationship  
*What is the name of Country? Country\_Name*
- Define Possessive relationships from the newly formed entities to either the table entity (location) if it is necessary and only as dictated by the context of the query.

- Define other relationships between other entities i.e. customer entity, and the formed structured entities:

Create Structured Entity: Location Name

Consists of: CITY\_NAME, STATE\_NAME, COUNTRY\_NAME

Link CITY\_ID, STATE, and COUNTRY entities to LOCATIONS and establish possessive relationships.

See Define Conceptual Table Entities under Edit Model- Phase 1 page 69

## 17. CASE 7: STRUCTURED ENTITIES

### Numerical data and report information:

A data processing department may wish to obtain reports that show the total resource usage i.e. the total CPU time, total Printer time and total DASD utilization for a given time period. This may be used for charge back reasons.

We create a structured ENTITY: RESOURCE USAGE

Entity: RESOURCE\_USAGE

Term: Total Resource Usage

Class: Number

Consists of CPU\_TIME, PRINT\_TIME, DASD\_TIME

Entity: CPU TIME

Term: total Cpu time

Class: Number

SQL: select CPU\_TIME from CPU\_TABLE

Entity: PRINT\_TIME term:

Total Print Time

Class: Number

SQL: Select PRINTER\_TIME from PRINT\_TABLE

Entity: DASD\_TIME

Term: total dasd time

Class: Number

SQL: Select DASD\_TIME from DASD\_TABLE

A variation of this would occur if the table information for CPU, PRINTERS and DASD contain horizontal information (i.e. a Column for every printer in-house) rather than putting all printers under one column. We then need to modify our SQL statements to reflect a horizontal summation of all PRINTER, CPU and DASD resources:

SQL: CPU A + CPU B + CPU C

SQL: PRINTER A + PRINTER B + PRINTER C

SQL: DASD A + DASD B +DASD C

See Structured Entities with Numerical Entity Components.

### Ordered-by reports:

Ergo handles order-by by including the columns ordered-by in the columns selected, i.e. *'List cpu time by system'* would select both CPU\_SECS and SYSTEM\_ID. In order to do this Ergo paraphrases the questions in the following way: *'List systems and their cpu times.'* Order by systems id. Therefore it is unnecessary to specify *'systems'* in the question, i.e. *'List systems and their cpu times by system'* is redundant since it mentions systems twice, and Ergo will create two paraphrases for that question.

The more terms in the order-by the longer it is going to take to for the engine to process the question:

*List CPU time by bid, by sid, by job system.*

There is therefore an upper limit to the number of order-by terms that can be inserted into the same sentence. This upper limit depends on the complexity of the sentence. We can ask questions with six order-by terms. Fortunately, structured entities can be used to group concepts and thereby reduce the number of terms in the order-by question.

*List CPU time by account order.*

Here account order is a structured entity, which consists of BID, SID, JOB SYSTEM. Realize that this might not be a realistic solution since if there are five possible order-by terms (BID, AID, SID, JOB NAME, and SYSTEM ID) and they could be used in any order, 120 different possible combinations of structured entities can be created. However, if any of these terms could be grouped together in composite entities, that would reduce the number of order-by terms in the questions and produce the SQL that are needed.

See Create Structured Entities for Identifiers and Names.

## 18. CASE 8: ENTITIES WITH NUMERIC DATA - MEASURED BY

Measured-by relationships enable one to rank the instances of the entities. We would like to find the agent with the highest commission amount. It is important to bear in mind that we should not define more than one measured by relationship for a given entity. If we have more than one measured-by the engine arbitrarily picks one of them.

Entity: Account Asset  
Term: account asset  
Class: Thing  
SQL: none

Entity: Account Commission  
Term: account commission  
Class: Thing  
SQL: none

Entity: Asset Amount  
Term: asset amount, asset  
Class: number, quantity  
SQL: Select x1.amount from account asset x1

Entity: Commission\_Amount  
Term: Commission  
Class: number, quantity  
SQL: Select x1.amount from commission x1

< The figure will be added in a later release >

*Figure 47 Measured by*

Classify the following entities 'number' and 'quantity' Commission Amount Asset Amount.

The reason for classifying these entities as both 'number' and 'quantity' is that it enables two types of questions to be asked. The 'number' classification enables questions of this type to be asked:

*List the commission amounts > 5000*

The 'quantity' classification enables questions of this type to be asked

*List the top ten commissions*

For the latter question to be asked a measured-by relationship between Commission Amount and Commission also has to be in place.

Establish a measured-by relationship between:  
Commission Amount and Commission  
Asset Amount and Account Asset

With this setup one can ask the following questions:

*List the asset years of the accounts*

*List the top ten commissions*

*List the accounts with asset amounts > 5000*

*List the top ten account assets*

*List the commission amounts > 5000*

*List the accounts with the top ten account assets*

See Entities Referring to numeric data.

## 19. CASE 9: NON-3NF TABLES - TABLES PROCESSED AS VERBS

Any table entity can be processed as a verb if the semantic context warrants it and if the join-paths are correctly set up. In this application, we would like to ask the following questions:

*Who studied chemistry and accounting?*  
*Who studied chemistry and accounting?*

This application contains several tables with multi-column primary keys. We mentioned earlier that some tables with multi-column keys can be treated as verbs, and the table (view), Education\_History, is one such case.

< The figure will be added in a later release >

*Figure 48 Tables processed as verbs*

### Customization

Table: V\_Education\_History should be processed as verb, say, 'study'.

Verb complement: *Who studies what*

Subject: 'employee' (entity: employee\_tent from EMPLOYEE)

Object: 'major' (new dummy entity: major\_dent)

Take away inclusion dependency between EMPLOYEE SNO and Education\_Hist SNO, although the join-paths remain.

Suggestions for column entities from V\_Education\_History:

Entity Name	Classification	Term	Linked to
SSNO ** DELETE			
MAJOR_CODE	Identifier	'college major id'	'major'
MAJOR_NAME	Name	'college major name'	'major'
DEGREE_CODE	Identifier	'college degree id'	'college degree'
DEGREE_NAME	Name	'college degree name'	'college degree'
FINAL_YEAR	Date	'degree date'	'college degree'
			'major'

Two 'dummy entities' must be created:

Entity Name	Classification	Term	Linked to
MAJOR_DENT	Thing	major	Identified by MAJOR_CODE Named by MAJOR_NAME
DEGREE_DENT	Thing	college degree	Identified by DEGREE_CODE- Named by DEGREE_NAME

Because of the join-path set up and because the table entity is processed as a verb, the correct SSNOs will be found when asking questions for information across these two tables. If we have had an inclusion dependency in this case, we would have assumed that a link was created between two different concepts, e.g., employee/educated\_employee. The join-path and the excluded column assume that



the concepts identified by the SSNO column are the same, which in this case it is. (It may not be so in other cases. It depends on our DB conception of the world and what we want to talk about, e.g., certified instructor. With *'certified instructor'* we would have to create a special *'certified instructor name'* entity and link it to *'certified instructor'* if the concepts are truly different.)

Also because of the join-path and the direct employee/verb(study) link, the employee names are automatically generated. Hence we can ask now:

*what major did smith larry study*

To ask *'what major did larry smith study'* simply change the order of the components in the employee name structured entity.

Another verb may be created: *'receive'*

Relations:

*'employees study majors'*

*'employees received degrees'*

Syntax:

*'majors in/on something'*

*'degrees in/on something'*

You may now ask these questions:

*Who studied chemistry and accounting*

*Who received a bachelor of science*

*Who received a bachelor of science in 1990*

*Who received a bachelor of science in chemistry in 1990*

*What degree did smith larry receive*

These suggestions work in principle for the other History tables.

## 20. CASE 10: VERBS WITH EMBEDDED PREPOSITIONS

### Reply to/by

As mentioned in the topic of verbs, there are instances where we would want to add the preposition or the verb complement directly to the term of the verb. This is necessary in a special case when the SYNTAX of the query demands that another preposition follow the verb complement.

Entity: Communication  
Term: Communications  
Synonym: none

Verb: Reply\_to  
Term: reply to  
Synonyms: respond to, answer to  
SYNTAX: Who replies to what by something

Entity: Replying officer  
Class: Person  
Term: Officer  
Synonym: replying officer

*Figure 49 Verbs with embedded prepositions*

Here we link the verb REPLY\_TO to the entity *Officer*, and establish the relationship:

What does someone/something reply to by? Replying officer,  
since *Replying Officer* is the object of the sentence.

We link REPLY\_TO to the entity '*communication*' and establish the relationship:

*What is replied to? Communications.*

This enables us to answer the question:

*List communications replied to by officer John Smith.*

See Prepositional Complements under Creating Verbs

## 21. CASE 11: ADVERBIALS

In customizing adverbs (of manner) we normally create a verb that contains the adverb itself.

Supertype: Employee  
Term: Employee

Subtype: usually\_hard\_working\_employee  
Subclass of: Employee  
Term: usually hard working employee (single noun)  
Synonym: NONE  
SQL: Select x1.empno...FROM...WHERE x1.work rating='H'

Note: Here we've created a subtype that 'adjectivizes' the adverbial, making it modify 'employee'.

Verb: usually\_work\_hard  
Term: usually work hard  
Synonym: work hard usually  
Relation to: usually\_hard\_working\_employee  
Who usually works hard? Usually hard working employee

< The figure will be added in a later release >

*Figure 50 An adverbial*

Here we have linked the verb phrase, 'usually works hard', to just that group of employees who usually work hard. There may be other kinds of employees who work in other ways. If we make them disjoint in the classification hierarchy, then other kind of employees, for example, lazy employees, should not inherit the (verbal) property of 'usually working hard.' In any case, if you were to ask the question, 'list the lazy employees who usually work hard', you'll get an empty report. But as long as this entity is not disjoint with manager or salesrep, you should also be able to get those managers who usually work hard. With this setup we can ask the questions:

*Who usually works hard*  
*Which manager usually works hard*  
*Which salesrep usually works hard*

Assume salesrep and managers are subtypes of employee.

Now if you ask, 'who usually works hard', you should also get only one paraphrase, at the most 2. See Handling Adverbs.

## 22. CASE 12. COMPOUND VERBS WITH SINGLE SUBJECTS

This application is an example of a compound verb, where the thematic noun (customer) is a subject of one verb (to play golf) and the object of another verb (to be born). Let us first set-up the corresponding tables:

Join-path set up:

Table	Column	Joined to	Column (&exclude)	Table
Customer	Customer number	->	Customer number	Demographics
Customer	Customer number	->	Customer number	Promotion
Customer	Customer number	->	Customer number	Lifestyle

Assuming that '*Customer table*' represents the concept '*customer*', this will enable '*customer*' to possess any attribute-column in any of the other tables. Thus the '*customer number*' entities in the other tables should be excluded. We are assuming that the '*customer number*' columns contain a set of data values that is a subset of the set of customer numbers in the '*customer number*' column (superset) of the *Customer* table. We are also assuming that we're talking about a 1-column primary key.

The other table entities may now be processed as verbs: '*lives*' (demographic), '*be promoted*' (promotion), '*like to*' (lifestyle). Customer may possess all the columns in these tables. All the columns may be linked to the verb (and any other verbs you may create) if the semantics of your desired queries warrants it. These verbs seem to lean toward the passive voice and are called deponent verbs.

We would like to be able to ask the question

*How many customers like to play golf and were born after 1970*

This involves joining the customer table, the lifestyle table, and the demographic table. We find out that this needs further analysis in customization as compared to questions such as '*How many golfers were born after 1970*', by creating another entity golfer and setting this up with the demographics table, thus the customization is more straight forward.

The problem is that in the sentence '*How many customers like to golf and were born after 1970*' '*customers*' is the subject of the '*like*' verb but it is the object of the '*born*' verb. Therefore Ergo cannot interpret the question.

The solution is to create a verb '*BE BORN*' and make '*Customer*' the subject of that verb (WHO BE BORN? CUSTOMER). The forms of the '*BE BORN*' verb (passive voice) would be: BE BORN, ARE BORN, WERE BORN, BEEN BORN, BEING BORN. Note that we are using the plural form of the '*be*' verb (are, were). If you want to use the singular form as well, i.e. '*Is there a customer that is born after 1970*' then you need to create another verb '*BE BORN*' that uses the singular forms of the '*be*' verb.

< The figure will be added in a later release >

Figure 51 Compound verbs

### Customization

- Create verb '*like to*' and conjugate '*like*' in the dialog box, and select correct syntax (synonym: '*like to play*')
- Create verb '*be born*' and conjugate '*be*' in the dialog box and select correct syntax
- Link customer ('*people*') to '*like to*' and '*be born*' as subject Link '*like to*' to '*sport*' (or the appropriate column)
- Link '*be born*' to '*birth date*' (which must be DATE data value)

Verb: Like\_To

Term: Like to Play

Syntax: Who likes to play what?

Conjugate: like to, likes to, liked to liked to, liking to

Verb: Be Born

Term: Be born

Syntax: who is Born? Customer

Conjugation: be born, are born, were born, been born, being born

In addition to this, the time relations need to be set up between '*BE BORN*' and the entities *Birth date* and *Birth year*. See Section on Verbs.

### 23. CASE 13: NON-3NF DATABASES - AMBIGUOUS DATA/MULTIPLE COLUMN PRIMARY KEY

A database is in non-3NF if data vales in individual columns do not have a single semantic interpretation. An example of this occurs if a column contains data values with multiple interpretations by design. The following table exemplifies this case.

Table: Account

BASE ID .....	SSNO TAX NBR-	SSNO FLAG
233	171643628	S
331	944677531	S
774	656424566	T
133	984566545	T

The table is not in third normal form because column SSNO\_TAX\_NBR contains a set of numbers, which are either a Social Security number for individuals or a Tax Number for firms. In order to distinguish between these two values, a separate flagging column SSNO\_FLAG contains the values 'T' to signify a tax number and 'S' to signify a social security number. We are interested in addressing the following types of questions:

- List Tax Accounts
- List Social Security Accounts

< The figure will be added in a later release >

Figure 52 Ambiguous data

#### Customization steps

We should also classify each entity of the table accordingly:

Entity Name	Classification	Term	Linked To
ACCOUNT	thing	account	-
SSN_NBR	name	SSN number	ACCOUNT
SSNO_FLAG	thing	indicator	ACCOUNT
BASE_ID	identifier	Base id	ACCOUNT

Since we want to address either tax accounts or social security accounts and these concepts are ambiguous and imbedded in a single column called SSNO\_TAX\_NBR, then we have to create two subtypes of the ACCOUNT entity called TAX\_ACCOUNTS and SOC\_SEC\_ACCOUNTS (Tax accounts and Social Security Accounts), and create corresponding identifiers to each or make each entity a self-identifying concept.

Entity: Tax Account  
 Term: tax account  
 Class: Subtype of Account  
 SQL: Select (x1.SSN\_TAX\_NBR)  
 From Account x1  
 Where X1.SSN\_TAX\_FLAG = 'T'

Entity: Social\_Sec\_Account  
Term: Social Security account  
Class: Subtype of Account  
SQL: Select (x1.SSN\_TAX\_NBR)  
From Account x1  
Where X1.SSN\_TAX\_FLAG = 'S'

Through these subtypes, we can now address the questions

*List Tax Accounts*

*List Social Security Accounts*

See Section on Subtypes.

## 24. CASE 14: SUBTYPES OF ENTITIES WITH MULTI-COLUMN KEYS

When you have a subtype of an entity that has a multi-column identifier, the identifier of the subtype must also consist of several columns. Since none of the individual columns can identify the super-type, we have to create a structured entity, consisting of the primary key columns and classify it as an identifier. Note that the table contains a multiple column primary key composed of the BASE\_ID and the FIRM\_ID. For every combination of a FIRM\_ID and BASE\_ID, this will yield a unique social security number or a tax account number.

Table: Account

BASE_ID	FIRM_ID	SSNO TAX_NBR	SSNO_FLAG
233	A33	171643628	S
331	D12	944677531	S
774	W23	656424566	T
133	S22	984566545	T

In this case we create a "conceptual" entity ACCOUNT-ID and classified it as an identifier, which identifies the ACCOUNT entity concept. Account id is a structured entity, which consists of the Base\_ID and Firm id entities. (Use the consists-of function to build relationships). SSNO TAX\_NBR will Name the ACCOUNT entity.

Figure 53 Subtypes with multi-column keys

Since the entities TAX\_ACCOUNT and SOC\_SEC\_ACCOUNTS are subtypes of an entity ACCOUNT, which has an identifier ACCT\_ID, these subtypes should also have identifiers, which are also structured entities.

For example TAX\_ACCOUNTS will have a TAX\_ID and SOC\_SEC\_ACCOUNTS will have a SOC\_SEC\_ID. TAX\_ID will be a structured entity consisting of TAX\_BASE\_ID and FIRM\_ID. Only one of the columns that are part of the identifier that identifies the subtype should be a new column where the appropriate subtype subset is selected. All other columns that are part of the identifier that identifies the subtype should be the original columns that are part of the identifier that identifies the super-type. In this example, SOC\_SEC\_ID will be a structured entity composed of SSNO\_BASE\_ID and FIRM\_ID. Note that TAX\_BASE\_ID and SOC\_BASE\_ID are newly created subtypes that select the appropriate subsets of accounts. Using SQL, TAX\_BASE\_ID selects SSNO\_TAX\_NBR = 'T' and an SSNO\_BASE\_ID selects SSNO\_TAX\_NBR = 'S'. On the other hand FIRM\_ID is the same component of the structured identifier ACCOUNT\_ID of the super-type ACCOUNT entity.

### Additional new entities

COLUMN	CLASS	LINKAGE
ACCOUNT_ID	Identifier	ACCOUNT
TAX_ID	Identifier Consists of TAX_BASE_ID/FIRM_ID	TAX ACCOUNT



FIRM_ID	identifier	ACCOUNT/TAX_ID
TAX_BASE_ID	identifier	TAX_ID
SSNO_BASE-ID	identifier	SSN ACCT_ID
SSN_ACCT_ID	Identifier	SSNO_ACCOUNT
	Consists of	
	SSNO BASE ID/FIRM_ID	

Note here that we could have implemented the above in parallel by using subtypes of the FORM\_ID and embedding the same SQL code in them. This is left as an exercise.

Since the TAX\_ACCOUNT and SOC\_SEC\_ACCOUNT entities are subtypes of the ACCOUNT entity, they inherit all its properties, including the NAME which is the SOC\_SEC\_NUMBER. Thus a query that asks *'What are the TAX\_ACCOUNTS with LOANS > 500'*, will then pick up the SOC\_SEC\_NUMBER, and using the TAX\_ID, it picks up the subset of rows where SSNO\_TAX\_NBR = 'T' and finally applies the SQL predicate from the LOAN entity WHERE LOAN = >500.

See Subtypes.

## 25. CASE 15: ENTITIES WITH NUMERICAL DATA – COUNTED-BY RELATIONS

When we need to count any number of occurrences of a certain entity we ask the question prefaced with *"How many"*. We could for instance ask:

*"How many Policies are there" or  
"How many employees are there in each department?"*

There are special cases when we need to use the COUNTED-BY relationship and to create entities in order to count occurrences of entities. Here are some examples:

### A. Single-value field

We can use the Count by relationship to address a single quantifiable field in a certain column.

In order to obtain the exact value of a single column within a row which expresses an amount or quantity. For instance if you have a table COUNTRY and you wish to get the population of a certain country.

Table: COUNTRY\_TABLE  
Col1: COUNTRY\_NAME  
Col2: COUNTRY\_ID  
...  
Col(n): INHABITANTS

We create a new entity POPULATION and classify this as quantity. Set up a counted by relationship between inhabitants and population column by choosing the relationship *'How many inhabitants are there? Population'*

Entity: Population  
Term: Population  
Classify: quantity

This will enable us to answer the question

*How many inhabitants does Sweden have.*

If we failed to add this counted by entity and relationship, and asked

*How many inhabitants does Sweden have*

We may get the answer: 1 because there is only one row in the *'inhabitants'* column that show the number of inhabitants of Sweden.

< The figure will be added in a later release >

*Figure 54 Count-by relationship for single-column value*

## B. Counting instances

As mentioned earlier, the engine should normally be able to count the number of occurrences (ex. tax accounts), however, if the entity has a structured identifier (composite key), then the engine will not be able to count because the engine does not know which entity to use in order to count. Therefore we create a COUNTED\_BY entity called TAX\_ACCOUNT\_COUNT, and classify it as a QUANTITY. In this case we are interested in counting the number of TAX ACCOUNTS which exist in a given client file.

Entity: Tax Account  
Terms: tax account  
Class: Subclass of Account  
SQL: none

Entity: Tax Account Count  
Terms: count  
Class: quantity  
SQL: Select count (distinct x1.SSN\_TAX\_NBR)  
From Account X1  
Where SSN\_TAX\_NBR = 'Y'

Entity: Tax Number  
Terms: none  
Class: identifier/ Consists of Tax\_base\_id  
acct-firm-id  
SQL: none

We then set up a counted-by relationship between TAX\_ACCOUNT\_COUNT and TAX\_ACCOUNT.

*How many tax accounts are there? Tax account count.*

Within TAX ACCOUNT COUNT, set up SQL to express

```
Select count (X1. SSN TAX-NBR)
From Account X1
Where SSN_TAX_NBR = 'Y'
```

You may now ask the following questions:

*How many tax accounts are there*

The drawback of this implementation is that we cannot ask for both "List tax accounts" and "How many tax accounts are there?" In order to fix this we have a workaround:

we create two TAX ACCOUNT entities and connect only one of these to TAX\_ACCOUNT\_COUNT, which satisfies the question

How Many tax accounts are there?

Note that *'List tax accounts'* will produce 2 paraphrases.

### c. Summing instances

We can also use the counted-by relationship to sum the instances of rows on a certain column. Note that this implementation is basically limited to satisfy the need to sum up the values for a given column but it cannot be related to other entities. For example: *'How many saws did customer A order?'* cannot sum the rows for a certain customer. However, *'how many saws were ordered'* can be summed by this technique. In this example, we are given a product table, and we would like to find the number of saws that were ordered. We create a conceptual entity called sum saws and classify it as a subclass of quantity, then setup the corresponding SQL to sum total number of ordered saws.

Entity: Ordered\_Saw

Terms: ordered saw, saw

Class: Subclass of order\_info (the order entity)

```
SQL: SELECT X1.ORDERNO  
FROM EPE.SALES X1  
WHERE X1.PRODNO = '205'
```

Entity: Sum\_Saws

Terms: sum orders for saw

Class: quantity

```
SQL: select sum(x1.quantity)  
from epe.sales x1  
where x1.prodno = 205
```

The Count saw entity is connected to the Ordered\_saw entity with a counted-by relationship.

The Ordered\_saw entity is the object of the verbs order and sell, i.e.

*What is ordered/sold? ordered saws*

The Ordered\_saw entity also possesses sales quantity. Ordered\_saw is a subtype of the Order\_Info entity and not the Ordered\_Products entity. These are the types of questions that you can ask with this model:

Hour many saws are ordered

How many ordered saws are there How many saws are sold

List the number of ordered saws

See Entities referring to Numerical Data.

## 26. CASE 16: NON-3NF DATABASE: MIGRATED HIERARCHICAL TABLES

There are instances when a DB2 database has been converted from a hierarchical database. The hierarchical sequential key gets stored in a column of a table, which then causes the interpretation of that column to be ambiguous. The column has meaning based on the key's functional interpretation and an interpretation based on its place in the hierarchy. For example, let us look at a database with two tables to simplify this concept. The tables are a BILLING table and a STAFF table. Note that these two tables do not have any referential integrity constraints established (i.e. no join-path can be established).

Billing Table

BILL NO	ITEM	RESPONSIBLE DRG	' LOC
001	PENS	VPO1_2LBM_GWCR_222	LDG
002	NAILS	VPO1_____212	LDG
003	CLIPS	VP02_2LBM_____232	LDG

Staff Table

ENO	ENAME	ORG_VP	ORG_BM	ORG_M	LOC
123	HANS	VP03	2LBM	1LGW	LDG
456	CHARLIE	VPO1	2LBB	1LGW	LDG
789	SONIA	VP02	2LPG	1LPG	PHL

The Billing table contains the Hierarchical sequential key in column Responsible\_Org. The data VP01\_2LBM\_GWCR\_222 means that a bill was generated through an order made by a first line manager (GWCR) who passes approval to a second line manager 2LBM then to a vice president (VPO1). The employee serial number of the manager that placed the order is 2222. In a hierarchical database this structure is more apparent since the root segment can contain data for the Vice President, a parent segment can contain data for the 2nd Line Manager, a child segment may contain data on the first line manager and so on.... If a Vice President was responsible for making the order, then the data would be represented as VPO1 -----3333'. If a second line manager was responsible for the order, then it would be represented as 'VP01 2LBM -----3333'. What this column then denotes is that it can contain information regarding WHO is responsible for the billing and the billing level i.e. first, second or third line.

The objective here would be to address the different concepts embodied in the hierarchical key in a language construct such that we can address both WHO were responsible for the orders and WHAT their level of management is.

### Customization steps:

For the customization of this case there are three cases. In all of the cases the data values in the RESPONSIBLE\_ORG column has to follow a pattern that allows us to disambiguate the values with LIKE statements. We disambiguate the column RESPONSIBLE\_ORG by creating two new concepts called 'Responsible in Billing' and

**Case 1:** There is no referential integrity between the Billing table and any other table, i.e. the hierarchical sequential key in the billing table (RESPONSIBLE\_ORG) does not refer to any other column in any other table.

We disambiguate the data values by creating subtypes that look like this:

Entity: VP\_LEVEL

SQL:

```
Select x1.responsible_org  
FROM billing x1  
WHERE x1.responsible org LIKE 'VP01____%'
```

With this subtype we can ask:

*What bills were handled at the VP level*

As you can see all we are able to get is the entire responsible org column. It is not possible to break it up into pieces. Also if you want to ask:

*Who is responsible for bill 001*

You can only get the entire responsible\_org column as an answer.

**Case 2a:** In this case there is a referential integrity between the hierarchical sequential key of the billing table and the hierarchical sequential key of another table. These two keys would be subsets of each other and the concepts of the keys would be the same, i.e. they would both represent organizational levels.

A join-path would be set up between the two hierarchical sequential keys. One of the keys would then be disambiguated as in Case 1. The other key would not have to be disambiguated since there would be a join-path between them.

**Case 2b:** This is the case where there is hierarchical sequential key that is related to several columns in another table. For instance, (as is shown in the current picture in Case 4) there would be columns representing VP, BM and M in another table and these columns would be related to the *responsible org* column in the billing table.

In this case, the information from the different tables cannot be related without adding columns to the billing table. These new columns would be a breakup of the responsible org column, and could be linked to the columns in the employee table.

# PART 3. Appendices

## Appendix A. The technology - natural language processing

Natural Language Processing is a branch of Artificial Intelligence (AI) technology with an ambitious goal of enabling people and computers to communicate in "natural" (human) language, such as English, rather than in a computer language, thus making computers more accessible to non-technical users. The study of natural language as used in computers is also known as computational linguistics.

Ergo uses computational linguistics and conceptual modeling, among other technologies to interpret natural language questions. These technologies are not based on keyword searches, but provide grammatical analyses and map input elements in the database. The input element is analyzed semantically and syntactically. It is then converted into internal logical representations that are mapped onto SQL through a conceptual model of the database. Ergo can then identify the correct tables, columns, and elements of the database that are required to answer the questions.

Ergo features and functions can be described under two fields of study involved in natural language processing:

Natural language understanding - the goal is to enable computers to language so that the computer understands people more readily comprehend instructions in ordinary human

Natural language generation - the goal is to have computers produce ordinary human language, so that people can understand computers more easily.

### Natural language understanding

In order to understand a natural language, Ergo uses the following methods and techniques for analyzing text.

#### A sample relational database

The users work with the data, which is kept in tables of the relational database. Let us take a simple case where the database contains some information about several countries:

Ergo.CO

CNTRY	CPTL	CNT_ID
FRANCE	PARIS	5
JAPAN	TOKYO	3
SWEDEN	STOCKHOLM	5
BRAZIL	BRASILIA	2

Ergo.EXPORT

PRDCR	PRDCT
JAPAN	CAR
JAPAN	TEXTILE
SWEDEN	STEEL



SWEDEN	TEXTILE
FRANCE	WINE
FRANCE	CAR
FRANCE	STEEL
FRANCE	TEXTILE

CNTNNT	ID
AFRICA	1
AMERICA	2
ASIA	3
AUSTRALIA	4
EUROPE	5

In the case of our simple example of the database, the structure of the database suggests that:

- Countries and capitals are related Countries and continents are related,
- Countries export products,
- Countries, capitals, and products are identified by their names, and
- Continents are identified by their names and an identification number.

Accordingly, the following conceptual model can be created for the database:

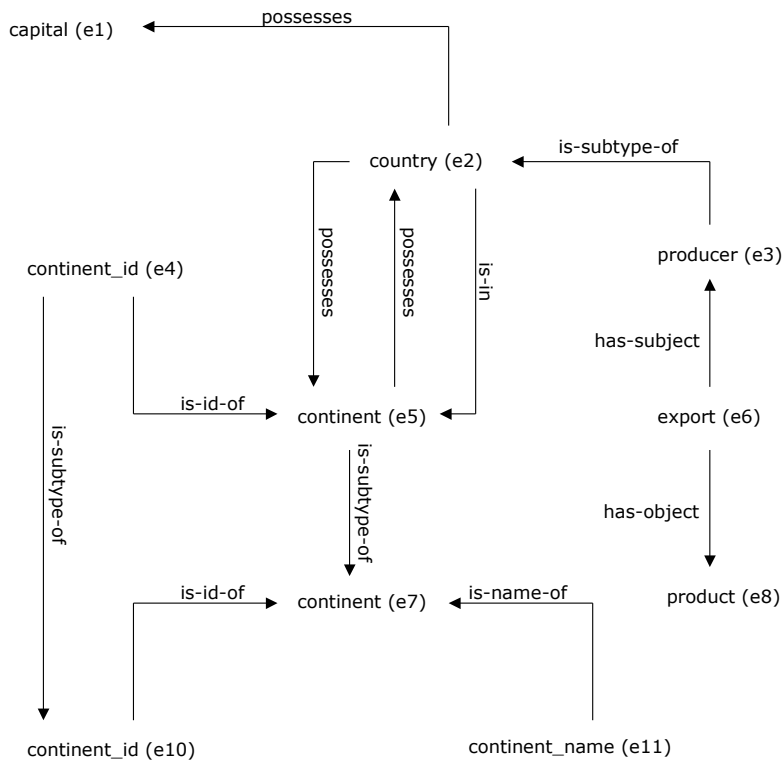


Figure 55 Conceptual model of the database

The model is stored internally as a set of logical facts:

```
possesses(e2, e1).
possesses(e2, e5).
possesses(e5, e2).
nom(e6, e3).
acc(e6, e8).
subtype(e3, e2).
subtype(e4, e10).
subtype(e5, e7).
identifies(e4, e5).
identifies(e10, e7).
name(e11, e7).
lp(e2, e5).
```

### Three-layered schema

The conceptual model is built in Ergo through the use of the Customization Tool. This mechanism provides a three-layered approach. The description of the world is made on the intermediate, or conceptual level, which fits very neatly between two other levels made up from the linguistic facts and the relational database. What this means is that a set of words representing a language governed by rules of grammar, can be represented in the conceptual model as entities and relationships.

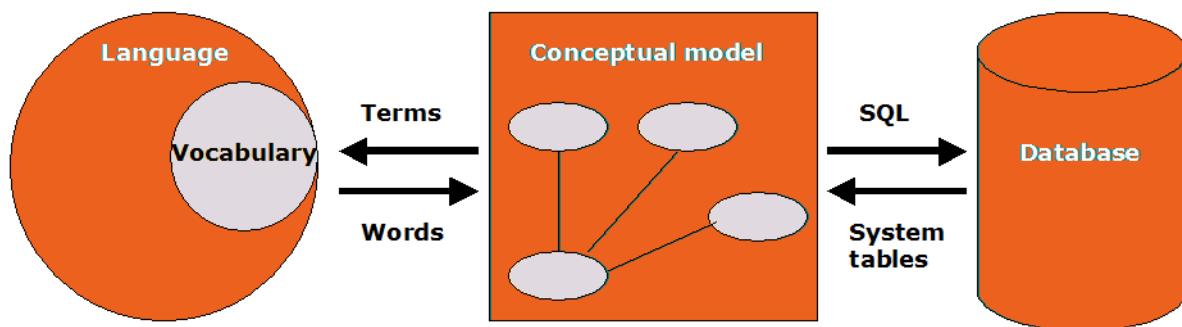


Figure 56 Three layered schema

These entities and relationships can easily be mapped to table names and columns in the database. An example of this mapping can be the word the "white house," which represents a physical object in the real world, which occupies time and space characteristics.

Therefore something exists that uniquely identifies the White House in the real world. On the other hand, if we compare this to the word the 'A president's house' there is no direct mapping to the real world, because this represents the concept of a set of

presidential houses, which is ambiguous in conceptual layer. Therefore we need to disambiguate this concept by mapping it via a conceptual model, to some physical object in the real world. In a database, there may be data that contain information on presidential houses such as the "*White House*", "*Casa Rosa!*", "*Malacanang Palace*" etc. Note that a name is different from a concept.

## Vocabulary definition

To connect the conceptual model to the natural language, the customizer must define the terms, which are likely to be used by the users.

The task of vocabulary definition includes connecting natural language terms to the entities and providing morphological information on them.

For example, for the sample country database, the following terms may be defined:

- (e1) -> 'capital' (t1) - noun, plural: 'capitals', pronoun: 'it'
- (e2) -> 'country' (t2) - noun, plural: 'countries', pronoun: 'it'
- (e7) -> 0 'continent' (t3) - noun, plural: 'continents', pronoun: 'it'
- (e8) -> 'product' (t4) - noun, plural: 'products' pronoun: 'it'
- (e6) -> 'export' (t5) - verb, forms: 'exports', 'exported', 'exported', 'exporting'
- (e6) -> 'produce' (t6) – verb, forms: 'produces', 'produced', 'produced', 'producing'

The customizer can define nouns, verbs, and adjectives and connect them to the entities. Note that one entity may be connected to zero, one, or several terms in natural language. Moreover, the same term may be connected to more than one entity.

There is no need to define base vocabulary, which is common to all applications (e.g., list, show, what, who, when, is). A base dictionary is shipped with Ergo.

The above definitions are also stored internally as logical facts:

```
image(e1, t1).
image(e2, t2).
image(e7, t3).
image(e8, t4).
image(e6, t5).
image(e6, t6).
category(t1, noun).
category(t2, noun).
category(t3, noun).
category(t4, noun).
category(t5, verb).
category(t6, verb).
term(t1, 'capital').
term(t2, 'country').
term(t3, 'continent').
term(t4, 'product').
term(t5, 'export').
term(t5, 'produce').
syntax(t1, 'capital'. 'capitals'. 'i'. nil).
syntax(t2, 'country'. 'countries'. 'i'. nil).
syntax(t3, 'continent'. 'continents'. 'i'. nil).
syntax(t4, 'product'. 'products'. 'i'. nil).
syntax(t5, 'export'. 'exports'. 'exported'. 'exported'. 'exporting'. nil).
syntax(t6, 'produce'. 'produces'. 'produced'. 'produced'. 'producing'. nil).
```

Dictionary entries are also created during the vocabulary definition, which are added to the application dictionary.

## Database links

In order to relate natural language queries to the relational database, it is necessary to link concepts of the model to the database.

Not all concepts are related to the database, but there can only be one direct database link for a specific concept. Of course, several different links may be introduced, if necessary, through definition of new concepts.

Concepts in our sample database will be connected to the database in the following way:

```
(e1) -> SELECT CPTL FROM Ergo.CC
(e2) -> SELECT CNTRY FROM Ergo.CO
(e8) -> SELECT PRDCT FROM Ergo.EXPORT
(e3) -> SELECT PRDCR FROM Ergo.EXPORT
(e11) -> SELECT CNTNNT FROM Ergo.CNT
(e4) -> SELECT CNT_ID FROM Ergo.CO
(e10) -> SELECT ID FROM Ergo.CNT
```

Note that the links to the database can be very complicated SQL expressions. is stored internally in DBLF format (explained later):

```
db(e2, set(V1, relation(Ergo.co(V1 = cntry)))).
db(e1, set(V1, relation(Ergo.co(V1 = cptl)))).
db(e4, set(V1, relation(Ergo.co(V1 = cnt-id)))).
db(e3, set(V1, relation(Ergo.export(V1 = prdcr)))).
db(e8, set(V1, relation(Ergo.export(V1 = prdct)))).
db(e11, set(V1, relation(Ergo.cnt(V1 = cntnnt)))).
db(e10, set(V1, relation(Ergo.cnt(V1 = id)))).
```

The information on links to the database

## Making queries

The following list shows examples of queries the user can make against the sample database vocabulary definition once the modeling and is completed as suggested.

*What is the capital of Sweden?*  
*What country is in what continent?*  
*List the countries of Europe.*  
*Who exports what?*  
*Does Japan produce wine?*  
*What products are exported by France?*  
*Show me the countries, which export cars.*  
*Isn't Brazil in Europe?*  
*What is the continent of Brazil?*  
*List the products.*  
*What countries in Europe export wine?*  
*What countries export all products?*

The process of translating natural language queries into SQL is described in the following sections.

### Dictionary look-up and syntactic analysis

Syntactic analysis determines the role played by each word in a sentence. For example, the meaning of "*The girl ate the candy*", and "*The candy ate the girl*", are different due to the roles played by each word in the sentence, since both sentences contain the same words. Syntactic analysis determines which word is the subject, and which word is the direct object. This is similar to sentence diagramming which involves some kind of parsing technique.

Parsing is the first step in processing a natural language query. The parser scans the input string character by character and, by using dictionary entries and analysis grammar rules, finds all possible combinations of patterns, which are grammatical.

The parser produces, as one of its outputs, a parse tree (or several parse trees in case of ambiguous queries), which describes how dictionary look-ups and application of different syntactic rules resulted in recognition of an input string as grammatical. Parse trees are built using a bottom-up chart parsing technique. For example, for the query '*who exports all products*', the following analysis grammar representation and rules can be used.

Q: *Who exports all products?*

Pronoun -> Who

NP -> pronoun

Verb -> exports

Ending -> s

Quantifier -> all

Noun -> product

Ending -> s

Final-punc -> .

np -> quantifier

noun vp-> verb np

S-> np vp

S-maj-> S final-punc

For example, for the query '*who exports all products*', the following parse tree (or syntax tree) will be produced by the parser:

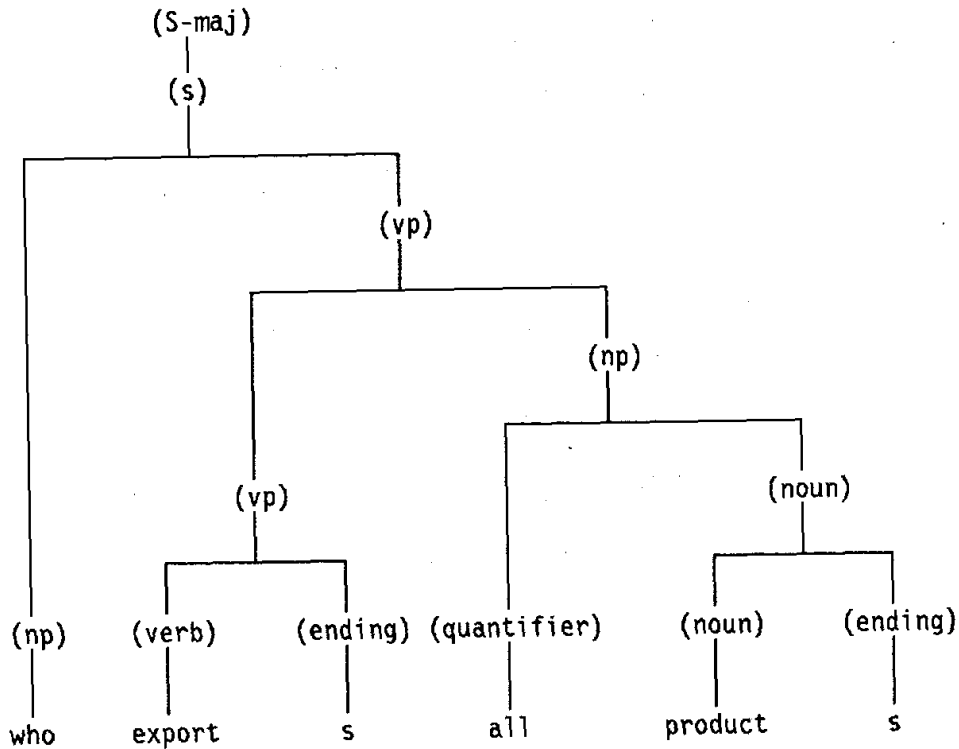


Figure 57 Parse-tree

Every syntactic rule (grammar rule) is associated with zero, one, or more semantic routines and the parser produces, as a second output, a semantic tree in association with each syntax tree. In short the syntactic tree is transformed into a semantic tree.

Semantic trees are nested structures containing semantic routines which, when executed, produce the internal representation form of the queries. For our query example, the semantic tree built by the parser will be a structure like:

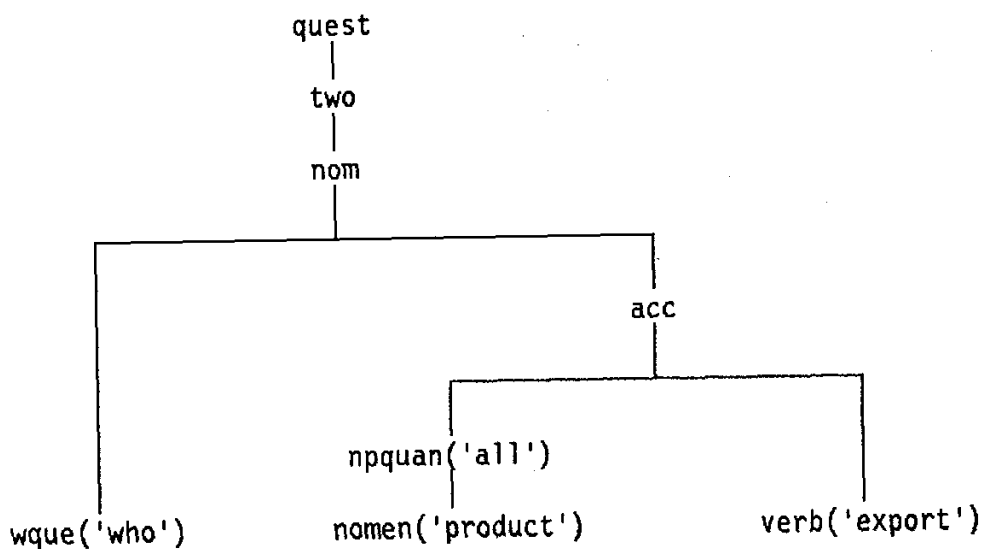


Figure 58 Semantic tree

The completed parse tree stores the syntactic relationship between each word in the sentence and acts as a template for building the appropriate semantic structure for the sentence, but does not represent the meaning of the sentence. This is where semantic processing is needed to interpret the meaning of the sentence.

## Semantic processing

Initially there is a definition for each word in a sentence. Each word definition is a mini-semantic structure. The individual meanings are combined according to how the parse tree indicates they syntactically relate to one another. When one word modifies another its semantic structure alters the semantic structure of the other word. These mini-semantic structures are merged word by word, then phrase by phrase, and ultimately, clause by clause. As more and more words participate in the structure, the meaning of the entire sentence is generated.

Evaluation of the semantic trees results in an internal representation form of the original query in which the meaning of the query, as far as the universe of discourse is concerned, is preserved.

The semantic tree of our example will internally be represented as something like:

```
quest(p01,
      two (p02,
          nom(p03,
              wque(p04, 'who'),
              acc(p05,
                  npquan(p06, 'all',
                      nomen(p07, 'product')),
                  verb(p08, 'export'))))))).
```

where p's are pointers to the internal structures created for the input query.

The semantic trees are evaluated by executing the nodes in the tree which all corresponds to semantic routines.

Using the conceptual model, the semantic routines generate a representation of the natural language queries called Conceptual Logical Form (CLF). CLF is a logical expression containing quantifiers, set operators, aggregate functions, a number of relations, variable names, entity names and constants.

The CLF representation of the query in the example we have been tracing will then be:

```
query(
  report,
  set (y1,
      all(y2,
          instance(e8, y2) ->
          exist(y3,
              instance(e6, y3) &
              acc(y3,y2) &
              nom(y3,y1)))))).
```

which simply means the user wants a report (as opposed to a yes/no answer or a chart) of everything which exports all products and by 'all products' here the user can only mean products which appear in the database.

The CLF is then verified, completed, and disambiguated by checking the conceptual model. For instance, if the verb 'export' is defined in the conceptual model such that it may take subjects from two different entities, then two CLF's must be produced, one for each case respectively. On the other hand, if there is no subject for the verb 'export' in the model, then the CLF must be aborted.

In the case of our example, navigating through the conceptual model results in a more completed CLF as follows:

```
query(
  report,
  set(y1,
    all(y2,
      instance(e8, y2) ->
        exist(y3,
          instance(e6, y3) &
          instance(e3, y1) &
          acc(y3,y2) &
          nom(y3,y1)))))).
```

where the added information is that the list the user wants is a list of countries (super-type of the concept e3, producer).

Contextual references are also resolved at this stage and results in completing the CLF statement by inserting the referents from the current or the previous CLF.

### Natural language generation - paraphrasing

In order to allow users to verify Ergo's interpretation of a query and/or to select from several viable interpretations, the meanings expressed in Conceptual Logical Form must first be converted to natural language. The natural language generator performs this task using a generation grammar.

To generate natural language from CLF, first CLF is translated into a set of trees called Initial Trees. Initial Trees contain such information as what the focus of the query is, what concepts are involved in the query, and what are the relationships between them. For example, the following set of trees will be generated for our sample CLF:

```
noun((id = 3).(group = 1).(scope = nil).(var = y1).(entity = e3).(focus = 1).nil).
noun((id = 1).(group = 1).(scope = nil).(var = y1).(entity = e8).(all = 1).nil).
verb((id = 2).(group = 1).(scope = y2.nil).(var = y1).(entity = e6).(acc = y2).(nom = y1).nil).
```

The natural language generator then combines the initial trees according to the generation grammar rules, and by using the syntactic information on the related



concepts from the conceptual model, generates an unambiguous natural language equivalent of the CLF.

In this example, the output of the natural language generator will be:  
List the countries that export all products

### SQL generation and optimization

When the user has confirmed; selected the interpretation, the corresponding CLF is translated into SQL. The process of SQL generation includes two steps: CLF to DBLF translation, and DBLF to SQL translation.

DBLF (Data Base-oriented Logical Form) has a similar format to CLF except that the entities are replaced by their DB-links from the conceptual model and the appropriate joins between SQL tables have been established.

In the case of our simple example, the following DBLF will be generated:

```
query(  
  report,  
  set(y1,  
    relation(Ergo.co(cntry=y1)) &  
    all(y2,  
      relation(Ergo.export(prdct=y2)) ->  
      relation(Ergo.export(prdct=y2,cntry=y1))))))
```

DBLF contains all information necessary to construct the SQL query. Moreover, Ergo can handle natural language queries, which cannot be expressed in (a single) SQL query. What DBLF is translated to is then beyond pure SQL and is called an Answer set.

The Answer set contains information on the data representation (yes/no, report, chart) and, in the cases when the data cannot be retrieved by a single SQL query, it also includes information necessary to create intermediate relations. Intermediate relations are temporary relations created as SQL tables containing data to be retrieved by a query later.

The Answer set is defined as

#### **SQL set:**

An SQL set consist of one single SELECT statement or the temporary-table triplet, CREATE TABLE, INSERT INTO, followed by SELECT. Temporary table constructs are necessary for generating correct SQL for some kinds of queries.

#### **Edit function:**

The edit function consists of the triplet First row, Last row, Ascending/Descending. It specifies how much of the data should be presented to the user.

#### **Display function:**

The Display function specifies how the answer should be presented to the user and is controlled by how the query was formulated. It consists of one of the following:

- REPORT

- CHART(chart\_type)
- YES NO(on\_data, on\_no\_data)

The SQL for our example will be:

```
CREATE TABLE t1 (entry , card)
```

```
INSERT INTO t1 (entry , card)
  SELECT x1.centry, COUNT( DISTINCT x1.prdct
  FROM test.export x1 GROUP BY x1.centry
```

```
SELECT DISTINCT x1.centry
  FROM test.co x1,t1 x3
 WHERE x1.centry = x3.centry
 AND x3.card = (
  SELECT COUNT( DISTINCT x2.prdct )
  FROM test.export x2)
```

```
NIL
REPORT
```

which results in a temporary relation created as the SQL table T1 with the columns CENTRY and CARD. The column CENTRY is copied from the column CENTRY of the table TEST.EXPORT and the values in the column CARD will be calculated as the number of distinct products (PRDCT column in TEST.EXPORT) related to each country.

The final query is made against the T1 table and will result in a list of countries, which export as many products number distinct products found in the database - only France in this case.

### Providing meta-knowledge for Ergo

In general, a system is said to have meta-knowledge if it can answer queries regarding own knowledge and about the kind of information available in its database.

In Ergo, meta-knowledge is implemented in a simple and elegant way. By keeping 'knowledge of Ergo' in SQL tables, users can use Ergo facilities to query that knowledge and thus request for meta-knowledge. In this way, there will be no difference between ordinary queries and meta-knowledge queries - neither from the user's point of view nor from Ergo's internal processing point of view. That means that meta-knowledge queries are also translated into SQL' and paraphrased as ordinary queries.

The conceptual model for meta-knowledge is created in advance as a part of 'base' conceptual model and is shipped to the users as part of Ergo.

Since the conceptual model is different for different applications, each application corresponds to a different set of tables where the model is stored. Although each set of tables may be considered as a different application when being customized, they can all be covered by one customization so that there is only one conceptual model

for meta-knowledge. The reason why we can have only one conceptual model for meta-knowledge of different applications is that the structure of the tables for storing meta-knowledge is the same even if their contents are different.

The SQL tables, which are used for storing the conceptual model are called with dummy unique names when customized. Then, during the CLF-to-DBLF translation, when these dummy table-names appear in the db-images (i.e., SQL statements associated with entities), they are replaced with the right table name corresponding to the current application. For example, the table where a list of all tables included in the application is kept can be called 'appl\_tabs' when creating the conceptual model for meta-knowledge. Then, when running 'entry' application, the CLF-to-DBLF translator replaces 'appl\_tabs' with 'cntrytabs' in the db\_mages.

The (related part of the) conceptual model for each application is stored in SQL tables automatically during the customization.

For our sample set of tables (which we will call 'entry' application) the tables contained in the meta-knowledge application include:

- EPEDB - System Database
- EPEDIATECH.APPLS -table name and creator name of application tables
- EPEDIATECH.ACCESS - information on application and authorized user
- CNTRYENTS - information on concepts defined in the model
- CNTRYIMAGE - information on links between the concepts and NL terms
- GROUP.applicTERMS - information on defined NL terms
- CNTRYRELS - information on relationships between the concepts
- CNTRYPRELS - information on relationships between the concepts involving prepositions.

For example, the CNTRYTERIVIS for entry application will include:

IMAGE	TERM	CATEGORY
T1	CAPITAL	NOUN
T2	COUNTRY	NOUN
T3	CONTINENT	NOUN
...	...	...
T6.	PRODUCE	VERB

where is are used for identifying the terms and joining among SQL tables.

By loading the conceptual model data in these tables and customizing them properly, the following sort of queries may be made in Ergo:

- What do you know?*
- What nouns do you know?*
- List the verbs you understand.*
- How many words are there?*
- Show country's category.*
- Is country related to export?*
- How many adjectives are defined?*

Besides the tables created during customization for storing the conceptual model, three more tables are contained in the *'meta-knowledge application'*.

The first table contains information on the applications. It is created during Ergo installation and updated whenever applications are created or updated. It is called APPLS and has one row of information for each application customized for Ergo (application name, customization date, customizer name, and application description).

The other two tables are SQL catalog tables, which are automatically maintained by SQL database management system. They are SYSTEM.SYSCATALOG where there is one row of information for each table and SYSTEM.SYSCOLUMNS where there is one row of information for each field of every table.

By properly customizing the above-mentioned tables, the user may ask queries such as:

*When was cntry application customized?*

*Who has created cntry application?*

*How many tables are there in cntry application?*

*List the columns of co table.*

*What applications are there?*

## **Appendix B. Defining the database structure in corpus-centric applications.**

Please contact Dialogue Technologies for more information on this subject.